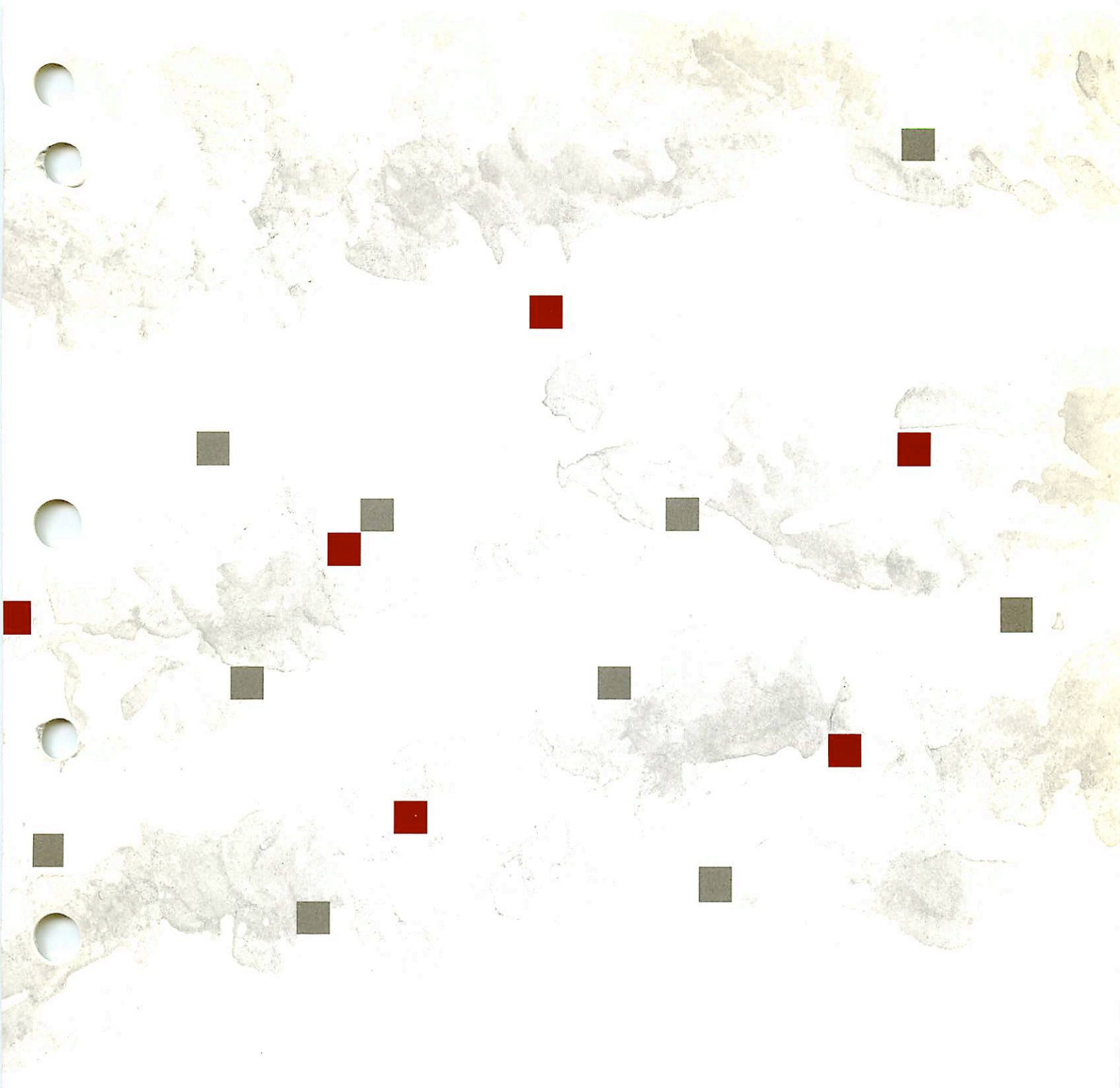


Prime®

COBOL85 Reference Guide

Release 1.0-22.0

DOC10166-1LA

COBOL85 Reference Guide

First Edition

Matthew Carr

This document reflects the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision 22.0 (Rev. 22.0).

Prime Computer, Inc., Prime Park, Natick, MA 01760

Copyright Information

Printing History

First Edition (DOC10166-1LA) October 1988 for Release 1.0-22.0

Credits

Project Editor: Barbara Fowlkes

Project Support: Wendy Blatt, Andre Lacroix, Roy Lemmon, Margaret W. Taft

Project Illustrator: Anna Spoerri

Document Preparation: Kathy Normington, Mary Mixon

Production: Judy Gordon

Composition: Julie Cyphers, Sharon Temple

Design: Leo Maldonado

How to Order Technical Documents

Follow the instructions below to obtain a catalog, a price list, and information on placing orders.

United States Only: Call Prime Telemarketing, toll free, at 800-343-2533, Monday through Friday, 8:30 a.m. to 5:00 p.m. (EST).

International: Contact your local Prime subsidiary or distributor.

Customer Support Center

Prime provides the following toll-free numbers for customers in the United States needing service:

1-800-322-2838 (Massachusetts)
1-800-541-8888 (Alaska and Hawaii)
1-800-343-2320 (within other states)

For other locations, contact your Prime representative.

Surveys and Correspondence

Please comment on this manual using the Reader Response Form provided in the back of this book. Address any additional comments on this or other Prime documents to:

Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, MA 01701

Contents

About This Book

Purpose and Audience

This document is a programmer's guide to the COBOL85 language as it is implemented on 50 Series™ systems, which run under the PRIMOS® operating system. The guide provides the necessary information for compiling, linking, executing, and debugging COBOL85 programs. It is designed to be used as a reference guide for an experienced COBOL programmer. If you are unfamiliar with the COBOL language, read one of the many commercially available instruction books. Examples are

- Feingold, Carl. *Fundamentals of Structured COBOL Programming*. Dubuque: Wm. C. Brown Company, 1978.
- Garfunkel, Jerome. *The COBOL 85 Example Book*. New York: John Wiley & Sons, 1987.
- Nelson, Donald. *COBOL 85 for Programmers*. New York: Elsevier Science Publishing Company, Inc., 1988.
- McCracken, Daniel D. *A Simplified Guide to Structured COBOL Programming*. New York: John Wiley, 1976.
- Philippakis, Andreas S. and Kazimier, Leonard J. *Advanced COBOL*. New York: McGraw-Hill, 1982.
- Popkin, Gary S. *Comprehensive Structured COBOL*. Boston: Kent Publishing Company, 1984.

Organization

This document has fifteen chapters:

- Chapter 1 introduces COBOL85, including supporting utilities, systems, and software, and differences from the ANSI standard.
- Chapter 2 provides information on the use of the COBOL85 compiler.
- Chapter 3 describes the process of loading and executing COBOL85 programs.
- Chapter 4 discusses the basic elements of the COBOL85 language.
- Chapters 5 through 8 discuss the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION, respectively.

- Chapters 9 through 12 focus on the processing of sequential files, indexed files, relative files, and magnetic tape files, respectively.
- Chapter 13 provides information on interprogram communication.
- Chapter 14 covers the SORT and MERGE verbs in detail.
- Chapter 15 discusses the source text manipulation statement: COPY.

The document also has sixteen appendices:

- Appendix A contains formats for all IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, and DATA DIVISION entries, and PROCEDURE DIVISION verbs.
- Appendix B contains COBOL85 reference tables.
- Appendix C discusses compile time and runtime error messages.
- Appendix D includes three tables listing PRISAM status codes and their corresponding COBOL85 status codes.
- Appendix E describes the COBOL85 interface to the Source Level Debugger.
- Appendix F lists all of the Standard COBOL features available in COBOL85, and Prime extensions to the ANSI standard.
- Appendix G lists all language elements on the ANSI obsolete language element list.
- Appendix H documents all of the differences between CBL and COBOL85 that may require attention during program conversion.
- Appendix I lists implementation-dependent features of COBOL85.
- Appendix J lists subroutines contained in the COBOL85 library.
- Appendix K contains a sample program listing that includes a map created with the –MAP compiler option.
- Appendix L contains a sample program followed by a cross-reference listing created with the –XREF compiler option.
- Appendix M discusses loading and executing COBOL85 programs with the SEG utility.
- Appendix N provides information on interactive file assignments with the –FILE_ASSIGN compiler option.
- Appendix O contains COBOL85 sample programs that process variable-length records.
- Appendix P is a glossary.

Suggested References

Refer to the following documents for more information about the COBOL 85 standard and about Prime resources and utilities.

The ANSI Standard

The definitive reference for standard COBOL 85 is *American National Standard Programming Language COBOL*, X3.23-1985. Every installation that uses COBOL85 extensively should have a copy of this standard, which may be obtained from American National Standards Institute, 1430 Broadway, New York, NY 10018.

Prime Documents

Several books describe other Prime utilities that help you with your programming on Prime equipment These documents are listed below.

- *System Architecture Reference Guide*, DOC9473-3LA
- *Instruction Sets Guide*, DOC9474-3LA
- *Subroutines Reference I: Using Subroutines*, DOC10080-2LA and its update UPD10080-21A
- *Subroutines Reference II: File System*, DOC10081-1LA and its updates UPD10081-11A and UPD10081-12A
- *Subroutines Reference III: Operating System*, DOC10082-1LA and its updates UPD10082-11A and UPD10082-12A
- *Subroutines Reference IV: Libraries and I/O*, DOC10083-1LA and its updates UPD10083-11A and UPD10083-12A
- *Subroutines Reference V: Event Synchronization*, DOC10213-1LA
- *PRIMOS User's Guide*, DOC4130-5LA
- *New User's Guide to EDITOR and RUNOFF*, FDR3104-101B
- *EMACS Primer*, IDR6107-183P
- *EMACS Reference Guide*, DOC5026-2LA
- *EMACS Extension Writing Guide*, DOC5025-2LA
- *Source Level Debugger User's Guide*, DOC4033-193L and its updates UPD4033-21A, UPD4033-22A, and UPD4033-23A
- *SyncSort/PRIME Reference Manual*, MAN10048-2LA
- *Programmer's Guide to BIND and EPFs*, DOC8691-1LA and its update UPD8691-11A
- *SEG and LOAD Reference Guide*, DOC3524-192L and its update UPD3524-21A
- *MIDASPLUS User's Guide*, DOC9244-2LA
- *PRISAM User's Guide*, DOC7999-4LA
- *FORMS Programmer's Guide*, PDR3040-163P
- *Assembly Language Programmer's Guide*, DOC3059-3LA
- *Magnetic Tape User's Guide*, DOC5027-2LA and its updates UPD5027-21A and UPD5027-22A
- *Advanced Programmer's Guide*, (4 volumes)
  - *Volume 0: Introduction and Error Codes*, DOC10066-3LA
  - *Volume I: BIND and EPFs*, DOC10055-1LA and its update UPD10055-11A
  - *Volume II: File System*, DOC10056-2LA
  - *Volume III: Command Environment*, DOC10057-1LA
- *Pascal Reference Guide*, DOC4303-4LA
- *CBL to COBOL85 Conversion Program Guide*, DOC10276-1PA

Other Sources of Information

In addition to the documents listed above, consider the following sources when looking for information about the COBOL85 compiler:

- The *Software Release Document*, also called an MRU, released at each software revision contains a summary of new features and changes in 50 Series user software.
- Online HELP files, which contain information on PRIMOS commands, can be displayed at your terminal. The command HELP DOCUMENTS provides an online list of current Prime manuals and updates.

Acknowledgment

Prime Documentation Conventions

The following conventions are used throughout this document. The examples illustrate the uses of these conventions.

Convention	*Explanation*	*Example*
UPPERCASE	In command or statement formats, words in uppercase bold indicate the names of commands, options, statements, and keywords. Enter them in either uppercase or lowercase. Underlined uppercase words are keywords. Uppercase words that are not underlined are optional words.	**RECORD IS VARYING**
italic	In command or statement formats, words in lowercase bold italic indicate variables for which you must substitute a suitable value. In text and in messages, variables are in non-bold lowercase italic.	COBOL85 ***program-name*** Supply a value for *x* between 1 and 10.
Abbreviations in format statements	If a command or option has an abbreviation, the abbreviation is placed immediately below the full form.	**SET_QUOTA** **SQ**
Brackets [ ]	Brackets enclose a list of one or more optional items. Choose none, one, or several of these items.	**LD** [**–BRIEF** / **–SIZE**]
Braces { }	Braces enclose a list of items. Choose one and only one of these items.	**CLOSE** { ***filename*** / **ALL** }
Braces within brackets [{}]	Braces within brackets enclose a list of items. Choose either none or only one of these items; do not choose more than one. If one of the items contains only uppercase words that are not underlined, that item is the default item.	[{ **COMPRESSED** / **UNCOMPRESSED** }]
Parentheses ( )	In command or statement formats, you must enter parentheses exactly as shown.	***data-name*** **(*index*)**

User input in examples	In examples, user input is in bold italic but system prompts and output are not.	`OK,` ***`RESUME MY_PROG`***
		`This is the output`
		`of MY_PROG.CPL`
		`OK,`

Ellipsis . . .	An ellipsis indicates that you have the option of entering several items of the same kind in the program.	***item-x*** **[*item-y*]...**
Hyphen –	Wherever a hyphen appears as the first character of an option, it is a required part of that option.	COBOL85 ***name*** **–LISTING**
Default indicator *	In a list of options, an asterisk indicates the default choice, if one exists. If you do not select an option, the system chooses the default option.	**–LISTING** * **–NO_LISTING**

Convention	*Explanation*	*Example*
Red text	Text printed in red indicates a Prime extension to or restriction on the ANSI standard.	**COMPRESSED**

Filename Conventions

Filenames may be comprised of 1 to 32 characters inclusive, the first character of which must be nonnumeric. Names must not begin with a hyphen (-) or underscore (_). Filenames may be composed only of the following characters: A-Z, 0-9, _ # $ - * . and /.

Note

On some devices, the underscore (_) may print as a back arrow (<-).

Convention	*Explanation*	*Example*
filename.language	Source file	MYPROG.COBOL85
filename.BIN	Binary (object) file	MYPROG.BIN
filename.LIST	Listing file	MYPROG.LIST
filename.RUN	Saved executable runfile	MYPROG.RUN

See Chapter 2 for an explanation of how the various names for source, object, listing, and runtime files relate to each other.

1

Overview of COBOL85

This chapter introduces COBOL85. It discusses the Prime implementation of ANSI Standard COBOL and highlights several Prime extensions to the ANSI Standard. The chapter also discusses the implementation of COBOL85 on Prime computers, and the characteristics of various programming environments. Finally, the chapter presents the broad range of system and file management resources that support COBOL85.

Language Standards

COBOL85 implements the intermediate level of American National Standards Institute (ANSI) COBOL, as defined in the document *American National Standard Programming Language COBOL* X3.23-1985, published by the American National Standards Institute, New York, 1985. COBOL85 also implements a number of high-level features of Standard COBOL. Appendix F provides a complete list of Standard COBOL features available in COBOL85.

Prime Extensions to the ANSI Standard

Prime has added a number of extensions to the ANSI standard. Throughout this book, Prime extensions are printed in red to identify them as such. Appendix F includes a complete list of these extensions. Some of the more important Prime extensions are listed here:

- You can use either the apostrophe or the single quotation mark as a delimiter for nonnumeric literals.
- You can specify data types COMPUTATIONAL-1 and COMPUTATIONAL-2 (single-precision floating point and double-precision floating point).
- You can specify the COMPRESSED or UNCOMPRESSED attribute in a file description.
- You can specify as many as eight levels of subscripting.
- You can specify subscripts that are subscripted themselves. You also can specify subscripts that are arithmetic expressions.

- You can use arithmetic expressions in place of data-names in many circumstances.
- You can use the CORRESPONDING phrase with IF, MULTIPLY, DIVIDE, and COMPUTE.
- You can use OTHERWISE with IF.
- You can use literals as operands of some clauses after INSPECT.
- You can use the EJECT, EXHIBIT, GOBACK, NOTE, READY TRACE, REMARKS, and RESET TRACE statements in the PROCEDURE DIVISION.
- In a CALL statement you can pass arguments of a level-number other than 01 or 77 (except 66 or 88).
- You can specify the RECORDING MODE clause in a file description.

COBOL85 on Prime Computers

Implementation

COBOL85 runs on the PRIMOS operating system. COBOL85 runfiles operate in Virtual Addressing mode (V mode); therefore, COBOL85 runs on all Prime models that support V mode. For more information on V mode, see the *System Architecture Reference Guide.*

Prime processors execute an extended set of instructions directly, including decimal arithmetic and character edits. They maximize execution time efficiency better than processors that substitute only an equivalent software routine. The Prime instruction sets are presented in the *System Architecture Reference Guide* and the *Instruction Sets Guide.*

Operating Environment

One version of PRIMOS exists for all 50 Series computers. PRIMOS features paged and segmented virtual memory management. The operating system is based on demand paging from disk with 2048 bytes per page. A page-sharing feature reduces overhead time. The system thus immediately and automatically satisfies paging requirements for the application program.

COBOL85 runs on PRIMOS at Rev. 22.0 and higher.

Conversion From CBL to COBOL85

Because of changes in ANSI Standard COBOL, you may need to modify programs that you compiled with the Prime CBL compiler, if you wish to recompile them with COBOL85. If you do recompile CBL programs with COBOL85, load and execute them with PRIMOS Rev. 22.0 or higher.

Appendix H lists the differences between CBL and COBOL85 and discusses conversion issues.

Program Environments

Under PRIMOS, you can compile, load, and execute COBOL85 programs in one of three environments:

- Interactive
- Phantom
- Batch

Interactive Environment

You can handle all phases of COBOL85 compilation through an interactive terminal. You can create, edit, compile, list, debug, execute, and save a program in a single interactive session.

You can initiate program execution directly. Programs run in real time, and the terminal is dedicated to the program during execution. You can display program output, as well as error messages, at the terminal. Common uses of the interactive environment include

- Use of interactive software, such as ED, EMACS, and DBG, to develop, test, and debug programs
- Execution of programs that require short execution time
- Execution of data entry programs, such as order entry or payroll

Note

All examples of compiling, loading, and execution in this book are given for an interactive environment.

Phantom Environment

The phantom environment allows you to execute programs without a dedicated terminal. **Phantom users** are programs that accept input from a command file instead of a terminal; output directed to a terminal is either ignored by the system or directed to a file.

Common uses of phantoms include

- Execution of programs that require long execution time, such as sort programs
- Execution of certain system utilities, such as the line printer spooler
- Execution of any program that must leave the terminal free for another use

See the *PRIMOS User's Guide* for more detailed information on command files and phantom users.

Batch Environment

Because the number of phantom users on a system is limited, phantoms are not always available. The batch environment allows users to submit noninteractive command files as

batch jobs at any time. The batch monitor (itself a phantom) queues these jobs and runs them, one to six at a time, as phantoms become available.

See the *PRIMOS User's Guide* for more detailed information on command files and batch processing.

System Resources Supporting COBOL85

COBOL85 shares with all Prime programming languages a broad range of system and file management resources. Such resources as system libraries, the text editor, the Source Level Debugger, and the BIND utility expand the scope and efficiency of the Prime interactive environment. Compatible file management systems provide standardized file management, and allow you to create and maintain data files separately from the application program. COBOL85 programs can call other programs compiled in any of several languages, including CBL.

Libraries

COBOL85 programmers may find system library functions and subroutines of use in some applications. Appendix J lists the modules in the COBOL85 library (COBOL85LIB).

Note

COBOL85 programs do not have access to any part of the CBL library. Likewise, CBL programs do not have access to any part of the COBOL85 library. All CBL and COBOL85 entry points are unique.

The *Subroutines References, I* through *V,* include complete descriptions of all Prime library and system subroutines.

Language Interfaces

Because all Prime high-level languages are alike at the object-code level, object modules produced by the COBOL85 compiler can call and be called by modules produced by the CBL, F77, Pascal, PMA, PL1, and PL1G compilers, provided that you observe the following restrictions:

- Write all I-O routines in the same language.
- Ensure that data types for variables being passed as arguments do not conflict.

For more information on language interfaces, see Table 13-1 and the *Subroutines Reference I.*

Editors

ED is a line-oriented text editor that enables you to enter and modify source code and text files. For information on how to use ED, see the *PRIMOS User's Guide* or the *New User's Guide to EDITOR and RUNOFF.*

EMACS is a separately priced screen-oriented editor. For information on EMACS, see the *EMACS Primer*, the *EMACS Reference Guide*, and the *EMACS Extension Writing Guide.*

Source Level Debugger

The Source Level Debugger (DBG) is a separately priced symbolic debugger that supports breakpoints, single-stepping, tracing, change of program flow, and modification of data. The COBOL85 compiler option that interfaces to the Debugger is documented in Chapter 2 and Appendix E. The Debugger is described in the *PRIMOS User's Guide* and the *Source Level Debugger User's Guide.*

PRIMOS SORT and SyncSort/PRIME

The SORT verb invokes either PRIMOS SORT or the separately priced SyncSort®/PRIME, whichever is installed on the system. A COBOL85 program can also invoke either PRIMOS SORT or SyncSort/PRIME through a subroutine call. For information on PRIMOS SORT, see the *PRIMOS User's Guide.* For information on SyncSort/PRIME, see the *SyncSort/ PRIME Reference Manual.*

BIND and SEG

BIND is the PRIMOS linking utility that creates Executable Program Formats (EPFs). Chapter 3 discusses BIND, and describes the commands needed to link programs. The *Programmer's Guide to BIND and EPFs* contains a complete dictionary of all BIND subcommands as well as a dictionary of EPF-related PRIMOS commands and the subroutines that apply to EPFs. The book also discusses programming restrictions and limitations on EPFs, and provides instructions on how to build an EPF library.

SEG is a loading and executing utility that creates static-mode runfiles, combining separately compiled program modules, subroutines, and libraries into an executable runfile. The SEG utility has many functions. Appendix M discusses the minimum functions necessary for a COBOL85 programmer. The *SEG and LOAD Reference Guide* presents advanced features.

Note

If your COBOL85 program is larger than one segment, you *must* use BIND to link and execute it. If your program is less than one segment, you can use either BIND or SEG.

Multiple Index Data Access System (MIDASPLUS)

MIDASPLUS™ is a system of utilities and subroutines for creating and maintaining indexed and relative files. MIDASPLUS provides the COBOL85 programmer with a transparent multilevel file structure. MIDASPLUS subroutines called automatically from the COBOL85 library routines perform all housekeeping functions on the index and data subfiles. MIDASPLUS files created by programs written in one language can be accessed and manipulated by programs written in other languages.

MIDASPLUS offers several features of interest to the COBOL85 programmer:

- A MIDASPLUS file can have as many as 17 alternate record keys.
- MIDASPLUS can retrieve multiple records for a single key value through the use of duplicate keys.
- A COBOL85 program can access a single MIDASPLUS file both sequentially and randomly.

For more information, see the *MIDASPLUS User's Guide.*

The Prime Recoverable Indexed Sequential Access Method (PRISAM)

PRISAM™ is a data management software system designed to provide automatic recovery, simple file structures, and strong performance in a transactional multiuser environment. Major features of PRISAM include

- Management of sequential, indexed, and relative files
- Support for user defined and mixed transactions
- System halt recovery
- Media failure recovery
- Software error recovery

For more information, see the *PRISAM User's Guide.*

Forms Management System (FORMS)

FORMS allows you to create and maintain screen forms for use in interactive application programs. These screen forms are useful for the applications programmer writing data entry programs in which data fields must be displayed in one or more formats. FORMS keeps the application programs, the forms, and the devices they use separated until runtime. Thus, changes to one area do not necessarily affect the other two. For information on how to use FORMS, see the *FORMS Programmer's Guide.*

2

Compiling the Program

In order to execute a COBOL85 program, you must first compile and link it successfully. This chapter explains how to compile COBOL85 programs. It discusses the COBOL85 command, compiler error messages and output files, COBOL85 file-naming conventions, and all COBOL85 compiler options. Chapter 3 explains how to link and execute a successfully compiled COBOL85 program.

The COBOL85 Command

The COBOL85 compiler is invoked by the COBOL85 command in the format

COBOL85 ***pathname [options]***
COB

where the variables in this format have the following meanings:

Variable	*Meaning*
pathname	The pathname of the COBOL85 source file. Pathnames are explained in the *PRIMOS User's Guide*.
options	One or more options controlling compiler functions, such as generating listings and invoking the Source Level Debugger. The options are explained in the section Compiler Options, later in this chapter. A hyphen (–) must precede each option.

COB is a valid abbreviation for the COBOL85 command and for the COBOL85 source filename suffix.

An example of the COBOL85 command is

```
OK, COBOL85 MYPROG -LISTING HOME>MYPROG.LIST
```

You can specify the source filename without the COBOL85 suffix. In this example, the COBOL85 compiler first looks for a source file named MYPROG.COBOL85. If MYPROG.COBOL85 does not exist, the compiler looks for a source file named MYPROG.COB. If MYPROG.COB does not exist, the compiler looks for a source file named MYPROG. If more than one of these files exist, COBOL85 compiles the first one it

finds. In this example, the compiler also opens a listing file having the pathname HOME>MYPROG.LIST.

If compilation is successful, the screen displays a message in the following format:

```
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]
[0 ERRORS IN PROGRAM:  <MFD>MYDIR>MYPROG.COBOL85]
OK,
```

If errors occur during compilation, error messages are output to the terminal. If you specify a listing or error file, error messages are also output to these files. After compilation, control returns to PRIMOS.

Compiler Error Messages

The general format of the error message is

```
ERROR err  SEVERITY sev  LINE line  COLUMN col  [  text    ]
diagnostic
```

where the variables in this format have the following meanings:

Variable	*Meaning*	
err	The COBOL85 error number	
sev	The severity number:	
	1	Observation
	2	Warning
	3	Fatal (no object code produced)
	4	Abortion of compilation (no object code produced)
line	The line where the error occurs	
col	The column where the error begins	
[*text*]	A description of the severity level and type:	
	SYNTAX	Fatal, caused by violation of syntax rules or format
	SEMANTICS	Caused by violation of syntax rules or general rules
diagnostic	The COBOL85 compiler error diagnostic	

An example of a compilation that generates an error message is

```
OK, COBOL85 MYPROG -LISTING
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]

ERROR 407  SEVERITY 2  LINE 15  COLUMN 19  [WARNING, SEMANTICS]
The initial value for "A9" exceeds the range of values allowed by
the PICTURE or by the default implementation size.  The initial value
may be truncated or unpredictable.
```

```
[1 WARNING IN PROGRAM: <MFD>MYDIR>MYPROG.COBOL85]
OK,
```

Compiler Output

By default, COBOL85

- Sends error messages to the terminal
- Aborts if the number of fatal errors exceeds 100
- Creates an object file having a default filename
- Suppresses the listing file
- Suppresses the error file

To alter these defaults, you can

- Use the –SILENT option to suppress error messages of the severity that you specify
- Use the –ALLERRORS option to continue compilation and error message generation beyond the limit of 100 fatal errors
- Use the –NO_ERRTTY to suppress the display of error messages at the terminal
- Use the –ERRORFILE option to request that all error messages be written to an error file
- Use the –BINARY option to specify an object pathname
- Use the –NO_BINARY option to suppress the object file
- Use the –LISTING option to request a source listing, and to specify a listing pathname
- Use the –EXPLIST, –MAP, –MAPSORT, –MAPWIDE, –XREF, and –XREFSORT options to create various expanded listings

For a discussion of all compiler options, see the section Compiler Options, later in this chapter. For a discussion of default file-naming conventions, see the following section.

COBOL85 Default File-naming Conventions

Four types of files can be involved in compilation: the source file, a listing file, an error file, and an object file. Of these, the listing, error, and object files are compiler-generated. If you specify filenames in the compile command line for the listing, error, and object files, the COBOL85 compiler opens these files under the filenames that you specify. If you do not specify filenames for these files, two default methods of file-naming are possible.

Normal Default File-naming

If the source filename ends in .COBOL85 or .COB, the default binary filename is *filename*.BIN. If you request a listing or error file, the default listing name is *filename*.LIST and the default error filename is *filename*.ERROR.

Thus, for MYPROG.COBOL85, if the compile command line is

```
OK, COBOL85 MYPROG -LISTING -ERRORFILE
```

the files produced are MYPROG.BIN, MYPROG.LIST, and MYPROG.ERROR.

Obsolete Default File-naming

The compiler uses an outmoded file-naming convention if the source filename does not end in .COBOL85. In this case, the default filename for a listing file is L_*filename*, and the default filename for an object file is B_*filename*. However, the default filename for an error file is *filename*.ERROR.

Thus, for a source file named SAM, if the compile command line is

```
OK, COBOL85 SAM -LISTING -ERRORFILE
```

the listing and object files are opened in the current directory as L_SAM and B_SAM, respectively. If errors are generated, they are recorded in SAM.ERROR.

Table 2-1 summarizes the two file-naming conventions.

TABLE 2-1
Default File-naming Conventions

Source	*Binary*	*List*	*Errors*
filename.COBOL85	*filename*.BIN	*filename*.LIST	*filename*.ERROR
filename.COB	*filename*.BIN	*filename*.LIST	*filename*.ERROR
filename	*B_filename*	*L_filename*	*filename*.ERROR

Setting Default Names From PRIMOS

If you want the listing or object files to have default names other than those outlined above, you must invoke the PRIMOS commands LISTING or BINARY with arguments prior to compilation. The *PRIMOS Commands Reference Guide* discusses these commands.

Default Directories

If you specify a pathname for the source file, such as

<MFD>*directory1*>SAM

where *directory1* is not the current directory, the compiler output files are, nevertheless, created in the current directory, unless you specify another directory. This convention is true for both file-naming conventions discussed above.

Compiler Options

This section discusses all COBOL85 compiler options in alphabetical order. Most of the options come in pairs, which act as switches to enable or disable a particular action. Table 2-3, at the end of this chapter, provides a summary of the compiler options and abbreviations.

You can specify compiler options in any order. However, if you specify conflicting or redundant options, the compiler generates an error message.

Some compiler options require that you specify an argument in addition to the option. The argument follows the option, and is *not* preceded by a hyphen. If applicable, the following discussions include lists of valid arguments.

In the absence of user-specified options, the compiler assumes certain options by default. These default options are indicated by asterisks (*) throughout this section. Your System Administrator can change these default options.

*** –64V**

–64V specifies the object code addressing mode. 64V is a segmented virtual addressing mode for 32-bit machines. It is the default and only addressing mode available for COBOL85.

–ALLERRORS
–ALL

–ALLERRORS produces diagnostics for all errors detected in the source program. If you do not specify –ALL, the compilation aborts when the compiler has issued 100 fatal diagnostics.

–ANSI_OBSOLETE / * –NO_ANSI_OBSOLETE
–AO / –NAO

–ANSI_OBSOLETE generates observations for all language elements that ANSI X3.23-1985 lists as obsolete language elements. The presence of an element on this list means that the element will not appear in the next ANSI COBOL standard. However, this does not imply that the element will be removed from the COBOL85 compiler.

–NO_ANSI_OBSOLETE suppresses generation of observations for obsolete language elements.

*** –BIG_TABLES / –NO_BIG_TABLES**
–BT / –NBT

–BIG_TABLES enables compile time checks for segment spanning of multi-segment data blocks.

–NO_BIG_TABLES disables compile time checks for segment spanning of multi-segment data blocks, and thus reduces compile time. Use –NO_BIG_TABLES only when you are certain that you have correctly defined any multi-segment data blocks.

*** –BINARY [*pathname*] / –NO_BINARY**
–B / –NB

–BINARY produces an object file having the name that you specify in *pathname*. By default, –BINARY produces an object file having the name *source-program*.BIN.

–NO_BINARY suppresses creation of the object file. Use this option when you want only a syntax check or source listing.

*** –CALCINDEX / –NO_CALCINDEX**
–CALC / –NCALC

–CALCINDEX performs address calculations for indexed references at the time the indexed item is referenced, instead of when the index is changed by a SET, SEARCH, or PERFORM statement.

–NO_CALCINDEX disables address calculations for indexed references at the time of reference.

–COMP / * –NO_COMP
/ –NCOMP

–COMP extends the range of computational variables from that specified by the PICTURE clause to the maximum value that can be contained by the hardware for that computational data type. Table 2-2 shows the range of computational variables allowed if you specify the –COMP option. Size errors do not occur if the precision of runtime values falls within this range. However, size errors do occur if the precision of runtime values exceeds this range. Specifying the –COMP option in a program that uses the COMP or BINARY data types enhances runtime speed because it eliminates some size checks and allows use of the native instruction set with the data type.

TABLE 2-2
Range of Computational Data Types

PICTURE Clause	*Storage*	*Range*
S9(1) through S9(4)	16 bits (2 bytes)	–32768 through 32767
S9(5) through S9(9)	32 bits (4 bytes)	–2147483648 through 2147483647

–NO_COMP suppresses use of the full hardware capacity (15 or 31 bits) for COMP fields.

See Chapter 4 for a full discussion of computational data types.

–CORRMAP / * –NO_CORRMAP
–CORM / –NCORM

–CORRMAP inserts a map of CORRESPONDING matches into the source listing.

–NO_CORRMAP suppresses insertion of a map of CORRESPONDING matches into the source listing.

–DATA_REP_OPT / * –NO_DATA_REP_OPT
–DRO / –NDRO

–DATA_REP_OPT performs optimization for decimal data type arithmetic operations when possible. This optimization enhances runtime performance, but increases compile time. This option includes –COMP implicitly.

Do not use this option if your program includes arithmetic operations on values that exceed nine significant digits. Consider all trailing zeros, up to the limit that you specify for the variables in the DATA DIVISION, as significant digits. If an arithmetic value exceeds nine significant digits, and overflow occurs, results are undefined. Use the –SIGNALERRORS compile option with –DATA_REP_OPT, so that the program aborts if overflow occurs.

–NO_DATA_REP_OPT suppresses optimization for decimal data type arithmetic operations.

–DEBUG / * –NO_DEBUG
–DBG / –NDBG

–DEBUG modifies the object code so that it runs under the Source Level Debugger (DBG). The code is not optimized, and execution time increases. Appendix E gives more information on the debugger interface.

–NO_DEBUG suppresses generation of debugger code.

–ERRORFILE / * –NO_ERRORFILE
–ERRF / –NERRF

If the compiler issues any diagnostics to the terminal, –ERRORFILE produces a file called *program-name*.ERROR, containing all diagnostics issued.

–NO_ERRORFILE suppresses creation of a file containing diagnostics.

*** –ERRTTY / –NO_ERRTTY**
–ERRT / –NERRT

–ERRTTY displays error messages at the terminal.

–NO_ERRTTY suppresses this function.

–EXPLIST / * –NO_EXPLIST
–EXP / –NEXP

–EXPLIST creates a source listing that includes an assembly code listing. Each statement in the source listing is followed by the Prime Macro Assembler (PMA) statements into which it was compiled. To use the listing, a knowledge of PMA is necessary. For information on PMA, see the *Assembly Language Programmer's Guide.*

–NO_EXPLIST suppresses printing of assembler statements in the listing.

–FILE_ASSIGN / * –NO_FILE_ASSIGN
–FA / –NFA

–FILE_ASSIGN enables manual runtime file assignment. For a detailed description of this option see Appendix N.

–NO_FILE_ASSIGN disables manual runtime file assignment.

*** –FORMATTED_DISPLAY / –NO_FORMATTED_DISPLAY**
–FDIS / –NFDIS

–FORMATTED_DISPLAY formats output of numeric fields. It suppresses leading zeros, moves the operational sign to the left of nonspace numeric fields, and inserts implied decimal points. Use of this option does not affect the way numeric fields are stored in memory.

–NO_FORMATTED_DISPLAY specifies that numeric fields be displayed just as they are stored in memory; that is, they are not formatted.

–FULL_HELP
–FH

–FULL_HELP is similar to the –HELP option, except that in addition to the usage summary, the screen displays a description of each compiler option.

–HELP
–H

–HELP displays a list of compiler options and their functions.

Enter this option with no arguments, as follows:

```
OK, COBOL85 -HELP
```

–HEXADDRESS / * –NO_HEXADDRESS
–HEX / –NHEX

In conjunction with –MAP or –EXPLIST, –HEXADDRESS prints addresses in the listing file in hexadecimal instead of octal notation.

–NO_HEXADDRESS suppresses the printing of addresses in the listing file in hexadecimal notation.

–LISTING [*argument*] / * –NO_LISTING
–L / –NL

–LISTING generates a source listing. The basic source listing contains the date and time of compilation, the options in effect, the source text, and a list of errors. You may use the arguments TTY, *pathname*, and SPOOL to specify the following actions:

Argument	*Action*
TTY	Displays the listing at the terminal.
pathname	Writes the listing to a specific file.
SPOOL	Spools the listing directly to the line printer.

–NO_LISTING suppresses the source listing.

–MAP / * –NO_MAP
–MA / –NMA

–MAP produces a listing that includes a map of data and procedure names. A sample data map with discussion appears in Appendix K. This option includes –LISTING implicitly.

–NO_MAP disables the creation of a map listing file.

–MAPSORT / * –NO_MAP
–MAPS / –NMA

–MAPSORT produces a listing that includes a map of data and procedure names sorted alphabetically. This option includes –LISTING implicitly.

–NO_MAP disables the creation of a map listing file.

–MAPWIDE / * –NO_MAP
–MAPW / –NMA

–MAPWIDE produces a listing that includes a map of data and procedure names. The map information is printed in 108-character lines instead of 80-character lines. This option includes –LISTING implicitly.

–NO_MAP disables the creation of a map listing file.

–OFFSET / * –NO_OFFSET
–OFF / –NOFF

–OFFSET produces a listing that includes the object address (octal offset from the Procedure Base register) of each PROCEDURE DIVISION statement. The format of each address is

line number*: *halfword offset

This option includes –LISTING implicitly.

–NO_OFFSET disables printing of object addresses in the listing file.

*** –OPTIMIZE [*decimal-integer*]**
–OPT

–OPTIMIZE controls the optimization phase of the compiler.

Optimized code runs more efficiently than nonoptimized code, but takes longer to compile. The use of this option performs such automatic tasks as keeping track of register contents and evaluating constant expressions.

You can set optimization at one of the following levels by specifying *decimal-integer* following –OPTIMIZE:

Level	*Meaning*
0	Turns optimization off
1	Same as level 0
2	Full optimization (default)
3	Reserved for future designation
4	Reserved for future designation

–PRODUCTION / * –NO_PRODUCTION
–PROD / –NPROD

–PRODUCTION is similar to –DEBUG, except that the code generated does not permit insertion of statement breakpoints. Execution time is not affected.

–NO_PRODUCTION disables this partial debugger functionality.

–RANGE / * –NO_RANGE
–RA / –NRA

–RANGE checks for out-of-bounds values in array subscripts and in OCCURS DEPENDING ON *data-names*. The compiler inserts error-checking code into the object file. During execution, if an item takes on a value outside the range that you specify in the DATA DIVISION entry, a runtime error diagnostic is issued, and the program terminates. The diagnostic contains the source line number, the runtime subscript or the value in the

OCCURS DEPENDING ON *data-name*, and the legal boundaries for the array reference that caused the error.

Because –RANGE increases both the compile time and execution time of your program, use it only as a debugging tool.

–NO_RANGE disables runtime array range checking.

–RANGE_NONFATAL / * –NO_RANGE
–RNF / –NRA

–RANGE_NONFATAL functions the same as the –RANGE option, except that program execution continues. However, continuing the program may lead to unpredictable results depending on the subsequent use of the invalid array reference.

–NO_RANGE disables runtime array range checking.

–RMARGIN
–RMARG

–RMARGIN extends Area B of each source line to column 160.

–SIGNALERRORS / * –NO_SIGNALERRORS
–SIG / –NSIG

–SIGNALERRORS aborts execution and signals the PRIMOS ARITH$ condition if division by zero, arithmetic overflow, or a conversion error occurs. Exponentiation errors always abort execution. This option overrides any ON SIZE ERROR actions that you specify in the program.

–NO_SIGNALERRORS does not signal a condition for such errors. Results are undefined if you do not specify ON SIZE ERROR in the program.

*** –SILENT** [*decimal-integer*]
–SI

–SILENT suppresses the display of error and warning messages of the severity you specify in *decimal-integer*. The error and warning messages are also omitted from any listing files generated. *decimal-integer* specifies one of the following levels:

Level	*Meaning*
0	Displays all messages
1	Suppresses observations; displays warnings and fatals
2	Suppresses observations and warnings; displays fatals
3	Suppresses all messages

–SILENT with no argument defaults to –SILENT 1. An appearance of

```
-SILENT -0
```

at the top of the listing file means that you did not specify –SILENT, and that all error messages are displayed by default. (Levels of messages are explained in the section Compiler Output, earlier in this chapter.)

–SLACKBYTES / * –NO_SLACKBYTES
–SLACK / –NSLACK

–SLACKBYTES issues a severity level 1 diagnostic for each elementary or group item that the compiler aligns on a 16-bit boundary. For example, COMP, BINARY, COMP-1, and COMP-2 data items must be allocated on 16-bit boundaries. When any of these items are members of a group, they are allocated on the current available location *if* that location is on a 16-bit boundary. Otherwise, they are shifted one byte to the right. The group item that contains the items is also shifted, if required. See the section Data Representation and Alignment in Chapter 4.

–NO_SLACKBYTES suppresses these diagnostics.

–SPACE / * –TIME

–SPACE specifies that object code size reduction be given preference over runtime speed during the optimization phase of the compiler.

–TIME specifies that runtime speed be given preference over object code size reduction during the optimization phase of the compiler.

–STANDARD / * –NO_STANDARD
–STAN / –NSTAN

–STANDARD generates observations for all Prime extensions.

–NO_STANDARD suppresses observations for Prime extensions.

–STATISTICS / * –NO_STATISTICS
–STAT / –NSTAT

–STATISTICS controls printout of compiler statistics.

The terminal displays a list of compilation statistics, such as program size and resources used during compilation, after each phase of compilation. Headings identify each phase:

Heading	*Phase*
LEX	Lexical analysis and parsing of IDENTIFICATION DIVISION
ED_PARSE	Parsing of ENVIRONMENT DIVISION
DD_PARSE	Parsing of DATA DIVISION
PD_PARSE	Parsing of PROCEDURE DIVISION
ALLOCATOR	Data allocation
GENERATOR	Optimization and object code generation
TOTAL	Total disk time for all phases

For each phase the list of statistics contains

Heading	*Statistic*
DISK	Number of reads and writes during the phase, excluding those needed to obtain the source file
SECONDS	Elapsed real time
SPACE	Internal buffer space used for symbol table, in units of 2K bytes
NODES	The number of symbol table nodes that the compiler uses in the program
PAGING	Disk I-O time
CPU	CPU time in seconds, followed by the clock time when the phase was completed

Storage allocation statistics are appended to the listing. All sizes are stated in 16-bit halfwords. The statistics are as follows:

Heading	*Statistic*
Code Size	The number of halfwords of object code generated from the PROCEDURE DIVISION of the source program. Appendix I lists the maximum allowable code size.
Static Size	The size of user-defined WORKING-STORAGE.
Source Lines	The number of lines in the source program.
Lines per Min	Lines compiled per minute.

–NO_STATISTICS suppresses display of compilation statistics.

*** –STORE_OWNER_FIELD / –NO_STORE_OWNER_FIELD**
–SOF / –NSOF

–STORE_OWNER_FIELD stores the identity of the current program in a known place for use by traceback routines, such as DMSTK.

–NO_STORE_OWNER_FIELD does not save this information, but does save a small code sequence for extremely time-critical programs.

*** –SYNTAXMSG / –NO_SYNTAXMSG**
–SYN / –NSYN

–SYNTAXMSG displays the messages SYNTAX CHECKING SUSPENDED and SYNTAX CHECKING RESUMED, which accompany error messages.

–NO_SYNTAXMSG suppresses display of these messages.

*** –VARYING / –NO_VARYING**
–VARY / –NVARY

–VARYING specifies that files containing variable-length record descriptions be processed as variable-length record files. This option also specifies that variable occurrence data items be processed as variable-length tables.

–NO_VARYING specifies that all files be processed as fixed-length record files during I-O operations. This option also specifies that variable occurrence data items be processed as fixed-length tables set at the maximum size.

Note

Specification of the RECORD IS VARYING clause or the RECORDING MODE IS V clause in a file description entry overrides the –NO_VARYING compiler option.

–XREF / * –NO_XREF
–XR / –NXR

–XREF creates a listing that includes a map and cross-reference. For every variable, the cross-reference lists the line number on which the variable is referenced. If the line number is preceded by an asterisk, the reference changes the variable's value. A sample cross-reference listing with discussion appears in Appendix L. This option includes –LISTING implicitly.

–NO_XREF suppresses the cross-reference listing.

–XREFSORT
–XRS

–XREFSORT creates a listing like the one generated by –XREF, but with *data-names* in alphanumeric order.

–NO_XREF suppresses the cross-reference listing.

TABLE 2-3
Summary of Compiler Options and Abbreviations

Option	*Abbreviation*	*Significance*
*-64V		Produces 64V-mode code
-ALLERRORS	-ALL	Overrides the limit of 100 fatal diagnostics
-ANSI_OBSOLETE *-NO_ANSI_OBSOLETE	-AO -NAO	Generates diagnostics for obsolete language elements
*-BIG_TABLES -NO_BIG_TABLES	-BT -NBT	Checks for segment spanning of multisegment data blocks
*-BINARY -NO_BINARY	-B -NB	Creates object file
*-CALCINDEX -NO_CALCINDEX	-CALC -NCALC	Calculates index offsets when referenced instead of set
-COMP *-NO_COMP	-NCOMP	Extends the range of computational variables
-CORRMAP *-NO_CORRMAP	-CORM -NCORM	Inserts a map of CORRESPONDING matches into source listing
-DATA_REP_OPT *-NO_DATA_REP_OPT	-DRO -NDRO	Performs optimization for decimal data type arithmetic operations
-DEBUG *-NO_DEBUG	-DBG -NDBG	Generates debugger code
-ERRORFILE *-NO_ERRORFILE	-ERRF -NERRF	Produces error file if any diagnostics are used
*-ERRTTY -NO_ERRTTY	-ERRT -NERRT	Displays diagnostics
-EXPLIST *-NO_EXPLIST	-EXP -NEXP	Produces expanded source listing
-FILE_ASSIGN *-NO_FILE_ASSIGN	-FA -NFA	Enables manual runtime file assignment
*-FORMATTED_DISPLAY -NO_FORMATTED_DISPLAY	-FDIS -NFDIS	Formats numeric items on DISPLAY
-FULL_HELP	-FH	Displays detailed list and usage of options
-HELP	-H	Displays list of options

* Default

TABLE 2-3
Summary of Compiler Options and Abbreviations - Continued

Option	Abbreviation	Significance
-HEXADDRESS *-NO_HEXADDRESS	-HEX -NHEX	Prints addresses in hexadecimal notation
-LISTING *-NO_LISTING	-L -NL	Creates source listing
-MAP *-NO_MAP	-MA -NMA	Produces a data map at the end of the listing
-MAPSORT	-MAPS	Prints map with names sorted alphabetically
-MAPWIDE	-MAPW	Prints map and cross-reference on 108-character lines
-OFFSET *-NO_OFFSET	-OFF -NOFF	Lists object address of procedure statements
*-OPTIMIZE [dec]	-OPT	Specifies the level of optimization to perform
-PRODUCTION *-NO_PRODUCTION	-PROD -NPROD	Generates production code
-RANGE *-NO_RANGE	-RA -NRA	Checks subscript ranges
-RANGE_NONFATAL	-RNF	Generates non-fatal runtime code that checks subscript ranges
-RMARGIN	-RMARG	Extends Area B to column 160
-SIGNALERRORS *-NO_SIGNAL_ERRORS	-SIG -NSIG	Aborts execution and signals overflow errors
-SILENT [dec]	-SI	Suppresses reporting of a specified level of errors
-SLACKBYTES *-NO_SLACKBYTES	-SLACK -NSLACK	Flags each item that is compiler-aligned
-SPACE		Reduces size of object code
-STANDARD *-NO_STANDARD	-STAN -NSTAN	Generates observations for Prime extensions

* Default

TABLE 2-3
Summary of Compiler Options and Abbreviations - Continued

Option	*Abbreviation*	*Significance*
–STATISTICS *–NO_STATISTICS	–STAT –NSTAT	Displays compilation statistics
*–STORE_OWNER_FIELD –NO_STORE_OWNER_FIELD	–SOF –NSOF	Generates module names into program code for debugging use
*–SYNTAXMSG –NO_SYNTAXMSG	–SYN –NSYN	Displays syntax-recovery messages
*–TIME		Generates fast object code
*–VARYING –NO_VARYING	–VARY –NVARY	Enables variable-length record/table processing
–XREF *–NO_XREF	–XR –NXR	Generates cross-reference
–XREFSORT	–XRS	Generates alphanumeric-order cross-reference

*Default.

3 Linking and Executing Programs

After you compile your program, as discussed in Chapter 2, you are ready to link and execute it. This chapter explains how to use BIND, the PRIMOS utility for linking virtual addressing mode (64V or 32I) programs. It discusses the BIND subcommands required to create an Executable Program Format (EPF) runfile, as well as other useful BIND subcommands. Finally, the chapter explains how to use the RESUME command to execute the EPF runfile you have created.

Note

If your COBOL85 program is larger than one segment, you *must* use BIND to link and execute it. If your program is smaller than one segment, you may use either BIND or SEG. For information on linking, loading, and executing programs with the PRIMOS SEG utility, see Appendix M.

Using BIND to Create an EPF

You can create an EPF using BIND in one of two ways:

- Interactively, by issuing BIND subcommands
- Directly, from the PRIMOS command line

Using BIND Interactively

To invoke BIND interactively, type the following command:

```
BIND
```

Then invoke BIND subcommands in response to the colon (:) prompt. The three BIND subcommands needed to create an EPF are LOAD, LIBRARY, and FILE. Enter these subcommands in the following sequence:

1. Use the LOAD subcommand to link your program, starting with the main procedure followed by subroutines. The LOAD subcommand has the following format:

LOAD *pathname-1* [*pathname-2, pathname-3, ...*]
LO

Each pathname is the name of an object file (binary file).

2. Use the LIBRARY subcommand to link system libraries. The LIBRARY subcommand has the following format:

   LIBRARY [*library-name-1*] [*library-name-2, library-name-3,...*]
   LI

   You must link the COBOL85 library, COBOL85LIB (abbreviated COBLIB), and the standard system library. Depending upon your application, link your own user-supplied libraries, and any application libraries that your program uses.
3. If you do not receive a `BIND COMPLETE` message, use the MAP subcommand to identify unresolved references (subroutines that are called but that you have not yet linked). The MAP subcommand is explained in the following section.
4. If you do receive a `BIND COMPLETE` message, use the FILE subcommand to save the EPF runfile and return to PRIMOS. The FILE subcommand has the following format:

   FILE [*EPF-filename*]

In response to the FILE subcommand, BIND processes the EPF, files it in your directory with a .RUN suffix, and returns control to PRIMOS. If you don't specify an *EPF-filename*, BIND adds the .RUN suffix to the first object file that you link, and saves the runfile in your directory.

When you use BIND interactively and specify the FILE subcommand before you receive a `BIND COMPLETE` message, BIND does the following:

- Resolves any unresolved references that remain, by creating dynamic links for those references
- Processes the entries
- Displays a `BIND COMPLETE` message at your terminal
- Files the EPF in your directory with a .RUN suffix
- Returns control to PRIMOS

A sample interactive linking session follows:

```
OK, BIND MYPROG          /*Invoke BIND and link object file
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
: LOAD ADD               /*Link ADD subroutine
: LOAD SUB               /*Link SUB subroutine
: LI COBOL85LIB          /*Link COBOL85 library
: LI                     /*Link System libraries
BIND COMPLETE
: FILE                   /*Save the EPF and return
                            control to PRIMOS
OK,
```

In the preceding example, the resulting runfile is MYPROG.RUN.

Using BIND From the Command Line

To invoke BIND from the PRIMOS command line type the following command:

BIND [*EPF-filename*] [*-options*]

EPF-filename is the name of the existing EPF or the name of the object file (binary file) that you want BIND to create. If you do not specify the *EPF-filename*, BIND uses the name of the first linked binary file for the *EPF-filename* base.

options given on the command line correspond to the BIND subcommands explained in the preceding section. You must precede each option with a hyphen.

A sample single-step linking session, which is equivalent to the previous interactive session, follows:

```
OK, BIND MYPROG -LOAD ADD SUB -LI COBOL85LIB -LI
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
BIND COMPLETE
OK,
```

When you invoke BIND from the command line, the FILE subcommand is appended by default to the end of the command line.

Other Useful BIND Subcommands

You can use the subcommands MAP, HELP, RELOAD, and QUIT during a BIND session, but you do not need them to create an EPF. You can use them at any time during the linking sequence. See the *Programmer's Guide to BIND and EPFs* for information on additional BIND subcommands.

Using the MAP Subcommand

Use the MAP subcommand to identify unresolved references if you do not receive a `BIND COMPLETE` message at your terminal. The MAP subcommand has the following format:

MAP [*pathname*] [*option*]

If you specify a pathname, the map is written to a file instead of displayed at your terminal.

For example,

```
:MAP MYMAP
```

writes a standard map of your program to the file MYMAP.

Use the –UNDEFINED option to obtain a list of any unresolved subroutine, program, or common block references.

For example,

```
:MAP -UNDEFINED
```

displays a list of the unresolved references.

Using the HELP Subcommand

You can enter the HELP subcommand only when you work with BIND interactively, not when you are at the command line. To invoke the HELP facility for BIND, enter the HELP subcommand at the colon (:) prompt. The format for HELP is as follows:

HELP { *subcommand-name* | **-LIST** }

Specifying a *subcommand-name* gives you a brief online description of the syntax, semantics, and abbreviation of that subcommand. The –LIST option displays a list of all BIND's subcommands available in the HELP facility.

Using the QUIT Subcommand

You can use the QUIT subcommand to return to PRIMOS without completing the binding process. If you are using BIND on the command line, the QUIT subcommand causes BIND not to create an EPF. The QUIT subcommand has the following format:

QUIT
Q

Because the QUIT subcommand ends a BIND session without saving the EPF being created, BIND asks you for verification before returning to PRIMOS. For example,

```
OK, BIND
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
:LOAD SAMPLE
:LI
BIND COMPLETE
:QUIT
EPF not filed, ok to quit? ('Yes', 'Y', 'No', 'N'):Yes
OK,
```

Using the RELOAD Subcommand

Use the RELOAD subcommand to relink a binary file into an already existing EPF. The RELOAD subcommand has the following format:

RELOAD *pathname-1* [*pathname-2*, *pathname-3*, ...]
RL

Each pathname is the name of an object file (binary file).

For example, suppose that you have created an EPF named BIG.PROGRAM.RUN, which contains a main program and six subroutines. Suppose also that when you run this program, you discover an error in one of the subroutines, SUB5. You correct the error and recompile SUB5. You now want to BIND the program again and rerun it to see that it is now working correctly.

To do this most quickly, first use the LOAD subcommand to link the existing EPF, then use the RELOAD subcommand to relink your new version of SUB5. The BIND sequence looks like this:

```
OK, BIND
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
:LOAD BIG.PROGRAM.RUN
:RELOAD SUB5
BIND COMPLETE
:FILE
OK,
```

If you do not get the `BIND COMPLETE` message after relinking your subroutine, reissue the LIBRARY commands that you used in your original BIND sequence. This situation happens if you add new library calls when you rewrite the subroutine.

Running Your Program

After you compile your program and create an EPF with BIND, you are ready to run your program by using the RESUME command. The RESUME command has the following format:

RESUME [*EPF-filename*]

For example, suppose you have used BIND to create a runfile with the name MYPROG.RUN. To execute MYPROG, for example, use the following PRIMOS command:

```
OK, RESUME MYPROG
```

PRIMOS looks in your directory for a file called MYPROG.RUN. If it finds such a file, PRIMOS begins executing it as an EPF runfile. If it does not find MYPROG.RUN, PRIMOS then searches for the following files, in order, and executes the first that it finds.

1. MYPROG.SAVE (static-mode runfile generated by LOAD or SEG)
2. MYPROG.CPL (CPL program)
3. MYPROG (static-mode runfile generated by LOAD or SEG)

If PRIMOS finds none of these files, it displays this error message:

```
Not found. MYPROG (std$cp)
ER!
```

For more information on running programs see the *PRIMOS User's Guide.*

Switch Settings at Runtime

If your COBOL85 program contains any of the switch-names CBLSW0 through CBLSW7 (defined in Chapter 6), when you execute the program, the system displays a request for switch settings:

```
SPECIFY ON SWITCHES:
```

Enter the numbers of all COBOL85 switches, from 0 through 7, that must be on during this execution. The numbers must be separated by either spaces or commas. If an entry is wrong, the system displays an error message and repeats the request.

As an example, for the sample program given in the section SPECIAL-NAMES in Chapter 6, if tape processing or no printout is desired, use the following dialog at runtime:

```
SPECIFY ON SWITCHES:

0 1
```

File Assignments at Runtime

If you compiled your program with the –FILE_ASSIGN compiler option, when you execute the program, the system displays a request for file assignments:

```
ENTER FILE ASSIGNMENTS:
>
```

Make one entry for each FD whose file ID you wish to assign. Syntax errors are generated during file assignment for improper formats. When no file assignments remain to be entered, use a slash mark (/) to conclude the session.

For example, suppose that a COBOL85 program contains the following statements:

```
FD  DISK-FILE
    VALUE OF FILE-ID IS 'FILE1'.
FD  TAPE-FILE
    LABEL RECORDS ARE STANDARD,
    VALUE OF FILE-ID IS 'FILE2'.
```

An appropriate runtime dialog is

```
ENTER FILE ASSIGNMENTS:
>FILE1 = MYDIRECTORY>DATA>DISBURSE
>FILE2 = $MT0, S, MYNAME, T1
>/
```

For additional information on the –FILE_ASSIGN option, see Chapter 2 and Appendix N.

4 Elements of COBOL85

This chapter discusses the basic elements of the COBOL85 language. The chapter begins with a summary of the divisions of a COBOL85 program, and the rules for coding a COBOL85 program. It then discusses the COBOL85 character set, and its use in the formation of COBOL85 words. The chapter explains the various levels, classes, and categories of data, and the manner in which data is stored internally. The chapter lists the rules for the qualification of data names, the formation of arithmetic and conditional expressions, the specification of variable-length records, and the processing of tables of repeating data items. The chapter concludes with a discussion of exception handling and file status codes.

Divisions of a COBOL85 Program: A Summary

A COBOL85 program consists of four divisions:

- IDENTIFICATION DIVISION
- ENVIRONMENT DIVISION
- DATA DIVISION
- PROCEDURE DIVISION

IDENTIFICATION DIVISION

The IDENTIFICATION DIVISION (ID DIVISION) assigns a name to the program and allows you to enter other information, such as your name, the date the program was written, and remarks.

ENVIRONMENT DIVISION

The ENVIRONMENT DIVISION specifies those aspects of a program that depend upon the physical characteristics of a specific computer, its peripheral devices, and file system. Two sections make up the ENVIRONMENT DIVISION: the CONFIGURATION SECTION and the INPUT-OUTPUT SECTION. The ENVIRONMENT DIVISION is optional.

The **CONFIGURATION SECTION** describes the computer on which the source program is compiled and the computer on which the compiled program is to be run. It also relates implementor-names used by the compiler to names introduced by the programmer in the source program.

The **INPUT-OUTPUT SECTION** describes each file, and associates the file with a peripheral device or a storage medium.

DATA DIVISION

The DATA DIVISION provides the compiler with a description of every data item used within the program. The DATA DIVISION has three sections: the FILE SECTION, the WORKING-STORAGE SECTION, and the LINKAGE SECTION. The DATA DIVISION is optional.

The **FILE SECTION** describes the structure of data files. Each file is defined by a *file-description-entry* and one or more *record-description-entries.*

The **WORKING-STORAGE SECTION** describes records and noncontiguous data items that are not part of external files, but are developed and processed internally.

The **LINKAGE SECTION** is meaningful only in a called program. This section describes data items that may be used by both the called and calling programs.

PROCEDURE DIVISION

The PROCEDURE DIVISION contains instructions (COBOL85 statements) to solve a data-processing problem. The PROCEDURE DIVISION is optional. This division contains two types of sections: declarative sections and procedural sections. The size of the PROCEDURE DIVISION can exceed one segment.

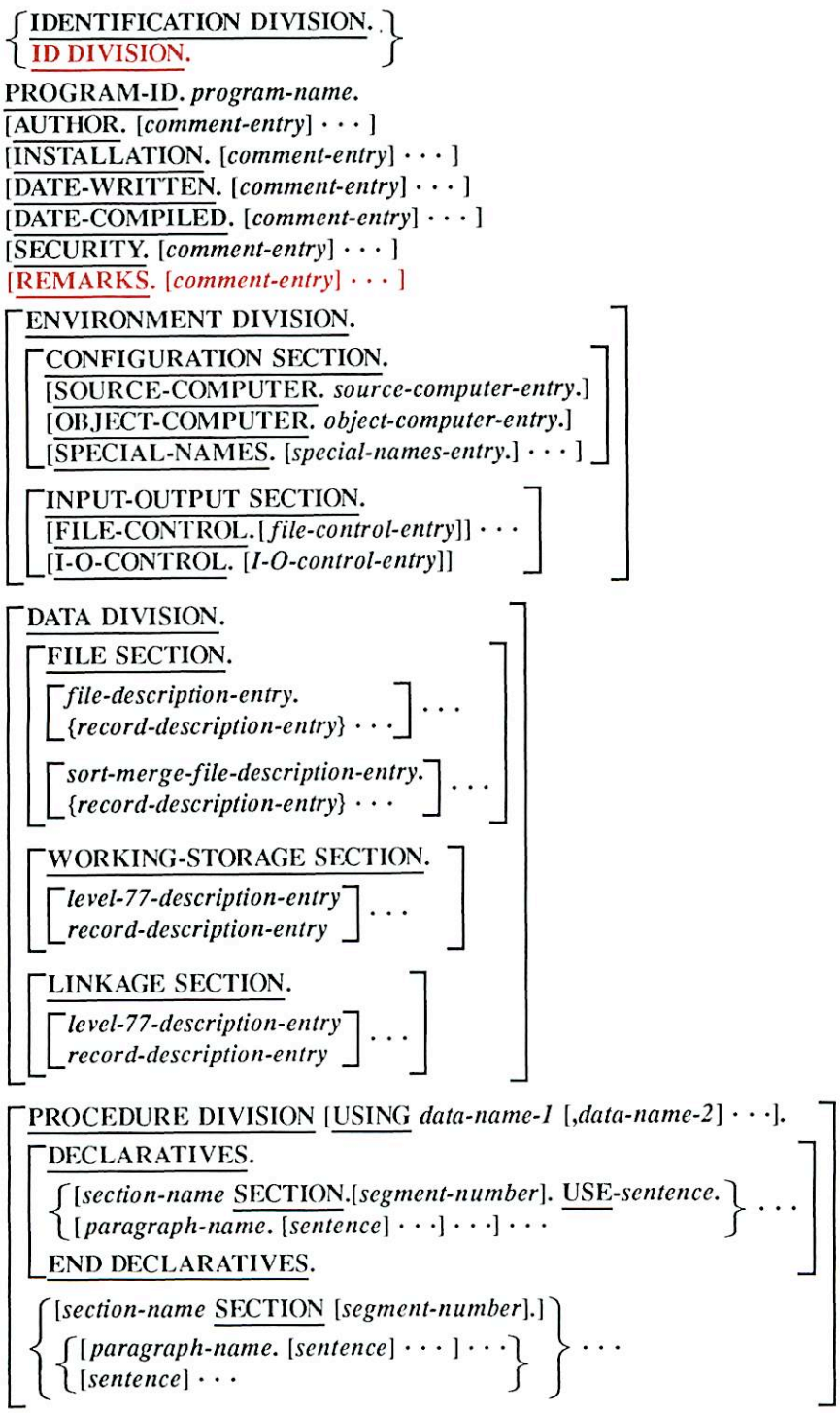
Declarative sections contain instructions that are not performed in the regular sequence of coding. Such procedures are executed only when an error condition is detected during a file operation.

Procedural sections contain zero or more paragraphs each. Each paragraph consists of a *paragraph-name* followed by zero or more COBOL85 sentences. Sentences, in turn, comprise one or more COBOL85 statements. Sections and paragraphs are optional.

Execution of the instructions in the PROCEDURE DIVISION begins with the first statement in the division, excluding declaratives. COBOL85 executes statements in the order in which they are written in the source program, until a PERFORM, GO TO, or other transfer of control is encountered.

Program Format and Example

The following diagram defines COBOL85 program format (conventions of notation are explained in the following section, Format Notation):



The following listing file for program SAMPLE illustrates COBOL85 program format. SAMPLE reads a file and prints its contents.

```
SOURCE FILE: <MYMFD>MYDIR>COBOL85>SAMPLE.COBOL85
COMPILED ON: WED, AUG 03 1988  AT: 11:59  BY: COBOL85 REV. 1.0-22.0
Options selected:  sample -listing
Optimization note: Currently "-OPTimize" means "-OPTimize 2",
Options used (* follows those that are not default):
  64V No_Ansi_Obsolete Big_Tables Binary CALCindex No_COMP No_CORrMap
  No_DeBuG No_Data_Rep_Opt No_ERRorFile ERRTty No_EXPlist No_File_Assign
  Formatted_DISplay No_HEXaddress Listing* No_MAp No_OFFset OPTimize(2)
  No_PRODuction No_RAnge No_SIGnalerrors SIlent(0) No_SLACKbytes TIME
  NO_STANdard No_STATistics Store_Owner_Field SYNtaxmsg No_TRUNCdiags
  VARYing No_XRef

    1            ID DIVISION.
    2            PROGRAM-ID. SAMPC4.
    3            INSTALLATION. PRIME 50 SERIES.
    4            DATE-WRITTEN. AUGUST 25, 1987.
    5            DATE-COMPILED. 880803.11:59:48.
    6            SECURITY. NONE
    7            REMARKS. THIS PROGRAM READS A FILE AND PRINTS ITS
    8                CONTENTS SEQUENTIALLY.
    9           ******************************************************
   10            ENVIRONMENT DIVISION.
   11            CONFIGURATION SECTION.
   12            SOURCE-COMPUTER. PRIME-9950.
   13            OBJECT-COMPUTER. PRIME-9950.
   14            INPUT-OUTPUT SECTION.
   15            FILE-CONTROL.
   16                SELECT PRINT-FILE ASSIGN TO PRINTER.
   17                SELECT INPUT-FILE ASSIGN TO PRIMOS.
   18           ******************************************************
   19            DATA DIVISION.
   20            FILE SECTION.
   21            FD  PRINT-FILE, LABEL RECORDS ARE OMITTED,
   22                DATA RECORD IS PRINT-LINE,
   23                VALUE OF FILE-ID IS FILE-NAME-PT.
   24            01  PRINT-LINE.
   25                05  OUT-LINE            PIC X(72).
   26            FD  INPUT-FILE,
   27                VALUE OF FILE-ID IS 'IN-DATA',
   28                DATA RECORD IS INPUT-IMAGE.
   29            01  INPUT-IMAGE             PIC X(72).
   30           *
   31            WORKING-STORAGE SECTION.
   32            01  HEADER.
   33                05  H1                  PIC X(71)
   34                     VALUE 'NAME                                STREET
   35           -    '      CITY          '.
   36            77  FILE-NAME-PT            PIC X(8) VALUE 'SAMP.RPT'.
   37            77  NO-MORE-INPUTS          PIC X    VALUE 'N'.
   38           ******************************************************
   39            PROCEDURE DIVISION.
   40           *
   41            DECLARATIVES.
```

```
42            INPUT-ERROR SECTION. USE AFTER ERROR PROCEDURE ON
43                  INPUT-FILE.
44            ONLY-PARAGRAPH. DISPLAY 'ERROR ON READ'.
45                  CLOSE INPUT-FILE, PRINT-FILE.
46                  STOP RUN.
47            END DECLARATIVES.
48           *
49            BEGINNING SECTION.
50              100-CREATE-FILE.
51                OPEN INPUT INPUT-FILE.
52                OPEN OUTPUT PRINT-FILE.
53                PERFORM 300-NEW-PAGE.
54                PERFORM 150-READ-PRINT.
55                PERFORM LAST-SECTION.
56                STOP RUN.
57              150-READ-PRINT.
58                READ INPUT-FILE AT END MOVE 'Y' TO NO-MORE-INPUTS.
59                PERFORM 155-PROCESS UNTIL NO-MORE-INPUTS = 'Y'.
60            155-PROCESS.
61                MOVE INPUT-IMAGE TO OUT-LINE.
62                WRITE PRINT-LINE.
63                READ INPUT-FILE AT END MOVE 'Y' TO NO-MORE-INPUTS.
64            LAST-SECTION SECTION.
65              200-CLOSE-ALL.
66                CLOSE INPUT-FILE, PRINT-FILE.
67                DISPLAY 'END OF FILE'.
68            THIRD-LEVEL SECTION.
69              300-NEW-PAGE.
70                MOVE SPACES TO PRINT-LINE.
71                WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
72                MOVE SPACES TO PRINT-LINE.
73                WRITE PRINT-LINE AFTER ADVANCING 1.
```

Format Notation

Throughout this document, formats are prescribed for various clauses or statements. They are presented in ANSI COBOL notation, except for the use of brackets and the Prime extensions discussed below.

ANSI Notation

Words: All underlined uppercase words are called key words and are required when the clauses containing them are used. Uppercase words that are not underlined are optional. Uppercase words, whether underlined or not, must be spelled correctly.

Lowercase words are generic terms used to represent variable COBOL words, *literals*, PICTURE *character-strings*, *comment-entries*, or a complete syntactical entry that must be supplied by the user. Where generic terms are repeated, a number appendage to the term identifies that term. These terms appear in italic.

Level-numbers: When specific *level-numbers* appear in *data-description-entry* formats, those specific *level-numbers* are required. In this document, the forms 01, 02, and so on indicate *level-numbers* 1 through 9.

Brackets and Braces: When a portion of a general format is enclosed in brackets, [ ], that portion may be included or omitted at your choice. Braces, { }, enclosing a portion of a general format mean that you must select one of the options contained within the braces. In both cases, a choice is indicated by vertically stacking the possibilities. However, if the items within brackets are themselves enclosed in brackets, then the header of that section or division is required if any other items are used. If a line within brackets is indented, it is part of the preceding line. When brackets or braces enclose a portion of a format, but only one possibility is shown, the brackets or braces delimit that portion of the format to which a following ellipsis applies. If an option within braces contains only reserved words that are not key words, then the option is a default option (selected unless one of the other options is explicitly indicated).

The Ellipsis: In the general formats, the ellipsis represents the position at which repetition may occur at your option. The portion of the format that may be repeated is determined as follows: scanning right to left, determine the] or } immediately to the left of the ellipsis. Continue scanning right to left and determine the matching [or {; the ellipsis applies to the words between this matching pair of delimiters.

Format Punctuation: The punctuation characters comma and semicolon are shown in some formats. They are optional and you may include or omit them. In the source program these two punctuation characters are interchangeable. Neither may appear immediately preceding the first clause of an entry or paragraph.

If desired, you can use a semicolon or comma between statements in the PROCEDURE DIVISION.

Paragraphs within the IDENTIFICATION and PROCEDURE DIVISIONs, and the entries within the ENVIRONMENT DIVISION and DATA DIVISION, must be terminated by the period.

Special Characters: The characters + – > < = when appearing in formats, although not underlined, are required.

Examples: In the program format at the beginning of this chapter, PROGRAM-ID is a key word that you must use, and you must follow it with a literal that you supply. The *sort-file-description-entry* is optional. If you use it, you must follow it with one or more record descriptions, which are described in Chapter 7. The entire ENVIRONMENT DIVISION is optional; however, if you include CONFIGURATION SECTION, you must include ENVIRONMENT DIVISION.

Prime Extensions to ANSI Notation

The following Prime terms are added to ANSI notation.

The term *data-name* means a user-defined name for a variable that may be subscripted or qualified.

Clause, Statement, Entry: Certain entries in the formats consist of one or more capitalized words followed by the word **clause, statement,** or **entry.** These designate clauses or statements described in other formats in appropriate sections of the text.

Underscore: For easier reference in the text, some lowercase words are followed by an underscore and a digit or letter. This modification does not change the syntactical definition of the word.

Multiple Formats: For a given COBOL85 verb, separate formats are mutually exclusive options.

Coding Rules

Figure 4-1 illustrates COBOL85 program coding rules, which are listed below.

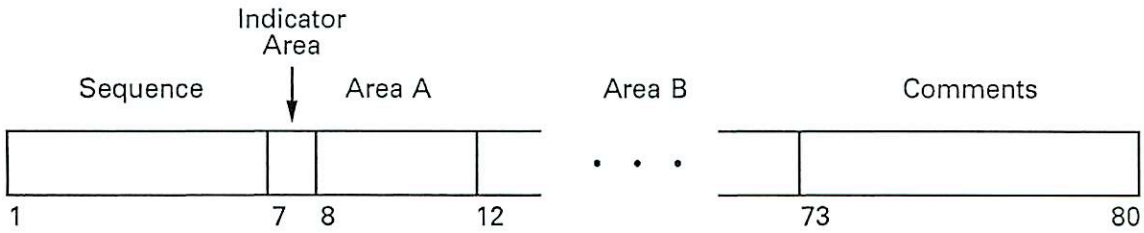


FIGURE 4-1
Standard COBOL Coding Areas

1. Each line of code may have a six-digit sequence number in positions 1-6, arranged so that the source statements are in ascending order. Blanks are also permitted in positions 1-6.
2. Position 7 is used for four special coding symbols:
   - An asterisk (*) in position 7 of the line causes that line to be treated as a comment. Any characters may follow on that line. The asterisk and the characters are produced on the source listing but serve no other purpose.
   - If a slash (/) appears in position 7, the current line is treated as a comment line, and the next line is printed at the top of a new page of the compiler-generated listing.
   - A hyphen (-) is used to continue any word or literal from one line to another. Refer to the section titled Continuation of Literals in this chapter for continuation rules.
   - The letter D in position 7 causes the line to be treated as a comment, unless the program is compiled with the –DEBUG option, in which case the line is treated as a normal source statement.
3. Division, section, and paragraph headers must begin in Area A (positions 8-11). *paragraph-names* must also appear in Area A (at the point where they are defined). All *level-numbers* may appear in Area A or Area B.
4. All other program elements must be confined to Area B.
5. Positions 73-80 are ignored by the compiler unless the –RMARGIN option is in effect. Frequently, these positions contain the program identification.

Punctuation and Separators

A **separator** is a string of one or more punctuation characters. The rules for formation of separators are as follows:

- The space is a separator. Anywhere a space is used, more than one space may be used.
- The comma, semicolon, and period, when immediately followed by a space, are separators. These separators may appear in a COBOL85 source program only where explicitly permitted by the general formats, by syntax rules, or by statement and sentence structure definitions. The period followed by a space also serves as a statement terminator for conditionals.
- The right parenthesis and left parenthesis are separators. Parentheses may appear only in balanced pairs of left and right parentheses delimiting subscripts, indexes, arithmetic expressions, or conditions.
- The quotation mark is a separator. An opening quotation mark must be immediately preceded by a space or left parenthesis; a closing quotation mark must be immediately followed by a space, comma, semicolon, period, or right parenthesis. Quotation marks may appear only in balanced pairs delimiting nonnumeric literals except when the literal is continued. (See the section titled Continuation of Literals, later in this chapter.)
- The pseudo-text delimiter, ==, is a separator. An opening pseudo-text delimiter must be immediately preceded by a space; a closing pseudo-text delimiter must be immediately followed by one of the following separators: space, comma, semicolon, or period. Pseudo-text delimiters may appear only in balanced pairs delimiting pseudo-text in a COPY...REPLACING statement. Chapter 15 discusses the COPY...REPLACING statement.
- The separator, space, is optional immediately preceding all separators except
    - As specified by syntax rules.
    - When the separator is a closing quotation mark. In this case, a preceding space is considered part of the nonnumeric literal and not a separator.
    - Before the opening pseudo-text delimiter, where the preceding space is required.
- The separator, space, may immediately follow any separator except the opening quotation mark. In this case, a following space is considered as part of the nonnumeric literal and not as a separator.

Any punctuation character that appears as part of a PICTURE *character-string* or numeric *literal* is not considered a punctuation character, but rather an element of that PICTURE *character-string* or numeric *literal*. PICTURE *character-strings* are delimited only by the following separators: space, comma, semicolon, or period.

The rules for forming separators do not apply to the characters that make up the contents of nonnumeric *literals*, *comment-entries*, or comment lines.

The standard character set used by Prime is the ASCII character set. The set of characters, with binary, decimal, octal, and hexadecimal equivalents is presented in Appendix B.

The COBOL85 Character Set

The basic and indivisible unit of the language is the character. The COBOL85 language character set, shown in Table 4-1, has the following characters: the numbers 0 through 9, the 26 uppercase and lowercase letters of the English alphabet, the space (blank), and special characters. You use this character set to form COBOL85 character strings and separators. In the case of nonnumeric *literals*, *comment-entries*, and comment lines, the character set is expanded to include the computer's entire character set as defined in Appendix B.

Prime Extensions

A single quotation mark (apostrophe) is accepted as an equivalent of double quotation marks. Quotation marks preceding and following a given item must be identical. (Note that you can use double quotation marks and the question mark when entering COBOL85 programs with ED, only if other deletion characters have been established with the PRIMOS command TERM or the ED command SYMBOL, unless they are prefaced by the escape character.)

Use the backslash (\) to denote the beginning of a nonnumeric literal mnemonic that corresponds to a character value in the Prime ECS character set. For more information see the section, Nonnumeric Literals, later in this chapter, and Table B-3.

TABLE 4-1
COBOL85 Character Set

Class	*Character*	*Meaning*
Numeric	**0...9**	Digit
Alphabetic	**A...Z**	Uppercase letters
	a...z	Lowercase letters
	Space	Blank
Special characters	+	Plus sign
	–	Minus sign
	_	Underscore
	*	Asterisk (star)
	=	Equal sign
	$	Currency sign
	,	Comma
	;	Semicolon
	.	Period
	"	Quotation marks
	'	Apostrophe
	(	Left parenthesis
	)	Right parenthesis
	>	Greater than
	<	Less than
	/	Slash (stroke)
	\	Backslash

Collating Sequence

Each character in the Prime character set has a unique value, which establishes the collating sequence for the character set. The characters are arranged in ascending collating sequence in Table B-3 of Appendix B. The collating sequence can be modified by the PROGRAM COLLATING SEQUENCE clause.

The Prime Extended Character Set

As of Rev. 21.0, Prime expanded its character set. The basic character set is the same as it was before Rev. 21.0: it is the ANSI ASCII 7-bit set (called ASCII-7), with the 8th bit turned on. However, the 8th bit is now significant; when it is turned off, it signifies a different character. Thus, the size of the character set has doubled from 128 to 256 characters. This expanded character set is called the Prime Extended Character Set (Prime ECS). Table B-3 shows the Prime Extended Character Set.

Note

The extra characters are not available on most printers and terminals. Check with your System Administrator to find out whether you can use all of the characters in Prime ECS.

Specifying Prime ECS Characters

Direct Entry: On terminals that support Prime ECS, you can enter the printing characters directly; the characters appear on the screen as you type them. For information on how to do this, see the appropriate manual for your terminal.

A terminal supports Prime ECS if

- It uses ASCII-8 as its internal character set.
- The TTY8 protocol is configured on your asynchronous line.

If you do not know whether your terminal supports Prime ECS, ask your System Administrator.

On terminals that do not support Prime ECS, you can enter any of the ASCII-7 printing characters (characters with a decimal value of 160 or higher) directly by just typing them.

Octal Notation: If you use the Editor (ED), you can enter any Prime ECS character by typing

^octal-value

where *octal-value* is the three-digit octal number given in Table B-3. You must type all three digits, including leading zeros.

Before you use this method to enter any of the ECS characters that have decimal values between 32 and 127, first specify the following ED command:

MODE CKPAR

This command permits ED to print as ^*nnn* any characters that have a first bit of 0.

Note

Prime ECS characters with decimal values less than 128 (octal 200) cannot be used in the formation of COBOL85 words. They are only meaningful as nonnumeric literals.

character-strings

A ***character-string*** is a character or a sequence of contiguous characters that forms a PICTURE *character-string*, a COBOL85 word, a *literal*, or a *comment-entry*. A *character-string* is delimited by one of the separators defined in the previous section.

picture-strings

A PICTURE *character-string (picture-string)* consists of certain combinations of characters in the COBOL85 character set used as a template. See PICTURE in Chapter 7, for a description of the *picture-string* and the rules governing its use. A punctuation character that is part of a *picture-string* is not considered as a punctuation character, but as a symbol in that *picture-string*.

Word Formation

A COBOL85 word is a *character-string* of a maximum of 32 characters chosen from the following set of 64 characters:

0 through 9 (digits)
A through Z (uppercase letters)
a through z (lowercase letters)
– (hyphen)
_ (underscore)

All words except *level-numbers, section-names, segment-numbers,* and *paragraph-names* must contain at least one alphabetic character, an underscore, or a hyphen. A word must not begin or end with a hyphen or an underscore. It is delimited by a space, or by proper punctuation. A word may contain more than one embedded hyphen or underscore; consecutive embedded hyphens or underscores are also permitted.

Examples of valid words are

ITEM1
1STITEM
1ST-ITEM
3_5

All words are reserved words, implementor-names, or programmer-defined words.

Reserved Words

A **reserved word** is one of a specified list of words that may be used in COBOL85 source programs, but which may not appear as programmer-defined words. Use them only as specified in the general formats. Reserved words are of four types:

- Required Words
  - Key words
  - Special-character words
- Optional words
- Figurative constants
- Connectives

All COBOL85 reserved words are listed in Table B-2 of Appendix B.

Required Words

A required word is a word whose presence is required when the format in which the word appears is used in a source program.

Key Words: A **key word** is required when the statement in which the word appears is used in a source program. Within each statement format in this book, such words are uppercase and underlined.

Special-character Words: The arithmetic operators and relation characters are reserved words. Table 4-2 lists the operators and their meanings:

TABLE 4-2
Arithmetic and Relational Operators

Operator	*Meaning*
Arithmetic:	
+	Addition
–	Subtraction
*	Multiplication
/	Division
**	Exponentiation
Relational:	
=	Equal to
<	Less than
>	Greater than

Optional Words

Within each format, uppercase words that are not underlined are **optional words.** The presence or absence of an optional word does not alter the meaning of the COBOL85 program in which it appears, but, when present, the word improves readability of the program.

Figurative Constants

Figurative constants are reserved words used to name and reference specific constant values. A figurative constant represents as many instances of the associated character as required in the context of the statement. The singular and plural forms are equivalent and may be used interchangeably.

A figurative constant may be used wherever *literal* appears in a format description; except that, whenever the literal is restricted to numeric characters, the only figurative constant permitted is ZERO (ZEROS, ZEROES). A figurative constant must not be bounded by quotation marks.

Table 4-3 lists the figurative constants with their meanings.

TABLE 4-3
Figurative Constants

Constant	*Meaning*
ZERO **ZEROS** **ZEROES**	The ASCII character represented by hexadecimal B0.
LOW-VALUE **LOW-VALUES**	The character whose hexadecimal representation is 00 (lowest character in the ASCII or EBCDIC collating sequence).
HIGH-VALUE **HIGH-VALUES**	The character whose hexadecimal representation is FF (highest character in the ASCII or EBCDIC collating sequence).
QUOTE **QUOTES**	The quotation mark, whose hexadecimal representation is A2 (").
SPACE **SPACES**	The blank character represented by hexadecimal AO.
ALL *literal*	Represents one or more of the string of characters comprising the literal. The literal must be either a nonnumeric literal or a figurative constant other than ALL literal. When a figurative constant is used, the word ALL is redundant and is used for readability only.

Connectives

The three types of connectives are

- **Qualifier connectives:** OF and IN can associate a *data-name, condition-name, text-name,* or *paragraph-name* with its qualifier. Qualifiers are discussed in the section Qualification of Names, later in this chapter.
- **Series connectives:** Comma (,) or semicolon (;) can link two or more consecutive operands.
- **Logical connectives:** AND, OR, and NOT can function in the formation of conditions.

Implementor-names

Implementor-names include *device-names* and *switch-names* unique to Prime computers. These are listed in Chapter 6, The ENVIRONMENT DIVISION.

Programmer-defined Words

A **programmer-defined** word is one supplied by the programmer to satisfy the format of a clause or statement. Each is constructed according to the rules for word formation. The categories for these words include

level-numbers

data-names

file-names

condition-names

mnemonic-names

paragraph-names

section-names

segment-numbers

alphabet-names

index-names

class-names

With the exception of *paragraph-names, segment-numbers,* and *section-names,* all programmer-defined words must contain at least one alphabetic character, an underscore, or a hyphen.

If a programmer-defined word is not unique, there must be a unique method of referencing it by using qualifiers (for example, TAX-RATE IN STATE-TABLE). Qualifiers are explained in the section Qualification of Names, later in this chapter.

level-numbers

level-numbers are one-digit or two-digit, programmer-defined numbers in the DATA DIVISION. They group items within the data hierarchy of the Record Description.

The range of levels is 01 through 49, and 66, 77, and 88. *level-numbers* 1 through 9 may be written as single digits. The use of *level-numbers* is discussed in Chapter 7, The DATA DIVISION.

data-names

A *data-name* is a word made up by the user to identify a data item used in a program. A *data-name* always refers to a field of data, not to a particular value. It is formulated according to the rules for word formation above. It must not be identical to a reserved word.

data-names are used in all divisions of a COBOL85 program. When referenced in the PROCEDURE DIVISION, a *data-name*, if not unique, must be followed by a syntactically correct combination of qualifiers, subscripts, or indexes sufficient to ensure uniqueness.

file-names

A **file** is a collection of data records of a similar class or application. A *file-name* is preceded by an FD entry in the DATA DIVISION's FILE SECTION. Rules for composition of the name are identical to those for *data-names*. (See the section titled Word Formation, above.) References to a *file-name* appear in PROCEDURE DIVISION I-O statements as well as in the ENVIRONMENT DIVISION and DATA DIVISION.

condition-names

A *condition-name* is a name assigned to a specific value, set of values, or range of values, within a complete set of values that a data item may assume. The data item is called a **conditional variable**. *condition-names* are allowed in the FILE, WORKING-STORAGE, and LINKAGE sections of the DATA DIVISION, as well as in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

Define a *condition-name* within the DATA DIVISION in a level-88 entry subordinate to the associated data item name, or in the SPECIAL-NAMES paragraph, assigned to the ON STATUS or OFF STATUS of switches. Rules for the formation of *condition-name* words are the same as those specified in the section titled Word Formation. Additional information concerning *condition-names* and procedural statements employing them is given in the chapters on the DATA DIVISION and PROCEDURE DIVISION.

mnemonic-names

A *mnemonic-name* is assigned in the ENVIRONMENT DIVISION in the SPECIAL-NAMES paragraph for reference in ACCEPT or DISPLAY statements or in switch-condition tests. A *mnemonic-name* is composed according to the rules for word formation above.

paragraph-names and section-names

paragraph-names and *section-names* identify paragraphs and sections, respectively, in the PROCEDURE DIVISION. They may be a maximum of 32 characters long, and may be all alphabetic, all numeric, or alphanumeric.

Examples of valid *paragraph-names* are

```
050-NEXT-ITEM
050
NEXT-ITEM
```

segment-numbers

A *segment-number* must be an integer in the range 0 through 99.

alphabet-names

An *alphabet-name* is a programmer-defined word in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION that assigns a name to a specific character set or collating sequence.

index-names

An *index-name* is a programmer-defined word that names an index associated with a specific table.

class-names

A *class-name* is a programmer-defined word in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION that assigns a name to a specific set of characters to be used in class condition tests.

Literals

A *literal* is a programmer-defined constant value. It is not identified by a *data-name* in a program, but is completely defined by its own identity. A *literal* is either nonnumeric or numeric.

Nonnumeric Literals

A **nonnumeric literal** may be any combination of printable or nonprintable characters in the ASCII character set defined in Appendix B. This includes characters that can be represented by Prime mnemonics.

A nonnumeric literal must be delimited by matching quotation marks or apostrophes.

Note

You can use double quotation marks when entering a COBOL85 program with ED only if you change the erase character from quotation marks to some other character with the PRIMOS command TERM or the ED command SYMBOL, or preface the double quotation marks with the escape character.

All spaces enclosed by the delimiters are included as part of the literal. If the delimiter itself is to be used within the literal string, you must write it twice. The last example below shows a single quotation mark within a literal delimited by single quotation marks. The length of a nonnumeric literal is computed excluding the delimiters. A nonnumeric literal must not exceed 160 characters in length. The minimum length is 1.

If a Prime mnemonic is used to represent a character, the mnemonic must be enclosed within parentheses and preceded by a backslash. The mnemonic must not be enclosed within quotation marks or apostrophes, and embedded spaces are not allowed. The mnemonic must be one of the mnemonics listed in Appendix B. The character represented by the mnemonic is stored as a character string having a length of 1.

A nonnumeric literal that contains a mnemonic follows the same coding rules as any other nonnumeric literal. A mnemonic is not converted to its character equivalent within comments. If a mnemonic is enclosed within quotation marks or apostrophes, the system treats it as any other nonnumeric literal.

Note

ED interprets the backslash (\) as a logical tab. If you are using ED to enter a character represented by a mnemonic into a source program, you must define another character as your logical tab in order to create the backslash required for the mnemonic.

A character represented by a mnemonic can be specified by itself, or juxtaposed with one or more additional mnemonics, or juxtaposed with one or more quoted character strings. This juxtaposition results in an implicit concatenation operation, unless you use a comma or semicolon to separate the character represented by the mnemonic from the juxtaposed string or character.

For example,

```
'hello' \(BEL)
```

is stored as a single character string having a length of 6. However,

```
'hello', \(BEL)
```

is stored as two separate character strings having lengths of 5 and 1, respectively.

Caution

Because of the implicit concatenation described above, you must use caution when using a character represented by a mnemonic in conjunction with any verb or *data-description-entry* that allows contiguous literals in its format.

The following examples illustrate several types of nonnumeric literals:

```
"ILLEGAL CONTROL CARD"

'IT WAS A DARK AND STORMY NIGHT'

"123"

'1001'

"3.1414"

'-6'

'LINE1'\(CR)'LINE2'

'SPACES ARE ALLOWED' \(CR) 'TO AID READABILITY'

'ABSENCE OF SEPARATORS' \(BEL) 'IMPLIES CONCATENATION'

'PRESENCE OF SEPARATORS', \(BEL), 'RESULTS IN SEPARATE LITERALS'

\(BEL)'HELLO!'\(CR)

\(BEL) \(CR)

"DO'S AND DON'TS"

'HERE''S LOOKING AT YOU'
```

Numeric Literals

A **numeric literal** must contain at least one and not more than 18 digits. A numeric literal may consist of the characters (digits) 0 through 9 (optionally preceded by a sign) and a decimal point, or a comma in the case, DECIMAL-POINT IS COMMA, discussed below. It may contain only one sign character (except for COMP-1 and COMP-2, which can contain two) and only one decimal point. The sign, if present, must appear as the leftmost character of the numeric literal. If a numeric literal is unsigned, it is assumed to be positive.

A decimal point may appear anywhere within the numeric literal, except as the rightmost character. If a numeric literal does not contain a decimal point, it is considered to be an integer.

If a literal conforms to the rules for the formation of numeric literals, but is enclosed in quotation marks, it is a nonnumeric literal and it is treated as such by the compiler.

The following examples are numeric literals:

```
72     +1011     3.14159     -6     -.333     1.23E10
```

The last example uses floating-point format, a Prime extension that is described in the section titled Data Representation and Alignment, later in this chapter.

By use of the clause, DECIMAL-POINT IS COMMA, the functions of the period and comma may be interchanged, putting the European notation into effect. In this case, for example, the literal value one thousand and one tenth is written as 1.000,1.

Continuation of Literals

When a literal is too long to fit on one line, the following conventions apply to the next line of coding (continuation line):

- A hyphen in the indicator area (column 7) of a line indicates that the current line is a continuation line and the preceding line is the continued line.
- Area A of a continuation line must be blank.
- If the continued line contains a nonnumeric literal, the following rules apply:
    - The first nonblank character of the continuation line must be a quotation mark and the last nonblank character of the continued line must not be a quotation mark.
    - The continuation starts with the character immediately after that quotation mark.
    - All spaces at the end of the continued line are considered part of the literal.

  The next two lines illustrate continuation of a nonnumeric literal:

```
      MOVE 'NOW IS THE TIME FOR ALL GOOD MEN TO COME TO
     -  ' THE AID OF THE PARTY.' TO HEADER.
```

- If the continued line contains a numeric literal, the first nonblank character of the current line is the successor of the last nonblank character of the preceding line without any intervening space.

Data Levels

The two levels of data are elementary and group.

Elementary Item

An **elementary item** is a data item containing no subordinate items. An elementary item must contain a PICTURE clause, except when usage is described as COMPUTATIONAL-1, COMPUTATIONAL-2, or INDEX. An elementary item must not span segment boundaries. The maximum size of an elementary item is given in Appendix I.

Group Item

A **group item** is defined as one having further subdivisions, so that it contains one or more elementary items or other groups. The maximum size of a group item is given in Appendix I.

Classes and Categories of Data

The classes of data are alphabetic, numeric, and alphanumeric. Within each class, the categories of data are alphabetic, numeric, numeric edited, alphanumeric, and alphanumeric edited. Every elementary item except an index data item belongs to a class and to a category, as defined by its PICTURE or USAGE clause.

COBOL85 uses the category of a data item to determine the validity of operations such as MOVE and COMPUTE, and for alignment. The categories have the following characteristics. (More detail is given in the section PICTURE, in Chapter 7.)

Alphabetic Item

An **alphabetic item** consists of any combination of the 26 uppercase and the 26 lowercase characters of the English alphabet and the space character. It is defined by PICTURE A.

Numeric Item

A **numeric item** consists only of digits, no more than one assumed decimal point, and an optional sign. It is defined by PICTURE 9 or by one of the following:

USAGE IS BINARY
USAGE IS COMPUTATIONAL
USAGE IS COMPUTATIONAL-1
USAGE IS COMPUTATIONAL-2
USAGE IS PACKED-DECIMAL
USAGE IS COMPUTATIONAL-3

Numeric Edited Item

A **numeric edited item** consists only of digits and special editing characters or editing characters alone, as described in the section entitled PICTURE, in Chapter 7. It is defined by PICTURE 9 plus editing characters.

Alphanumeric Item

An **alphanumeric item** consists of any combination of ANSI characters plus lowercase letters, defined by PICTURE X.

Alphanumeric Edited Item

An **alphanumeric edited item** is an alphanumeric item defined by PICTURE X plus editing characters described in the section entitled PICTURE, in Chapter 7.

Relationship of Classes and Categories of Data

The class rather than the category is used in some relation conditions, and for determining the validity of operations on group items. For alphabetic and numeric elementary items, classes and categories are the same. For elementary items, the alphanumeric class includes the categories of alphanumeric edited, numeric edited, and alphanumeric. The class of a group item is treated at execution time as alphanumeric regardless of the class of elementary items subordinate to that group item. Table 4-4 depicts the relationship of the classes and categories of data items.

TABLE 4-4
Classes and Categories of Data

Level of Data	*Class*	*Category*
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Numeric edited
		Alphanumeric edited
		Alphanumeric
Group	Alphanumeric	Alphabetic
		Numeric
		Numeric edited
		Alphanumeric edited
		Alphanumeric

Data Representation and Alignment

Data is further categorized by the format in which it is stored in the computer. The formats are display or unpacked decimal, packed decimal, binary, index, and floating-point. These formats are specified by the USAGE clause, as outlined below.

USAGE	*Machine Description*
DISPLAY	Unpacked decimal
PACKED-DECIMAL	Packed decimal
COMPUTATIONAL-3	Packed decimal
BINARY	Binary
COMPUTATIONAL	Binary
INDEX	Binary
COMPUTATIONAL-1	Single-precision floating-point
COMPUTATIONAL-2	Double-precision floating-point

Note

You can use data items of all formats together in computations, although you can often save time by ensuring that all data items used in any given computation are in the same format.

COBOL85 operates on five types of decimal data: leading separate sign, trailing separate sign, packed decimal, leading embedded sign, and trailing embedded sign. The last two types may be entered with an overpunch. Table B-12 in Appendix B summarizes the characteristics of each decimal data type and the sign values.

Unpacked Decimal Item (DISPLAY)

An **unpacked decimal** item is one in which one byte (eight bits) is employed to represent one digit as well as the sign. An exception is such an item with the SIGN IS SEPARATE clause, discussed in Chapter 7. The PICTURE clause for an external decimal item may contain only 9, S, V, and P. The USAGE for an unpacked decimal item is always DISPLAY, whether implicit or explicit. Maximum size is 18 digits. Figure 4-2 represents the storage of such an item.

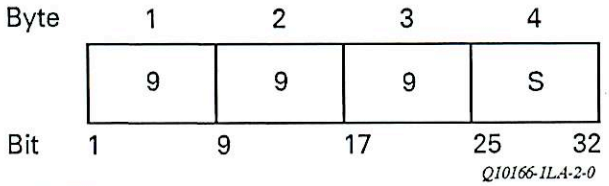


FIGURE 4-2
Unpacked Decimal Storage

Packed Decimal Item (PACKED-DECIMAL or COMP-3)

A **packed decimal item** is one in which each byte represents two digits. It is defined by the PACKED-DECIMAL or COMPUTATIONAL-3 (COMP-3) USAGE clause. The maximum size is 18 digits.

Its PICTURE clause may contain only 9, S, V, and P. A packed decimal item defined by *n* nines in its PICTURE occupies (*n*/2)+1 bytes in memory. All bytes, except the rightmost, contain a pair of digits.

The rightmost half-byte of a packed item contains a representation of the sign. Bit string 1100 represents a positive sign, 1101 represents a negative sign. Four bits are always reserved for the sign in a packed field, even if the picture lacks the leading character S. For this reason, the optimal space allocation for a packed decimal item is an odd-size field. Figure 4-3 represents the storage of such an item.

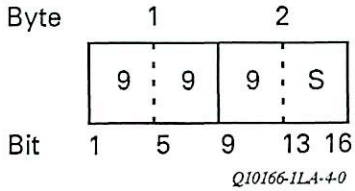


FIGURE 4-3
Packed Decimal Storage

Binary Item (BINARY or COMP)

A **binary item** uses the base-2 system to represent an integer. The item occupies the following storage: 16 bits if 1 to 4 nines are specified in the PICTURE clause, 32 bits if 5 to 9 nines are specified, and 64 bits if 10 to 18 nines are specified. The maximum size is 18 digits. If no PICTURE is specified, the default PICTURE is S9(4) (16 bits).

The leftmost bit of the storage area is the operational sign: 0 is positive, 1 is negative. (BINARY and COMPUTATIONAL data types are stored in two's-complement form.) The sign is optional in the PICTURE clause. If it is omitted, the value in this field is always treated as positive. You must specify USAGE IS BINARY or USAGE IS COMPUTATIONAL. Because this data type is represented in hardware by a signed data type (fixed binary), the compiler generates extra code to return the absolute value for every reference to this field when it is declared without a sign. In addition, if the PICTURE clause of BINARY or COMP items specifies more than nine digit positions, or specifies positions to the right of the decimal point, the compiler generates extra code to convert the contents of such fields when they are referenced. Figure 4-4 represents the storage of this item.

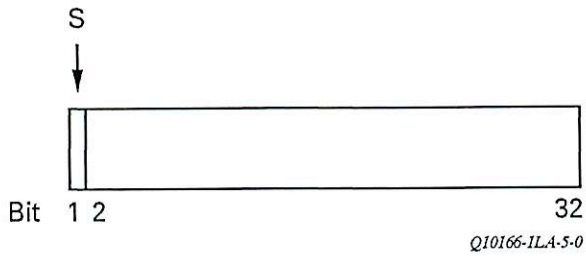


FIGURE 4-4
Binary Storage (PIC S9(9))

COMP and BINARY items are aligned by the compiler on halfword (16-bit) boundaries. 16-bit binary items have a maximum range of –32768 to +32767 when using the –COMP option. 32-bit binary items have a range of –2147483648 to 2147483647, also when using the –COMP option. (These two items correspond to INTEGER*2 and INTEGER*4, respectively, in FORTRAN.) 64-bit binary items are converted to decimal when referenced and, therefore, have the same range as 18-digit decimal data, that is, a PIC of from 9(18) to V9(18).

If you do not specify the –COMP compile line option, the number of 9s in the picture clause is used in determining the range of values allowed. Even though 16 or 32 bits of storage are allocated, they are not all accessible. If PIC S9(4) COMP is used, 16 bits of storage are allocated, but 9999 is the maximum value allowed. A BINARY or COMP data item declared as a PIC S9(1) has a range of –9 through +9.

See Chapter 2 for a description of the –COMP compiler option.

Index Item

An **index item** is defined with USAGE IS INDEX or INDEXED BY. It may not have a PICTURE clause. It is a 64-bit signed binary item, the first half of which contains the occurrence number; the last half of the offset. The maximum value of index items is listed in Appendix I. Figure 4-5 represents the storage of this item.

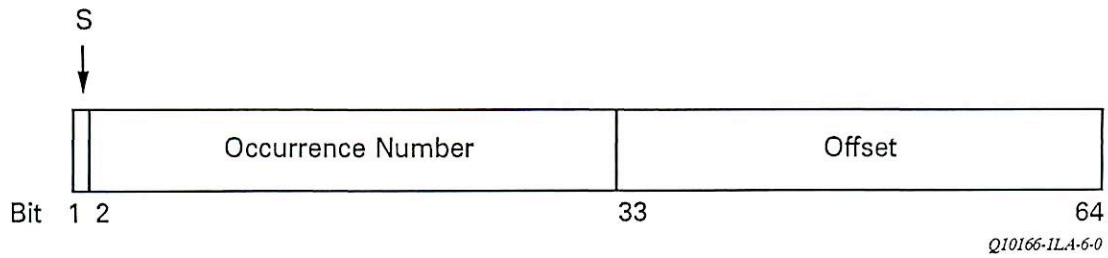


FIGURE 4-5
Index Storage

Prime Extension: Floating-point Item (COMP-1, COMP-2)

A **single-precision floating-point item** is defined by a USAGE clause of COMPUTATIONAL-1 or COMP-1. No PICTURE clause is allowed. The item occupies 32 bits of which bit 1 (the leftmost bit) is the sign, bits 2-24 are the mantissa, and bits 25-32 contain the exponent. The sign and mantissa are treated as a two's-complement number, and the exponent is an unsigned excess-128 binary exponent. Effective precision is between 22 and 23 bits (± 8,388,607). The exponent range is –128 through +127 (10 to the ±38 power). Figure 4-6 represents the storage of this item.

FIGURE 4-6
Single-precision Floating-point Storage

A **double-precision floating-point item** is defined by a USAGE clause of COMPUTATIONAL-2 or COMP-2. No PICTURE clause is allowed. The item occupies 64 bits of which bit 1 (the leftmost bit) is the sign, bits 2-48 are the mantissa, and bits 49-64 contain the exponent. The sign and mantissa are treated as a two's-complement number, and the exponent is an unsigned excess-128 binary exponent. Effective precision is between 46 and 47 bits (± 737,488,355,327). The exponent range is –32896 through +32639 (10 to the +9823 or –9812 power). Figure 4-7 represents the storage of this item.

FIGURE 4-7
Double-precision Floating-point Storage

Floating-point format is a Prime extension to ANSI COBOL intended for use in scientific calculations, when very large or very small numbers must be represented, or when you wish to call FORTRAN or PL1G subroutines that operate on floating-point (real) numbers.

In a COBOL85 statement, the format of a floating-point number is

[(±)]*mantissa*E[(±)]*exponent*

The mantissa consists of one to seven digits for COMP-1 or one to fourteen characters for COMP-2 with a required decimal point. Examples are

```
MOVE 1.23456E-10 TO ITEM1.
IF TEST1 > 4.0E14 PERFORM 050-EXCESS.
```

Floating-point items are compiler-aligned on halfword (16-bit) boundaries.

Notes

Be careful when you use floating-point operands in computations with other operand types. In order to retain the precision of standard COBOL operand types, COMP-1 and COMP-2 operands may be converted to COBOL85 data types. In the process, the contents of the COMP-1 or COMP-2 operands may be truncated, because the range of floating-point operands exceeds that of standard COBOL operand types. On the other hand, because the precision of standard COBOL operands (1 to 18 digits) exceeds that of floating-point operands (7 or 14 digits), precision can be lost when conversion to floating-point is required.

Also be aware that mixed operations can cause the **nines syndrome,** where, for example, a value of 41 at the beginning of a mixed operation may end up as 40.9999.

In general, it is a good rule to use floating-point operands in a COBOL85 context only when strictly required, as is the case when operands with extremely large ranges are required, or when a COBOL85 program interacts with a FORTRAN, PL/I, or PL/I subset G program.

When using floating-point numbers, or results of operations using floating-point numbers, in relational tests, use tests of GREATER THAN or LESS THAN, not of EQUALS, or round the numbers before using or testing them.

Standard Alignment Rules

The COBOL85 compiler automatically aligns data as needed at compilation time. DISPLAY, PACKED-DECIMAL, and COMP-3 items are aligned on byte boundaries, with the exceptions discussed in the next section. All other items are aligned on 16-bit boundaries. At execution time, the standard rules by which the compiler positions data within an elementary item depend on the category of the receiving item. These rules are

- If the receiving data item is described as numeric, the data is aligned by decimal point and is moved to the receiving digit positions with zero filling or truncation at either end, as required.
- When an assumed decimal point is not explicitly specified, the data item is treated as if it had an assumed decimal point immediately following its rightmost digit. It is aligned as in the rule above.
- If the receiving data item is numeric edited, the data moved to the edited data item is aligned by decimal point. Zero filling or truncation at either end occurs as required except where editing requirements cause replacement of the leading zeros.
- If the receiving data item is alphanumeric (other than a numeric edited data item), alphanumeric edited, or alphabetic, the sending data is aligned at the leftmost character position in the receiving data item. Space filling or truncation occurs to the right, as required.

If the JUSTIFIED clause is specified for the receiving item, these standard rules are modified as described in the section titled JUSTIFIED, in Chapter 7.

The alignment examples in Table 4-5 show the results of moving various length alphabetic and alphanumeric items into an 11-character alphanumeric field. (b = blank.)

TABLE 4-5
Alphanumeric Alignment

Data to be Stored	*Receiving Field Before Transfer*	*Receiving Field After Transfer*
ABC	XXXXXXXXXXX	ABCbbbbbbbb
ABCDEF1234	XXXXXXXXXXX	ABCDEF1234b
AAABBBCCCDD	XXXXXXXXXXX	AAABBBCCCDD
AAABBBCCCDDDE	XXXXXXXXXXX	AAABBBCCCDD

The examples in Table 4-6 show the results of moving various length numeric items into a six-character numeric field. (^ = implied decimal point.)

TABLE 4-6
Numeric Alignment

Data to be Stored	*Receiving Field Before Transfer*	*Receiving Field After Transfer*
3^4	999V999	003^400
345^678	999V999	345^678
12345^67890	999V999	345^678
34^	999V999	034^000
1234567890	999V999	890^000
1234567890	9999V99	7890^00
3^4	999999	000003

Prime Extension: Alignment of Substructures Within Structures

The compiler automatically aligns certain elements on 16-bit boundaries in order to allow substructures to be passed to called programs correctly. Alignment follows these rules:

- Each level-01 or level-77 item is allocated on a 16-bit boundary.
- Each group item subordinate to a level-01 item is aligned on the largest boundary required by any item contained in it.

COBOL85 Type	*Alignment Required*
DISPLAY	Byte
BINARY	16 bits
COMP	16 bits

COMP-1	16 bits
COMP-2	16 bits
PACKED-DECIMAL	Byte
COMP-3	Byte

- Compiler-generated filler is inserted into structures where necessary to make substructures align on the proper boundary.
- If you specify the –SLACKBYTES option at compile time, the compiler issues a diagnostic when filler is added to align a substructure. If you specify the –MAP option, each data item so aligned is indicated by the phrase COMPILER-ALIGNED.

Examples: The following structure is to be passed to a called program. A and B may be byte-aligned, while C and S2 require 16-bit alignment.

```
01  STRUC1.
    02 A              PIC X.
    02 S2.
        03 B          PIC X.
        03 C          COMP.
```

The compiler actually allocates the structure as

```
01 STRUC1.
    02 A              PIC X.
    02 FILLER         PIC X.
    02 S2.
        03 B          PIC X.
        03 FILLER     PIC X.
        03 C          COMP.
```

When S2 is passed to a called program, this automatic alignment allows the programmer to pass the subgroup because it is already aligned to correspond to a level-01 or level-77 group in the called program. Thus, the argument can be described as a level-01 group in the called program, as the following example illustrates.

```
LINKAGE SECTION.
01  M.
    02  B  PIC X.
    02  C  COMP.
  .
  .
PROCEDURE DIVISION USING M.
  .
  .
```

Algebraic Signs

Algebraic signs fall into two categories: operational signs and editing signs. **Operational signs** are associated with signed numeric data items and signed numeric literals to indicate their algebraic properties. **Editing signs** appear on edited reports to identify the sign of the item.

The SIGN clause permits you to state explicitly the location of the operational sign. Editing signs are inserted into a data item with the editing symbols of the PICTURE clause.

Qualification of Names

You must be able to identify, uniquely, every name that defines an element in a COBOL85 source program. You can make the name unique in its spelling or hyphenation, or by using qualifier names.

Qualifiers are names of higher-level items (that is, of a lower level-number) preceded by the word OF or IN. A series of items connected by OFs or INs may qualify one name. The general formats for qualification are

Format 1

$$\left\{\begin{matrix}\textit{data-name-1}\\ \textit{condition-name}\end{matrix}\right\}\left[\left\{\begin{matrix}\underline{\text{OF}}\\ \underline{\text{IN}}\end{matrix}\right\}\textit{data-name-2}\right]\cdots$$

Format 2

$$\textit{paragraph-name}\left[\left\{\begin{matrix}\underline{\text{OF}}\\ \underline{\text{IN}}\end{matrix}\right\}\textit{section-name}\right]$$

Format 3

$$\textit{'file-name'}\left[\left\{\begin{matrix}\underline{\text{OF}}\\ \underline{\text{IN}}\end{matrix}\right\}\textit{'directory-name'}\right]$$

Format 4

status-name OF *switch-name*

The rules for qualification are

- Each qualifier must be of a higher level and within the same hierarchy as the name it follows.
- The same name must not appear at two levels in the same hierarchy.
- If a *data-name* or a *condition-name* is assigned to more than one item in a source program, the name must be qualified each time it is referred to.

- A *paragraph-name* may be qualified only by its *section-name*. Therefore, two identical *paragraph-names* must not appear in the same section. When a paragraph is qualified by a *section-name*, the word SECTION must not appear. A *paragraph-name* need not be qualified when referred to within the same section.
- A name can be qualified even though it does not need qualification. If more than one combination of qualifiers can make a name unique, any combination can be used. The complete set of qualifiers for a *data-name* must not be the same as any partial set of qualifiers for another *data-name*.
- The maximum number of qualifiers is one for a *paragraph-name* and 48 for a *data-name* or *condition-name*. *file-names* may be qualified only in a COPY statement. *mnemonic-names* and *section-names* must not be qualified.

In the following example, the *data-name* YEAR requires qualification for reference because it defines two elementary items, one in HIRE-DATE and one in TERMINATION-DATE.

```
01  EMPLOYEE-RECORD
    05  NAME
    05  ADDRESS
    05  HIRE-DATE
        10  YEAR
        10  MONTH
        10  DAYY
    05  TERMINATION-DATE
        10  YEAR
        10  MONTH
        10  DAYY
```

YEAR OF HIRE-DATE is a qualified reference that differentiates between year fields in HIRE-DATE and TERMINATION-DATE. YEAR OF HIRE-DATE IN EMPLOYEE-RECORD is also a valid qualifier for the first YEAR field.

Arithmetic Expressions

An arithmetic expression must be one of the following:

- A name of a numeric elementary item
- A numeric literal
- Such names and literals separated by arithmetic operators
- Two arithmetic expressions separated by an arithmetic operator
- An arithmetic expression enclosed in parentheses

Any arithmetic expression may be preceded by a unary operator. The permissible combinations of variables, numeric literals, arithmetic operators, and parentheses are given in Table 4-7.

TABLE 4-7
Symbol Combinations in Arithmetic Expressions

First Symbol	*Variable*	*Second Symbol* * / + – **	*Unary + or –*	*(*	*)*
Variable	X	P	X	X	P
* / + – **	P	X	X	P	X
Unary + or –	P	X	X	P	X
(	P	X	P	P	X
)	X	P	X	X	P

P = Permitted, X = Invalid.

Names and literals appearing in an arithmetic expression must represent either numeric elementary items or numeric literals on which arithmetic may be performed.

Arithmetic Operators

The characters in Table 4-8 represent the binary and unary arithmetic operators.

TABLE 4-8
Binary and Unary Arithmetic Operators

Symbol	*Meaning*
Binary Arithmetic Operators	
+	Addition
–	Subtraction
*	Multiplication
/	Division
**	Exponentiation
Unary Arithmetic Operators	
+	The effect of multiplication by +1 (sign normalization)
–	The effect of multiplication by –1 (sign inversion)
Parentheses	
( )	Encloses expressions to specify the sequence in which conditions are evaluated

Rules

Follow these general rules for arithmetic expressions:

- Use parentheses in arithmetic expressions to specify the order in which elements are to be evaluated. Expressions within parentheses are evaluated first. Within nested parentheses, evaluation proceeds from the innermost set to the outermost set.
- When you do not use parentheses, the following hierarchical order of execution is implied:

  1. Unary plus and minus
  2. Exponentiation
  3. Multiplication and division
  4. Addition and subtraction

  The order of execution of consecutive operations of the same hierarchical level is from left to right. For example,

  ```
  A + B / ( C – D * E )
  ```

  This expression is evaluated in the following sequence:

  1. Compute the product D times E, considered as intermediate result R1.
  2. Compute intermediate result R2 as the difference C – R1.
  3. Divide B by R2, providing intermediate result R3.
  4. Compute the final result by addition of A to R3.

  Without parentheses, the expression A + B / C – D * E is evaluated as

  ```
  R1 = B/C
  R2 = D * E
  R3 = A + R1
  ```

  The final result is R3 – R2.
- When you use parentheses, use the following punctuation rules:

  1. Precede a left parenthesis with one or more spaces.
  2. Follow a right parenthesis with one or more spaces.

- Combine operators, variables, and parentheses in arithmetic expressions as summarized in Table 4-7.
- Begin an arithmetic expression only with one of the symbols (+ – or a variable; end it only with a) or a variable. Each left parenthesis must be to the left of its corresponding right parenthesis.
- **Prime Extension**: Spaces are always optional before and after the * / and ** binary arithmetic operators. The following examples are valid arithmetic expressions.

  ```
  X = A*B
  X = A/2
  X = A**2
  ```

- **Prime Extension**: When the + binary arithmetic operator separates two numeric elementary data items, spaces are optional before and after the + operator. The following example is a valid arithmetic expression.

```
X = A+B
```

- When the + binary arithmetic operator immediately precedes a numeric literal, a space must separate the + operator and the numeric literal. The following examples are valid arithmetic expressions.

```
X = A+ 5
X = 10+ 5
```

But these are *invalid* arithmetic expressions.

```
X = A+5
X = 10 +5
```

- When the – binary arithmetic operator immediately precedes a numeric literal, a space must separate the – operator and the numeric literal. The following examples are valid arithmetic expressions.

```
X = A – 5
X = 10– 5
```

But these are *invalid* arithmetic expressions.

```
X = A–5
X = 10 –5
```

- When the – binary arithmetic operator immediately follows a numeric elementary data item, a space must separate the data item and the – operator. The following examples are valid arithmetic expressions.

```
X = A –B
X = A – 5
```

But these are *invalid* arithmetic expressions.

```
X = A–B
X = A– 5
```

Arithmetic Statements

The arithmetic statements are the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements. These statements have several common features.

- The data descriptions of the operands need not be the same; any necessary conversion and decimal point alignment is supplied throughout the calculation.
- The maximum size of each operand is 18 decimal digits. The **composite of operands**, which is a hypothetical data item resulting from the superimposition of operands

aligned on their decimal points, must not contain more than 18 decimal digits. An example is given in the section titled Arithmetic Statements in the PROCEDURE DIVISION in Chapter 8.

Overlapping Operands

When a sending and a receiving item in an arithmetic statement or an INSPECT, MOVE, SET, STRING, UNSTRING, or other statements share a part of their storage areas, the result of the execution of such a statement is undefined and unpredictable.

Conditional Expressions

Conditional expressions identify conditions that are tested to enable the object program to select between alternate paths of control depending upon the truth value of the condition.

Simple Conditions

The **simple conditions** are the relation, class, *condition-name*, switch-status, and sign conditions. A simple condition has a truth value of true or false. The inclusion in parentheses of simple conditions does not change the simple truth value.

Relation Condition: A **relation condition** causes a comparison of two operands. A relation condition may have one of these formats:

Format 1

operand relation operand

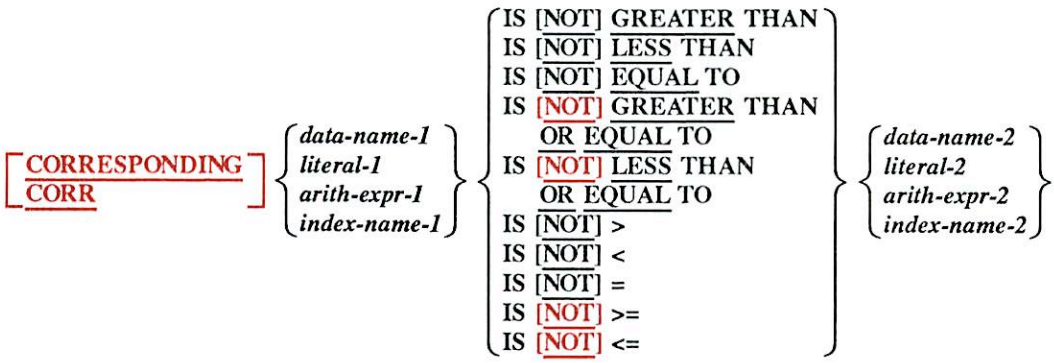
Format 2

[CORRESPONDING / CORR] ***operand relation operand***

The *relation* is a relational operator: EQUAL, GREATER, LESS, or the negation of one of these. A relation condition has a truth value of true if the relation exists between the operands. The *operand* is a *data-name*, *literal*, arithmetic expression, or figurative-constant.

Prime Extension: The CORRESPONDING option in relation conditions is a **Prime extension** and can be used only when the operands are group items.

The general format of a relation condition is as follows:



Note

Although required where indicated in formats, the relational characters < > and = are not underlined in this text.

The relational operator specifies the type of comparison to be made in a relation condition. A space must precede and follow each reserved word comprising the relational operator. When used, NOT and the next key word or relation character form one relational operator defining the comparison to be executed for truth value. Thus NOT EQUAL is a truth test for an unequal comparison; NOT GREATER is a truth test for an equal or less comparison.

Numeric and Nonnumeric Comparisons: Comparison of two numeric operands of different formats is permitted. If either operand is nonnumeric, the comparison is nonnumeric.

- **Numeric comparisons**: For elementary operands whose class is numeric, a comparison is made with respect to their algebraic value. The length of the operands is not significant. Zero is considered a unique value regardless of the sign. It is neither positive nor negative, and fails these sign tests.

  Comparison of these operands is permitted regardless of their usage. Unsigned numeric operands are considered positive.

  The data operands are compared after alignment of their decimal points. An *index-name* or index item may appear in a numeric comparison.

- **Nonnumeric comparisons**: For nonnumeric operands, a comparison is made with respect to the Prime collating sequence of characters. The value associated with each ASCII character in the Prime computer is the basis for the comparison. The collating sequence can be modified by the PROGRAM COLLATING SEQUENCE clause. (Refer to Appendix B for all ASCII character representations and the Prime collating sequence.)

  Comparison proceeds by comparing characters in corresponding character positions starting from the high-order (left) end and continuing until either a pair of unequal characters is encountered or the low-order end of the operand is reached. The operands are determined to be equal if all pairs of characters compare equally through the last pair, when the low-order end is reached.

  The first pair of unequal characters encountered is compared to determine their relative position in the collating sequence. The operand that contains the character positioned higher in the collating sequence is considered to be the greater operand.

The size of an operand is the total number of characters in the operand.

If the operands are of unequal size, comparison proceeds as though sufficient spaces were added to the left of the shorter operand to make the operands of equal size.

If one operand is a literal, the data class of the two operands must be the same.

If one of the operands is specified as numeric, it must be an integer data item or an integer literal and

- If the nonnumeric operand is an elementary data item or a nonnumeric literal, the numeric operand is treated as though it were an elementary alphanumeric data item of the same size as the numeric data item.
- If the nonnumeric operand is a group item, the numeric operand is treated as though it were a nonnumeric item of the same size as the numeric data item.
- A noninteger numeric operand cannot be compared to a nonnumeric operand.
- Numeric and nonnumeric operands may be compared only when their usage is the same.

Class Condition: The **class condition** determines whether an operand is numeric, alphabetic, lowercase alphabetic, uppercase alphabetic, or contains only the characters in the set of characters specified by the CLASS clause as defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

The general format for the class condition is

data-name **IS** [NOT] { **NUMERIC** / **ALPHABETIC** / **ALPHABETIC-LOWER** / **ALPHABETIC-UPPER** / *class-name-1* }

data-name must be described, implicitly or explicitly, as USAGE IS DISPLAY.

A NUMERIC data item consists entirely of the digits 0 through 9, with or without the operational sign.

The NUMERIC test cannot be used with *data-name* described as alphabetic or as a group item composed of elementary items whose data description indicates the presence of operational signs.

If the data description of the *data-name* being tested does not contain an operational sign, the *data-name* is determined to be numeric only if the contents are numeric and an operational sign is not present.

If the data description of the *data-name* being tested contains an operational sign, the *data-name* is determined to be numeric only if the contents are numeric and a valid operational sign is present.

An ALPHABETIC data item consists entirely of the uppercase letters A through Z, space, or the lowercase letters a through z, space, or any combination of the uppercase and lowercase letters and spaces.

An ALPHABETIC-UPPER data item consists entirely of the uppercase letters A through Z, and space.

An ALPHABETIC-LOWER data item consists entirely of the lowercase letters a through z, and space.

The ALPHABETIC, ALPHABETIC-UPPER, and ALPHABETIC-LOWER tests cannot be used with a *data-name* described as numeric. The *data-name* being tested is determined to be alphabetic only if the contents consist of any combination of the uppercase and lowercase letters and spaces. The *data-name* being tested is determined to be alphabetic-upper only if the contents consist of any combination of the uppercase letters and spaces. The *data-name* being tested is determined to be alphabetic-lower only if the contents consist of any combination of the lowercase letters and spaces.

A *class-name-1* data item consists entirely of characters included in the set of characters identified by *class-name-1* in the CLASS clause in the SPECIAL-NAMES paragraph.

The *class-name-1* test cannot be used with a *data-name* described as numeric.

Condition-name Condition: In a **condition-name condition**, a conditional variable is tested to determine whether or not its value is equal to a value associated with one of its *condition-names* in a level-88 entry of the DATA DIVISION. The general format for the *condition-name* statement is as follows:

[NOT] ***condition-name***

If the *condition-name* is associated with a range or ranges of values, then the conditional variable is tested to determine whether or not its value falls in this range, including the end values. (See Chapter 7 for details.)

The rules for comparing a conditional variable with a *condition-name* value are the same as those specified for relation conditions.

The result of the test is true if the content of the field associated with the *condition-name* equals one of the values specified for that *condition-name*.

Switch-status Condition: A **switch-status condition** determines the ON or OFF status of a switch. The *switch-name* and the ON or OFF value associated with the condition must be named in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. The general format for the switch-status statement is as follows:

[NOT] ***status-name***

The result of the test is true if the switch is set to the ON or OFF status associated with the switch in the SPECIAL-NAMES paragraph.

Sign Condition: The **sign condition** determines whether or not the algebraic value of an arithmetic expression is less than, greater than, or equal to 0. The general format for a sign condition is

$$\left\{\begin{matrix}\textit{data-name}\\ \textit{arith-expr}\end{matrix}\right\} \text{ IS } [\underline{\text{NOT}}] \left\{\begin{matrix}\underline{\text{POSITIVE}}\\ \underline{\text{NEGATIVE}}\\ \underline{\text{ZERO}}\end{matrix}\right\}$$

Complex Conditions

A complex condition is formed by combining simple conditions, combined conditions and/or complex conditions with logical connectors (logical operators AND and OR), or by negating these conditions with logical negation (the logical operator NOT). The truth of a complex condition is calculated as described in the section titled Condition Evaluation Rules, below. The logical operators are the following:

Logical Operator	*Meaning*
AND	Logical conjunction; the truth value is true if both of the conjoined conditions are true; false if one or both of the conjoined conditions is false.
OR	Logical inclusive OR; the truth value is true if one or both of the included conditions are true; false if both included conditions are false.
NOT	Logical negation is the reversal of the truth value; the truth value is true if the condition is false, and false if the condition is true.

Logical operators must be preceded and followed by a space.

Negated Simple Conditions: The general format of a **negated simple condition** is

NOT *simple-condition*

Thus, the *simple-condition* is negated through the use of the logical operator NOT.

The truth value of a negated simple-condition is the opposite of the truth value for a *simple-condition*. The negated condition is true if the *simple-condition* is false, and false if the *simple-condition* is true.

Inclusion in parentheses of a negated *simple-condition* does not affect the truth value.

Combined and Negated Combined Conditions: A **combined condition** is two or more conditions connected by one of the logical operators AND or OR. A combined condition has the format

$$[\underline{\text{NOT}}]\ \textit{condition-1}\ \left\{\left\{\begin{matrix}\underline{\text{AND}}\\ \underline{\text{OR}}\end{matrix}\right\}\ [\underline{\text{NOT}}]\ \textit{condition-2}\right\}\ \ldots$$

where *condition* is

- A simple condition
- A negated simple condition
- A combined condition
- A negated combined condition, that is, the logical operator NOT followed by a combined condition enclosed in parentheses
- Combinations of the above

Table 4-9 sets forth the permissible combinations of conditions, logical operators, and parentheses.

TABLE 4-9
Permissible Combinations of Conditions, Logical Operations, and Parentheses

Element	*Place in Expression*		*When not first, the element can be immediately preceded only by*	*When not last, the element can be immediately followed only by*
	First	*Last*		
Simple-condition	Yes	Yes	OR, NOT, AND, (	OR, AND,)
OR and AND	No	No	Simple-condition,)	Simple-condition, NOT, (
NOT	Yes	No	OR, AND, (	Simple-condition, (
(	Yes	No	OR, NOT, AND, (	Simple-condition, NOT, (
)	No	Yes	Simple-condition,)	OR, AND,)

Multiple Conditions: Multiple conditions refer to complex conditions grouped in parentheses.

Parentheses are permitted to an arbitrary depth. Often, however, you can enhance clarity by rewriting the condition without parentheses.

For example, in the statement

```
IF a = b AND (c = d OR e = f)
```

explicit grouping may be achieved by coding

```
IF a = b AND c = d OR a = b AND e = f
```

Abbreviated Combined Conditions

Abbreviated combined conditions are conditions with implied subjects or implied operators. That is, you can omit the subject of the relation condition, or both the subject and the relational operator, if they are the same as those in the preceding clause.

The format for an abbreviated combined condition is

[NOT] *relation-condition* { { AND / OR } [NOT] [*relational-operator*] *operand* } ...

You can use either form of abbreviation: the omission of subject, or the omission of subject and relational operator. The effect of such abbreviations is that of inserting the previously stated subject in place of the omitted subject, or the previously stated relational operator in place of the omitted operator. All insertions terminate once a complete simple condition is encountered within a complex condition.

In all instances, the results must comply with the rules outlined in Table 4-9 above.

If the word NOT is used in an abbreviated condition, it is evaluated as follows:

- NOT participates as part of the relational operator if the word immediately following NOT is GREATER, >, LESS, <, EQUAL, or =.
- Otherwise, NOT is interpreted as a logical operator with the result that the implied insertion of subject or relational operator results in a negated relation condition.

Below are examples of abbreviated combined conditions with their expanded equivalents:

Abbreviated Combined and Negated Combined Relation Conditions	*Expanded Equivalent*
`a = b OR c OR d`	`a = b OR a = c OR a = d`
`a > b AND NOT < c OR d`	`((a > b) AND (a NOT < c)) OR (a NOT < d)`
`NOT a = b OR c`	`(NOT (a = b)) OR (a = c)`
`a NOT EQUAL b OR c`	`(a NOT EQUAL b) OR (a NOT EQUAL c)`
`NOT (a GREATER b OR < c)`	`NOT ((a GREATER b) OR (a < c))`
`NOT (a NOT > b AND c AND NOT d)`	`NOT ((((a NOT > b) AND (a NOT > c)) AND (NOT (a NOT > d))))`

Condition Evaluation Rules

COBOL85 uses the following order of logical evaluation to determine the truth value of a condition.

1. Conditions within parentheses are evaluated first. Within nested parentheses, evaluation proceeds from the least inclusive (innermost) condition to the most inclusive (outermost) condition.

2. Truth values for simple conditions are evaluated in the following order:

   Relation (following the expansion of any abbreviated relation condition)

   Class

   Condition-name

   Switch-status

   Sign

3. Truth values for negated simple conditions are established.
4. Truth values for combined conditions are established with this hierarchy:

   AND logical operators

   OR logical operators

5. Truth values for negated combined conditions are established.
6. When the sequence of evaluation is not completely specified by parentheses, the order of evaluation of consecutive operations of the same hierarchical level is from left to right.

The following examples illustrate the condition evaluation rules:

1. The condition below contains both AND and OR connectors.

   IF X = Y AND FLAG = "Z" OR SWITCH = 0, GO TO PROCESSING.

   Execution is as follows, depending on various data values listed in Table 4-10:

TABLE 4-10
Combined Condition Evaluation

Data Values				
X	*Y*	*FLAG*	*SWITCH*	*GOTO Executes*
10	10	'Z'	1	Yes
10	11	'Z'	1	No
10	11	'Z'	0	Yes
10	10	'p'	1	No
6	3	'p'	0	Yes
6	6	'p'	1	No

2. A < B OR C = D OR E NOT > F: The evaluation is equivalent to (A < B) or (C = D) or NOT (E > F) and is true if any of the three individual parenthesized simple conditions is true.

3. The following time-card record description includes three *condition-names*. It is followed by an evaluation.

```
01 TIME-CARD
    05 EMP-STATUS        PIC X.
      88 W     VALUE 'W'.
      88 H     VALUE 'H'.
      88 E     VALUE 'E'.
    05 HOURS             PIC 99.
     .
     .
     .
    IF W AND HOURS NOT = 0 ...
```

   The evaluation is equivalent to

   IF (EMP-STATUS = 'W') AND NOT (HOURS = 0)...

   and is true only if both the simple conditions are true.

4. A = 1 AND B = 2 AND G > –3 OR P NOT EQUAL TO "SPAIN" is evaluated as

   [(A = 1) AND (B = 2) AND (G > –3)] OR NOT (P = "SPAIN")

   If P = "SPAIN", the complex condition can be true only if all three of the following are true:

   A = 1
   B = 2
   G > –3

   However, if P is not equal to "SPAIN", the complex condition is true regardless of the values of A, B, and G.

Variable-length Records

A variable-length record is a record associated with a file whose *file-description-entry* permits records to contain a varying number of character positions. Using variable-length record functionality can save substantial disk space, depending upon the application. The following paragraphs discuss variable-length record support and methods of specifying variable-length records in COBOL85.

File Types That Support Variable-length Records

The following file types support variable-length records:

- PRIMOS sequential disk files
- PRISAM sequential, indexed, and relative files
- MIDASPLUS indexed files
- Magnetic tape files

Variable-length records for any of these file types can vary in length from one to the maximum length supported for the particular file type. Any keys defined within records must be within the fixed portion of the record descriptions associated with a variable-length file.

Note

COBOL85 does not support COMPRESSED and PRINTER variable-length file types.

File Formats of Variable-length Records

For information on variable-length file formats for sequential disk files, indexed files, relative files, and sequential tape files, see Chapters 9, 10, 11, and 12, respectively.

For additional information on MIDASPLUS and PRISAM file types that support variable-length records, refer to the *MIDASPLUS User's Guide* and the *PRISAM User's Guide*.

Specifying Variable-length Records in a Program

Use any of the following methods to specify variable-length records in a COBOL85 program:

The –VARYING Default Compiler Option: The –VARYING compiler option, which is specified by default, causes COBOL85 to treat as variable-length files all files that have file descriptions containing multiple *record-description-entries* of different sizes. –VARYING also causes all files having file descriptions that contain record descriptions with variable occurrence data items to be treated as variable-length files. Variable occurrence data items defined in the WORKING-STORAGE SECTION are also treated as variable in length. For additional information on the –VARYING and –NO_VARYING compiler options, see Chapter 2.

The RECORD IS VARYING and RECORDING MODE IS V Clauses: The specification of the following clauses in a file description entry specifies variable-length records:

```
RECORD IS VARYING IN SIZE [[FROM integer-1]
                          [TO integer-2] CHARACTERS ]

RECORDING MODE IS V
```

Specifying either of these clauses overrides the –NO_VARYING compiler option. Also, specifying these clauses in file descriptions that do not contain records of different sizes or variable occurrence data items allows such files to be formatted as variable-length files. For additional information on these clauses refer to Chapter 7.

Variable Occurrence Data Items: A **variable occurrence data item** is a table element that is repeated a variable number of times. The specification of the following clause in a record description entry specifies a variable occurrence data item:

```
OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name
```

Using the OCCURS DEPENDING ON clause in conjuction with the –VARYING option, or within a file description that contains a RECORD IS VARYING or RECORDING MODE IS V clause, denotes a variable-length table whose size is dependent upon the value of a *data-name*. For additional information on the OCCURS DEPENDING ON clause refer to Chapter 7.

Specifying Variable-length Records in CREATK

To create a MIDASPLUS file of variable-length records, use CREATK as follows:

1. Begin the CREATK dialog, indicating that you are creating a keyed-indexed access file, the only type of MIDASPLUS file that can contain variable-length records.
2. When CREATK prompts `DATA SIZE IN WORDS`, enter a 0 followed by two values. The first value sets the minimum record size; the second value sets the maximum record size. The minimum must be at least 1; the maximum can be at most 32767.

Three CREATK commands, INITIALIZE, GET, and PRINT, help you administer variable-length record files.

For more information, see the *MIDASPLUS User's Guide*.

Specifying Variable-length Records in DDL

To specify a PRISAM file of variable-length records, use one of the following methods:

- Specify multiple record descriptions of varying lengths
- Specify an OCCURS clause with the DEPENDING phrase

Note

Relative files are always preallocated as maximum length records whether or not you specify them as fixed or variable.

For more information, see the *PRISAM User's Guide*.

Table Handling

This section discusses tables of repeating data items and various means of referring to those items according to their positions in the tables.

Table Definition

To define a data item as a **table element**, use an OCCURS clause in the item's *data-description-entry*. The OCCURS clause specifies that the item is to be repeated as many times as stated. The item's name and description apply to each repetition or occurrence. Since each occurrence of a table element does not have a unique name, to refer to a particular occurrence, you must specify both the name and the occurrence number of the table element. The occurrence number is called a **subscript**.

You can specify the number of occurrences of a table element to be fixed or variable. Use the DEPENDING phrase of the OCCURS clause to specify **variable occurrence data items**. See the OCCURS clause in Chapter 7 for more information.

Table Initialization

Table initialization, if required, may be achieved either in the WORKING-STORAGE SECTION or in the PROCEDURE DIVISION. The VALUE clause is not permitted in a *data-description-entry* specifying an OCCURS or REDEFINES clause, or in any entry subordinate to one specifying an OCCURS or REDEFINES clause. The following paragraphs suggest means of assigning values to table elements.

In the WORKING-STORAGE SECTION of the DATA DIVISION, tables can be initialized in one of two ways:

- If the elements in a table do not need to be individually initialized, you can specify the VALUE clause in the *data-description-entry* containing the table name. Then give the subordinate *data-description-entry* an OCCURS clause defining the structure of the table.

  For example,

```
01  A-TABLE        VALUE ZEROS.
  05  B-TABLE     PIC 9(3)  OCCURS 100 TIMES.

01  STATE-TABLE   VALUE 'CALAMAPAVA'.
  05  STATE       PIC XX  OCCURS 5 TIMES.
```

- If the elements in a table need to be individually initialized, you must first define the table as a nontable structure with the desired number of characters. You can then specify a VALUE clause in each element entry of the nontable structure and then redefine the structure as a table with REDEFINES plus a subordinate entry containing an OCCURS clause.

  For example,

```
01  WAREHOUSE.
    05  FILLER     PIC 99       VALUE 10.
    05  NAME       PIC X(22)    VALUE 'BOSTON DISTRICT BRANCH'.
    05  FILLER     PIC 99       VALUE 11.
    05  FILLER     PIC X(22)    VALUE 'NEW YORK CITY BRANCH  '.
    05  FILLER     PIC 99       VALUE 12.
    05  FILLER     PIC X(22)    VALUE 'HOUSTON HOME OFFICE   '.
01  WARE-HOUSE REDEFINES WAREHOUSE.
    05  HOUSES   OCCURS 3 TIMES.
        10  HOUSE-NO       PIC 99.
        10  HOUSE-NAME     PIC X(22).
```

In the PROCEDURE DIVISION, you can initialize a table with MOVE statements:

```
MOVE '10BOSTON DISTRICT BRANCH11NEW YORK CITY BRANCH  12HOUST
-    'ON HOME OFFICE   ' TO WAREHOUSE.
```

Using Subscripts

Use subscripts to refer to an individual element within a table of like elements that have not been assigned individual *data-names*.

Format

$$\left\{\begin{matrix}\textit{condition-name}\\ \textit{data-name-1}\end{matrix}\right\}\ (\ \left\{\begin{matrix}\textit{integer-1}\\ \textit{data-name-2}\ [\{\pm\}\ \textit{integer-2}]\\ \textit{index-name}\ [\{\pm\}\ \textit{integer-3}]\\ \textit{arith-expr}\ [\{\pm\}\ \textit{integer-4}\ ]\end{matrix}\right\}\ \ldots\ )$$

Syntax Rules

1. The subscript can be represented by a numeric literal, by a numeric *data-name*, by an *index-name*, or by an arithmetic expression. The *data-name* subscripts may themselves be qualified or subscripted.
2. The subscript may be signed. It must have a positive integer value. The lowest possible subscript value is 1. This value points to the first element of the table. The next elements of the table are pointed to in turn by subscripts whose values are 2, 3, and so on. The highest permissible subscript value, in any particular case, is the maximum number of occurrences of the item as specified in the OCCURS clause.

   Note

   Literal subscripts are range-checked at compile time. Variable subscripts can be checked at runtime if the –RANGE option is specified at compile time.

3. The subscript that identifies a table element is delimited by the balanced pair of separators, left parenthesis and right parenthesis, following the table element *data-name*. When more than one subscript is required, write them in the order of successively less inclusive dimensions of the table organization.
4. An *index-name* is a name specified in the INDEXED BY phrase in the table definition. The value of an index corresponds to the occurrence number of an element in the associated table.
5. An *index-name* must be initialized before it is used in a table reference. An *index-name* can be given an initial value by a SET, a SEARCH ALL, or a Format 4 PERFORM statement.
6. An *index-name* can be modified only by the SET, SEARCH, and Format 4 PERFORM statements.
7. Data items described by the USAGE IS INDEX clause permit storage of the values associated with *index-names*. Such data items are called **index data items**.

Types of Subscripting

You can use any of the following types of subscripting to specify the occurrence number of a particular table element:

- Literal subscripting
- *data-name* subscripting

- Arithmetic expression subscripting
- Direct indexing
- Relative indexing

Literal Subscripting: An integer in parentheses is used for literal subscripting. For example, given the following three-element array,

```
01 ARRAY.
   05 ELEMENT    PICTURE S9(4)    OCCURS 3 TIMES.
```

the statement below results in the contents of the second ELEMENT of ARRAY being moved to the field called PART-NO.

```
MOVE ELEMENT(2) TO PART-NO.
```

The integer 2 is a literal subscript.

data-name Subscripting: An additional *data-description-entry* is required to define a *data-name* to be used as a subscript (SUBSCRIPTNO in this example):

```
01 ARRAY.
    05 ELEMENT    PICTURE S9(4)    OCCURS 10 TIMES.
    .
    .
    .
01  SUBSCRIPTNO              PIC 99.
01  PART-NO                  PIC X(4).
01  ONE                      PIC S9(1)      VALUE 1.
01  TWO                      PIC S9(1)      VALUE 2.
01  THREE                    PIC S9(1)      VALUE 3.
    .
    .
    .
    MOVE 2 TO SUBSCRIPTNO.
    PERFORM 050-TABLERUN.
    .
    .
    .
050-TABLERUN.
    MOVE ELEMENT(SUBSCRIPTNO) TO PART-NO.
```

Prime Extension: Arithmetic Expression Subscripting: This form of subscripting is similar to *data-name* subscripting except that any subscript may be an arithmetic expression. The arithmetic expression, at the time of reference, must evaluate to a positive integer. For example, given the array and subscript defined above, the following statement results in the contents of the seventh ELEMENT of ARRAY being moved to PART-NO.

```
MOVE ELEMENT(ONE + TWO * THREE) TO PART-NO.
```

ONE + TWO * THREE is an arithmetic expression that evaluates to a positive 7.

Direct Indexing: **Direct indexing** is specified by using an *index-name* alone within parentheses, for example, ELEMENT(INDX1).

Consider the following illustration:

```
      .
      .
      .
01 TABLE-A.
   05 ELEMENT OCCURS 6 TIMES INDEXED BY INDX1.
      .
      .
      .
    SET INDX1 TO 4.
    MOVE ELEMENT(INDX1) TO PRINT-FIELD.
      .
      .
      .
```

ELEMENT(INDX1) in the example above refers to the fourth element of the table. The MOVE statement moves the contents of the fourth occurrence of ELEMENT to a field called PRINT-FIELD.

Relative Indexing: **Relative indexing** uses an arithmetic expression to compute the location of a table element. Using the sample TABLE-A defined in the example above, the same results could be achieved with relative indexing. If INDX1 has a value of 1, the fourth element of TABLE-A can be moved to PRINT-FIELD with this statement:

```
MOVE ELEMENT(INDX1 + 3) TO PRINT-FIELD.
```

In relative indexing, *index-name* is followed by a space, followed by one of the operators + or –, followed by another space, followed by an unsigned integer numeric literal or arithmetic expression, all delimited by the balanced pair of separators left parenthesis and right parenthesis.

The occurrence number resulting from relative indexing is determined by incrementing or decrementing the index by the value of the literal or arithmetic expression.

Multidimensional Tables

When a table has more than one dimension, the *data-name* of the desired item is followed by a list of subscripts, one for each OCCURS clause to which the item is subordinate.

In such a list, the first subscript applies to the first OCCURS clause to which the item is subordinate. The second subscript applies to the next most encompassing level. The third subscript applies to the next lower level OCCURS clause being accessed, and so on.

The following example presents DATA DIVISION entries for a multidimensional table, TABLE-PLUS.

```
01 TABLE-PLUS.
   05 TYPE OCCURS 10 TIMES.
      10 PART-NO          PIC X(4).
      10 COLOR            PIC X OCCURS 10 TIMES.
      10 CONTROL OCCURS 7 TIMES.
         15 C1            PIC X.
         15 C2            PIC XX OCCURS 4 TIMES.
```

The statement

```
MOVE C2(8, 6, 4) TO TEMP.
```

moves the contents of the fourth occurrence of the field C2, in the sixth occurrence of the field CONTROL, in the eighth occurrence of the field TYPE, to a field called TEMP.

Similarly, the statement

```
MOVE C2(10, 7, 4) TO TEMP.
```

moves the contents of the last occurrence of the field C2 to the field labeled TEMP.

Subscripting Qualified data-names

To subscript a qualified *data-name*, code the subscript at the end of the fully qualified *data-name*.

For example, suppose a program contains the following data items in WORKING-STORAGE:

```
01  RECORD-1.
    05  GROUP-1.
        10  ELEMENT-1 OCCURS 10 TIMES PIC X(4).
        10  ELEMENT-2                 PIC X(4).

01  RECORD-2.
    05  GROUP-1.
        10  ELEMENT-1 OCCURS 10 TIMES PIC X(4).
        10  ELEMENT-2                 PIC X(4).
```

To refer to the fifth occurrence of ELEMENT-1 in GROUP-1 of RECORD-1 in the PROCEDURE DIVISION, code

```
ELEMENT-1 OF GROUP-1 OF RECORD-1(5)
```

not

```
ELEMENT-1(5) OF GROUP-1 OF RECORD-1
```

Table Handling Example

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.                    BUDGET.
 AUTHOR.                        PRIMATE1.
 INSTALLATION.                  PRIME.
 DATE-COMPILED.
*****************************************************************
 REMARKS. THE PROGRAM READS A FILE CONTAINING BUDGET LIMITS AND
     EXPENDITURES.  BUDGET LIMIT RECORDS HAVE A "B" CODE IN THE
     FIRST POSITION; EXPENDITURE RECORDS HAVE AN "E" CODE.
     IF ANY BUDGET LIMIT ENTRIES ARE MISSING, ZERO AMOUNTS ARE
     ASSUMED.  THE EXPENDITURE RECORDS MUST BE IN ORDER BY
     CATEGORY WITHIN ACCOUNT.

     THE PROGRAM BUILDS A TABLE CONTAINING BUDGET LIMITS AND
     EXPENDITURES FOR EACH BUDGET CATEGORY WITHIN EACH ACCOUNT.

     THE PROGRAM PRODUCES ONE REPORT.  FOR EACH ACCOUNT NUMBER,
     IT SEARCHES ALL CATEGORIES, COMPARES EXPENDITURES
     WITH BUDGETED LIMITS, AND PRINTS ANY CATEGORIES THAT ARE
     OVER THE BUDGETED LIMIT.

     NOTE: THE PROGRAM DOES NOT VALIDATE INPUT.  IT ASSUMES THAT
           AN INPUT DATA ITEM THAT IS TO BE USED AS AN INDEX IS
           WITHIN THE VALID RANGE.  IT ALSO ASSUMES THAT NUMERIC
           DATA ITEMS CONTAIN ONLY VALID NUMERICS.

*****************************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PRIME.
 OBJECT-COMPUTER. PRIME.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT INPUT-FILE ASSIGN TO PRIMOS.
     SELECT PRINT-FILE ASSIGN TO PRINTER.
*****************************************************************
 DATA DIVISION.
 FILE SECTION.
 FD  INPUT-FILE COMPRESSED,
     VALUE OF FILE-ID IS 'BUDGET.DATA'.
 01  ENTRY                      PIC X(80).
*
 FD  PRINT-FILE,
     LABEL RECORDS ARE OMITTED.
 01  PRINT-LINE                 PIC X(132).
*
 WORKING-STORAGE SECTION.
 77  ACCT-SAVE-WS               PIC X(3).
 77  EXCESS-COUNT-WS            PIC S999 COMP-3    VALUE 0.
 77  NO-MORE-INPUT              PIC X              VALUE 'N'.
*************************************************************
*TWO-DIMENSIONAL TABLE FOR ACCTS AND CATEGORIES, INDEXED.    *
*************************************************************
```

```
 01  BUDGET-TABLE.
     05 ACCOUNTS         OCCURS 8 TIMES    INDEXED BY ACCT-IDX.
        10 CATEGORIES    OCCURS 10 TIMES   INDEXED BY CAT-IDX.
           15  BUDGETED-AMT     PIC 9(5)V99.
           15  AMT-SPENT        PIC 9(10)V99.
************************************************************
* ONE-DIMENSIONAL TABLE FOR ACCOUNT NAMES IN EXCESS-LINE  *
************************************************************
 01  ACCOUNT-NAMES.
     05 FILLER            PIC X(03) VALUE 'ABC'.
     05 FILLER            PIC X(03) VALUE 'DEF'.
     05 FILLER            PIC X(03) VALUE 'GHI'.
     05 FILLER            PIC X(03) VALUE 'JKL'.
     05 FILLER            PIC X(03) VALUE 'MNO'.
     05 FILLER            PIC X(03) VALUE 'PQR'.
     05 FILLER            PIC X(03) VALUE 'STU'.
     05 FILLER            PIC X(03) VALUE 'VWX'.
 01  ACCOUNT-NAMES-R   REDEFINES   ACCOUNT-NAMES.
     05 ACCOUNT-NAME   OCCURS 8 TIMES  PIC X(03).
************************************************************
* ONE-DIMENSIONAL TABLE FOR CATEGORY NAMES IN EXCESS-LINE *
************************************************************
 01  CATEGORY-NAMES.
     05 FILLER            PIC X(10) VALUE 'AUTO      '.
     05 FILLER            PIC X(10) VALUE 'CLOTHING  '.
     05 FILLER            PIC X(10) VALUE 'FOOD      '.
     05 FILLER            PIC X(10) VALUE 'INSURANCE '.
     05 FILLER            PIC X(10) VALUE 'MAINTENANC'.
     05 FILLER            PIC X(10) VALUE 'MEDICAL   '.
     05 FILLER            PIC X(10) VALUE 'MORTGAGE  '.
     05 FILLER            PIC X(10) VALUE 'RECREATION'.
     05 FILLER            PIC X(10) VALUE 'UTILITIES '.
     05 FILLER            PIC X(10) VALUE 'MISC      '.
 01  CATEGORY-NAMES-R   REDEFINES   CATEGORY-NAMES.
     05 CATEGORY-NAME   OCCURS 10 TIMES  PIC X(10).
*
************************************************************
*    WORK LINES                                            *
************************************************************
*LIMIT-LINE IS BUDGET LIMIT:
 01  LIMIT-LINE.
     05 CODE-LT                  PIC X.
     05 ACCT-LT                  PIC X(3).
     05 CAT-LT                   PIC 99.
     05 DATE-LT.
        10 MONTH-LT              PIC 99.
        10 DAY-YR                PIC 9(4).
     05 AMT-LT                   PIC 9(5)V99.
     05 FILLER                   PIC X(61).
*EXPENSE-LINE IS EXPENDITURES:
 01  EXPENSE-LINE       REDEFINES      LIMIT-LINE.
     05 CODE-EX                  PIC X.
     05 ACCT-EX                  PIC X(3).
     05 CAT-EX                   PIC 99.
     05 DATE-EX.
        10 MONTH-EX              PIC 99.
```

```
           10 DAY-EX                PIC 99.
           10 YEAR-EX               PIC 99.
        05 AMOUNT-EX                PIC 9(5)V99.
        05 FILLER                   PIC X(61).
*
 01  EXCESS-WORK.
        05 CATEGORY                 PIC X(10)      VALUE SPACES.
        05 AMOUNT                   PIC 9(10)V99   VALUE 0 COMP-3.
        05 BUDGET-LIMIT             PIC 9(10)V99   VALUE 0 COMP-3.
        05 OVER                     PIC 9(10)V99   VALUE 0 COMP-3.
        05 PERCENT                  PIC 999V99     VALUE 0 COMP-3.
*
*****************************************************************
*                PRINT LINES                                    *
*****************************************************************
 01  HEADING1.
        05 CTL1                     PIC X      VALUE '1'.
        05 FILLER                   PIC X(11)  VALUE 'ACCOUNT    '.
        05 ACCT-PRINT               PIC X(3)   VALUE SPACES.
        05 FILLER                   PIC X(115)
          VALUE '             CATEGORIES EXCEEDING BUDGET
-       '                                                    '.
 01  HEADING2.
        05 CTL2                     PIC X      VALUE '0'.
        05 FILLER                   PIC X(71)
           VALUE 'CATEGORY        YTDSPENT       BDGTD AMT    AMOUNT
-       'OVER              '.
 01  EXCESS-PRINT.
        05 CTRL-EXC                 PIC X      VALUE ' '.
        05 CATEGORY                 PIC X(10)  VALUE SPACES.
        05 AMOUNT                   PIC Z(9)9.99.
        05 FILLER                   PIC X      VALUE SPACES.
        05 BUDGET-LIMIT             PIC Z(9)9.99.
        05 FILLER                   PIC XX   VALUE SPACES.
        05 OVER                     PIC Z(9)9.99.
 01  MESSAGE-LINE                   PIC X(49)
          VALUE '1NO CATEGORIES HAVE AN AMOUNT THAT EXCEEDS BUDGET'.
*****************************************************************
*
 PROCEDURE DIVISION.
*
 DECLARATIVES.
   INPUT-ERROR SECTION. USE AFTER ERROR PROCEDURE ON INPUT-FILE.
   FIRST-PARAGRAPH.
     DISPLAY '*** I-O ERROR ON INPUT FILE:  ***'.
     STOP RUN.
 END DECLARATIVES.
*
 0000-MAINLINE.
     PERFORM 1000-INITIALIZATION.
     PERFORM 2000-PROCESS-LIMITS UNTIL CODE-LT = 'E'.
     MOVE ACCT-EX TO ACCT-SAVE-WS.
     PERFORM 3000-PROCESS-EXPENSES UNTIL NO-MORE-INPUT = 'Y'.
     CLOSE PRINT-FILE, INPUT-FILE.
     STOP RUN.
*
```

```
 1000-INITIALIZATION.
     PERFORM 1100-ZERO-TABLES
             VARYING ACCT-IDX FROM 1 BY 1
             UNTIL ACCT-IDX > 8.
     PERFORM 1200-OPEN-FILES.
*
 1100-ZERO-TABLES.
     PERFORM 1110-ZERO-TABLES
             VARYING CAT-IDX FROM 1 BY 1
             UNTIL CAT-IDX > 10.
*
 1110-ZERO-TABLES.
     MOVE ZEROS TO BUDGETED-AMT(ACCT-IDX, CAT-IDX).
     MOVE ZEROS TO AMT-SPENT(ACCT-IDX, CAT-IDX).
*
 1200-OPEN-FILES.
     OPEN INPUT INPUT-FILE,
          OUTPUT PRINT-FILE.
     MOVE SPACES TO PRINT-LINE.
     WRITE PRINT-LINE AFTER ADVANCING PAGE.
     READ INPUT-FILE INTO LIMIT-LINE,
         AT END DISPLAY 'EMPTY FILE',
      CLOSE INPUT-FILE, PRINT-FILE
      STOP RUN.
*
 2000-PROCESS-LIMITS.
*************************************************************
*  READ BUDGET LIMITS INTO TABLE.  USE ACCT-LT AND          *
*  CAT-LT ON LIMIT-RECORD TO SET TABLE INDEXES:             *
*************************************************************
     SET  CAT-IDX TO CAT-LT.
     SET  ACCT-IDX TO 1.
     SEARCH ACCOUNT-NAME
            VARYING ACCT-IDX
       WHEN ACCT-LT = ACCOUNT-NAME (ACCT-IDX)
            NEXT SENTENCE.
     MOVE AMT-LT TO BUDGETED-AMT(ACCT-IDX, CAT-IDX).
     READ INPUT-FILE INTO LIMIT-LINE,
         AT END MOVE 'E' TO CODE-LT,
                MOVE 'Y' TO NO-MORE-INPUT,
                DISPLAY 'NO EXPENDITURES'.
*
 3000-PROCESS-EXPENSES.
     IF ACCT-EX NOT EQUAL ACCT-SAVE-WS,
         PERFORM 3200-NEXT-ACCT.
     PERFORM 3100-MAKE-TABLES.
     READ INPUT-FILE INTO EXPENSE-LINE,
         AT END MOVE 'Y' TO NO-MORE-INPUT,
             PERFORM 3200-NEXT-ACCT.
*
 3100-MAKE-TABLES.
*********************************************************
* SET ACCT-IDX, CAT-IDX FOR TABLE, ADD TO THOSE TOTALS. *
*********************************************************
     SET CAT-IDX TO CAT-EX.
     SET  ACCT-IDX TO 1.
```

```
           SEARCH ACCOUNT-NAME
                  VARYING ACCT-IDX
             WHEN ACCT-EX = ACCOUNT-NAME (ACCT-IDX)
                  NEXT SENTENCE.
           ADD AMOUNT-EX TO AMT-SPENT(ACCT-IDX, CAT-IDX).
       *
        3200-NEXT-ACCT.
           MOVE ACCT-SAVE-WS TO ACCT-PRINT.
           MOVE ACCT-EX TO ACCT-SAVE-WS.
           MOVE ZEROS TO EXCESS-COUNT-WS.
           MOVE SPACES TO PRINT-LINE.
           WRITE PRINT-LINE FROM HEADING1 AFTER ADVANCING 4.
           WRITE PRINT-LINE FROM HEADING2 AFTER ADVANCING 2.
           PERFORM 3210-SEARCH-FOR-EXCESS
                   VARYING CAT-IDX FROM 1 BY 1
                   UNTIL CAT-IDX > 10.
           IF EXCESS-COUNT-WS = 0
              MOVE SPACES TO PRINT-LINE
              WRITE PRINT-LINE FROM MESSAGE-LINE.
           MOVE 0 TO EXCESS-COUNT-WS.
       *
        3210-SEARCH-FOR-EXCESS.
       ************************************************************
       * LINEAR SEARCH OF ONE ACCOUNT BY CATEGORIES:              *
       ************************************************************
           IF  AMT-SPENT(ACCT-IDX, CAT-IDX) > BUDGETED-AMT
                   (ACCT-IDX, CAT-IDX),
               ADD 1 TO EXCESS-COUNT-WS,
               PERFORM 3211-PRINT-EXCESS.
       *
        3211-PRINT-EXCESS.
           MOVE CATEGORY-NAME(CAT-IDX) TO CATEGORY OF EXCESS-WORK.
           MOVE AMT-SPENT(ACCT-IDX, CAT-IDX) TO AMOUNT OF EXCESS-WORK.
           MOVE BUDGETED-AMT(ACCT-IDX, CAT-IDX) TO
               BUDGET-LIMIT OF EXCESS-WORK.
           COMPUTE OVER OF EXCESS-WORK = AMT-SPENT(ACCT-IDX, CAT-IDX)
               - BUDGETED-AMT(ACCT-IDX, CAT-IDX).
           MOVE CORR EXCESS-WORK TO EXCESS-PRINT.
           MOVE SPACES TO PRINT-LINE.
           WRITE PRINT-LINE FROM EXCESS-PRINT.
```

Compile, link, and execute this program, stored as BUDGET.TABLE.COBOL85, with the following dialog. Sample input and output is given below.

```
OK, COBOL85 BUDGET.TABLE -L
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]
[0 ERRORS IN PROGRAM: BUDGET.TABLE.COBOL85]

OK, BIND -LOAD BUDGET.TABLE -LI COBOL85LIB -LI
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
BIND COMPLETE

OK, RESUME BUDGET.TABLE
OK,
```

Input File (BUDGET.DATA):

```
OK, SLIST BUDGET.DATA
BABC010132780300200
BABC020232780346200
BABC030332780020000
BABC040432780000500
BABC050532780200060
BDEF080532780200000
BGHI080332780098300
BGHI090432780090000
EABC031025789998930
EABC031025780000116
EABC051021780000984
EABC021004780000386
EABC031004780008512
EABC031004780004000
EABC011001780030000
EABC041001780001000
EABC030802780008651
EABC030730780000450
EABC030725780008015
EDEF080720780000430
EGHI090615780025600
EGHI080614780003050
OK,
```

Output File (PRINT-FILE):

```
ACCOUNT    ABC              CATEGORIES EXCEEDING BUDGET

CATEGORY         YTDSPENT        BDGTD AMT     AMOUNTOVER
FOOD           100286.74         200.00       100086.74
INSURANCE          10.00           5.00            5.00



ACCOUNT    DEF              CATEGORIES EXCEEDING BUDGET

CATEGORY         YTDSPENT        BDGTD AMT     AMOUNTOVER
NO CATEGORIES HAVE AN AMOUNT THAT EXCEEDS BUDGET



ACCOUNT    GHI              CATEGORIES EXCEEDING BUDGET

CATEGORY         YTDSPENT        BDGTD AMT     AMOUNTOVER
NO CATEGORIES HAVE AN AMOUNT THAT EXCEEDS BUDGET
```

Exception Handling

During the execution of any input or output operation, unusual conditions may arise that preclude normal completion of the operation. COBOL85 communicates these conditions to the object program in the following ways:

- Exception declaratives
- Optional phrases associated with the imperative statement causing the condition
- I-O status

Exception Declaratives

If you specify in the declaratives section a USE procedure for a file, the procedure is executed whenever an input or output condition arises that results in an unsuccessful input-output operation. However, this exception declarative is not executed for the invalid key condition if you specify the INVALID KEY phrase, nor for the AT END condition if you specify the AT END phrase.

Optional Phrases

You can specify the INVALID KEY phrase in the DELETE, READ, REWRITE, START, and WRITE statements. Some of the conditions that give rise to an invalid key condition are

- A requested key does not exist in the file (DELETE, READ, and START statements).
- A key is already in a file, and duplicates are not allowed (WRITE statement).
- A key does not exist in the file, or it is not the last key read (REWRITE statement).

If the invalid key condition occurs during the execution of a statement for which you specify the INVALID KEY phrase, the statement identified by that INVALID KEY phrase is executed.

You can specify the AT END phrase in a READ statement. The AT END condition occurs in a sequentially accessed file when

- No next logical record exists in the file.
- The number of significant digits in the relative record number is larger than the size of the relative key data item.
- An optional file is unavailable.
- A READ statement is attempted, and the AT END condition already exists.

If the AT END condition occurs during the execution of a statement for which you specify the AT END phrase, the statement identified by that AT END phrase is executed.

I-O Status

The I-O status is a two-byte conceptual entity whose value is set by COBOL85 to indicate the status of an input-output operation. When you specify a two-character file status field in the

WORKING-STORAGE SECTION or the LINKAGE SECTION, and you reference that field in the FILE STATUS clause in the *file-control-entry* for a particular file, the file control system moves a value into the file status field following the execution of every OPEN, CLOSE, READ, REWRITE, WRITE, START, or DELETE statement that references that file. This value indicates the execution status of the statement. The following section discusses all COBOL85 file status codes.

File Status Codes

The first digit in each status code indicates the following categories of error:

Digit	*Error Category*
0x	**Successful Completion**: The input-output statement was executed successfully. Additional information may also be provided. Control returns to the statement following the input-output statement or to the imperative statement following any NOT AT END or NOT INVALID KEY phrase.
1x	**At End**: A sequential READ statement was executed unsuccessfully, as a result of an AT END condition. Control returns to the imperative statement following the AT END phrase, if specified. Otherwise, the applicable USE procedure is invoked.
2x	**Invalid Key**: The input-output statement was unsuccessfully executed as a result of an invalid key condition. Control returns to the imperative statement following the INVALID KEY phrase, if you specify one. Otherwise, the applicable USE procedure is invoked.
3x	**Permanent Error**: The input-output statement was unsuccessfully executed as a result of a permanent error condition that precluded further processing of the file. Any specified USE procedures are executed, and the program terminates.
4x	**Logic Error**: The input-output statement was unsuccessfully executed as a result of performing an improper sequence of input-output operations on the file, or as a result of violating a user-defined limit. Any specified USE procedures are executed, and the program terminates.
8x	**Prime Specific (Permanent)**: The input-output statement was unsuccessfully executed as a result of a permanent error condition that is specified by the implementor. Any specified USE procedures are executed, and the program terminates.
9x	**Prime Specific (Recoverable)**: The input-output statement was unsuccessfully executed as a result of a condition that is specified by the implementor. Control returns to the imperative statement following the INVALID KEY phrase, if you specify one. Otherwise, the applicable USE procedure is invoked. If no applicable USE procedure is present, the program terminates.

If a critical error (3x, 4x, 8x) occurs, further I-O operations are not permitted on the file during the execution of the USE procedure. I-O operations on other files may be possible, as long as such operations do not create additional errors.

Caution

Performing I-O operations during declaratives processing is not recommended. The cases listed below can be handled at runtime, but there may be other similar situations that produce unexpected results. Use care to avoid such situations when coding USE procedures.

The following operations cause the program to terminate immediately without completing the USE procedure:

- Any I-O operation on the same file
- Any I-O operation on another file that results in a fatal error
- Any I-O operation on another file that results in a non-fatal error (1x, 2x) and invokes the USE procedure currently being executed

The following sections list all COBOL85 I-O status codes. Deviations from the ANSI standard, whether restrictions or extensions, are documented for the applicable status code.

Unless otherwise noted, all status codes relating to indexed and relative I-O apply to MIDASPLUS and PRISAM files, and all status codes relating to sequential I-O apply to PRISAM, PRIMOS, and magnetic tape files.

Status Code 00

Successful completion for all input-output operations.

Status Code 02 (Indexed)

The input-output statement is successfully executed, but a duplicate is detected.

WRITE or REWRITE: When a WRITE or REWRITE statement creates a duplicate secondary key, status code 02 is returned.

READ Secondary Key of Reference: When the key value for the current key of reference is equal to the value of the same key in the next record within the current key of reference, status code 02 is returned. The last duplicate in the chain returns status code 00.

Note

This status code applies only to indexed files that contain secondary indexes.

Status Code 04 (Sequential, Relative, Indexed)

A READ statement is successfully executed, but the length of the record being processed does not conform to the minimum or to the maximum record sizes specified for the file. This applies to variable-length records only. The buffer area contains undefined positions for short records or the record has been truncated for long records. This is an informational status code only.

Status Code 05 (Sequential, Indexed, Relative)

An OPEN statement is successfully executed, but the referenced OPTIONAL file is not present at the time the OPEN statement is executed.

- If the open mode is I-O or EXTEND, the file is created.
- If the open mode is INPUT, the first sequential READ statement returns status code 10 (logical end of file). The first START or random READ returns status code 23 (not found).

- If the open mode is OUTPUT, status code 05 is not applicable.

Prime Restriction: OPTIONAL files assigned to PRISAM that are unavailable and opened in I-O or EXTEND mode are not created.

Status Code 07 (Sequential)

If an OPEN or CLOSE statement is specified with optional phrases relating to reel/unit media, and the referenced file is on a nonreel/unit medium, status code 07 is returned.

The optional phrases applicable for a CLOSE statement are the NO REWIND and REEL/UNIT phrases. For an OPEN statement, the applicable optional phrases are the NO REWIND and REVERSED phrases.

Note

A CLOSE statement with the REEL/UNIT phrase for a non-reel/unit file causes the file to remain in the open mode.

Status Code 10 (Sequential, Relative, Indexed)

The AT END condition exists because

- A sequential READ statement is attempted, and no next logical record exists in the file because the end of file has been reached.
- A sequential READ statement is attempted for the first time on an optional input file that is not present.

Status Code 14 (Relative)

A sequential READ statement is attempted for a relative file and the number of significant digits in the relative record number is larger than the size of the relative key data item described for the file. Therefore, an apparent logical end-of-file condition exists.

Status Code 21 (Indexed)

A sequence error exists for an indexed file because

- A READ statement is required prior to a REWRITE (see Status Code 43), and the primary key value is changed by the program between the successful execution of a READ and the execution of the next REWRITE statement for that file.
- For a sequentially accessed file, the successive primary record key values are not in ascending order during a WRITE operation.

Status Code 22 (Relative, Indexed)

An attempt is made to

- WRITE a record that would create a duplicate primary key in an indexed or relative file.

- WRITE or REWRITE a record that would create a duplicate alternate record key in an indexed file for which you do not specify the DUPLICATES phrase.

Prime Restriction: For PRISAM files, if you do not specify the DUPLICATES phrase in the program, and the DDL specification allows duplicates, status code 22 is returned, but during sequential access of the file with a secondary key of reference, the current file position is undefined.

Status Code 23 (Relative, Indexed)

An invalid key condition exists because

- An attempt is made to randomly access a record that does not exist in the file.
- A START or random READ statement is attempted on an optional input file that is not present.

Status Code 24 (Relative)

A boundary violation exists because

- An attempt is made to write beyond the externally defined boundaries of a relative file. This boundary is the maximum number of records allocated during CREATK or FAU invocation. For MIDASPLUS files this maximum can be increased by using the DATA function of CREATK.

  Note

  MIDASPLUS allocates space in pages, not in records. Therefore, a boundary violation (status code 24) only occurs when the relative record number times the record size exceeds the allocated space in pages. The space allocated meets or exceeds the amount required for the number of records requested.

- A sequential WRITE statement is attempted for a relative file and the number of significant digits in the relative record number is larger than the size of the relative key data item described for the file. The relative key data item is undefined.

Status Code 30 (Sequential, Relative, Indexed)

A permanent error exists because

- For indexed and relative files, disk full or quota exceeded is encountered during a WRITE operation.
- For all file types, an unrecoverable error is encountered, and no further information is available. This status code applies to all verbs for all file types. Status code 30 reports any fatal error returned from PRIMOS (that is, insufficient access rights, file in use, illegal name, and so on).

Status Code 34 (Sequential)

An attempt is made to write beyond the externally defined boundaries of a sequential file (that is, quota exceeded or disk full).

Status Code 35 (Sequential, Indexed, Relative)

A permanent error exists because an OPEN statement with the INPUT, I-O, or EXTEND phrase is attempted on a nonoptional file that is not present.

Status Code 37 (Sequential, Indexed, Relative)

A permanent error exists because an OPEN statement is attempted on a file that cannot support the open mode specified in the OPEN statement. Although status code 37 is used for runtime error checking, many of the associated problems can be detected during compilation. All the possible violations that reflect file type/mode conflicts are

- The EXTEND phrase is specified, but the file cannot support the write or positioning operations required for the file. Specifically, this violation applies to
  - Files assigned to MT9, TERMINAL, or OFFLINE-PRINT. The compiler generates a fatal diagnostic.
  - Files assigned to PFMS that are redirected to tape ($MT0, and so on) through the use of the –FILE_ASSIGN option. The OPEN statement returns status code 37.
  - A WRITE statement with the ADVANCING clause on a file assigned to PFMS. The WRITE statement returns status code 37.
  - An OPEN statement for an unavailable OPTIONAL file assigned to PRISAM. OPEN returns status code 37. COBOL85 does not create the file.
  - An OPEN statement for an unavailable OPTIONAL file assigned to MIDASPLUS that has nonalphanumeric keys. OPEN returns status code 37. COBOL85 does not create the file.
  - Nonoptional indexed or relative files. The compiler generates a fatal diagnostic.
  - An OPEN statement for an optional indexed or relative file that is present but that contains data. The compiler generates a warning, and the OPEN statement returns status code 37.
- The I-O phrase is specified, but the file cannot support the input and output operations that are permitted for a sequential file when opened in the I-O mode. Specifically, this violation applies to
  - A REWRITE statement for a compressed file. The compiler generates a fatal diagnostic for files assigned to PRINTER or files with the COMPRESSED clause in the *file-description-entry*.

    Also, if the first WRITE statement for a file assigned to PFMS has an ADVANCING clause, the file implicitly acquires the compressed attribute. At runtime, any REWRITE attempts return status code 37. (The ADVANCING clause is allowed only on files assigned to PFMS or PRINTER.)
  - Files assigned to MT9 or OFFLINE-PRINT. The compiler generates a fatal diagnostic.
  - Files assigned to PFMS that are redirected to tape ($MT0, and so on) through the use of the –FILE_ASSIGN option. The OPEN statement returns status code 37.
  - A REWRITE statement for a file assigned to TERMINAL. The REWRITE statement returns status code 37.

  - An OPEN statement for an unavailable OPTIONAL file assigned to PRISAM. OPEN returns status code 37. COBOL85 does not create the file.
  - An OPEN statement for an unavailable OPTIONAL file assigned to MIDASPLUS that has nonalphanumeric keys. OPEN returns status code 37. COBOL85 does not create the file.

- The INPUT phrase is specified, but the file cannot support read operations. Specifically, this violation applies to files assigned to PRINTER and OFFLINE-PRINT. The compiler generates a fatal diagnostic.
- The OUTPUT phrase is specified, but the file cannot support the implicit truncation or creation capabilities required. Specifically, this violation applies to
  - MIDASPLUS indexed and relative files, and PRISAM sequential, indexed, and relative files that contain data at the time of the OPEN statement. Such files cannot be truncated and, therefore, are not opened. Status code 37 is returned.
  - Any file that is assigned to PRISAM and is not present at the time of the OPEN statement. COBOL85 does not create the file. Status code 37 is returned.
  - Any file that is assigned to MIDASPLUS or PFMS has nonalphanumeric key definitions, and is not present at the time of the OPEN statement. COBOL85 does not create the file. Status code 37 is returned.

Status Code 39 (Sequential, Indexed, Relative)

The OPEN statement is unsuccessful because the fixed file attributes conflict with the attributes specified for the file in the program.

Specifically, this status code refers to file attributes such as the organization, minimum and maximum logical record sizes, record type (fixed versus variable), and for tape files, the block size. For indexed files, additional file attributes include the primary record key and alternate record keys.

This status code can also indicate the following invalid attributes:

- Invalid magnetic tape specifications during file assignment (*drive-number, label-type, file-id, volume-id*)
- Invalid assign device during tape file assignment
- RBF file not a valid PRISAM file (DBMS/archived)
- Variable-length records not supported for MIDASPLUS relative files
- Maximum tape block size exceeds 12288 characters
- Maximum tape record size for variable-length records exceeds 9995 characters
- The data management product used to create the file

For magnetic tape files, no attribute checking occurs for unlabeled tape.

For variable-length files, COBOL85 compares the minimum and maximum record sizes in the physical file with the minimum and maximum sizes specified in the RECORD IS VARYING clause of the associated file. These sizes must be equal. If you do not specify the

RECORD IS VARYING clause, COBOL85 uses the sizes of the smallest and largest *record-description-entries* for this comparison. For more information see the section, Specifying Variable-length Records, earlier in this chapter. See also the RECORD clause in Chapter 7.

Status Code 41 (Sequential, Relative, Indexed)

An OPEN statement is attempted for a file already open.

Status Code 42 (Sequential, Relative, Indexed)

A CLOSE statement is attempted for a file that is not open.

Status Code 43 (Sequential, Relative, Indexed)

Sequential: The last input-output statement executed for the file prior to the execution of a REWRITE statement was not a successfully executed READ statement.

Relative: In the sequential access mode, the last input-output statement executed for the file prior to the execution of a DELETE or REWRITE statement was not a successfully executed READ statement.

Indexed: In the sequential access mode, the last input-output statement executed for the file prior to the execution of a DELETE or REWRITE statement was not a successfully executed READ statement. For dynamic or random mode, see the REWRITE statement in Chapter 10 for additional instances when the last input-output statement executed for the file prior to the REWRITE statement must be a successfully executed READ statement.

Status Code 44 (Sequential, Relative, Indexed)

A boundary violation exists because

- An attempt is made to WRITE a record that is larger than the largest or smaller than the smallest record allowed for the associated variable-length file.
- An attempt is made to REWRITE a variable-length record, and the record is not the same size as the record being replaced.

Status Code 46 (Sequential, Relative, Indexed)

A sequential READ statement is attempted on a file open in INPUT or I-O mode, and no valid next record is established because

- The preceding START statement for an indexed or relative file was unsuccessful.
- The preceding READ statement was unsuccessful, but did not cause an AT END condition.
- The preceding READ statement caused an AT END condition.

Status Code 47 (Sequential, Relative, Indexed)

The execution of a READ or START statement is attempted on a file not open in the INPUT or I-O mode.

Status Code 48 (Sequential, Relative, Indexed)

Sequential: The execution of a WRITE statement is attempted on a file not open in OUTPUT or EXTEND mode.

Relative/Indexed: In sequential access mode, the execution of a WRITE statement is attempted on a file not open in OUTPUT or EXTEND mode. In dynamic or random mode, the execution of a WRITE statement is attempted on a file not open in OUTPUT or I-O mode.

Status Code 49 (Sequential, Relative, Indexed)

The execution of a DELETE or REWRITE statement is attempted on a file not open in the I-O mode.

Status Code 82

Prime Extension: Specifies a fatal tape error when end of tape (EOT) occurs on unlabeled tape during a READ or WRITE operation.

Status Code 90

A lock is requested on a MIDASPLUS record that is already locked. (See MIDASPLUS error code 10.)

Status Code 93

A FORMS validation error on a READ statement.

Status Code 94

Concurrency error. Another user may have deleted an active MIDASPLUS record. (See MIDASPLUS error code 13.)

Status Code 97

Concurrency error during a PRISAM operation. (See PRISAM error codes ER$ABT, ER$TIM, and ER$TAB.)

Status Code 98

An input-output operation is unsuccessful due to an error that may be recoverable, such as a badspot on a magnetic tape file.

Status Code 99

MIDASPLUS or PRISAM unexpected system error. This is a critical error. The program terminates.

5 The IDENTIFICATION DIVISION

This chapter discusses the first of the four major divisions of the COBOL85 program, the IDENTIFICATION DIVISION. The chapter discusses the format of the IDENTIFICATION DIVISION and concludes with an example.

IDENTIFICATION DIVISION

You must include the IDENTIFICATION DIVISION as the first division in the COBOL85 program. This division identifies the program. You can include additional user information, such as the date the program was written or the program author, in the appropriate paragraph in the format shown below. You can precede the IDENTIFICATION DIVISION by comment lines denoted by either an asterisk or a slash in column seven.

Format

```
{IDENTIFICATION DIVISION.}
{ID DIVISION.            }

PROGRAM-ID. program-name.

[AUTHOR. [comment-entry] . . .]

[INSTALLATION. [comment-entry] . . .]

[DATE-WRITTEN. [comment-entry] . . .]

[DATE-COMPILED. [comment-entry] . . .]

[SECURITY. [comment-entry] . . .]

[REMARKS. [comment-entry] . . .]
```

Syntax Rules

1. The IDENTIFICATION DIVISION must begin with the reserved words IDENTIFICATION DIVISION or ID DIVISION followed by a period and a space.

   Prime Extension: You can use ID instead of IDENTIFICATION.

2. The PROGRAM-ID paragraph is required and must immediately follow the division header. The *program-name* is the name of the entry-point or object module, and is the name by which you reference this program in a CALL statement. If you omit the *program-name*, the compiler uses MAIN by default.

3. The *program-name* follows the general rules for word formation listed in Chapter 4. It may be any alphanumeric string.

   Note

   If you use SEG as the loader, SEG retains only the first 8 characters of the *program-name*. Because they define the entry point name, these characters must be unique in any one runfile.

4. All remaining paragraphs are optional. A paragraph-header (a reserved word) identifies the type of information contained in each paragraph.

   Prime Extension: Optional paragraphs can appear in any order.

5. A *comment-entry* can be any combination of Prime characters. The continuation of a *comment-entry* by a hyphen in column 7 is not permitted; however, the *comment-entry* can appear on one or more lines. Limit to Area B any comment lines after the header.

6. DATE-COMPILED writes the date and time of compilation to the listing file on the same line. For example,

   ```
   DATE-COMPILED.  870814.14:04:28.
   ```

7. You can use PROGRAM-ID and DATE-COMPILED in the PROCEDURE DIVISION. For example,

   ```
   DISPLAY PROGRAM-ID.

   MOVE DATE-COMPILED TO PIC-X-15.
   ```

   In all references to PROGRAM-ID in the PROCEDURE DIVISION, the compiler substitutes the program name. In all references to DATE-COMPILED in the PROCEDURE DIVISION, the compiler substitutes the compilation date and time. The compiler treats both substituted items as nonnumeric literals. DATE-COMPILED is a 15-character field. Its value is in the format

   YYMMDD.HH:MM:SS

8. **Prime Extension:** The REMARKS paragraph is a Prime extension.

IDENTIFICATION DIVISION Example

This example forms one program with the examples at the end of Chapters 6, 7, and 8.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.      DISBURSE.
 AUTHOR.          MATT.
 INSTALLATION.    PRIME.
 DATE-WRITTEN.    AUGUST, 14, 1988.
 DATE-COMPILED.

*****************************************************************
 REMARKS.  THIS PROGRAM PRODUCES A MONTHLY CASH DISBURSEMENTS
     JOURNAL:  A PRINTED DETAIL LIST AND TOTALS BY DEPARTMENTS
     WITH GRAND-TOTAL (CROSS-TOTAL) BALANCED AGAINST A JOB TOTAL.
*
     USE DETAIL-LINES WITH
         COL. 1-3  CHECK NO.,
         COL. 4-9  MMDDYY,
         COL. 13-32 VENDOR,
         COL. 33-35 DEPT. OR ACCT. NO.,
         COL. 36-42 AMOUNT.
     TO WRITE TAPE RECORD, ENTER YES  FOR TAPE REQUEST.
*
     THE PROGRAM CHECKS FOR INPUT ERRORS OF INVALID ACCOUNT
     NUMBER, INVALID DATE, INVALID NUMERIC FIELDS.  IT DOES NOT
     CHECK FOR SEQUENCE ERRORS IN DATE OR CHECK NUMBER, OR FOR
     DUPLICATE ENTRIES.  THE PROGRAM ASSUMES AN UNSORTED DATA FILE.
*****************************************************************
```

6 The ENVIRONMENT DIVISION

This chapter discusses the second of the four major divisions of the COBOL85 program, the ENVIRONMENT DIVISION. The chapter describes the CONFIGURATION SECTION and its paragraphs: SOURCE-COMPUTER, OBJECT-COMPUTER, and SPECIAL-NAMES. It also describes the INPUT-OUTPUT SECTION and its paragraphs: FILE-CONTROL and I-O-CONTROL. The chapter concludes with an example of the ENVIRONMENT DIVISION.

ENVIRONMENT DIVISION

The ENVIRONMENT DIVISION defines those aspects of a program that depend on hardware considerations. This division is optional.

Format

```
ENVIRONMENT DIVISION.

[ CONFIGURATION SECTION.

  [SOURCE-COMPUTER. [computer-name.]]

  [ OBJECT-COMPUTER. [computer-name]
    [object-computer-entry]. ]

  [ SPECIAL-NAMES.
    [special-names-entry] . . . ]

]

[ INPUT-OUTPUT SECTION.

  [ FILE-CONTROL.
    {file-control-entry} . . . ]

  [ I-O-CONTROL.
    [I-O-control-entry] ]

]
```

Syntax Rules

1. The ENVIRONMENT DIVISION must begin with the header ENVIRONMENT DIVISION, followed by a period and a space.
2. The format above indicates the mandatory sequence of required and optional paragraphs in the ENVIRONMENT DIVISION.

   Prime Extension: The clauses in the I-O-CONTROL paragraph can appear in any order.
3. The CONFIGURATION SECTION and the INPUT-OUTPUT SECTION are optional.

General Rules

1. Each section within the ENVIRONMENT DIVISION begins with its *section-name*, followed by the word SECTION, and each paragraph within each section begins with its *paragraph-header*.
2. Use the ENVIRONMENT DIVISION to document hardware-dependent features of a program.
3. *computer-name* serves only as documentation. Use it to identify the computer on which the COBOL85 program is compiled. You can use *computer-name* as a programmer-defined word elsewhere in the program.

CONFIGURATION SECTION

The CONFIGURATION SECTION is the first of two optional sections in the ENVIRONMENT DIVISION. It contains three optional paragraphs: SOURCE-COMPUTER, OBJECT-COMPUTER, and SPECIAL-NAMES.

SOURCE-COMPUTER

Format

SOURCE-COMPUTER. [*computer-name* [WITH DEBUGGING MODE].]

General Rules

1. *computer-name* serves only as documentation. It identifies the computer on which the COBOL85 program is compiled. You can use *computer-name* as a programmer-defined word elsewhere in the program.
2. If you specify the WITH DEBUGGING MODE clause, COBOL85 compiles all debugging lines as it compiles all other source lines in the program.
3. If you do not specify the WITH DEBUGGING MODE clause, COBOL85 compiles all debugging lines as if they were comment lines.

OBJECT-COMPUTER

Format

OBJECT-COMPUTER. [*computer-name*]

[, MEMORY SIZE *integer* { WORDS / CHARACTERS / MODULES }]

[, PROGRAM COLLATING SEQUENCE IS *alphabet-name-1*]

[, SEGMENT-LIMIT IS *segment-number*].

General Rules

1. *computer-name* serves only as documentation. Use it to identify the computer on which the COBOL85 program is executed. You can use *computer-name* as a programmer-defined word elsewhere in the program.
2. The MEMORY SIZE clause serves only as documentation. *alphabet-name-1* is a programmer-defined word, which you define in the SPECIAL-NAMES paragraph as NATIVE, EBCDIC, STANDARD-1, or STANDARD-2.
3. If you do not specify the PROGRAM COLLATING SEQUENCE IS clause, the collating sequence defaults to NATIVE, which is the ASCII collating sequence defined in Table B-3.
4. When you specify the *alphabet-name* of the PROGRAM COLLATING SEQUENCE clause as EBCDIC, nonnumeric comparisons in conditional expressions are made with respect to the EBCDIC collating sequence defined in Table B-6.
5. When you specify the *alphabet-name* of the PROGRAM COLLATING SEQUENCE clause as STANDARD-1, nonnumeric comparisons in conditional expressions are made with respect to the standard ASCII collating sequence as defined by ANSI X3.4-1977. Table B-4 describes this collating sequence.
6. When you specify the *alphabet-name* of the PROGRAM COLLATING SEQUENCE clause as STANDARD-2, nonnumeric comparisons in conditional expressions are made with respect to the International Reference Version of the ISO 7-bit code defined in International Standard 646. Table B-5 describes this collating sequence.
7. If there is no correspondence between the characters of a specified *alphabet-name* and the native character set, the collating sequence of those characters is undefined.
8. Chapter 14 explains the use of the PROGRAM COLLATING SEQUENCE clause as it relates to SORT and MERGE statements.
9. The SEGMENT-LIMIT clause serves only as documentation.

Example

In the following example, the comparison and the first sort are performed with respect to the EBCDIC collating sequence. The second sort is performed with respect to the ASCII collating sequence described in Table B-3.

```
OBJECT-COMPUTER.
   PROGRAM COLLATING SEQUENCE IS ALPHABET1.
SPECIAL-NAMES.
   ALPHABET1 IS EBCDIC,
   ALPHABET2 IS NATIVE.
             .
             .
             .
PROCEDURE DIVISION.
             .
             .
             .
   IF NAME < 'SMITH' PERFORM INSERT-PARA.
             .
             .
             .
   SORT FILE-1 ON ASCENDING KEY NAME
        INPUT  PROCEDURE IS SORT-INPUT
        OUTPUT PROCEDURE IS SORT-OUTPUT.
             .
             .
             .
   SORT FILE-2 ON ASCENDING KEY NAME
        COLLATING SEQUENCE IS ALPHABET2
        INPUT  PROCEDURE IS SORT-INPUT2
        OUTPUT PROCEDURE IS SORT-OUTPUT2.
```

SPECIAL-NAMES

This paragraph is required only if you use one or more of its statements.

Format

```
SPECIAL-NAMES.

[CONSOLE IS mnemonic-name-1] . . .

[switch-name [IS mnemonic-name-2]
  [{ON STATUS IS condition-name-1 [OFF STATUS IS condition-name-2] }]]
  [{OFF STATUS IS condition-name-2 [ON STATUS IS condition-name-1] }]] . . .

                                  {STANDARD-1}
[ALPHABET alphabet-name-1 IS     {STANDARD-2}] . . .
                                  {NATIVE    }
                                  {EBCDIC    }

[CLASS class-name-1 IS {literal-1 [{THROUGH} literal-2]} . . .] . . .
                                   {THRU   }

[CURRENCY SIGN IS literal-3]

[DECIMAL-POINT IS COMMA].
```

Syntax Rules

1. Only the ACCEPT and DISPLAY statements can reference *mnemonic-name-1.*
2. Only a Format 3 SET statement can reference *mnemonic-name-2;* hence, the ON STATUS and OFF STATUS phrases are optional.
3. The eight *switch-names* are

   CBLSW0
   CBLSW1
   CBLSW2
   CBLSW3
   CBLSW4
   CBLSW5
   CBLSW6
   CBLSW7

4. The words THRU and THROUGH are equivalent.

5. The literals specified in the *literal-1* THRU *literal-2* phrase,

   - If numeric, must be unsigned integers and must have a value within the range of 0 through 255
   - If nonnumeric, must each be one character in length

General Rules

1. *mnemonic-name-1* is a programmer-defined word that is associated with CONSOLE throughout the program. The following example uses TTY as *mnemonic-name-1*. The coding causes the field YEAR OF HIRE-DATE to be displayed on the console.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
            .
            .
            .
SPECIAL-NAMES.  CONSOLE IS TTY.
            .
            .
            .
PROCEDURE DIVISION.
            .
            .
            .
    DISPLAY YEAR OF HIRE-DATE UPON TTY.
```

2. *mnemonic-name-2* is a programmer-defined word that is associated with the external switch to which it refers throughout the program. It can be set by responding to the automatic runtime prompt (see Chapter 3) or by using the SET statement within the program (see Chapter 8). Switches allow for changes in the application environment each time a program is run. For example, a switch can determine whether or not to add month-end processing. *switch-mnemonic-names* or *switch-status-conditions* are programmer-defined names. You can specify ON and OFF status by associating *condition-names* with each switch used in a program. You can determine the status of a switch by testing an associated *condition-name*, as in the following example.

   condition-names can be qualified by *mnemonic-names*, as SWITCH-ON OF SWITCH-ONE in this example.

```
SPECIAL-NAMES.
    CBLSW0 IS TAPE-SWITCH,
              ON STATUS  IS TAPE-SWITCH-ON,
              OFF STATUS IS TAPE-SWITCH-OFF,
    CBLSW1 IS SWITCH-ONE,
              ON STATUS  IS SWITCH-ON,
              OFF STATUS IS SWITCH-OFF,
    CBLSW2 IS SWITCH-TWO,
              ON STATUS  IS SWITCH-ON,
              OFF STATUS IS SWITCH-OFF,
```

```
      .
      .
      .
TEST SECTION.
  ONLY-PARAGRAPH.
    IF TAPE-SWITCH-OFF DISPLAY 'TAPE CANNOT BE PROCESSED'.
    IF SWITCH-ON OF SWITCH-ONE DISPLAY 'NO PRINT-OUT'.
```

3. Use the ALPHABET clause to relate a programmer-defined name to a specified character code set and/or collating sequence.
4. You can use *alphabet-name-1* as the object of the PROGRAM COLLATING SEQUENCE clause in the OBJECT-COMPUTER paragraph, as the object of the COLLATING SEQUENCE clause of SORT and MERGE statements, and as the object of the CODE-SET clause for magnetic tape files.
5. Use the CLASS clause to relate a name to a set of specific characters. You can reference *class-name-1* only in a class condition. *class-name-1* consists of the exclusive set of characters defined by the values of the literals in the CLASS clause. The value of each literal specifies

   - If numeric, the ordinal number of a character within the native character set. This value must not exceed 255.
   - If nonnumeric, the actual character within the native character set. If the value of the nonnumeric literal contains multiple characters, each character in the literal is included in the set of characters identified by *class-name-1*.

   For example,

```
CLASS EXTENDED-ALPHA IS
      "A" THRU "Z",
      "a" THRU "z",
      " "  ","  ";"  "'" "-".
.
.
.
01  CUST-NAME  PIC X(50) VALUE "O'BRIEN, JEAN-CLAUDE".
.
.
.
IF CUST-NAME IS EXTENDED-ALPHA
.
.
.
```

6. If you specify the THROUGH phrase, the contiguous characters in the native character set beginning with the character specified by the value of *literal-1*, and ending with the character specified by the value of *literal-2*, are included in the set of characters identified by *class-name-1*. In addition, the contiguous characters specified by a given THROUGH phrase can specify characters of the native character set in either ascending or descending sequence.

7. *literal-3* represents the currency sign to be used in the PICTURE clause. It is a single-character, nonnumeric literal that replaces the dollar sign as the currency sign. The designated character must not be a single or double quotation mark, or any of the characters defined for PICTURE representations.
8. The DECIMAL-POINT IS COMMA clause exchanges the functions of comma and period in the *character-string* of PICTURE clauses and in numeric literals.
9. You can use the ten *implementor-names* CONSOLE, CBLSW0 through CBLSW7, and EBCDIC as programmer-defined words elsewhere in the program.

INPUT-OUTPUT SECTION

The INPUT-OUTPUT SECTION is the second of two optional sections in the ENVIRONMENT DIVISION. Use this section when the program processes data files. Use it to specify peripheral devices and information needed to transmit and handle data between the devices and the program. The section has two optional paragraphs: FILE-CONTROL and I-O-CONTROL.

FILE-CONTROL

Each file requires one *file-control-entry*. The format you use depends on file organization.

Format 1

```
SELECT [OPTIONAL] file-name-1
  ASSIGN TO { device-name }
            { literal-1   }

[ RESERVE integer-1 [ AREA  ] ]
                    [ AREAS ]

[ [ORGANIZATION IS] SEQUENTIAL]

[ ACCESS MODE IS SEQUENTIAL]

[ FILE STATUS IS data-name-1].
```

Format 2

```
SELECT [OPTIONAL] file-name-1
  ASSIGN TO { device-name }
            { literal-1   }

[ RESERVE integer-1 [ AREA  ] ]
                    [ AREAS ]

[ORGANIZATION IS] RELATIVE

[ ACCESS MODE IS { SEQUENTIAL [, RELATIVE KEY IS data-name-1] }
                 { { RANDOM  }                                }
                 { { DYNAMIC } , RELATIVE KEY IS data-name-1  } ]

[ FILE STATUS IS data-name-2].
```

Format 3

```
SELECT [OPTIONAL] file-name-1
  ASSIGN TO { device-name }
            { literal-1   }

[ RESERVE integer-1 [ AREA  ] ]
                    [ AREAS ]

[ORGANIZATION IS] INDEXED

[ ACCESS MODE IS { SEQUENTIAL } ]
                 { RANDOM     }
                 { DYNAMIC    }

RECORD KEY IS data-name-1

[ ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLICATES]] . . .

[ FILE STATUS IS data-name-3].
```

General Rules

1. *file-name* is a programmer-defined name described in the DATA DIVISION. Each file specified here must have a *file-description-entry* in the DATA DIVISION. The ASSIGN clause associates the *file-name* with a storage medium or input-output hardware. The *device-names* are COBOL85 *implementor-names* that you can use as programmer-defined names elsewhere in the program. Table 6-1 lists allowable *device-names*.

TABLE 6-1
Device Specifications

device-name	*Hardware Device*
PRIMOS	Disk storage (sequential file)
PRISAM	Disk storage (sequential, indexed, or relative file)
MIDASPLUS	Disk storage (indexed or relative file)
PRINTER	System printer (goes to disk, can be spooled)
MT9	9-track magnetic tape drive
PFMS (obsolete)	Disk storage (Prime File Management System); at runtime, is one of the above devices
TERMINAL	CRT terminal or TTY terminal
OFFLINE-PRINT	FORMS (PRINTER) interface

For example,

```
SELECT SCREEN-FILE ASSIGN TO TERMINAL.
SELECT DISK-SEQUENTIAL-FILE ASSIGN TO PRIMOS.
SELECT TAPE-FILE1 ASSIGN TO MT9.
SELECT INDEXED-FILE ASSIGN TO PRISAM.
SELECT RELATIVE-FILE ASSIGN TO MIDASPLUS.
```

literal-1 must contain one of the allowable *device-names*.

ASSIGN TO PRIMOS designates a PRIMOS disk sequential file. It can be fixed-length or variable-length; compressed or uncompressed. Its organization must be SEQUENTIAL.

ASSIGN TO PRISAM designates a PRISAM managed file. Its organization can be SEQUENTIAL, INDEXED, or RELATIVE.

ASSIGN TO MIDASPLUS designates a MIDASPLUS managed file. Its organization can be INDEXED or RELATIVE.

ASSIGN TO PFMS designates a PRIMOS, PRISAM, or MIDASPLUS file. You can also use PFMS for tape files if runtime file assignment is to be made to a magtape device. You can also use it to indicate a PRINTER file if the first WRITE statement contains an ADVANCING clause.

Note

PFMS is included for compatibility with CBL. To achieve optimum compile-time and runtime performance from your COBOL85 program, specify the file management system (PRIMOS, PRISAM, MIDASPLUS, and so on) to be used.

2. The OPTIONAL phrase applies only to files opened in the INPUT, I-O, or EXTEND mode. The OPTIONAL phrase does not apply to files opened in OUTPUT mode. You can specify it for files that are not always present each time the program is executed.

   If an optional file is unavailable at runtime, the successful execution of an OPEN statement with an EXTEND or I-O phrase creates the file. This creation takes place as if the following statements were executed in the order shown:

   OPEN OUTPUT *file-name.*
   CLOSE *file-name.*

   After these statements, the OPEN statement specified in the source program is executed.

   If an optional file is unavailable at runtime and is opened in INPUT mode, the first READ statement to the file returns a status code indicating `END OF FILE` or `RECORD NOT FOUND`.

   Table 6-2 lists the file types and OPEN modes currently supported.

TABLE 6-2
OPTIONAL File Types and OPEN Modes

	OPEN Mode		
File Type	*Input*	*I-O*	*Extend*
PRIMOS sequential disk	Y	Y	Y
PRISAM sequential	Y	N	N
MIDASPLUS indexed/relative	Y	Y	Y
PRISAM indexed/relative	Y	N	N
Labeled magtape	Y	N	N
Unlabeled magtape	N	N	N

   In all OPEN modes, OPTIONAL is ignored for files that you assign to TERMINAL and OFFLINE-PRINT.

   OPTIONAL is not supported for PRISAM files (SEQUENTIAL, RELATIVE, and INDEXED) for EXTEND and I-O modes. If the file is not available at runtime, an error condition exists.

3. The RESERVE clause is for documentation only. Whether or not you use it, the compiler assigns buffer areas necessary for processing.
4. The ORGANIZATION clause specifies the logical structure of a file. When you omit the clause, the default is SEQUENTIAL.
5. Use the ACCESS MODE clause to describe the sequence in which records are accessed. When you omit this clause, the default is SEQUENTIAL.
6. Chapter 11 discusses the RELATIVE KEY clause.
7. Chapter 10 discusses the RECORD KEY and ALTERNATE RECORD KEY clauses.
8. Use the FILE STATUS clause to specify a two-character field, a *data-name* described in the WORKING-STORAGE SECTION or the LINKAGE SECTION, as the file status field. The *data-name* can be qualified.

Prime Extension: The file status field can be either an alphanumeric or an unsigned numeric display field.

When you specify the FILE STATUS clause in the FILE-CONTROL paragraph, COBOL85 file control moves a value into the file status field after the execution of every statement that refers to that file. Thus, the FILE STATUS data item is updated during the execution of the OPEN, CLOSE, READ, WRITE, REWRITE, DELETE, and START statements. The value in the file status field indicates to the COBOL85 program the status of execution of the statement.

Chapter 4 includes a complete discussion of COBOL85 file status codes.

I-O-CONTROL

Format

```
I-O-CONTROL.

[ SAME [ RECORD ]                                                    ]...
       [ SORT   ] AREA FOR file-name-1, {file-name-2} ... ]...
       [ SORT-MERGE ]                                                ]

[ RERUN [ON file-name-1]                                             ]
[                                                                    ]
[ EVERY { { END OF { REEL } } OF file-name-3                          }... ]
[       {   {      { UNIT } }                                        }    ]
[       {   integer-4 RECORDS                                        }    ]
[       {   integer-5 CLOCK-UNITS                                    }    ]
[       {   condition-name                                           }    ]

[ MULTIPLE FILE TAPE CONTAINS file-name-4
      [POSITION integer-6 ]
      [ file-name-5 [POSITION integer-7 ]] ... ] ...
```

Syntax Rule

The I-O-CONTROL paragraph is optional.

General Rules

1. The compiler treats SAME AREA as SAME RECORD AREA. Use the SAME AREA or SAME RECORD AREA clause to share the same memory areas for files that are not sort or merge files. This feature saves memory space and eliminates MOVEs from one record to another, thus saving execution time. Do not list a file in more than one SAME AREA or SAME RECORD AREA clause.

   The sample program in Chapter 10 contains an example.

2. The SAME AREA or SAME RECORD AREA clause specifies that two or more files are to use the same memory area for processing the current logical record. All the files can be open at the same time. COBOL85 considers a logical record in the SAME

RECORD AREA both as a logical record of each opened output file whose *file-name* appears in this SAME RECORD AREA clause, and as a record of the most recently read input file whose *file-name* appears in this SAME RECORD AREA clause. This is equivalent to an implicit redefinition of the area; that is, records are aligned on the leftmost character position.

3. If one or more *file-names* of a SAME AREA clause appear in a SAME RECORD AREA clause, all the *file-names* in the first clause must appear in the second clause. However, additional *file-names* appearing in the SAME RECORD AREA clause need not appear in the SAME AREA clause.
4. The files referenced in the SAME AREA or SAME RECORD AREA clause need not all have the same organization or access.
5. The RERUN clause is checked for syntax only.
6. Chapter 12 discusses the MULTIPLE-FILE clause.

ENVIRONMENT DIVISION Example

This example forms one program with the examples at the end of Chapters 5, 7, and 8.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  PRIME.
OBJECT-COMPUTER.  PRIME.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TAPE-FILE, ASSIGN TO MT9.
    SELECT PRINT-FILE, ASSIGN TO PRINTER.
    SELECT DISK-FILE, ASSIGN TO PRIMOS.
*****************************************************************
```

7 The DATA DIVISION

This chapter discusses the third of the four major divisions of the COBOL85 program, the DATA DIVISION. The chapter describes the three optional sections of the DATA DIVISION: the FILE SECTION, the WORKING-STORAGE SECTION, and the LINKAGE SECTION. It also describes the clauses of the *file-description-entry* and the clauses of the *record-description-entry*. The chapter concludes with an example of the DATA DIVISION.

DATA DIVISION

The DATA DIVISION of the COBOL85 source program defines the nature and characteristics of the data that the program processes. Data to be processed falls into three categories:

- Data in files, which enter or leave the internal memory of the computer from or to a specified storage area or areas
- Data developed internally and placed into intermediate or working storage
- Data passed to the program from a calling program

The DATA DIVISION is optional and consists of three optional sections. If you use the sections, they must appear in the following order:

1. FILE SECTION. This section describes files and records in files.
2. WORKING-STORAGE SECTION. This section defines memory space for the storage of items that are not part of external data files but are intermediate processing results.
3. LINKAGE SECTION. This section, in a called program, describes data available to both the called program and the calling program.

Format

```
DATA DIVISION.

[ FILE SECTION.
  [ file-description-entry.
    [record-description-entry] ... ] ...
  [ sort-merge-file-description-entry.
    {record-description-entry} ... ] ... ]

[ WORKING-STORAGE SECTION.
  [ level-77-data-description-entry
    data-description-entry ] ... ]

[ LINKAGE SECTION.
  [ level-77-data-description-entry
    data-description-entry ] ... ]
```

Syntax Rules

1. If you include the DATA DIVISION, begin it with the header DATA DIVISION, followed by a period and a space.
2. If you include the optional sections of the DATA DIVISION, they must appear in the order shown above.
3. **Prime Extensions:** *file-description-entries* and *sort-merge-file-description-entries* can appear in any order. In WORKING-STORAGE, level-01 and level-77 items can appear in any order.

General Rules

1. Each section within the DATA DIVISION begins with its *section-name*, followed by a period and a space.
2. In WORKING-STORAGE, a *data-description-entry* uses the same format as a *record-description-entry*.

FILE SECTION

The FILE SECTION is the first of three optional sections in the DATA DIVISION. Use this section to define the structure of data files. Define each file with a *file-description-entry* (FD) or a *sort-merge-file-description-entry* (SD), and with one or more associated *record-description-entries*.

Format

FILE SECTION.

```
[file-description-entry. {record-description-entry} ...                ]
[sort-merge-file-description-entry. {record-description-entry} ...     ] ...
```

Syntax Rules

1. If you use the FILE SECTION, begin it with the header FILE SECTION, followed by a period and a space.
2. The FILE SECTION contains FD and SD entries. Follow each entry immediately with one or more associated *record-description-entries*. The number of FD and SD entries in the FILE SECTION is unlimited. The number of files that can be open at once is limited by PRIMOS and is listed in Appendix I.

General Rule

Use a SELECT statement in the ENVIRONMENT DIVISION to associate each FD or SD entry with an I-O device.

Note

This chapter describes the format and the clauses required in an FD entry for nonsort files. Chapter 14 discusses the SD entry for sort-merge files.

WORKING-STORAGE SECTION

The WORKING-STORAGE SECTION of the DATA DIVISION describes **noncontiguous** data (level-77 data with no hierarchical relationship) and records that are not part of external files, but are developed and processed internally. You can use the VALUE clause to assign initial values to data items in this section.

Format

WORKING-STORAGE SECTION.

```
[level-77-description-entry ]
[record-description-entry   ] ...
```

Syntax Rules

1. The WORKING-STORAGE SECTION is optional. If you include it, begin it with the words WORKING-STORAGE SECTION, followed by a period and a space.
2. Noncontiguous item names and record names in the WORKING-STORAGE SECTION must be unique; they cannot be qualified. Subordinate *data-names* need not be unique if you can make them unique by qualification, or if you never reference them.

3. Apply the *level-number* 77 to noncontiguous elementary data items. Define each noncontiguous item in a separate *data-description-entry*. The following data clauses are required in each noncontiguous *data-description-entry*:

   - *level-number* 77
   - *data-name*
   - Either the PICTURE clause or the USAGE IS INDEX, BINARY, COMP, COMP-1, or COMP-2 clause

   Other data description clauses are optional. If necessary, use them to complete the description of the item.

4. Group data items in the WORKING-STORAGE SECTION that have a hierarchical relationship according to the rules for formation of record descriptions. Any clause that you can use in a record description in the FILE SECTION can be used in a data description in the WORKING-STORAGE SECTION. (See the section, record-description-entry, later in this chapter, for more information.)

General Rules

1. WORKING-STORAGE items described in this chapter include the following:

   - Noncontiguous elementary items having the *level-number* 77. These items have no hierarchical relationship. You cannot group them into records or further subdivide them.
   - Data items in records not associated with an input-output device and not part of external data files, but developed and processed internally. These items have *level-numbers* 01 through 49.

2. VALUE clauses, prohibited in the FILE SECTION, are permitted throughout WORKING-STORAGE to specify the initial value of an item, except for an index data item.

Note

Use the VALUE clause or PROCEDURE DIVISION statements to initialize all WORKING-STORAGE data items before using them. Unexpected values may appear in uninitialized data items.

LINKAGE SECTION

The LINKAGE SECTION describes data previously defined in a calling program that is available to a called program.

Format

LINKAGE SECTION.

[*level-77-description-entry*]
[*record-description-entry*] . . .

Syntax Rules

1. The LINKAGE SECTION is optional. It is meaningful only in a called program. If you include a LINKAGE SECTION, begin it with the words LINKAGE SECTION, followed by a period and a space.
2. Each LINKAGE SECTION *record-name* and noncontiguous item name must be unique within the called program; the names cannot be qualified.
3. Apply *level-number* 77 to noncontiguous elementary data items. Define each *level-number* 77 data item in a separate *data-description-entry*. The following data clauses are required in each noncontiguous *data-description-entry*:

   - *level-number* 77
   - *data-name*
   - The PICTURE clause or the USAGE IS INDEX, BINARY, COMP, COMP-1, or COMP-2 clause

   Other data description clauses are optional. If necessary, use them to complete the description of the item. However, the EXTERNAL clause has no meaning in the LINKAGE SECTION.
4. Group data items in the LINKAGE SECTION that have a hierarchical relationship according to the rules for formation of record descriptions. (See the section, record-description-entry, later in this chapter, for more information.)
5. In the LINKAGE SECTION, only items that have a *level-number* 88 can have initial values.

General Rules

1. The LINKAGE SECTION of the DATA DIVISION is meaningful only if the program containing it is called by another program whose CALL statement contains a USING phrase.
2. Use the LINKAGE SECTION to describe data that is available through the calling program, and that is referred to in both the calling program and the called program. COBOL85 does not allocate space in the called program for data items defined in the LINKAGE SECTION of that program. Instead, PROCEDURE DIVISION references to these data items are resolved at runtime by equating the reference in the called program to the location used in the calling program.
3. Data items that you define in the LINKAGE SECTION of the called program can be used within the PROCEDURE DIVISION of the called program only if you specify them as operands of the USING phrase in the PROCEDURE DIVISION header, or subordinate to such operands, and the object program is under the control of a CALL statement that specifies a USING phrase.

Note

Chapter 13 includes a LINKAGE SECTION example.

file-description-entry

The file description provides information concerning the physical structure, identification, and record names of a nonsort file. Write *file-description-entries* only in the FILE SECTION of the DATA DIVISION. The sections that follow this section describe the clauses that you can specify in a *file-description-entry*.

Format

```
FD file-name [ { COMPRESSED   } ]
             { UNCOMPRESSED }

[ ; IS EXTERNAL]

[ ; BLOCK CONTAINS [integer-1 TO] integer-2 { RECORDS    } ]
                                            { CHARACTERS }

[ ; CODE-SET IS alphabet-name]

[ ; DATA { RECORD IS  } data-name-1 [, data-name-2] ... ]
         { RECORDS ARE }

[ ; LABEL { RECORD IS  } { STANDARD } ]
          { RECORDS ARE } { OMITTED  }

[ ; RECORD { CONTAINS integer-3 CHARACTERS                                    }
           { IS VARYING IN SIZE [[FROM integer-4] [TO integer-5] CHARACTERS] }
           { CONTAINS integer-6 TO integer-7 CHARACTERS                      }
           { IS NOT VARYING IN SIZE                                          } ]

[ ; RECORDING MODE IS {F, U, S, V}]

[ ; VALUE OF FILE-ID IS { data-name-3 } ... ]
                        { literal-1   }

[record-description-entry] ...
```

Syntax Rules

1. The level indicator FD identifies the beginning of a file description and must precede the *file-name*.
2. The *file-name* follows the general rules for word formation listed in Chapter 4.

3. Use the COMPRESSED/UNCOMPRESSED clause only with sequential files. The default is UNCOMPRESSED.
4. The *file-description-entry* is a sequence of clauses that you must terminate with a period. The number of *file-description-entries* allowed in the FILE SECTION is unlimited; the number of files that can be open at one time is limited by PRIMOS and is listed in Appendix I.
5. If you use the DATA RECORD clause, follow the *file-description-entry* with one or more *record-description-entries*. See the section, record-description-entry, later in this chapter, for more information.
6. All clauses that follow *file-name* are optional.

The following sections describe the clauses that you can specify in a *file-description-entry*.

COMPRESSED/UNCOMPRESSED — Prime Extension

The UNCOMPRESSED clause enables a disk READ or WRITE based on record length, while the COMPRESSED clause enables a READ or WRITE by compression control characters.

Compression is the elimination of multiple blank characters. File compression is effected by replacing any string of three or more blank characters with a control character plus a count. When a record is written with compression control, the first space character in such a string is replaced by the ASCII control character DC1, and the second space is replaced by a binary count (3 through 255) of spaces in the string. The 3rd through 255th spaces are then deleted. When the same record is read with compression control, each combination of DC1 plus number is replaced by that number of spaces before the record is made available to the program.

WARNING

If your program reads a compressed file as uncompressed, a premature end of file results, and data is transferred to seemingly inappropriate fields.

Format

FD *file-name* [{ COMPRESSED / UNCOMPRESSED }]

General Rules

1. If you specify neither clause, the default is UNCOMPRESSED.
2. You must use the UNCOMPRESSED clause when your program reads sequential I-O files containing nondisplay numeric data, such as packed or binary data.

3. You must use the COMPRESSED clause to read sequential disk files in compressed format (for example, files that you create using ED or EMACS, or files written in compressed format by other programs).

EXTERNAL

The EXTERNAL clause in a *file-description-entry* specifies that the file is external. Therefore, the file's data items are available to every program in the run unit that describes that file. An external data item is often referred to as a **common block**. When one COBOL85 program calls another, defining a file used by both programs as EXTERNAL saves storage space. The called program need not define the file in its linkage area. The file need not be closed and reopened to be accessed by different programs in the run unit. If a calling program opens the file, a called program can read or write to the file, and the calling program or another program can close it.

You can also use the EXTERNAL clause in *record-description-entries* in WORKING-STORAGE. See the EXTERNAL section later in this chapter for more information.

Format

IS EXTERNAL

Syntax Rules

1. You can specify the EXTERNAL clause in a *file-description-entry*.
2. In the same program, the *file-name* that you specify as the subject of the entry must be neither the same *data-name* that you specify for any other *data-description-entry* that includes the EXTERNAL clause, nor the same *program-name* that you specify in the PROGRAM-ID paragraph.
3. When you specify EXTERNAL in a *file-description-entry*, the *record-description-entries* associated with that file are implicitly defined as EXTERNAL.
4. When you specify EXTERNAL in a *file-description-entry*, the WORKING-STORAGE items associated with that file, such as FILE STATUS, RELATIVE KEY, or FILE-ID, are implicitly defined as EXTERNAL.

General Rules

The file named by the EXTERNAL clause can be accessed and processed by any program in the run unit that describes it according to the following rules.

1. Within a run unit, if two or more programs describe the same external file, the associated *data-description-entries*, including all subordinate *data-names* and their redefinitions, must be identical.
2. The file control block associated with this file is external.

Note

If you use SEG as the loader, ensure that all external items (explicit or implicit) in the run unit have variable names that are unique through the first eight characters.

BLOCK CONTAINS

The BLOCK CONTAINS clause specifies the size of a physical record.

Format

BLOCK CONTAINS [*integer-1* TO] *integer-2* { RECORDS / CHARACTERS }

Chapter 12 discusses the BLOCK CONTAINS clause.

CODE-SET

The CODE-SET clause specifies the character code set used to represent data on external media.

Format

CODE-SET IS *alphabet-name*

Chapter 12 discusses the CODE-SET clause.

DATA RECORDS

The DATA RECORDS clause serves only as documentation for the names of data records with their associated file.

Format

DATA { RECORD IS / RECORDS ARE } *data-name-1* [, *data-name-2*] . . .

Syntax Rule

data-name-1 and *data-name-2* are the names of the data records for the FD entry. You must specify them as 01-level items following the file description, and they must follow the general rules for word formation listed in Chapter 4.

General Rules

1. More than one *data-name* indicates that the file contains more than one type of data record. These records can have different sizes and formats. The order in which you list them is not significant.

2. Conceptually, all data records within a file share the same area, regardless of the number or types of data records within the file.

LABEL RECORDS

The LABEL RECORDS clause specifies whether labels exist for the file.

Format

```
LABEL { RECORD IS  } { STANDARD }
      { RECORDS ARE} { OMITTED  }
```

General Rules

1. If you do not specify the LABEL RECORDS clause, LABEL RECORDS OMITTED is assumed.
2. OMITTED specifies that no explicit labels exist for the file or device to which the file is assigned.
3. STANDARD specifies that a label exists for the tape file and that the label conforms to system specifications.
4. Each Prime device requires a specific LABEL option, as shown in Table 7-1.

Note

See Chapter 12 and the *Magnetic Tape User's Guide* for more information on writing standard labels for magnetic tape.

TABLE 7-1
LABEL Clause Requirement

Device	*STANDARD*	*OMITTED*
PRIMOS		X
PRISAM		X
MIDASPLUS		X
PFMS (Disk)		X
TERMINAL		X
PRINTER		X
MT9 (Tape)	X	X
OFFLINE-PRINT		X

RECORD

The RECORD clause specifies the exact size of a fixed-length record, or the minimum and maximum sizes of a variable-length record.

Format 1

RECORD CONTAINS *integer-1* CHARACTERS

Format 2

RECORD IS VARYING IN SIZE [[FROM *integer-2*] [TO *integer-3*] CHARACTERS]

Format 3

RECORD CONTAINS *integer-4* TO *integer-5* CHARACTERS

Format 4

RECORD IS NOT VARYING IN SIZE

Notes

The RECORD clause specifies the size of fixed-length or variable-length records for indexed, relative, and sequential files. Fixed-length records contain the same number of character positions in each record. Variable-length records can have a range of record sizes. If a file's organization is relative, all records are preallocated as maximum length records whether you specify them as fixed or variable.

The maximum allowable size of a single data record is listed in Appendix I.

Syntax Rules

1. In Format 1, no *record-description-entry* for the file may specify a number of character positions greater than *integer-1*.
2. In Format 1, at least one *record-description-entry* must specify a number of character positions equal to *integer-1*.
3. In Format 2, no *record-description-entry* for the file may specify a number of character positions less than *integer-2* or greater than *integer-3*.
4. In Format 3, *integer-4* and *integer-5* refer to the minimum number of characters in the smallest record and the maximum number of characters in the largest record, respectively.

General Rules

All Formats:

1. If you do not specify the RECORD clause, the size of each data record is completely defined in the *record-description-entry*. If you specify multiple *record-description-entries* having different lengths and/or containing a variable occurrence data item, and you specify the –VARYING compiler option, COBOL85 processes records as if you specified a Format 3 RECORD clause. If you specify multiple *record-description-entries* having different lengths and/or containing a variable occurrence data item, and you

specify the –NO_VARYING compiler option, COBOL85 processes records as if you specified a Format 1 RECORD clause.

2. If you specify the RECORDING MODE IS V clause, COBOL85 processes records as if you specified a Format 3 RECORD clause with an implied *integer-4* and *integer-5* that represent the smallest and largest *record-description-entry* for the file. RECORDING MODE IS V overrides the –NO_VARYING compiler option.
3. If you specify the RECORD IS NOT VARYING clause, COBOL85 processes records as if you specified a Format 1 RECORD clause with an implied *integer-1* that represents the largest *record-description-entry* for the file. RECORD IS NOT VARYING overrides the –VARYING compiler option.
4. The number of character positions required to store the logical record, regardless of the type of characters used to represent the items within the logical record, determines the size of each data record. For variable-length records, the size of the record is the sum of the number of character positions in all fixed-length elementary items plus the sum of the maximum number of character positions in any variable occurrence data item subordinate to the record. This sum includes any compiler-generated filler (see the section Alignment of Substructures Within Structures in Chapter 4).
5. During the execution of a sending or receiving operation, the number of character positions in a variable-length record being sent or received is determined as follows:

   - If the record does not contain a variable occurrence data item, the size of the record is the number of character positions in the record.
   - If the record contains a variable occurrence data item, the size of the record is the sum of the fixed portion of the record and that portion of the variable occurrence data item described by the number of occurrences at the start of statement execution.

6. If a record description contains a variable occurrence data item, and you specify the –VARYING compiler option,

   - The minimum number of table elements described in the record determines the minimum number of character positions associated with the record description.
   - The maximum number of table elements described in the record determines the maximum number of character positions associated with the record description.

Format 1:

1. Use Format 1 to specify fixed-length records.
2. If you specify the –VARYING compiler option, it is ignored.
3. If you specify multiple *record-description-entries* having different lengths, COBOL85 uses the maximum size record length in all I-O operations.

Format 2:

1. Use Format 2 to specify variable-length records, especially when the *record-description-entries* do not specify the largest or the smallest actual record size associated with the file.
2. If you specify the –NO_VARYING compiler option, it is ignored.

3. COBOL85 uses *integer-2* and *integer-3* to check the size file attributes during an OPEN operation. The actual file attributes (minimum and maximum logical record sizes) must equal *integer-2* and *integer-3*, respectively. COBOL85 ignores the minimum and maximum sizes in *record-description-entries* during file attribute checking.
4. If you specify Format 2 for a file whose record descriptions are the same length, the library I-O routines process the records as variable-length records.
5. If you do not specify *integer-2*, the minimum number of character positions contained in any record of the file is equal to the minimum number of character positions described for a record in that file.
6. If you do not specify *integer-3*, the maximum number of character positions contained in any record of the file is equal to the maximum number of character positions described for a record in that file.

Format 3:

1. Use Format 3 to specify variable-length records when the *record-description-entries* specify the largest and the smallest actual record size associated with the file.
2. If you specify the –NO_VARYING compiler option, COBOL85 ignores *integer-4* and processes records as if you specified a Format 1 RECORD clause.
3. COBOL85 uses *integer-4* and *integer-5* to check the size file attributes during an OPEN operation. The actual file attributes (minimum and maximum logical record sizes) must equal *integer-4* and *integer-5*, respectively.

RECORDING MODE

The RECORDING MODE clause specifies the format of the logical records in a file.

Format

RECORDING MODE IS {F, U, S, V}

General Rules

1. You can specify **F mode** (fixed-length format) when all the records in a file are the same length. The specification of the RECORDING MODE IS F clause overrides the –VARYING compiler option.
2. You can specify **U mode** (unspecified format) for any combination of record descriptions. This mode is ignored by the COBOL85 compiler.
3. You can specify **S mode** (spanned format) for any combination of record descriptions. This mode is ignored by the COBOL85 compiler.
4. You can specify **V mode** (variable-length format) when the records of a file vary in length. You can use this mode for single *record-description-entries* or for multiple *record-description-entries*. See the preceding section for more information on variable-length record specification. The specification of the RECORDING MODE IS V clause overrides the –NO_VARYING compiler option.

5. If you specify the RECORDING MODE IS V clause for a file whose record descriptions are the same length, the library I-O routines handle and format the records as variable-length records.

VALUE OF FILE-ID

The VALUE OF FILE-ID clause associates the internal filename with a disk file or a tape file, thus allowing for the linkage of internal and external filenames.

Format

VALUE OF FILE-ID **IS** { *data-name-3* / *literal-2* }

Syntax Rules

1. Disk filenames in *data-name-3* and *literal-2* must have the following format:

   [[[***MFD-name***] ***directory-name***] ***sub-directory-name*** ...] ***file-name***
   [**:** ***share-mode*** [**,** ***wait-mode***]]

   where the elements of this format have the following meanings:

Element	*Meaning*	
file-name	The disk filename	
colon (:)	Punctuation required to specify optional modes for PRISAM files	
share-mode	Optional share mode for PRISAM files:	
	FSH	Fully shared (default)
	EXC	Exclusive
	PRO	Protected
comma (,)	Punctuation required to specify optional wait-mode for PRISAM files	
wait-mode	Optional wait mode for PRISAM files:	
	NWT	No wait (default)
	WAT	Wait

   For example,

```
VALUE OF FILE-ID IS 'MY-FILE:EXC,WAT'
```

   identifies a PRISAM file and specifies share and wait modes.

Notes

To specify a wait-mode, you must first specify a share-mode.

If *file-name* is not a PRISAM file, share-mode and wait-mode are ignored.

If you specify an invalid share-mode or wait-mode, the program aborts at runtime.

See the *PRISAM User's Guide* for more information on share and wait modes.

2. Tape filenames in *data-name-3* and *literal-2* must have one of the formats described in Chapter 12.
3. Do not use the VALUE OF FILE-ID clause for files assigned to TERMINAL or OFFLINE-PRINT devices.
4. You can use the *implementor-name* FILE-ID as a programmer-defined word elsewhere in the program.
5. *literal-2* is a nonnumeric literal that must not exceed 128 characters.
6. *data-name-3* must be an alphanumeric item defined in the WORKING-STORAGE SECTION. It can be qualified, but it must not be subscripted or indexed. The value of *data-name-3* must not exceed 128 characters.

General Rules

1. If you do not specify the VALUE OF FILE-ID clause, the compiler uses the *file-name* following FD as the name of the file.
2. If you specify the VALUE OF FILE-ID clause, the compiler uses *literal-2* or the value in *data-name-3* as the name of the file or pathname. The COBOL85 program must assign to *data-name-3* a value that is the pathname or filename.
3. To change the name of a file during the execution of the run unit, you can assign a value to *data-name-3* with ACCEPT or MOVE statements. For an example, see the sample program at the end of Chapter 11.
4. If the program changes the value of *data-name-3*, it must close and then reopen the file in order to implement the new value of *data-name-3*.
5. Use the –FILE_ASSIGN compiler option if you wish to modify the value of *literal-2* or *data-name-3* at the beginning of program execution. See Appendix N for more details.

Examples

You can associate a PRIMOS file named FILEX with a logical COBOL85 file named TEST-FILE in any of the following ways.

1. VALUE OF FILE-ID IS *literal*:

```
FD TEST-FILE
   LABEL RECORDS OMITTED
   VALUE OF FILE-ID 'X>Y>FILEX'.
```

2. VALUE OF FILE-ID IS *data-name*:

```
FD TEST-FILE
   LABEL RECORDS OMITTED
   VALUE OF FILE-ID IS TFILE-NAME.
    .
    .
WORKING-STORAGE SECTION.
77 TFILE-NAME                PIC X(24).
```

You can associate an actual filename with the logical filename TEST-FILE by executing COBOL85 statements. For example,

```
IF NEW-FILE = 1
              MOVE "FILEX" TO TFILE-NAME,
ELSE IF NEW-FILE = 2
              MOVE "OTHER" TO TFILE-NAME,
     ELSE
              MOVE "ARTHUR>TESTFILE" TO TFILE-NAME.
```

You can also use the ACCEPT verb to associate an actual filename with a logical filename. For example,

```
MOVE SPACES TO TFILE-NAME
DISPLAY "ENTER TEST-FILE NAME: " WITH NO ADVANCING.
ACCEPT TFILE-NAME.
```

record-description-entry

A record description describes the characteristics of a particular record. You can code *record-description-entries* in the FILE SECTION, the WORKING-STORAGE SECTION, and the LINKAGE SECTION of the DATA DIVISION. The following sections describe the clauses that you can specify in a *record-description-entry*.

Format 1

```
level-number [ data-name-1 ]
             [ FILLER      ]

[ ; BLANK WHEN ZERO]

[ ; IS EXTERNAL]

[ ; { JUSTIFIED } RIGHT ]
    { JUST      }

[ ; OCCURS { integer-1 TO integer-2 TIMES DEPENDING ON data-name-2 }
           { integer-2 TIMES                                       }
  [ { ASCENDING  } KEY IS data-name-3 [, data-name-4] . . . ] . . .
    { DESCENDING }

  [INDEXED BY index-name-1 [, index-name-2] . . . ]

[ ; { PICTURE } IS character-string ]
    { PIC     }

[ ; REDEFINES data-name-5 ]

[ ; [SIGN IS] { LEADING  } [SEPARATE CHARACTER] ]
              { TRAILING }

[ ; { SYNCHRONIZED } [ LEFT  ] ]
    { SYNC         } [ RIGHT ]

[ ; [USAGE IS] { BINARY                } ]
               { COMPUTATIONAL         }
               { COMP                  }
               { COMPUTATIONAL-1       }
               { COMP-1                }
               { COMPUTATIONAL-2       }
               { COMP-2                }
               { COMPUTATIONAL-3       }
               { COMP-3                }
               { DISPLAY               }
               { INDEX                 }
               { PACKED-DECIMAL        } ]

[; VALUE IS literal]
```

Format 2

```
66 data-name-1 ; RENAMES data-name-2 [ { THROUGH } data-name-3 ]
                                       { THRU    }
```

Format 3

```
88 condition-name; { VALUE IS  } literal-1 [ { THROUGH } literal-2 ]
                   { VALUES ARE }           { THRU    }

   [ , literal-3 [ { THROUGH } literal-4 ] ] ...
                   { THRU    }
```

Syntax Rules

1. The *level-number* in Format 1 can contain a value of 01 through 49, or 77.
2. In Format 1, you can write clauses in any order, except the REDEFINES clause. When used, this clause must immediately follow the *data-name-1* phrase.
3. In Format 1, you must specify the PICTURE clause for every elementary item except those items whose usage is described as binary (COMPUTATIONAL or BINARY), floating-point (COMPUTATIONAL-1 or COMPUTATIONAL-2), or INDEX. A group item cannot contain a PICTURE clause.
4. Do not specify the OCCURS clause in a *data-description-entry* that has a 66, 77, or an 88 *level-number*.
5. Format 2 permits alternative, possibly overlapping, groups of elementary items.
6. The words THRU and THROUGH are equivalent.

General Rules

1. A *record-description-entry* can appear in the FILE SECTION, WORKING-STORAGE SECTION, and LINKAGE SECTION of the DATA DIVISION.
2. You must describe all records in each *file-description-entry* by *record-description-entries*.

The following sections describe the clauses that you can specify in a *record-description-entry*.

level-number

The *level-number* shows the position of a data item within the hierarchy of a logical record. It also specifies entries for *condition-names*, the RENAMES clause, and data items in the WORKING-STORAGE SECTION and LINKAGE SECTION.

Format

level-number

Syntax Rules

1. A *level-number* is required as the first element in each *data-description-entry*.
2. *record-description-entries* subordinate to an FD or SD entry must have *level-numbers* 01 through 49, 66, or 88.
3. *data-description-entries* in the WORKING-STORAGE SECTION and LINKAGE SECTION must have *level-numbers* 01 through 49, 66, 77, or 88.

General Rules

1. Use *level-numbers* to subdivide a record so that your program can refer to each item in the record. You can divide a record into subdivisions, and you can further divide each subdivision. An item that you do not further subdivide is called an **elementary item**. A record can itself be an elementary item.
2. A **group** consists of one or more consecutive elementary items; groups can, in turn, be combined into other groups. A group includes all group and elementary items following it until the next item with a *level-number* less than or equal to the *level-number* of that group.
3. *level-numbers* range from 01, the most inclusive level, to 49, the least inclusive level. Any *level-number* except 49 can denote a group.
4. *level-number* 01 identifies the first entry in each record description. A reference to a level-01 *data-name* in the PROCEDURE DIVISION is a reference to the entire record.
5. Multiple level-01 entries subordinate to one FD represent implicit redefinitions of the same area.
6. Use the following special *level-numbers* to identify certain entries in which no real concept of hierarchy exists.

   - *level-number* 77 identifies noncontiguous WORKING-STORAGE or LINKAGE data items. Use them only as described in Format 1 of the *record-description-entry*.

     Level-77 data items are elementary items that cannot be subdivided.

   - *level-number* 88 identifies entries that define *condition-names* associated with a conditional variable. Chapter 4 discusses *condition-names* and the conditional variable. Use level 88 only with Format 3 of the *record-description-entry*.

     Level-88 entries can contain individual values, series of individual values, or a range of values.

     For example,

```
01  TEST-AREA PIC X.
    88  TEST-VALUE-1 VALUE '1'.
    88  TEST-VALUE-2 VALUE '1', '2'.
    88  TEST-VALUE-3 VALUE '1' THRU '8'.
    88  TEST-VALUE-4 VALUE '1' THRU '4', '6', '7'.
```

The VALUE clause is required in a level-88 entry, and must be the only clause in the entry. THRU and THROUGH are equivalent.

A level-88 entry must be preceded either by another level-88 entry, or by an elementary item, called the **conditional variable**.

The *condition-name* can be qualified by the name of the conditional variable. You can use a *condition-name* in the PROCEDURE DIVISION in place of a relational condition, as discussed in Chapter 4. If you use a *condition-name* in the PROCEDURE DIVISION, you must subscript it if its conditional variable is subscripted. The type of literal in a *condition-name* VALUE clause must be consistent with the data type of the conditional variable.

In the following example, PAYROLL-PERIOD is the conditional variable. The PICTURE clause associated with it limits the value of the 88 *condition-name* to one digit.

```
02  PAYROLL-PERIOD      PICTURE IS 9.
    88  WEEKLY              VALUE IS 1.
    88  SEMI-MONTHLY        VALUE IS 2.
    88  MONTHLY             VALUE IS 3.
```

Using the above description, you can write the following *condition-name* test in the PROCEDURE DIVISION:

```
IF MONTHLY PERFORM DO-MONTHLY.
```

An equivalent statement is

```
IF PAYROLL-PERIOD = 3 PERFORM DO-MONTHLY.
```

- *level-number* 66 identifies RENAMES entries, which are discussed later in this chapter. Use *level-number* 66 only with Format 2 of the *record-description-entry*.

Example

The structure chart in Figure 7-1 illustrates the level concept. The chart represents the structure of a weekly time card record. The record is divided into four major items: name, employee number, pay period end date, and hours worked, with more specific information included within name and pay period end date.

You can describe the time card record with DATA DIVISION entries having the following *level-numbers*, *data-names*, and *picture-definitions*.

```
01  TIME-CARD.
    05  NAME.
        10   LAST-NAME          PICTURE X(18).
        10   FIRST-INIT         PICTURE X.
        10   MIDDLE-INIT        PICTURE X.
    05  EMPLOYEE-NUM           PICTURE 99999.
    05  PERIOD-ENDED.
        10   MONTH              PIC 99.
        10   DAYY               PIC 99.
        10   YEAR               PIC 99.
    05  HOURS-WORKED           PICTURE 99V99.
```

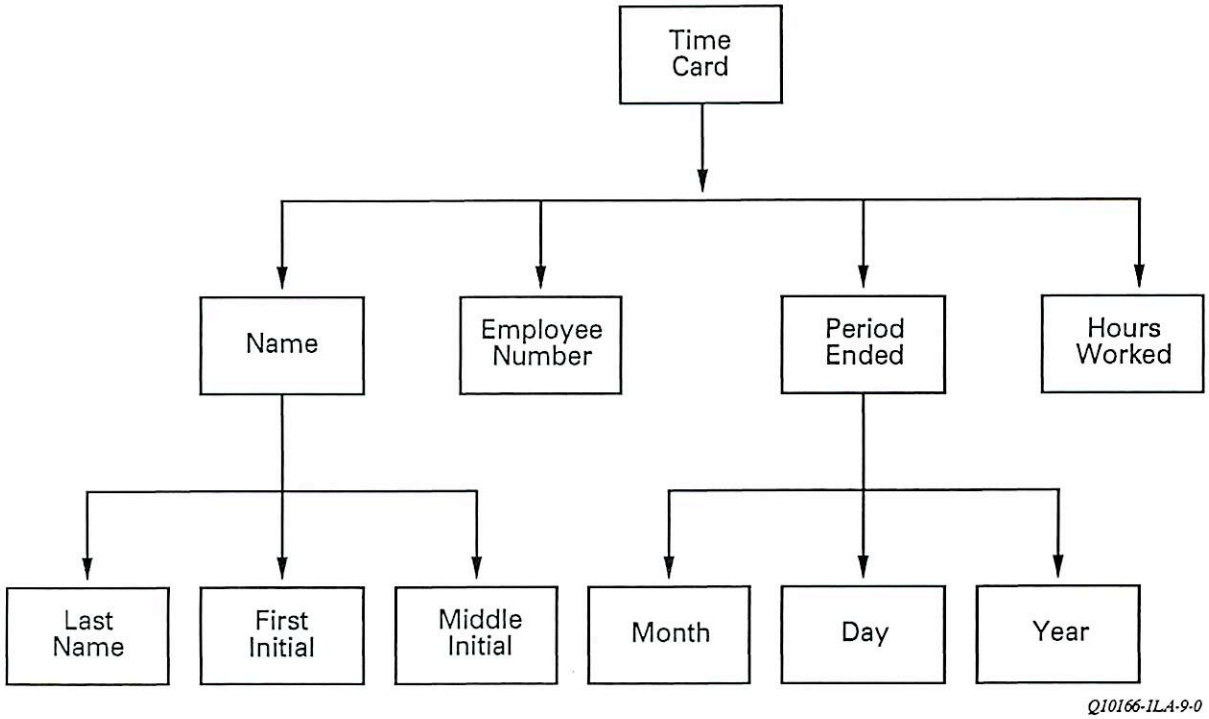


FIGURE 7-1
Weekly Time Card Record

data-name or FILLER

A *data-name* specifies the name of the *data-description-entry*. FILLER specifies an elementary item that you cannot refer to explicitly.

Format

[*data-name*]
[FILLER]

General Rules

1. **Prime Extension:** You can use FILLER to name a group item as well as an elementary item.
2. You cannot refer to a FILLER item explicitly. However, you can use FILLER as a conditional variable. Such use does not require explicit reference to the FILLER item, but rather to its value.
3. You can use a VALUE clause with a FILLER item.
4. If you specify neither *data-name* nor FILLER, the data item is treated as though you specified FILLER.

Example

```
01  INPUT-RECORD.
    05  FILLER                              PIC 9.
        88 FULL-TIME    VALUE 1.
        88 PART-TIME    VALUE 2.
    05  NAME                                PIC X(40).
    05  FILLER                              PIC X(10).
    05  HOURS                               PIC 99.
```

In this example, you cannot change the FILLER items except by moving data into the group item INPUT-RECORD. You cannot refer to the second FILLER item individually, but you can test the first FILLER item with code such as the following:

```
IF FULL-TIME PERFORM 070-FULL-TIME.
```

BLANK WHEN ZERO

The BLANK WHEN ZERO clause sets an item to blanks when the item's value is zero.

Format

BLANK WHEN ZERO

Syntax Rules

1. You can use the BLANK WHEN ZERO clause only for an elementary numeric or numeric edited item.
2. The item must be described, either implicitly or explicitly, as USAGE IS DISPLAY.

General Rules

1. The BLANK WHEN ZERO clause specifies that the data item be set to blanks when the value is all zeros. This clause does not suppress leading zeros. Table 7-2 illustrates some uses for this clause.
2. If you specify the clause for a numeric item, the compiler interprets the category of the item as numeric edited.
3. Do not use an asterisk as the zero-suppression symbol in a *picture-string* with this clause.

TABLE 7-2
Examples: BLANK WHEN ZERO

Value	*Description of OUT-COST*		*Result*
0012.34	9999.99	BLANK WHEN ZERO	0012.34
0123.45	$9999.99	BLANK WHEN ZERO	$0123.45
01.2345	$9999.99	BLANK WHEN ZERO	$0001.23
0000.04	$$$$$.99	BLANK WHEN ZERO	$.04
0000.00	$$$$$.99	BLANK WHEN ZERO	bbbbbbb
0000.00	$$$$$.99		$.00
0000.00	ZZZZVZZ	BLANK WHEN ZERO	bbbbbb
0000.04	ZZZZVZZ	BLANK WHEN ZERO	4
0000.00	ZZZZ.ZZ	BLANK WHEN ZERO	bbbbbbb
0000.04	ZZZZ.ZZ	BLANK WHEN ZERO	.04
0000.00	ZZZZ.99	BLANK WHEN ZERO	bbbbbbb
0000.00	ZZZZ.99		.00

b = blank.

EXTERNAL

The EXTERNAL clause specifies that a data item is external. Such an item is also referred to as a **common block**. The data item, including all subordinate data items, is available to every program in the run unit that describes that data item. Use of the EXTERNAL clause saves space within the programs of the run unit.

For a discussion of EXTERNAL *file-description-entries*, see the discussion of the EXTERNAL clause earlier in this chapter.

Format

IS EXTERNAL

Syntax Rules

1. You can specify the EXTERNAL clause for *data-description-entries* in the WORKING-STORAGE SECTION whose *level-number* is 01 or 77.
2. Do not specify the EXTERNAL clause and the REDEFINES clause in the same *data-description-entry*.
3. Do not use the VALUE clause in any *data-description-entry* that includes or is subordinate to an entry that includes the EXTERNAL clause. However, you can specify the VALUE clause for *condition-name* entries associated with such *data-description-entries*.

Note

If you use SEG as the loader, ensure that all external items in the run unit have variable names that are unique through the first eight characters.

General Rules

1. The data contained in the record named by *data-name* is external and can be accessed and processed by any program in the run unit that describes and, optionally, redefines it, subject to the following general rule.
2. Within a run unit, if two or more programs describe the same external data record, the associated *data-description-entries*, including all subordinate *data-names* and data items and their redefinitions, must be identical. However, a program that describes an external record can contain a *data-description-entry* that redefines the complete external record. This complete redefinition need not occur identically in other programs in the run unit.

Example

This example uses one program, CALLER.COBOL85, that calls a second program, CALLED.COBOL85. Both use a file called PFMS-FILE, which is defined as EXTERNAL. The called program does not need to refer to PFMS-FILE in the LINKAGE SECTION. The called program writes one record, 'XX', to this file. After the main program calls the second program, it checks that the PFMS-FILE file contains the record 'XX'.

```
OK, SLIST CALLER.COBOL85
       ID DIVISION.
       PROGRAM-ID. FILID.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT PFMS-FILE  ASSIGN PFMS.
       DATA DIVISION.
       FILE SECTION.
       FD PFMS-FILE  EXTERNAL  VALUE OF FILE-ID 'MY-FILE'.
       01 RECORD1   PIC XX.
       01 PFMS-REC  PIC XX.
       PROCEDURE DIVISION.
           CALL 'PFMS1'.
           CLOSE PFMS-FILE.
           OPEN INPUT PFMS-FILE.
           READ PFMS-FILE END DISPLAY 'END-OF-FILE'.
           IF PFMS-REC = 'XX' DISPLAY ' PASS' ELSE DISPLAY ' FAIL'.
           CLOSE PFMS-FILE.
           STOP RUN.
OK, SLIST CALLED.COBOL85
       ID DIVISION.
       PROGRAM-ID. PFMS1.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT PFMS-FILE ASSIGN PFMS.
       DATA DIVISION.
       FILE SECTION.
       FD PFMS-FILE  EXTERNAL
          VALUE OF FILE-ID IS 'MY-FILE'.
```

```
01 RECORD1 PIC XX.
PROCEDURE DIVISION.
    OPEN OUTPUT PFMS-FILE.
    MOVE 'XX' TO RECORD1.
    WRITE RECORD1.
    EXIT PROGRAM.
```

This program, when executed, produces the following results:

```
PASS
```

JUSTIFIED

The JUSTIFIED clause specifies right alignment of data within a field.

Format

$$\left\{\begin{matrix}\underline{\text{JUSTIFIED}}\\ \underline{\text{JUST}}\end{matrix}\right\} \text{ RIGHT}$$

Syntax Rules

1. You can specify the JUSTIFIED clause only at the elementary level.
2. JUST is a valid abbreviation of JUSTIFIED.
3. Do not use the JUSTIFIED clause for data items described as numeric.
4. Do not use the JUSTIFIED clause for data items for which you specify editing.

General Rules

1. When you include the JUSTIFIED clause, values are stored right to left. In a MOVE operation, if the sending field is shorter than the JUSTIFIED receiving field, space filling occurs in the leftmost positions of the receiving field. If the sending field is longer, the leftmost characters of the sending field are truncated. If the sending field is the same size as the JUSTIFIED receiving field, the result is a straight MOVE, including spaces.
2. When you omit the JUSTIFIED clause, the standard alignment rules listed in Chapter 4 apply.

Example

Of the following two fields, Y is right justified and Z is left justified. Because the MOVE statement involves a literal smaller than the picture definition of the data fields, the contents of both Y and Z are appropriately justified.

```
    01 Y PIC X(4) JUST RIGHT.
    01 Z PIC X(4).

MOVE 'AB' TO Y Z.
EXHIBIT Y '  ' Z.      terminal displays Y = bbAB    Z = ABbb
                                        where b = blank
```

OCCURS

The OCCURS clause permits you to define related sets of repeated data, such as tables, arrays, and lists. The clause also provides required information for the application of subscripts and indexes.

Format 1

OCCURS *integer-2* TIMES

[{ ASCENDING / DESCENDING } KEY IS *data-name-2* [, *data-name-3*] ...] ...

[INDEXED BY *index-name-1* [, *index-name-2*] ...]

Format 2

OCCURS *integer-1* TO *integer-2* TIMES DEPENDING ON *data-name-1*

[{ ASCENDING / DESCENDING } KEY IS *data-name-2* [, *data-name-3*] ...] ...

[INDEXED BY *index-name-1* [, *index-name-2*] ...]

Syntax Rules

1. Do not use the OCCURS clause in a *data-description-entry* that
   - Has a 66, 77, or 88 *level-number*
   - Describes an item whose size is variable (that is, an item that includes a subordinate item containing Format 2 of the OCCURS clause)
2. *integer-1* must be greater than or equal to zero. If you use both *integer-1* and *integer-2*, *integer-2* must be greater than *integer-1*. The maximum allowable table size is listed in Appendix I.
3. You must define *data-name-1* as an integer. If *data-name-1* is signed, the value of *data-name-1* must be positive. You can describe *data-name-1* within the record entry or outside of it.

4. *data-name-2* must be the name of either the entry containing the OCCURS clause or an entry subordinate to the entry containing the OCCURS clause. *data-name-3* and any additional *data-names* in the KEY IS phrase must be subordinate to the entry containing the OCCURS clause.
5. The *data-names* in the KEY IS phrase must not contain an OCCURS clause, except where *data-name-2* is the subject of the entry.
6. Entries must not contain an OCCURS clause between the *data-names* in the KEY IS phrase and the subject of the entry, except where *data-name-2* is the subject of the entry.
7. All *data-names* used in the OCCURS clause can be qualified; however, they must not be subscripted or indexed.
8. An INDEXED BY phrase is required if the subject of this entry, or an entry subordinate to this entry, is to be referenced by indexing. The *index-names* identified by this phrase are not defined elsewhere. Because *index-names* do not represent data, you cannot associate them with any data record or refer to them in a USING phrase.
9. Each *index-name* must be unique within the program.
10. If you define the subject of the OCCURS clause as EXTERNAL, and you do not define *data-name-1* as EXTERNAL, *data-name-1* inherits the EXTERNAL attribute. If you define *data-name-1* outside of the EXTERNAL variable occurrence data item, it must be an 01 or 77 level item.
11. A *data-description-entry* that contains Format 2 of the OCCURS clause can be followed, within that record description, only by *data-description-entries* that are subordinate to it.
12. In Format 2, the data item defined by *data-name-1* must not fall within the range of the first character position defined by the *data-description-entry* containing the OCCURS clause and the last character position defined by the *record-description-entry* containing that OCCURS clause.

General Rules

1. The OCCURS clause defines tables of repeating data items. When you use the OCCURS clause, the *data-name* that is the subject of the entry, and any items subordinate to the subject, must be referred to by subscripting or indexing, except when used with the SEARCH verb.
2. Except for the OCCURS clause itself, all data description clauses associated with an item containing an OCCURS clause apply to each occurrence of the item being described.
3. The KEY IS phrase indicates that the repeated data is arranged in ascending or descending order according to the values contained in *data-name-2*, *data-name-3*, and so on. The rules for the evaluation of relation conditions, discussed in the section Conditional Expressions in Chapter 4, determine the ascending or descending order. The *data-names* are listed in their descending order of significance.
4. When you use the INDEXED BY phrase, an index is assigned to a table; *index-name* identifies the individual occurrences of items in the table. For example, the following code defines a table MONTH-TAB of 12 items, indexed by INDX.

```
      .
      .
      .
05  MONTH-TAB OCCURS 12 TIMES
    ASCENDING KEY MONTH-NO
    INDEXED BY INDX.
    10  MONTH-NO      PIC 99.
    10  MONTH-VALUE   PIC XXX.
      .
      .
      .
```

This code creates a storage area as shown in Figure 7-2.

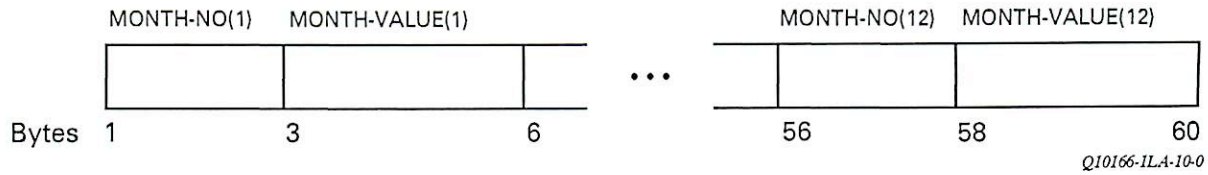


FIGURE 7-2
A 12-element Table

5. You can refer to an individual item in the table by means of an index or subscript. Assuming INDX and DATA-NM both have the value 4, you can refer to the item MONTH-NO of the fourth element of MONTH-TAB in any of the following three ways:

```
MONTH-NO(4)
MONTH-NO(INDX)
MONTH-NO(DATA-NM)
```

6. You can modify an *index-name* only by the SET verb, the SEARCH verb, and the PERFORM verb.
7. You can specify a maximum of eight indexes in an indexed reference.
8. The section Data Representation and Alignment in Chapter 4 describes the format of an *index-name*. Its maximum value is listed in Appendix I.
9. When you omit the INDEXED BY phrase, you must use subscripting alone to refer to an individual element within a table.
10. The number of occurrences of the subject entry is defined as follows:

    - If you do not use the DEPENDING phrase, the value of *integer-2* represents the exact number of occurrences.
    - If you use the DEPENDING phrase, the current value of the data item referenced by *data-name-1* represents the number of occurrences.

    The DEPENDING phrase specifies that the subject of this entry has a variable number of occurrences. The value of *integer-2* represents the maximum number of occurrences and the value of *integer-1* represents the minimum number of occurrences. Using the

DEPENDING phrase does not imply that the length of the subject of the entry is variable, but that the number of occurrences of the subject is variable.

The value of *data-name-1* must fall within the range *integer-1* through *integer-2*. The contents of data items whose occurrence numbers exceed the value of *data-name-1* are unpredictable.

Note

If you specify the –RANGE or –RANGE_NONFATAL compiler option, COBOL85 checks the value of *data-name-1* each time the program refers to the associated variable occurrence data item.

11. When you refer to a group data item that contains a variable occurrence data item, the part of the table area used in the operation is determined as follows:

    - If the data item referenced by *data-name-1* is outside the group, only that part of the table area specified by the value of *data-name-1* at the start of the operation is used.
    - If the data item referenced by *data-name-1* is included in the same group, and the group data item is referenced as a sending item, only that part of the table area specified by the value of *data-name-1* at the start of the operation is used. If the group is the receiving item, the maximum length of the group is used.

12. When you refer to a record or group item that contains an OCCURS DEPENDING ON clause while using the Source Level Debugger (DBG), the value of the OCCURS DEPENDING ON item at the start of statement execution determines the part of the table that is defined.

Example

The following is an example of an FD entry for a file of variable-length records. The record description contains an OCCURS DEPENDING ON clause, which allows from zero to twenty-five students to be enrolled in each course. Figure 7-3 illustrates how data is stored in such a file.

```
FD COURSE-FILE
   RECORD CONTAINS 28 TO 428 CHARACTERS.
01 COURSE-REC.
   05  COURSE-ID                PIC X(06).
   05  COURSE-TITLE-SHORT       PIC X(10).
   05  INSTRUCTOR-NAME-SHORT    PIC X(10).
   05  NUMBER-OF-STUDENTS       PIC 9(04) COMP.
   05  STUDENT-RECORD           OCCURS 0 TO 25 TIMES
                                DEPENDING ON
                                NUMBER-OF-STUDENTS.
       10  STUDENT-ID           PIC 9(06).
       10  STUDENT-NAME-SHORT   PIC X(10).
```

Record Number 1

ENG100	ENG COMP 1	SMITH	00

Record Number 2

HIS100	AM HIST 1	JONES	02	123456	THOMAS	654321	JOHNSON

Record Number 3

MUS100	MUSIC APP	WILSON	01	654321	JOHNSON

Q10166-1LA-11-0

FIGURE 7-3
Variable-length Data Storage

PICTURE

The PICTURE clause describes the general characteristics and editing requirements of an elementary item.

Format

$$\left\{\begin{matrix}\textbf{PICTURE}\\ \underline{\textbf{PIC}}\end{matrix}\right\} \textbf{IS } \textit{character-string}$$

Syntax Rules

1. You can specify a PICTURE clause only at the elementary item level.
2. A PICTURE *character-string* consists of certain allowable combinations of characters in the COBOL85 character set used as symbols. The allowable combinations determine the category of the elementary item.
3. The maximum number of characters allowed in a PICTURE *character-string* is 32. In the shorthand notation X(n), the repeat integer *n* must be within the range of 1 through 32767.
4. You must specify the PICTURE clause for every elementary item except binary and floating-point items. The PICTURE clause is optional for items with COMP or BINARY usage. (The default is PICTURE S9999.) The PICTURE clause is not allowed with items whose usage is COMPUTATIONAL-1, COMPUTATIONAL-2, or INDEX.
5. PIC is an abbreviation for PICTURE.
6. A PICTURE *character-string* must include at least one of the characters Z A * X 9 or at least two consecutive appearances of the characters + – or the currency symbol. You can break a PICTURE *character-string* so that part is on the continuation line. A hyphen in the indicator area indicates that the string has no blank before the continued part. If you omit the hyphen, then a blank is inserted into the PICTURE *character-string* before the continued part, which creates an erroneous PICTURE *character-string*.

General Rules

The following general rules govern the PICTURE clause as it relates to

- Data categories
- Data item size
- Symbol functions

Data Categories: You can define five categories of data with a PICTURE clause: alphabetic, numeric, alphanumeric, alphanumeric edited, and numeric edited. Their PICTURE *character-strings* have the following restrictions.

- Alphabetic: The PICTURE *character-string* can contain only the characters A and B. Contents of the data field so described are restricted to any combination of the letters of the English alphabet and the COBOL85 space character. See Example 3 of Table 7-3.

TABLE 7-3
Examples of PICTURE Clauses and Conversions

Example	*Value*	*Nonedited PICTURE*	*Stored as*
1	37	P999	0
	.5	P999	0
	.05	P999	.05
2	37	999PP	0
	3700	999PP	3700
Example	*Value*	*Edited PICTURE*	*Prints as*
3	JDOE	ABAAAA	J DOE
4	JDOE	XBXXXX	J DOE
5	113113	XXX0XXX	1130113
6	459243333	XXX/XX/XXXX	459/24/3333
7	10000	99,999	10,000
8	9999	999.9	999.0
9	999	999.9	999.0
10	+5500	+9999	+5500
11	–5500	+9999	–5500
12	+5500	9999CR	5500
13	+5500	9999DB	5500
14	+5500	9999+	5500+
15	+5500	ZZZ.99	500.00
16	123	$$$.$$	$23.00
17	123	$$$.99	$23.00
18	003	$$$.$$	$3.00
19	000	$$$.$$	
20	00345	$$$.$$	$45.00

TABLE 7-3
Examples of PICTURE Clauses and Conversions - Continued

Example	*Value*	*Edited PICTURE*	*Prints as*
21	0000	ZZ,ZZZ	
22	00345	ZZ,ZZZ	345
23	0000	$*****	
24	0000	$ZZZZZ	
25	00	$***.99	$***.00
26	00	ZZ,ZZZ.99	.00
27	0.03	ZZ,ZZZ.ZZ	.03
28	0.00	ZZ,ZZZ.ZZ	

- Numeric: The PICTURE *character-string* can contain only the symbols 9, P, S, and V. You can represent 1 through 18 digit positions with this PICTURE *character-string*; the contents of this field are restricted to a combination of the digits 0 through 9, plus an optional sign.
- Alphanumeric: The PICTURE *character-string* is a combination of the data description characters X, A, or 9, and the item is treated as if the string contained all Xs. Do not specify alphanumeric PICTURE *character-strings* with all 9s or all As. Item contents can be any character from the computer's ASCII character set defined in Table B-3.
- Alphanumeric edited: The PICTURE *character-string* is restricted to certain combinations of the following symbols: A, X, 9, B, 0, /. The section Symbol Functions, below, discusses allowable combinations. Contents of the field can be any character from the computer's ASCII character set defined in Table B-3.
- Numeric edited: The PICTURE *character-string* is a certain combination of the editing symbols Z . CR DB , + * B 0 – / 9 V P, and the currency symbol. It must contain at least one of the editing symbols Z . CR DB , + * B 0 – /. Field contents must be numerals. The maximum number of digit positions is 18.

Data Item Size: The size of an elementary item (the number of character positions occupied by the item in standard data format) is determined by the number of symbols that represent character positions, as listed below.

An integer enclosed in parentheses, following the symbols A X 9 P Z * B / 0 + – or the currency symbol, indicates the number of consecutive occurrences of that symbol. This is the **repeat integer**.

Symbol Functions: Symbols used in a PICTURE *character-string* to define an elementary item have the following functions:

Symbol	*Function*
A	Each A represents a character position containing either a letter of the alphabet in upper-case or lowercase, or a space.
B	Each B represents a character position into which a space character is inserted. The rules for simple insertion editing, listed in the next section, govern its use.

P — Each P indicates an assumed decimal scaling position. The P specifies the location of an assumed decimal point that does not appear in the data item. The P is not counted in the size of the data item, but is counted in determining the maximum number of digit positions in numeric edited items or numeric items.

In conversions, each digit position described by a P is considered to contain the value zero. The assumed decimal point is considered to be to the left of the P that is leftmost in a string, or to the right of Ps that are rightmost in a string. Thus the effect of each P is to divide or multiply a data item by one power of ten.

The scaling position character P can appear only to the left or right of the other characters in the string, except that the sign character S and the assumed decimal point V can appear to the left of a leftmost string of Ps. Because the character P implies an assumed decimal point, the symbol V is redundant as either the leftmost or rightmost character within such a PICTURE description.

The character P and the insertion character period (.) cannot both occur in the same *character-string*.

See the two nonedited examples in Table 7-3.

S — The *character-string* symbol S indicates the presence of a sign in a data item, but implies nothing about the actual format or location of the sign in storage.

The symbol S is not counted in determining the size of the elementary item, unless the entry is subject to a SIGN clause with the SEPARATE qualifier. (See SIGN.)

When you use the S symbol, you must write it as the leftmost character in a PICTURE *character-string*.

V — The character V indicates the position of an assumed decimal point. Because a numeric item cannot contain an actual decimal point, an assumed decimal point is used to provide information concerning the alignment of items involved in computations. The V does not represent a character position and, therefore, is not counted in the size of the item. Only one V is permitted in any single picture.

X — Each X represents a character position that can contain any allowable character from the computer's character set.

Z — Each Z character is a replacement character that represents a leading numeric position that is replaced by a space when its contents are zero. Each Z is counted in the size of the item. The rules for suppression and replacement editing, listed in the next section, govern its use.

9 — Each 9 in a PICTURE *character-string* represents a character position that contains a numeral and is counted in the size of the item.

0 — Each zero in the PICTURE *character-string* represents a character position into which the numeral 0 is inserted. The 0 is counted in the size of the item. The rules for simple insertion editing, listed in the next section, govern its use.

/ — Each slash mark in the PICTURE *character-string* represents a character position into which the slash character is inserted. The slash is counted in the size of the item. The rules for simple insertion editing, listed in the next section, govern its use.

, — The comma character specifies insertion of a comma between digits. Each such character is counted in the size of the data item, but does not represent a digit position. When DECIMAL-POINT IS COMMA is specified, the explanations for period and comma are reversed to apply to comma and period, respectively. The rules for simple insertion editing, listed in the next section, govern its use.

A period character in a PICTURE *character-string* is an editing symbol representing the decimal point for alignment purposes. The character also serves to indicate the position for decimal point insertion. The rules for special insertion editing, listed in the next section, govern its use.

Numeric character positions to the right of an actual decimal point in a PICTURE *character-string* must consist of characters of one type. The period character is counted in the size of the item. When DECIMAL-POINT IS COMMA is specified, the explanations for period and comma are understood to apply to comma and period, respectively.

The period character can be the last character in the PICTURE *character-string*.

+
–
CR
DB

These symbols are used as editing sign control symbols and represent the character position into which the editing sign control symbol is placed. The symbols are mutually exclusive in any one PICTURE *character-string*, and each character used in the symbol is counted in determining the size of the data item. That is, CR and DB require two character positions each; + and – require one character position each. The rules for fixed insertion editing and floating insertion editing, listed in the next section, govern their use.

*

Each asterisk in a PICTURE *character-string* is a replacement character. Leading zeros in the affected item are suppressed and replaced by asterisks. Each asterisk is counted in the size of the item. The rules for suppression and replacement editing, listed in the next section, govern its use.

$
cs

The dollar sign or other currency symbol represents a character position into which the currency symbol is placed. The currency symbol is the dollar sign or the character specified in the CURRENCY SIGN clause. It is counted in the size of the item. The rules for floating insertion editing, listed in the next section, govern its use.

Editing Rules

The rules below define the ways of **editing** a data field, that is, of adding, changing, or suppressing certain characters in it. Examples appear in Table 7-3.

1. The maximum length of an edited field is 255 characters. The maximum length of an edited PICTURE *character-string* is 32 characters.
2. The PICTURE clause provides two methods for editing: character insertion, and character suppression and replacement.

   The four types of insertion editing are

   Simple insertion
   Special insertion
   Fixed insertion
   Floating insertion

   The two types of suppression and replacement editing are

   Zero suppression and replacement with spaces
   Zero suppression and replacement with asterisks

   The type of editing that you can perform upon an item depends upon the category to which the item belongs, as defined in Chapter 4. Table 7-4 specifies which type of editing you can perform upon a given category.

TABLE 7-4
Categories of Data and Editing

Category of Data	*Type of Editing Allowed*
Alphabetic	Simple insertion (B only)
Numeric	None
Alphanumeric	None
Alphanumeric edited	Simple insertion (0 B and /)
Numeric edited	All, subject to rules for fixed insertion editing

3. **Simple insertion editing** uses the four symbols B 0 , / as insertion characters. The insertion characters are counted in the size of the item and represent the position in the item into which the character is inserted.

   If the insertion character comma (,) is the last symbol in the PICTURE *character-string*, the PICTURE clause must be the last clause of the data description entry and must be immediately followed by the separator period. This results in the combination ,. appearing in the *data-description-entry*. If you use the DECIMAL-POINT IS COMMA clause, this results in two consecutive periods appearing in the *data-description-entry*. See examples 4 through 7 for edited fields in Table 7-3.

4. **Special insertion editing** refers to decimal point insertion. The period is the insertion character. It also represents the decimal point for alignment purposes. The period is counted in the size of the item. Do not use the assumed decimal point, represented by the symbol V, and an actual decimal point, represented by the insertion character, in the same PICTURE *character-string*.

   If the insertion character is the last symbol in the PICTURE *character-string*, the PICTURE clause must be the last clause of that *data-description-entry* and must be immediately followed by the separator period. This results in two consecutive periods appearing in the *data-description-entry*. If you use the DECIMAL-POINT IS COMMA clause, this results in the combination ,. appearing in the *data-description-entry*. If you use special insertion editing, the insertion character appears in the item in the same position as in the PICTURE *character-string*. See examples 8 through 10 for edited fields in Table 7-3.

5. **Fixed insertion editing** uses the currency sign and editing sign control symbols as insertion characters. The four editing sign control symbols are + – CR DB.

   Use only one currency symbol and only one editing sign control symbol in a given PICTURE *character-string*. When the symbols CR or DB are used, they represent two character positions in determining the size of the item. They must represent the rightmost character positions to be counted in the size of the item. If you use the symbol + or -, it must be either the leftmost or rightmost character position to be counted in the size of the item. The currency symbol must be the leftmost character position to be counted in the size of the item, except that it can be preceded by either a + or a – symbol. If you use fixed insertion editing, the insertion character occupies the same character position in the edited item that it occupies in the PICTURE *character-string*. Editing sign control symbols produce the results shown in Table 7-5, depending upon the value of the data item.

TABLE 7-5
Results of Sign Control Symbols in Editing

	Result	
Editing Symbol in picture-string	*Data Item Positive or Zero*	*Data Item Negative*
+	+	–
–	space	–
CR	2 spaces	CR
DB	2 spaces	DB

See examples 11 through 14 for edited fields in Table 7-3.

6. **Floating insertion editing** uses **floating insertion characters**. These are the currency symbol and the editing sign control symbols + or –. As floating insertion characters, these are mutually exclusive in a given PICTURE *character-string*. These characters cause leading zeros to be replaced with blanks, except for the leftmost zero, which is replaced with the insertion character.

   A **floating string** is a leading, continuous series of $ + or –, or a string composed of one such character interrupted by one or more insertion commas and/or a decimal point. For example,

```
$$,$$$,$$$
++++
--,---,--
+(8).++
$$,$$$.$$$
```

   Indicate floating insertion editing in a PICTURE *character-string* by including in it a string of at least two of the floating insertion characters. This floating string can contain any of the fixed insertion symbols. The leftmost character of the floating insertion string represents the leftmost limit of the floating symbol in the data item. The rightmost character of the floating string represents the rightmost limit of the floating symbols in the data item.

   The second floating character from the left represents the leftmost limit of the numeric data that can be stored in the data item. Nonzero numeric data can replace all the characters to the right of this limit.

   Indicate floating insertion editing in a PICTURE *character-string* in one of the following ways:

   - Represent any or all of the leading numeric character positions on the left of the decimal point by the insertion character.
   - Represent all of the numeric character positions in the PICTURE *character-string* by the insertion character.

   See examples 15 through 16 for edited fields in Table 7-3.

If the insertion characters are only to the left of the decimal point in the PICTURE *character-string*, a single floating insertion character is placed in the character position immediately preceding the first nonzero digit in the data item. If all data item digits to the left of the decimal are zero, the floating insertion character is placed in the character position immediately preceding the decimal point. The character positions preceding the insertion character are replaced with spaces.

If you represent all numeric character positions in the PICTURE *character-string* by the insertion character, the result depends on the value of the data. If the value is zero, the entire data item contains spaces. If the value is not zero, the result is the same as if the insertion character were only to the left of the decimal point. See examples 17 through 19 for edited fields in Table 7-3.

To avoid truncation, the minimum size of the PICTURE *character-string* for the receiving data item must equal the sum of the number of characters in the sending data item, plus the number of nonfloating insertion characters being edited into the receiving data item, plus one for the floating insertion character. That is, to define *n* digit positions, a floating string must contain *n* + 1 occurrences of $ or + or –.

When a comma appears to the right of a floating string, the comma is not retained if no digits are retained before it. Table 7-6 provides examples.

TABLE 7-6
Commas in Floating Strings

Picture-string	*Numeric Value*	*Developed Item*
$$$999	14	$014
--,---,999	–456	–456
$$$$$$	14	$14

A floating string need not constitute the entire PICTURE *character-string* of a numeric edited item. However, the characters from the right of a decimal point to the end of the PICTURE *character-string*, excluding the fixed insertion characters +, –, CR, DB (if present), are subject to the following restrictions:

- Only one type of digit position character can appear. That is, the three characters Z * and 9 are mutually exclusive, and the floating-string digit position characters $ + and – are mutually exclusive.
- If you represent any of the numeric character positions to the right of a decimal point by + or – or $ or Z, then you must represent all of the numeric character positions in the PICTURE *character-string* by the same character.
- No symbol can precede a floating string except + or –.

7. **Suppression and replacement editing** includes two types: zero suppression and replacement with spaces, and zero suppression and replacement with asterisks.

   Floating insertion editing and editing by zero suppression and replacement are mutually exclusive in a PICTURE clause.

   Use the mutually exclusive suppression symbols Z or * in a PICTURE *character-string* to indicate the suppression of leading zeros in numeric character positions. Each suppression symbol is counted in determining the size of the item. If you use Z, the

replacement character is the space. If you use *, the replacement character is the asterisk. Any of the characters B 0 , / embedded in the string of suppression symbols, or to the immediate right of this string, is part of the string. See examples 20 through 24 for edited fields in Table 7-3.

Represent zero suppression in a PICTURE *character-string* in one of the following ways:

- Represent any or all leading numeric character positions to the left of the decimal point by suppression symbols.
- Represent all numeric character positions in the PICTURE *character-string* by suppression symbols.

If the suppression symbols appear only to the left of the decimal point, any leading zero in the data that corresponds to a symbol in the string is replaced by the replacement character. Suppression terminates either at the first nonzero digit in the data represented by the suppression symbol string, or at the decimal point, whichever is first.

If all numeric character positions in the PICTURE *character-string* are represented by suppression symbols, and the value of the data is not zero, the result is the same as if the suppression characters were only to the left of the decimal point. If the value is zero and the symbol is Z, the entire data item contains spaces. If the value is zero and the symbol is *, the entire data item (except for the actual decimal point) contains asterisks. See examples 25 through 27 for edited fields in Table 7-3.

8. The following symbols can appear only once in a given PICTURE *character-string*: S V . CR DB.

REDEFINES

The REDEFINES clause allows you to describe the same computer storage area with different *data-description-entries*. The clause is useful in table handling.

Format

level-number [***data-name-1*** / **FILLER**] [; REDEFINES ***data-name-2***]

Note

level-number, *data-name-1*, and the semicolon are not part of the REDEFINES clause, but are included to show the context.

Syntax Rules

1. The REDEFINES clause is optional; when you specify it, it must immediately follow *data-name-1* or FILLER. The entry identified by *data-name-1* must follow the entry identified by *data-name-2*.
2. *level-numbers* of *data-name-1* and *data-name-2* must be identical, but must not be 66 or 88.

3. **Prime Extension:** The entry for *data-name-1* can be separated from *data-name-2* by other data descriptions of the same level provided that the number of character positions in *data-name-2* is greater than or equal to the number of character positions in *data-name-1*. Otherwise, no restrictions apply to the lengths of the two entries.
4. Do not use the REDEFINES clause in level 01 entries in the FILE SECTION.
5. The *data-description-entry* for *data-name-2* can contain a REDEFINES clause.
6. The *data-description-entry* for *data-name-2* must not contain an OCCURS clause, and *data-name-1* must not be subordinate to an entry containing an OCCURS clause.
7. *data-name-2* can be qualified but not subscripted.
8. The entries for *data-name-1* and *data-name-2* cannot be separated by entries with lower *level-numbers*. In other words, they must be in the same group.

General Rules

1. Redefinition starts at *data-name-1* and ends when a *level-number* less than or equal to the *level-number* of *data-name-1* is encountered. In the following example, redefinition of FIELD-1 by FIELD-2 ends when FIELD-3 is encountered.

```
05  FIELD-1            PICTURE A(3).
05  FIELD-2 REDEFINES FIELD-1.
    10  ITEM-A         PICTURE A.
    10  ITEM-B         PICTURE AA.
05  FIELD-3            PICTURE X.
```

Figure 7-4 represents the projection of these two *data-names* on one storage area through the REDEFINES clause.

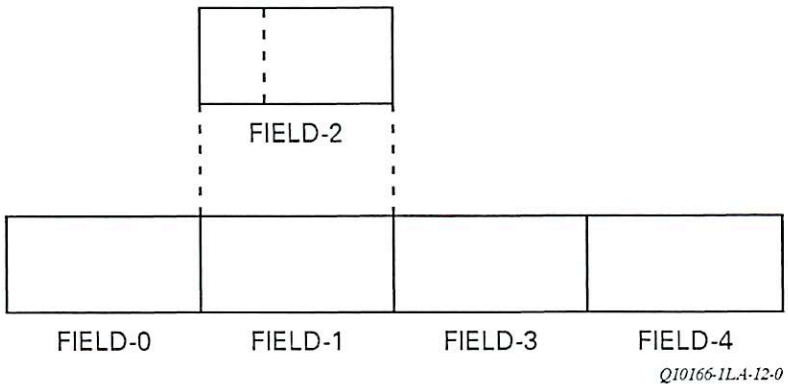


FIGURE 7-4
A Storage Area Redefined

2. The entries that specify the new description of the area must not contain VALUE clauses except in *condition-name* entries.
3. You can redefine to a depth greater than one level. Thus, the nested REDEFINES outlined below is valid:

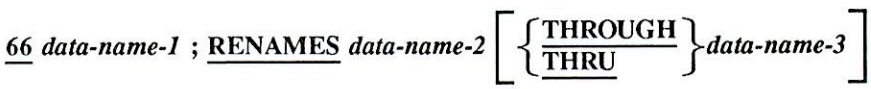
```
01  FIELD-A             PIC X(10).
01  FIELD-B  REDEFINES FIELD-A.
    05  FIELD-C         PIC X(5).
    05  FIELD-D  REDEFINES FIELD-C.
        10  FIELD-E1    PIC X(3).
        10  FIELD-E2    PIC X(2).
    05  FIELD-F         PIC X(5).
```

4. When you assign a value to one *data-name* for a redefined storage area, all *data-names* for that area have the same value.

RENAMES

The RENAMES clause allows you to define alternative, possibly overlapping, groups of elementary items.

Format



Note

level-number 66, *data-name-1*, and the semicolon are not part of the RENAMES clause, but are included to show the context.

Syntax Rules

1. You can write any number of RENAMES entries for a logical record. They must all immediately follow the last entry of that record.
2. *data-name-1* cannot be used as a qualifier but can itself be qualified by the 01 or FD entries. *data-name-2* and *data-name-3* can neither contain an OCCURS clause, nor be subordinate to an entry that has an OCCURS clause in its *data-description-entry*.
3. *data-name-2* and *data-name-3* must be the names of elementary items or groups of elementary items in the same record and cannot have the same *data-name*.
4. A level-66 entry cannot rename an entry having *level-number* 01, 66, 77, or 88.
5. The beginning of the area described by *data-name-3* must not be to the left of the beginning of the area described by *data-name-2*. The end of the area described by *data-name-3* must be to the right of the end of the area described by *data-name-2*. Therefore, *data-name-3* cannot be subordinate to *data-name-2*.
6. *data-name-2* and *data-name-3* can be qualified.
7. The words THRU and THROUGH are equivalent.

General Rules

1. When you specify *data-name-3*, *data-name-1* is a group item that includes all elementary items starting with *data-name-2* (if that is an elementary item) or the first elementary item in *data-name-2* (if *data-name-2* is a group item), and concluding with *data-name-3* (if that is an elementary item) or the last elementary item in *data-name-3* (if *data-name-3* is a group item).
2. When you do not specify *data-name-3*, *data-name-2* can be either a group or an elementary item. When *data-name-2* is a group item, *data-name-1* is treated as a group item, and when *data-name-2* is an elementary item, *data-name-1* is treated as an elementary item.

Example

In the following example, the 66-level name VENDOR-ENTRY establishes a new group item from a part of the elements in the record ENTRY-IMAGE. Note that in this example, RENAMES redefines two level-10 items followed by three level-05 items, while REDEFINES would have redefined only one elementary or one group item.

```
01  ENTRY-IMAGE.
    05 ENTRY-MONTH.
      10 DY                        PIC 99.
      10 MO                        PIC 99.
      10 YR                        PIC 99.
    05 ENTRY-VENDOR                PIC X(20).
    05 ENTRY-ACCT-NO               PIC 999.
    05 ENTRY-AMOUNT                PIC 9(5)V99.
66 VENDOR-ENTRY      RENAMES MO THRU ENTRY-AMOUNT.
```

SIGN

The SIGN clause specifies the position and the mode of representation of the operational sign when you need to describe these properties explicitly.

Format

[SIGN IS] $\left\{\begin{matrix}\text{LEADING}\\\text{TRAILING}\end{matrix}\right\}$ [SEPARATE CHARACTER]

Syntax Rules

1. Specify the SIGN clause only for a numeric data item whose PICTURE *character-string* contains the character S, or for a group item containing at least one such elementary item. If the PICTURE *character-string* does not contain an S, the item is considered unsigned (capable of storing only absolute values), and the SIGN clause is prohibited.
2. You must explicitly or implicitly describe as USAGE IS DISPLAY numeric data items to which the SIGN clause applies.
3. If you specify the CODE-SET clause in a *file-description-entry*, you must describe with the SIGN IS SEPARATE clause any signed numeric item associated with that file.

General Rules

1. When you specify S in a PICTURE *character-string*, but do not include the SIGN clause in an item's description, the default is SIGN IS TRAILING.
2. If you do not specify the optional SEPARATE CHARACTER phrase, then
   - The operational sign is presumed to be associated with the leading (or trailing) digit position of the elementary numeric data item.
   - The character S in the PICTURE *character-string* is not counted in determining item size.
3. If you include the SEPARATE CHARACTER phrase, then
   - The operational sign is presumed to be the leading (or trailing) character position of the elementary numeric data item; this character position is not a digit position.
   - The letter S in a PICTURE *character-string* is counted in determining the size of the item (in terms of standard data format characters).
   - The operational signs for positive and negative are the standard data format characters + and –, respectively.
4. Every numeric *data-description-entry* whose PICTURE *character-string* contains the character S is a signed numeric *data-description-entry*. If a SIGN clause applies to such an entry and conversion is necessary for purposes of computation or comparisons, conversion takes place automatically.
5. Table 7-7 depicts sign representations for the various SIGN clause options.
6. If you specify a SIGN clause in a group item subordinate to a group item for which you specify a SIGN clause, the SIGN clause you specify in the subordinate group item takes precedence for that subordinate group item.
7. If you specify a SIGN clause in an elementary numeric *data-description-entry* subordinate to a group item for which you specify a SIGN clause, the SIGN clause you specify in the subordinate elementary numeric *data-description-entry* takes precedence for that elementary numeric data item.

TABLE 7-7
Sign Representation

SIGN Clause	*Sign Representation*
TRAILING	Embedded in rightmost byte
LEADING	Embedded in leftmost byte
TRAILING SEPARATE	Stored in separate rightmost byte
LEADING SEPARATE	Stored in separate leftmost byte

See the section Data Representation and Alignment in Chapter 4 for a detailed description of SIGN formats and conventions.

SYNCHRONIZED

The SYNCHRONIZED clause specifies the alignment of an elementary item on its natural addressing boundaries in the computer's memory.

Format

```
{SYNCHRONIZED} [LEFT ]
{SYNC        } [RIGHT]
```

Syntax Rules

1. SYNC is an abbreviation for SYNCHRONIZED.
2. COBOL85 treats the SYNCHRONIZED clause as a comment.
3. COBOL85 automatically aligns group items and elementary items of certain data types. Level-77 and level-01 items are always aligned on halfword (16-bit) boundaries. Group items with *level-numbers* greater than 01 (subgroups) are aligned on halfwords if they contain fields that require such alignment (COMP, BINARY, COMP-1, and COMP-2 fields). The data map created with the –MAP compiler option flags items that are aligned by the compiler. In addition, the –SLACKBYTES compiler option issues an observational diagnostic for each compiler-aligned item.

   See the section Data Representation and Alignment in Chapter 4.

USAGE

The USAGE clause describes the form in which the compiler represents numeric data.

Format

```
[USAGE IS] { BINARY             }
           { COMPUTATIONAL      }
           { COMP               }
           { COMPUTATIONAL-1    }
           { COMP-1             }
           { COMPUTATIONAL-2    }
           { COMP-2             }
           { COMPUTATIONAL-3    }
           { COMP-3             }
           { DISPLAY            }
           { INDEX              }
           { PACKED-DECIMAL     }
```

Syntax Rules

1. COMP, COMP-1, COMP-2, and COMP-3 are abbreviations for COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2, and COMPUTATIONAL-3, respectively.
2. **Prime Extensions:** COMPUTATIONAL-1, COMPUTATIONAL-2, and COMPUTATIONAL-3 are Prime extensions to ANSI COBOL.
3. If you specify USAGE as COMPUTATIONAL-1, COMPUTATIONAL-2, or INDEX, you cannot use the PICTURE clause.
4. An elementary data item whose declaration contains, or an elementary data item subordinate to a group item whose declaration contains, a USAGE clause specifying BINARY, COMPUTATIONAL, COMPUTATIONAL-3, or PACKED-DECIMAL must be declared with a PICTURE *character-string* that describes a numeric item (that is, a PICTURE *character-string* that contains only the symbols P, S, V, and 9).
5. The USAGE IS INDEX clause describes an elementary item called an index data item. You can refer explicitly to an index data item only in a SEARCH or SET statement, a relation condition, the USING phrase of a PROCEDURE DIVISION header, or the USING phrase of a CALL statement.
6. You cannot use the SYNCHRONIZED, JUSTIFIED, PICTURE, VALUE, and BLANK WHEN ZERO clauses to describe group or elementary items that you describe with the USAGE IS INDEX clause.

General Rules

1. The USAGE IS BINARY or COMPUTATIONAL clause defines a binary item. COMPUTATIONAL-1 defines a single-precision floating-point number. COMPUTATIONAL-2 defines a double-precision floating-point number. The USAGE IS PACKED-DECIMAL or COMPUTATIONAL-3 clause defines a packed-decimal item. INDEX defines a binary item to be used for referencing tables. DISPLAY defines an item represented in external decimal format.

   The section Data Representation and Alignment in Chapter 4 discusses these items and their allowable PICTURE clauses. Also, the section SYNCHRONIZED, above, discusses alignment of COMP, BINARY, COMP-1, and COMP-2 items.
2. You can write the USAGE clause at any level. If you write the USAGE clause at a group level, it applies to each item in the group, including the elementary items within subgroups. The USAGE clause of an elementary item or subgroup cannot contradict the USAGE clause of a group item to which it belongs.
3. A COMPUTATIONAL, BINARY, COMPUTATIONAL-1, COMPUTATIONAL-2, COMPUTATIONAL-3, or PACKED-DECIMAL item can represent a value to be used in computations; therefore, the item must be numeric. When you specify one of these usages for a group item, only the elementary items in that group can be used in computations.
4. If you do not specify the USAGE clause, the system default is DISPLAY. You can use numeric display items in computations.
5. **Prime Extension:** If you specify USAGE as COMPUTATIONAL or BINARY for an item, and you do not include a PICTURE clause for that item, the compiler assumes a PICTURE of S9999 (16-bit signed binary integer).

6. COMP-1 and COMP-2 are intended for use in calling certain PRIMOS subroutines that require floating-point (real) arguments. They are also for use in scientific calculations that require a large range at the expense of absolute decimal precision.

   Use COMP, BINARY, PACKED-DECIMAL, and COMP-3 in most decimal calculations. COMP and BINARY allow the greatest efficiency both in storage space and in speed of calculations for integers. COMP-3 and PACKED-DECIMAL are more often used to save file and memory space. You can also use COMP and BINARY to specify scaling positions (for example, 99V99). However, such use may result in runtime inefficiency.

   DISPLAY is the only usage allowed for alphabetic characters and symbols. For numbers, DISPLAY is allowed in calculations but is less efficient than the other usages.

   INDEX is allowed only for a value to be used as the index to a table.

   If you mix any of these data types together in calculations, such a mixed calculation is allowed, but often at the expense of either precision or efficiency. For details, see the section Data Representation and Alignment in Chapter 4.

7. An index data item contains a value that must correspond to an occurrence number of a table element. The elementary item cannot be a conditional variable. If you describe a group item with the USAGE IS INDEX clause, the elementary items in the group are all index data items. The group itself is not an index data item, and you cannot use it in the SEARCH or SET statement or in a relation condition.
8. An index data item can be part of a group that you refer to in a MOVE or input-output statement, in which case no conversion takes place.
9. The format of the index data item is described in the section Data Representation and Alignment in Chapter 4. The maximum value of an index data item is listed in Appendix I.

VALUE

The VALUE clause defines the value of constants, the initial values of WORKING-STORAGE items, and the values associated with a *condition-name*.

Format 1

VALUE IS *literal*

Format 2

$$\left\{\begin{matrix}\text{VALUE IS}\\ \text{VALUES ARE}\end{matrix}\right\} \textit{literal-1} \left[\left\{\begin{matrix}\text{THROUGH}\\ \text{THRU}\end{matrix}\right\} \textit{literal-2}\right]$$

$$\left[, \textit{literal-3} \left[\left\{\begin{matrix}\text{THROUGH}\\ \text{THRU}\end{matrix}\right\} \textit{literal-4}\right]\right] \ldots$$

Syntax Rules

1. The words THROUGH and THRU are equivalent.

2. Do not use the VALUE clause in a *data-description-entry* that contains a REDEFINES clause, or that is subordinate to such an entry.
3. You can use a signed numeric literal in a VALUE clause only if the associated PICTURE *character-string* is a signed numeric item.
4. Numeric literals in a VALUE clause must have a value within the range of values indicated by the PICTURE clause, and must not have a value that would cause truncation of nonzero digits.

   Nonnumeric literals in a VALUE clause must not exceed the size indicated by the PICTURE clause.
5. The type of literal allowed in a VALUE clause depends on the type of data item, as specified in the PICTURE or USAGE clauses. For edited items, you must specify values as nonnumeric literals. A type conflict arises, producing a compile time error, if a figurative constant or literal is not compatible with the PICTURE clause. For example,

   ```
   PICTURE 9 VALUE 'A'
   ```

   produces a type conflict error, because 'A' is an alphabetic character and PICTURE 9 specifies a numeric item. Also, a size conflict produces a compile time error. For example,

   ```
   PICTURE X(2) VALUE 'ABCD'
   ```

   is invalid. The compiler issues a warning and truncates 'ABCD' to 'AB'.
6. Do not specify a VALUE clause in the FILE SECTION of the DATA DIVISION except in level-88 *condition-name* entries.
7. Edited elementary items with VALUE clauses are not initialized with their editing characteristics. For example,

   ```
   PICTURE ZZ,ZZ9.99 VALUE ZERO
   ```

   appears as all zeros with no suppression. However, a MOVE of ZERO to the data item achieves edited results.

   For an edited elementary item, you must express values in a VALUE clause as nonnumeric literals. For example,

   ```
   PICTURE ZZ,ZZ9.99 VALUE "   123.00"
   ```

General Rules

1. The VALUE clause must not conflict with other clauses in the data description of the item or in a data description within the hierarchy of the item.
2. Initialization occurs independently of any BLANK WHEN ZERO or JUSTIFIED clause that you specify.
3. You can specify the VALUE clause at the group level in the form of a correctly sized nonnumeric literal, or a figurative constant.
4. You can specify a figurative constant instead of a literal in both Format 1 and Format 2.
5. Format 1 is required to define an initial value for a data item or a constant.

6. Use Format 2 only for *condition-name* entries (level-88 items). The VALUE clause and the level-88 *condition-name* itself are the only two items permitted in the entry. The characteristics of a *condition-name* are implicitly those of its conditional variable. Wherever you use the THRU phrase, *literal-1* must be less than *literal-2*, *literal-3* less than *literal-4*, and so on.

   Level-88 specifications can contain individual values, series of individual values, a range of values, or a series of ranges of values. (See also *level-number*.)

7. Rules governing the VALUE clause differ in the respective sections of the DATA DIVISION as follows:

   - In the FILE SECTION, use the clause only in *condition-name* entries.
   - In the WORKING-STORAGE SECTION and in the LINKAGE SECTION, you must use the clause in *condition-name* entries. You can also use it in the WORKING-STORAGE SECTION to specify the initial value of any other data item, with the result that the item assumes the specified value at the start of the object program.
   - In the WORKING-STORAGE SECTION, do not use the VALUE clause in any *data-description-entry* that includes or is subordinate to any entry that includes the EXTERNAL clause. You can, however, specify the VALUE clause for *condition-name* entries associated with such *data-description-entries*.
   - When you do not specify an initial value, make no assumption regarding the initial contents of an item in WORKING-STORAGE.

8. If you use the VALUE clause in an entry at the group level, the literal must be a figurative constant or a nonnumeric literal. The group area is initialized without consideration for the individual elementary or group items contained within this group. You cannot specify a VALUE clause at the subordinate levels within this group.
9. Do not specify the VALUE clause for a group containing items with descriptions including JUSTIFIED, SYNCHRONIZED, or USAGE other than USAGE IS DISPLAY.
10. If you specify a VALUE clause in a *data-description-entry* of a data item that is associated with a variable occurrence data item, the data item is initialized as if the value of the data item referenced by the DEPENDING ON phrase in the OCCURS clause specified for the variable occurrence data item is set to the maximum number of occurrences specified by that OCCURS clause. A data item is associated with a variable occurence data item if it is one of the following:

    - A group data item that contains a variable occurrence data item
    - A variable occurrence data item
    - A data item that is subordinate to a variable occurrence data item

    If a VALUE clause is associated with the data item referenced by a DEPENDING ON phrase, that value is placed in the data item after the variable occurrence data item is initialized. See the OCCURS clause for additional information.

11. A Format 1 VALUE clause specified in a *data-description-entry* that contains an OCCURS clause or in an entry that is subordinate to an OCCURS clause causes every occurrence of the associated data item to be assigned the specified value.

Note

Use the VALUE clause or PROCEDURE DIVISION statements to initialize all WORKING-STORAGE data items before using them. Unexpected values may appear in uninitialized data items.

DATA DIVISION Example

This DATA DIVISION listing forms one program with the examples at the end of Chapters 5, 6, and 8.

```
 DATA DIVISION.
*
 FILE SECTION.
*
 FD  DISK-FILE COMPRESSED,
      VALUE OF FILE-ID IS 'DISBURSE',
     RECORD CONTAINS 42,
     DATA RECORD IS ENTRY-DETAIL.
 01  ENTRY-DETAIL.
     05 ENTRY-CHECK-NO           PIC X(3).
     05 ENTRY-MONTH.
        10 ENTRY-MM              PIC 99.
        10 ENTRY-DD              PIC 99.
        10 ENTRY-YY              PIC 99.
     05  FILLER                  PIC XXX.
     05  ENTRY-VENDOR            PIC X(20).
     05  ENTRY-ACCT-NO           PIC 999.
     05  ENTRY-AMOUNT            PIC 9(5)V99.
*
 FD  PRINT-FILE,
     LABEL RECORDS ARE OMITTED,
     DATA RECORDS ARE PRINT-LINE, ERROR-LINE.
 01  PRINT-LINE                  PIC X(70).
 01  ERROR-LINE.
     05 MESSAGE                  PIC X(20).
     05 ERR-CODE                 PIC 9.
*
 FD  TAPE-FILE,
     LABEL RECORD IS STANDARD,
     BLOCK CONTAINS 4 RECORDS,
     VALUE OF FILE-ID IS TAPENAME,
     DATA RECORD IS TAPE-LINE.
*
 01  TAPE-LINE                   PIC X(20).
*
 WORKING-STORAGE SECTION.
 77  BLANKS                  PIC X(2)              VALUE ZERO.
 77  CROSS-TOTAL             PIC S9(8)V99   COMP-3 VALUE ZERO.
 77  FINAL-TOTAL             PIC S9(8)V99   COMP-3 VALUE ZERO.
```

```
 77  GRAND-TOTAL              PIC S9(8)V99    COMP-3 VALUE ZERO.
 77  JOB-DATE                 PIC 9(6)               VALUE ZERO.
 77  LIMIT-DATE               PIC S9(5).
 77  LINECOUNT                PIC S99         COMP-3 VALUE ZERO.
 77  NO-MORE-RECORDS          PIC X                  VALUE 'N'.
 77  PAGECOUNT                PIC S9          COMP-3 VALUE 1.
 77  REJECT-TOTAL             PIC S9(7)V99    COMP-3 VALUE ZERO.
 77  TAPE-CHOICE              PIC XXX                VALUE 'NO '.
 77  TAPENAME                 PIC X(20)
           VALUE IS '$MT0, S, ANNE, T1'.
 77  TOTAL1                   PIC S9(7)V99    COMP-3 VALUE ZERO.
 77  TOTAL2                   PIC S9(7)V99    COMP-3 VALUE ZERO.
 77  TOTAL3                   PIC S9(7)V99    COMP-3 VALUE ZERO.
 77  TOTAL4                   PIC S9(7)V99    COMP-3 VALUE ZERO.
 77  TOTAL5                   PIC S9(7)V99    COMP-3 VALUE ZERO.
 77  TOTAL6                   PIC S9(7)V99    COMP-3 VALUE ZERO.
 77        VARIABLE                 PIC S9           VALUE 2.
*JOB-INFO IS ACCEPTED FROM CONSOLE:
 01  JOB-INFO.
     03 JOB-CODE              PIC XX.
       88 CORRECT-CODE        VALUE '25'.
*****************************************************************
*                PRINT-LINES
*****************************************************************
 01  HEADING1.
     03 CARRIAGE-CONTROL          PIC X.
     03 FILLER                    PIC X(18)      VALUE SPACES.
     03 FILLER                    PIC X(34)
          VALUE 'MONTHLY CASH DISBURSEMENTS JOURNAL'.
     03 FILLER                    PIC X(12)      VALUE SPACES.
 01  HEADING2.
     03 CARRIAGE-CONTROL          PIC X.
     03 FILLER                    PIC X(25)      VALUE SPACES.
     03 FILLER                    PIC X(7)       VALUE 'FOR    '.
     03 VARIABLE-MONTH            PIC X(15).
     03 FILLER                    PIC X(15)      VALUE SPACES.
     03 FILLER                    PIC X(4)       VALUE 'PAGE'.
     03 HEADING-PAGE              PIC ZZ9.
 01  HEADING3.
     03 CARRIAGE-CONTROL          PIC X.
     03 FILLER                    PIC X(24)      VALUE SPACES.
     03 VARIABLE-HEADING          PIC X(24)      VALUE SPACES.
     03 FILLER                    PIC X          VALUE SPACES.
 01  PRINT-DETAIL.
     05 FILLER                    PIC X(1)       VALUE SPACES.
     05 ENTRY-MONTH               PIC 9(6).
     05 FILLER                    PIC X(6)       VALUE SPACES.
     05 ENTRY-VENDOR              PIC X(20).
     05 FILLER                    PIC X(7)       VALUE SPACES.
     05 ENTRY-CHECK-NO            PIC X(3).
     05 FILLER                    PIC X(9)       VALUE SPACES.
```

```
     05 PRINT-ACCT-NO            PIC X(3).
     05 FILLER                   PIC X(6)       VALUE SPACES.
     05 PRINT-AMOUNT             PIC ZZZZZZ.99.
 01  HOME-ACCT-LINE.
     05 FILLER                   PIC X(13)      VALUE SPACES.
     05 HOME-NUMBER              PIC X(21).
     05 HOME-TOTAL               PIC Z(8).99.
 01  BALANCE-LINE.
     05 FILLER                   PIC X(9)       VALUE SPACES.
     05 FIELD-TOTAL              PIC Z(8).99.
     05 FILLER                   PIC X(8)       VALUE SPACES.
     05 FIELD-REJECT             PIC Z(8).99.
     05 FILLER                   PIC X(9)       VALUE SPACES.
     05 FIELD-DIFF               PIC Z(8).99.
     05 FILLER                   PIC X(11)      VALUE SPACES.
*****************************************************************
*       TAPE OUTPUT
*****************************************************************
 01  TAPE-HEADER.
     05 TAPE-MONTH               PIC X(15)      VALUE SPACES.
     05 FILLER                   PIC X(5)       VALUE SPACES.
 01  SAVE-TAPE.
     05 SAVE-DATE-TAPE           PIC 9(6).
     05 SAVE-ACCT-TAPE           PIC XXX.
     05 SAVE-TOTAL-TAPE          PIC S9(9)V99     COMP-3.
     EJECT
*****************************************************************
```

8 The PROCEDURE DIVISION

This chapter discusses the last of the four major divisions of the COBOL85 program, the PROCEDURE DIVISION. The chapter describes declarative sections, which allow you to provide error handling procedures for I-O operations, and scope terminators, which delimit the scope of PROCEDURE DIVISION statements. It also describes the special rules governing arithmetic statements and discusses all COBOL85 verbs in alphabetical order. The chapter concludes with an example of the PROCEDURE DIVISION.

PROCEDURE DIVISION

The PROCEDURE DIVISION contains instructions that specify the steps that the program performs. You can divide the PROCEDURE DIVISION into sections headed by *section-names*. You can further divide each section into paragraphs headed by *paragraph-names*. You can then write your program instructions as COBOL85 statements or sentences.

Format

```
PROCEDURE DIVISION [USING data-name-1 [, data-name-2] . . . [data-name-64]] .

[DECLARATIVES.
 { section-name SECTION [segment-number]. USE-sentence.
 { [paragraph-name. [sentence] . . . ] . . .                } . . .

 END DECLARATIVES.]

{   [section-name SECTION [segment-number].]
{ { [paragraph-name. [sentence] . . .] . . . }
{ { [sentence] . . .                         } } . . .
```

Syntax Rules

1. The PROCEDURE DIVISION is optional. When you include the PROCEDURE DIVISION, the first entry must be the words PROCEDURE DIVISION, followed by a period and a space unless you also include the USING clause.
2. Specify the USING clause only if both of the following occur:
   - The program is a CALLable subprogram that is to function under the control of a CALL statement in another program.
   - The CALL statement in the calling program contains a USING clause.
3. You must define each *data-name* operand in the USING clause as a data item in the LINKAGE SECTION of the program.
4. The program processes LINKAGE SECTION data items according to their data descriptions given in the program itself.
5. *level-numbers* of *data-names* in the USING clause must be 01 or 77.

   Prime Extension: *data-names* in the USING clause can contain a REDEFINES clause and can be redefined by other entries within the LINKAGE SECTION.

   See Chapter 13, Interprogram Communication, for a complete discussion.
6. Declarative sections are optional. When you include them, you must group them at the beginning of the PROCEDURE DIVISION. You must precede them by the keyword DECLARATIVES on a separate line and follow them by the keywords END DECLARATIVES on a separate line.
7. A declarative section must have a section header. A section header must consist of a *section-name*, followed by the word SECTION, an optional *segment-number*, and a period. Each *section-name* must appear on a line by itself; each *section-name* must be unique.
8. A section is an entry consisting of zero or more paragraphs, preceded by a section header. A section can consist entirely of sentences if no paragraph headers exist in the section. A section can be empty. A section must not contain a combination of sentences without paragraph headers and sentences with paragraph headers.
9. A paragraph is an entry consisting of zero or more sentences, preceded by a *paragraph-name*.
10. Within a section, sentences can appear without paragraph headers, providing no paragraphs exist in the section. Within the PROCEDURE DIVISION, sentences can appear without paragraph headers, providing no sections exist in the PROCEDURE DIVISION.
11. Within the PROCEDURE DIVISION, paragraphs can appear without section headers, providing no sections exist in the PROCEDURE DIVISION.
12. *segment-numbers* must be integers from 0 to 99. See SEGMENT-LIMIT entry in Chapter 6.
13. *paragraph-names* and *section-names* follow the general rules for word formation in Chapter 4. These names can be all numeric.
14. A **sentence** is a single statement or a series of statements terminated by a period and followed by a space.

15. A *statement* consists of a COBOL85 verb followed by appropriate operands (*literals* or *data-names*) and other clauses necessary for the completion of the statement. Statements are classed as imperative, conditional, and compiler-directing.

    Imperative statements specify one of the following:

    - An unconditional action to be taken by the object program. An unconditional action can be
        - An arithmetic statement without the ON SIZE ERROR or NOT ON SIZE ERROR phrase
        - An I-O statement without the INVALID KEY, NOT INVALID KEY, AT END, or NOT AT END clause
        - A STRING, UNSTRING, or CALL statement without the ON OVERFLOW or NOT ON OVERFLOW clause
    - A conditional statement that includes its explicit scope terminator. See the section Scope Terminators, below, for more information.

    Conditional statements test for conditions that determine whether the program flow takes an alternate path. See Chapter 4 for rules governing conditional statements.

    Conditional statements are the following:

    - IF, SEARCH, and RETURN with the AT END or NOT AT END clause
    - READ with the AT END, NOT AT END, INVALID KEY, or NOT INVALID KEY clause
    - WRITE, DELETE, REWRITE, and START with the INVALID KEY or NOT INVALID KEY clause
    - Arithmetic statements (ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT) with the ON SIZE ERROR or NOT ON SIZE ERROR phrase
    - STRING and UNSTRING with the ON OVERFLOW or NOT ON OVERFLOW clause
    - CALL with the ON OVERFLOW clause

    Compiler-directing statements cause the compiler to perform an action but have no effect on execution of the object program. USE, EJECT, and SKIP are directives to the compiler.

16. The only limit on code size in the PROCEDURE DIVISION is the amount of dynamic space allocated to the user by the System Administrator. The upper limit is 256 segments.

Note

If your COBOL85 program is larger than one segment, you must use BIND to link and execute it. If your program is less than one segment, you can use either BIND or SEG.

Declarative Sections

The sections under the DECLARATIVES header provide procedures that the program invokes when an I-O condition occurs that the program does not otherwise handle.

Because the program invokes such procedures only at the time an error occurs in an I-O operation, these procedures must not appear in the regular sequence of procedural statements. Instead, you must group them within the declarative section. You must group together error-handling procedures for each file and precede them by a separate USE sentence.

For additional information, see the USE statement in this chapter, and the example at the end of this chapter.

Format

DECLARATIVES.

{ *section-name* SECTION [*segment-number*]. USE-*sentence*.
  [*paragraph-name*. [*sentence*] ...] ... } ...

END DECLARATIVES.

Syntax Rules

1. Each declarative section includes a section header, a USE sentence, and, optionally, one or more paragraphs.
2. END DECLARATIVES must be followed by a period.

Scope Terminators

Scope terminators delimit the scope of PROCEDURE DIVISION statements. The two types of scope terminators are explicit and implicit.

Explicit Scope Terminators

You can terminate certain PROCEDURE DIVISION statements by using the following explicit scope terminators:

END-ADD	END-MULTIPLY	END-START
END-CALL	END-PERFORM	END-STRING
END-COMPUTE	END-READ	END-SUBTRACT
END-DELETE	END-RETURN	END-UNSTRING
END-DIVIDE	END-REWRITE	END-WRITE
END-IF	END-SEARCH	

Statements that include their explicit scope terminators are called **delimited scope statements**. When you nest a delimited scope statement within another delimited scope statement containing the same verb, each explicit scope terminator terminates the statement begun by the most recent and as yet unterminated occurrence of that verb.

In the following example, the END-ADD statement that immediately follows *imperative-statement-1* explicitly terminates the scope of the nested ADD statement. Likewise, the END-ADD statement that immediately follows *imperative-statement-2* explicitly terminates the scope of the containing ADD statement.

```
ADD data-name-1 TO data-name-2
    ON SIZE ERROR
       ADD data-name-3 TO data-name-4
           NOT ON SIZE ERROR
               imperative-statement-1
       END-ADD
    NOT ON SIZE ERROR
        imperative-statement-2
END-ADD.
```

Implicit Scope Terminators

You can terminate any PROCEDURE DIVISION statement by using either of the following implicit scope terminators:

- At the end of any sentence, the separator period terminates the scope of all previous statements not yet terminated either explicitly or implicitly.

  For example, in the following code the period implicitly terminates the READ and the IF statements.

  ```
  IF condition
     READ filename
          AT END
             imperative-statement-1.
  ```

  However, in the following example the period implicitly terminates only the IF statement. The END-READ explicitly terminates the READ statement.

  ```
  IF condition
     READ filename
          AT END
             imperative-statement-1
     END-READ
     imperative-statement-2.
  ```

- When you nest any statement within another statement, the next clause or phrase of the containing statement that follows the nested statement terminates the scope of any unterminated nested statement.

  For example, in the following code the ELSE clause implicitly terminates the READ statement.

  ```
  IF condition
     READ filename
          AT END
             imperative-statement-1
  ELSE
     imperative-statement-2.
  ```

Likewise, in the following example the NOT INVALID KEY phrase implicitly terminates the ADD statement.

```
READ filename-2
     INVALID KEY
        ADD data-name-1 TO data-name-2
     NOT INVALID KEY
        imperative-statement-2.
```

When you nest statements within statements that allow optional conditional phrases, COBOL85 considers any optional conditional phrase that it encounters as the next phrase of the most recent unterminated statement with which that phrase can be associated, but with which no such phrase has already been associated.

For example, in the following code COBOL85 considers the NOT ON SIZE ERROR phrase as the next phrase of the nested ADD statement.

```
ADD a to b
    ON SIZE ERROR
       ADD a to c
           NOT ON SIZE ERROR
               .
               .
               .
```

Arithmetic Statements in the PROCEDURE DIVISION

The five arithmetic verbs are ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE. The following rules govern arithmetic statements in the PROCEDURE DIVISION:

1. All *data-names* used in arithmetic statements must be elementary numeric data items defined in the DATA DIVISION of the program. However, when they are the operands of the GIVING phrase, they can be numeric edited. *index-names* and index items are not permitted in arithmetic statements.
2. All literals used in arithmetic statements must be numeric literals.
3. COBOL85 provides decimal-point alignment automatically throughout arithmetic computations.
4. The maximum size of each operand is 18 decimal digits.
5. The **composite of operands** for an arithmetic statement is a hypothetical data item resulting from superimposition of its operands aligned on their decimal points. For example, 12345678.9 and 1.23456789, superimposed, form a composite having 16 digits. No composite of operands can contain more than 18 decimal digits unless you use the composite with the COMPUTE statement or the GIVING phrase.

   Figure 8-1 shows the composite of the operands 12345678.9, 1.23456789, and 1234.56.

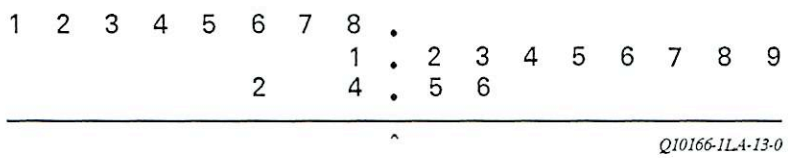


FIGURE 8-1
Composite of Operands

The six phrases that can appear in arithmetic statements are

- GIVING
- ROUNDED
- ON SIZE ERROR
- NOT ON SIZE ERROR
- CORRESPONDING
- Arithmetic scope terminator

The GIVING Phrase

When you specify the GIVING phrase, COBOL85 places the calculated result of the arithmetic operation into the *data-name* that follows the word GIVING. The *data-name* itself is not used in the computation and can be a numeric edited item. Do not use the GIVING phrase with the COMPUTE statement.

The ROUNDED Phrase

When you specify the ROUNDED phrase, if the most significant digit of the excess is greater than or equal to 5, COBOL85 increases the value of the least significant digit of the resultant *data-name* by 1. If you do not specify the ROUNDED phrase, truncation, and hence loss of precision, may occur.

COBOL85 rounds a computed negative result by rounding the absolute value of the computed result and making the rounded result negative.

Table 8-1 illustrates the relationship between a calculated result and the value stored in a receiving data item, with and without rounding.

TABLE 8-1
Rounding Results

	Item to Receive Calculated Result		
Calculated Result	*PICTURE*	*Value After Rounding*	*Value After Truncating*
–12.36	S99V9	–12.4	–12.3
8.432	9V9	8.4	8.4
35.6	99V9	35.6	35.6
65.6	S99V	66	65
.0055	SV999	.006	.005

The ON SIZE ERROR Phrase

Use the ON SIZE ERROR phrase to specify actions to be taken should a size error occur during an arithmetic operation. Write the ON SIZE ERROR phrase immediately after any arithmetic statement, as an extension of the statement.

If, after decimal-point alignment, the absolute value of a calculated result exceeds the largest value that the receiving field can hold, a size error condition exists.

Division by 0 always causes a size error condition and always terminates the arithmetic operation.

If you specify the ROUNDED phrase, rounding takes place before COBOL85 checks for size errors.

If you specify the ON SIZE ERROR phrase, and a size error condition arises, the value of the receiving *data-name* is unaltered, and COBOL85 executes the series of *imperative-statements* that you specified for the condition.

If you do not specify the ON SIZE ERROR phrase and a size error condition arises, the final result is undefined. Truncation usually results.

If you use the –SIGNALERRORS compiler option, program execution terminates when a size error condition exists.

An example of an ON SIZE ERROR phrase is

```
ADD 1 TO RECORD-COUNT
  ON SIZE ERROR
    MOVE ZERO TO RECORD-COUNT
    DISPLAY "LIMIT 99 EXCEEDED".
```

Assuming that you defined RECORD-COUNT as PICTURE 99, the MOVE and DISPLAY statements are not executed until RECORD-COUNT contains the value 99 and the ADD statement is executed. At that point a size error condition exists, and the MOVE and DISPLAY statements are executed.

The NOT ON SIZE ERROR Phrase

Use the NOT ON SIZE ERROR phrase to specify actions to be taken when a size error does not occur during an arithmetic operation. Write the NOT ON SIZE ERROR phrase immediately after any arithmetic statement, as an extension of the statement.

If the size error condition does not exist after the execution of the arithmetic operations specified by an arithmetic statement, COBOL85 ignores the ON SIZE ERROR phrase, if you specified one. COBOL85 then transfers control to the end of the arithmetic statement or to the *imperative-statements* in the NOT ON SIZE ERROR phrase, if you specified one.

The CORRESPONDING Phrase

You can use the CORRESPONDING phrase with all arithmetic statements, and with MOVE and IF statements. The CORRESPONDING phrase requires two operands, **group-1** and

group-2, each of which must refer to a group item. A pair of data items, one from group-1 and one from group-2, correspond if the following three conditions exist:

- A data item in group-1 and a data item in group-2 are not designated by the keyword FILLER and have the same *data-name* and the same qualifiers up to, but not including, group-1 and group-2.
- In arithmetic statements both data items are elementary numeric data items. In a MOVE or COMPUTE statement at least one data item is an elementary data item. (In IF statements, both items can be elementary or nonelementary.)
- The description of group-1 and group-2 does not contain *level-number* 66, 77, or 88 or the USAGE IS INDEX clause.

A data item that is subordinate to group-1 or group-2 and that contains a REDEFINES, RENAMES, OCCURS, or USAGE IS INDEX clause is ignored, as well as any items subordinate to it. However, group-1 and group-2 can have REDEFINES or OCCURS clauses or be subordinate to data items with REDEFINES or OCCURS clauses.

If you specify the ROUNDED phrase with the CORRESPONDING phrase, rounding as described above is performed on each matched receiving operand.

If you specify the ON SIZE ERROR phrase in conjunction with the CORRESPONDING phrase, COBOL85 checks for size errors for each pair of operands that are matched under the rules for CORRESPONDING, regardless of the results of any previous size error calculations in the statement. COBOL85 executes the *imperative-statement* that you specified in the ON SIZE ERROR phrase if any matching pairs of operands cause a size error.

Arithmetic Scope Terminators

You can explicitly terminate an arithmetic statement with one of the arithmetic scope terminators. The arithmetic scope terminators are END-ADD, END-SUBTRACT, END-MULTIPLY, END-DIVIDE, and END-COMPUTE. See the section Scope Terminators at the beginning of this chapter for more information.

Procedure Statements

This section describes COBOL85 verbs in alphabetical order. For a complete list of COBOL85 verbs and other reserved words, see COBOL85 Reserved Words, Table B-2 in Appendix B.

ACCEPT

Moves terminal or system data to the specified *data-name*.

Format 1

ACCEPT *data-name* [FROM *mnemonic-name*]

Format 2

ACCEPT *data-name* FROM { DATE / DAY / TIME }

Syntax Rules

1. You must specify the *mnemonic-name* in Format 1 with the CONSOLE IS clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.
2. Only one *data-name* is allowed after ACCEPT.

General Rules

1. The ACCEPT statement transfers data from the terminal or from the system clock. The transferred data replaces the contents of the field specified by *data-name*.
2. Execution of a Format 1 ACCEPT statement consists of the following steps:

   - Execution is suspended.
   - When you enter a carriage return, the program stores the keyed-in data preceding the carriage return in the field designated by *data-name*, and normal execution proceeds.
   - COBOL85 always considers terminal input as alphanumeric, and transfers it without conversion.
   - If the terminal input and *data-name* have unequal sizes, COBOL85 treats the data transfer as an alphanumeric to alphanumeric move, with space-fill on the right or right truncation.

3. The ACCEPT statement can transfer a single line of data with a maximum length of 256 characters.
4. A Format 2 ACCEPT statement transfers the requested information to the data item specified by *data-name* according to the rules of the MOVE statement. DATE, DAY, and TIME are reserved words; do not describe them in the COBOL85 program.
5. DATE has the following data elements: year, month, and day of the month, in that sequence. Thus, July 1, 1988 is expressed as 880701. DATE, when accessed by a COBOL85 program, is treated as though you describe it in the program as an unsigned elementary numeric integer data item six digits long.
6. DAY has the following data elements: year, and day of year, in that sequence. July 1, 1988 is expressed as 88183. DAY, when accessed by a COBOL85 program, is treated as though you describe it in the COBOL85 program as an unsigned elementary numeric integer data item five digits long.
7. TIME has the following data elements: hours, minutes, seconds, and hundredths of a second. TIME is based on time elapsed after midnight on a 24-hour basis; thus, 2:41 p.m., or 1441 hours, is expressed as 14410000. TIME, when accessed by a COBOL85 program, is treated as though you describe it in the program as an unsigned elementary numeric integer data item eight digits long. The minimum value of TIME is 00000000; the maximum value is 23595999.

Note

The ACCEPT statement does not perform functions declared in the definition of the *data-name*, such as BLANK WHEN ZERO or JUSTIFIED. All input, including numbers, is left-justified. When accepting numbers for calculations, use UNSTRING and INSPECT to prepare the data before doing calculations. The following example demonstrates this procedure.

Example

```
 ID DIVISION.
 PROGRAM-ID. CALC.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 DISPLAY-TOTAL                    PIC X(8).
 01 WORK-TOTAL                       PIC X(8) JUSTIFIED RIGHT.
 01 TOTAL-WORK                       PIC S9(6)V99.
*
 PROCEDURE DIVISION.
 000-INITIALIZE.
     DISPLAY 'WHAT IS INITIAL VALUE OF TOTAL?'.
     DISPLAY '      **  NOTE FORMAT MUST NOT USE DECIMAL POINT.'
     DISPLAY '      **  EX: TO REGISTER $45.25, ENTER 4525.'.
     ACCEPT DISPLAY-TOTAL.
     UNSTRING DISPLAY-TOTAL DELIMITED BY SPACE INTO WORK-TOTAL.
     INSPECT WORK-TOTAL REPLACING LEADING SPACES BY ZEROS.
     IF WORK-TOTAL NUMERIC
        MOVE WORK-TOTAL TO TOTAL-WORK
        DIVIDE 100 INTO TOTAL-WORK
     ELSE
        DISPLAY "INVALID ENTRY".
```

ADD

Adds two or more numeric values and stores the resulting sum.

Format 1

```
ADD { data-name-1   } [ , data-name-2   ] ...
    { literal-1     }   , literal-2
    { arith-expr-1  }   , arith-expr-2

TO data-name-3 [ROUNDED] [, data-name-n [ROUNDED]] ...

[ON SIZE ERROR imperative-statement-1]

[NOT ON SIZE ERROR imperative-statement-2]

[END-ADD]
```

Format 2

```
ADD { data-name-1 } [ , data-name-2 ] ... TO { data-name-3 }
    { literal-1   }   , literal-2           { literal-3   }
    { arith-expr-1}   , arith-expr-2        { arith-expr-3}

GIVING {data-name-4 [ROUNDED]} [, data-name-n [ROUNDED]] ...

[ON SIZE ERROR imperative-statement-1]

[NOT ON SIZE ERROR imperative-statement-2]

[END-ADD]
```

Format 3

```
ADD { CORRESPONDING } data-name-1 TO data-name-2 [ROUNDED]
    { CORR          }

[ON SIZE ERROR imperative-statement-1]

[NOT ON SIZE ERROR imperative-statement-2]

[END-ADD]
```

Syntax Rules

1. In Formats 1 and 2, each *data-name* must refer to an elementary numeric item, except that in Format 2 each item following GIVING can be either an elementary numeric item or an elementary numeric edited item.
2. Each literal must be a numeric literal.
3. The maximum size of each operand is 18 digits. That is, if all operands, excluding those following the word GIVING, are superimposed upon each other, and aligned by their implied decimal points, their composite must not exceed 18 decimal digits in length.
4. In Format 3, *data-name-1* and *data-name-2* must be group items.
5. **Prime Extension**: The use of arithmetic expressions in ADD statements is a Prime extension.

General Rules

1. A Format 1 ADD statement adds the sum of the values of the operands preceding the word TO to the current value of *data-name-3*, and stores the result in *data-name-3*. COBOL85 repeats this process for each operand following TO.
2. A Format 2 ADD statement adds the values of the operands preceding the word GIVING, and stores the sum as the new value of *data-name-4*.

3. A Format 3 ADD statement adds elementary items subordinate to *data-name-1* to elementary items subordinate to *data-name-2* that are defined with the same name, and stores the sums as the new values of the corresponding elementary items subordinate to *data-name-2*.
4. Use the ON SIZE ERROR phrase when the calculated answer can be larger than the result field can hold.
5. See the section Algebraic Signs in Chapter 4 for information on the rules for signs.
6. The ADD statement is governed by the rules for GIVING, ROUNDED, ON SIZE ERROR, NOT ON SIZE ERROR, and CORRESPONDING in the section Arithmetic Statements in the PROCEDURE DIVISION at the beginning of this chapter, and by the rules for arithmetic statements in Chapter 4.
7. The END-ADD clause delimits the scope of the ADD statement. For more information, see the section Scope Terminators at the beginning of this chapter.

Examples

```
ADD INTEREST, DEPOSIT TO BALANCE ROUNDED.
ADD REGULAR-TIME, OVERTIME GIVING GROSS-PAY.
ADD CORRESPONDING DETAIL-LINE TO TOTAL-LINE.
ADD INTEREST, DEPOSIT TO BALANCE
    ON SIZE ERROR
       DISPLAY 'SIZE ERROR'
    NOT ON SIZE ERROR
       MOVE BALANCE TO NEW-BALANCE
END-ADD.
```

The first statement results in the total sum of INTEREST, DEPOSIT, and BALANCE being rounded and placed in BALANCE. The second statement results in the sum of REGULAR-TIME and OVERTIME being placed in the item GROSS-PAY. The third statement adds the values of elementary items in DETAIL-LINE to elementary items of the same name in TOTAL-LINE. If the fourth statement causes a size error, the program displays a message; otherwise, the program updates NEW-BALANCE.

ALTER

Modifies a simple GO TO statement elsewhere in the PROCEDURE DIVISION, thus changing the sequence of execution of program statements.

Format

ALTER *procedure-name-1* TO [PROCEED TO] *procedure-name-2*

[, *procedure-name-3* TO [PROCEED TO] *procedure-name-4*] . . .

Syntax Rules

1. The *procedure-names 1, 3*, and so on, contain a single GO TO sentence without the DEPENDING clause.
2. The *procedure-names 2, 4*, and so on, name other paragraphs or sections in the PROCEDURE DIVISION.

General Rule

Execution of the ALTER statement modifies the GO TO statement of the first *procedure-name* so that subsequent executions of the modified GO TO statement transfer control to the second *procedure-name*.

CALL

Allows one program to communicate with one or more other programs. It transfers control from one object program to another within a runfile, with both programs having access to data items referred to in the CALL statement.

Format

CALL { *data-name-1* / *literal-1* } [USING *data-name-2* [, *data-name-3*] . . .]

[ON OVERFLOW *imperative-statement-1*]

[END-CALL]

Chapter 13, Interprogram Communication, describes the CALL statement in detail.

CANCEL

Releases the memory areas occupied by the referred to program.

Format

CANCEL { *identifier-1* / *literal-1* } [, *identifier-2* / , *literal-2*] . . .

Syntax Rules

1. Each *literal-1*, *literal-2*, and so on, must be a nonnumeric literal.
2. You must define each *identifier-1*, *identifier-2*, and so on, as an alphanumeric data item such that its value can be a program name.

General Rule

CANCEL is syntax-checked only.

CLOSE

Terminates the processing of files.

Format 1

CLOSE *file-name-1* [,*file-name-2*] . . .

Format 2

CLOSE { *file-name-1* [REEL / UNIT / WITH NO REWIND] } . . .

Chapters 9, 10, 11, and 12 describe the use of the CLOSE statement with sequential files, indexed files, relative files, and tape files, respectively.

COMPUTE

Evaluates an arithmetic expression and then stores the result in a designated item.

Format 1

COMPUTE *data-name-1* [ROUNDED] [, *data-name-2* [ROUNDED]] . . . = *arith-expr*

[ON SIZE ERROR *imperative-statement-1*]

[NOT ON SIZE ERROR *imperative-statement-2*]

[END-COMPUTE]

Format 2

COMPUTE { CORRESPONDING / CORR } *data-name-1* [ROUNDED] = *data-name-2*

[ON SIZE ERROR *imperative-statement-1*]

[NOT ON SIZE ERROR *imperative-statement-2*]

[END-COMPUTE]

Syntax Rules

1. In Format 1, *data-names* appearing to the left of the equal sign must refer to either an elementary numeric item or an elementary numeric edited item.
2. In Format 2, *data-name-1* and *data-name-2* must be group items.

General Rules

1. The COMPUTE statement is governed by the rules for ROUNDED, ON SIZE ERROR, NOT ON SIZE ERROR, and CORRESPONDING in the section Arithmetic *Statements* in the PROCEDURE DIVISION at the beginning of this chapter. It is also governed by the general rules for arithmetic expressions described in Chapter 4.
2. The COMPUTE statement allows you to combine arithmetic operations without the length restrictions on composite of operands and on receiving data items imposed by the arithmetic statements ADD, SUBTRACT, MULTIPLY, and DIVIDE.
3. In Format 1, an arithmetic expression can consist of a single *data-name* or *literal*. This format provides a method of setting the values of *data-name-1*, *data-name-2*, and so on, equal to the value of the arithmetic expression.
4. In Format 1, if more than one *data-name* precedes the equal sign, COBOL85 computes the value of the arithmetic expression, and then stores this value as the new value of each *data-name*.
5. In Format 2, COBOL85 sets data items in *data-name-1* to the contents of data items in *data-name-2* that are defined with the same name, subject to the rules for CORRESPONDING as discussed earlier in this chapter.
6. If you specify the CORRESPONDING phrase, only those data items that are defined as numeric are used in the computation.
7. The END-COMPUTE clause delimits the scope of the COMPUTE statement. For more information, see the section Scope Terminators at the beginning of this chapter.

CONTINUE

Indicates that no executable statement is present.

Format

CONTINUE

Syntax Rule

You can use the CONTINUE statement anywhere a conditional statement or an imperative statement is required.

General Rule

The CONTINUE statement has no effect on the execution of the program.

DELETE

Removes a record from an indexed or relative file.

Format

DELETE *file-name* RECORD

[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-DELETE]

Chapters 10 and 11 describe the use of the DELETE statement with indexed files and relative files, respectively.

DISPLAY

Displays low-volume data on the user terminal or on the supervisor terminal.

Format

DISPLAY { *data-name-1* / *literal-1* } [, *data-name-2* / , *literal-2*] ... [UPON *mnemonic-name*]

[[WITH] NO ADVANCING]

Syntax Rules

1. When you specify the UPON clause, you must also specify the *mnemonic-name* in the CONSOLE IS clause of the SPECIAL-NAMES paragraph in the ENVIRONMENT DIVISION.
2. The maximum number of characters that you can output without truncation is 256 per DISPLAY statement.
3. Display items are left-justified (truncated on the right).

General Rules

1. When you omit the UPON clause, the data is displayed on the user terminal.
2. If you specify a figurative constant as an operand, it is displayed as a single character.
3. **Prime Extension**: COBOL85 converts data whose usage is COMP, BINARY, COMP-1, or COMP-2 to trailing separate sign format and displays it as shown in Table 8-2, subject to the editing operations described in Rule 4. The size of the displayed data item, in characters, is the number of Ps plus the number of 9s in the PICTURE clause plus 3.

4. **Prime Extension**: COBOL85 performs the following editing operations for non-DISPLAY data types and unsigned DISPLAY data types:

   - Leading zeros are suppressed.
   - The insertion decimal point is used for implied decimal points.
   - The floating insertion character (-) is used to represent the operational sign.
   - Data is right-shifted 3 bytes.

   COBOL85 displays signed DISPLAY items as follows:

   - A trailing embedded operational sign is displayed.
   - Leading zeros are not suppressed.
   - The insertion decimal point is used for implied decimal points.
   - Data is right-shifted 3 bytes.

   The compiler option –NO_FORMATTED_DISPLAY (-NFDIS) prevents the occurrence of all editing operations for DISPLAY data types. COBOL85 displays data as it is formatted in memory. Operational signs are represented as trailing embedded.

5. **Prime Extension**: NO ADVANCING displays a line of information on the terminal but does not output a carriage return. It thus allows, for example, an answer to be input from the terminal on the same line with the question.
6. If the data item is a group item, then COBOL85 treats the displayed item as alphanumeric with a size equal to the total number of bytes in the group.

TABLE 8-2
DISPLAY of Binary Data Types
(After Conversion, If Necessary, to Display Type)

Original Data Type	*Bits*	*Size of Display Item in Characters (Bytes)*
Signed COMP or BINARY	16	9
	32	14
	64	22
Unsigned COMP or BINARY	16	9
	32	14
	64	22
COMP-1		14
COMP-2		23

Example

```
OK, SLIST DISP.COBOL85
    IDENTIFICATION DIVISION.
    PROGRAM-ID.       MAIN.
    ENVIRONMENT DIVISION.
    DATA DIVISION.
    WORKING-STORAGE SECTION.
```

```
    01  UNSIGNED-DISPLAY          PIC 999V99    VALUE 23.45.
    01  SIGNED-DISPLAY            PIC S999V99   VALUE -23.45.
    01  COMP3                     PIC S9(3)V99  VALUE -23.45  COMP-3.
    01  UNSIGNED_BINARY           PIC 9(3)      VALUE  23     COMP.
    01  SIGNED_BINARY             PIC S9(3)     VALUE -23     COMP.
    01  SIGNED_NON_INT_BINARY     PIC S9(3)V99  VALUE -23.45  COMP.
    01  SHORT_FLOAT                             VALUE -23.45  COMP-1.
    01  LONG_FLOAT                              VALUE 23.45   COMP-2.
    01  INDEX-ITEM                              VALUE 23      INDEX.
    01  FILLER.
        05  TABLE1 PIC XX OCCURS 50 INDEXED BY INDEX-NAME.
    01  ALPHANUMERIC_DATA  PIC X(10) VALUE 'SAMPLE'.
    PROCEDURE DIVISION.
    STARTIT.
        SET INDEX-NAME TO 23.
        DISPLAY UNSIGNED-DISPLAY.
        DISPLAY SIGNED-DISPLAY.
        DISPLAY COMP3.
        DISPLAY UNSIGNED_BINARY.
        DISPLAY SIGNED_BINARY.
        DISPLAY SIGNED_NON_INT_BINARY.
        DISPLAY SHORT_FLOAT.
        DISPLAY LONG_FLOAT.
        DISPLAY INDEX-ITEM.
        DISPLAY INDEX-NAME.
        DISPLAY ALPHANUMERIC_DATA.
        DISPLAY 'A NON-NUMERIC LITERAL'.
   * display numeric literal
        DISPLAY 9.
   * display figurative constant
        DISPLAY ZERO.
        STOP RUN.

OK, COBOL85 DISP -FDIS
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]
[0 ERRORS IN PROGRAM: DISP.COBOL85]

OK, BIND -LO DISP -LI COBOL85LIB -LI
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
BIND COMPLETE

OK, RESUME DISP
   23.45
  023.4N
  -23.45
      23
     -23
        -23.45
-2.345000E+01
 2.3450000000000E+0001
           23
           23
SAMPLE
A NON-NUMERIC LITERAL
9
0
```

If you specify the –NFDIS compiler option, DISPLAY data is displayed as it is formatted in memory.

```
OK, COBOL85 DISP -NFDIS
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]
[0 ERRORS IN PROGRAM: DISP.COBOL85]

OK, BIND -LO DISP -LI COBOL85LIB -LI
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
BIND COMPLETE

OK, RESUME DISP
02345
0234N
  -23.45
      23
     -23
        -23.45
-2.345000E+01
 2.3450000000000E+0001
            23
            23
SAMPLE
A NON-NUMERIC LITERAL
9
0
OK,
```

DIVIDE

Divides one numeric data item into another and stores the quotient and, optionally, the remainder.

Format 1

DIVIDE { *data-name-1* / *literal-1* / *arith-expr-1* } INTO *data-name-2* [ROUNDED] [, *data-name-3* [ROUNDED]] · · ·

[ON SIZE ERROR *imperative-statement-1*]

[NOT ON SIZE ERROR *imperative-statement-2*]

[END-DIVIDE]

Format 2

```
DIVIDE { data-name-1 }  { INTO }  { data-name-2 }
       { literal-1   }  { BY   }  { literal-2   }
       { arith-expr-1}           { arith-expr-2}

  GIVING data-name-3 [ROUNDED]  [, data-name-4 [ROUNDED]] . . .

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-DIVIDE]
```

Format 3

```
DIVIDE { data-name-1 }  { INTO }  { data-name-2 }
       { literal-1   }  { BY   }  { literal-2   }
       { arith-expr-1}           { arith-expr-2}

  GIVING data-name-3 [ROUNDED]

  REMAINDER data-name-4

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-DIVIDE]
```

Format 4

```
DIVIDE { CORRESPONDING }  data-name-1  { INTO }  data-name-2 [ROUNDED]
       { CORR          }               { BY   }

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-DIVIDE]
```

Syntax Rules

1. Each *data-name* must refer to an elementary numeric item, except that a *data-name* associated with the GIVING or REMAINDER phrase can refer either to an elementary numeric item or to an elementary numeric edited item.
2. Each literal must be a numeric literal.
3. The maximum size of each operand is 18 decimal digits. If all receiving data items are superimposed on each other, and aligned on their decimal points, their composite must not exceed 18 decimal digits in length.
4. Division by 0 always causes a size error condition if you specify the ON SIZE ERROR phrase. If you do not specify the ON SIZE ERROR phrase, but you do specify the –SIGNALERRORS compiler option, division by 0 causes the program to abort. Division by 0 under any other circumstances causes undefined results.
5. The DIVIDE statement is governed by the rules for GIVING, ROUNDED, ON SIZE ERROR, NOT ON SIZE ERROR, and CORRESPONDING at the beginning of this chapter, and by the rules for arithmetic statements listed in Chapter 4.
6. **Prime Extensions:** In Format 4, *data-name-1* and *data-name-2* must be group items. The use of arithmetic expressions in DIVIDE statements is a Prime extension.

General Rules

1. In Format 1, *data-name-1* or *literal-1* is divided into *data-name-2*, *data-name-3*, and so on; the quotient then replaces the dividend in *data-name-2*, *data-name-3*, and so on.
2. In Format 2, division occurs according to these rules:

   - If you use the keyword INTO, the value of the first operand is divided into the value of the second, and the result is stored in *data-name-3*, *data-name-4*, and so on.
   - If you use the keyword BY, the value of the second operand is divided into the value of the first, and the result is stored in *data-name-3*, *data-name-4*, and so on.

3. Use Format 3 when you need to store a remainder from the division operation. COBOL85 defines the remainder as the result of subtracting the product of the quotient *(data-name-3)* and the divisor from the dividend. If you define *data-name-3* as a numeric edited item, the quotient used to calculate the remainder is an intermediate field that contains the unedited quotient. If you specify the ROUNDED phrase, the quotient used to calculate the remainder is an intermediate field that contains the quotient of the DIVIDE statement, truncated rather than rounded.
4. The accuracy of the REMAINDER data item *(data-name-4)* is defined by the calculation described above. COBOL85 performs appropriate decimal alignment and truncation (not rounding) for *data-name-4*, as needed.
5. When you use the ON SIZE ERROR phrase in Format 3, the following rules pertain:

   - If the size error occurs on the quotient, no remainder calculation is meaningful. Thus, the contents of the data items referenced by both *data-name-3* and *data-name-4* remain unchanged.
   - If the size error occurs on the remainder, the contents of *data-name-4* remain unchanged.

6. In Format 4, elementary items subordinate to *data-name-1* and *data-name-2* that you define with the same name are divided. If you specify the INTO phrase, COBOL85 places the quotient in the matching elementary items subordinate to *data-name-2*. Otherwise, the quotient is placed in the matching elementary items subordinate to *data-name-1*.
7. The END-DIVIDE clause delimits the scope of the DIVIDE statement. For more information, see the section Scope Terminators at the beginning of this chapter.

EJECT — Prime Extension

Directs the compiler to start a new page for the program listing.

Format

EJECT

General Rule

This statement causes the compiler to insert a form feed in the program listing after the line containing EJECT. The statement can occur in any division in the program. You must code it in coding area B (Columns 12-72).

ENTER

Is used for documentation only. It has no effect on the compiled program.

Format

ENTER ***language-name*** **[*routine-name*].**

Syntax Rules

1. You can use the *language-name* and *routine-name* following ENTER as programmer-defined words elsewhere in the program.
2. The *language-name* and *routine-name* must each contain at least one alphabetic character.

EXHIBIT

Displays data at the user terminal. It is useful for debugging.

Format

EXHIBIT $\left\{\begin{matrix} literal \\ [NAMED]\ data\text{-}name \end{matrix}\right\} \ldots$

General Rules

1. You can insert the EXHIBIT statement anywhere in the PROCEDURE DIVISION to provide debugging information. The data that you specify is exhibited on the terminal, in the format shown for the DISPLAY statement.
2. The EXHIBIT statement differs from DISPLAY in that both the *data-name* and its value, connected by an = character, are displayed. A space precedes and follows the = character.
3. EXHIBIT is the same as EXHIBIT NAMED.

Example

The program statement

```
EXHIBIT NAMED EMPLOYEE-NO
```

when executed, produces the output

```
EMPLOYEE-NO = 950
```

EXIT

Provides an endpoint for a procedure or series of procedures.

Format

EXIT.

Syntax Rules

1. The EXIT statement is optional.
2. When you use the EXIT statement, it must appear in a sentence by itself.
3. The EXIT sentence can be the only sentence in its paragraph, but it must be the last sentence in its paragraph.

General Rule

An EXIT statement enables you to enhance the readability of a program by designating a termination point for a procedure or series of procedures. The EXIT statement has no other effect on the compilation or execution of the program; COBOL85 ignores it during compilation and execution.

EXIT PROGRAM

Marks the logical end of a called program.

Format

EXIT PROGRAM.

Chapter 13 discusses the EXIT PROGRAM statement.

GO TO

Transfers control from one part of the PROCEDURE DIVISION to another.

Format 1

GO TO [*procedure-name*]

Format 2

GO TO *procedure-name-1* [, *procedure-name-2*] · · · [, *procedure-name-n*]

DEPENDING ON $\left\{ \begin{matrix} \textit{data-name} \\ \textit{arith-expr} \end{matrix} \right\}$

Syntax Rules

1. A *paragraph-name* or *section-name* referenced by an ALTER statement must consist only of that *procedure-name* followed by a Format 1 GO TO statement.
2. In Format 2, *data-name* must be an elementary numeric integer data item. The *arith-expr* must have an integer value.
3. A *procedure-name* can be either a *paragraph-name* or a *section-name*.
4. A Format 1 GO TO statement without a *procedure-name* must be the only statement in its paragraph.
5. Prime Extension: The use of an arithmetic expression in a GO TO statement is a Prime extension.

General Rules

1. In Format 1, if you do not specify *procedure-name*, an ALTER statement referring to this GO TO must be executed before the GO TO is executed; otherwise, control passes to the next statement.
2. When a Format 1 GO TO statement is executed, control is transferred to *procedure-name*, or to another *procedure-name* if the GO TO statement has been modified by an ALTER statement.
3. When a GO TO statement represented by Format 2 is executed, control is transferred to *procedure-name-1*, *procedure-name-2*, and so on, depending on the value of the *data-name* or the arithmetic expression. This value must be between 1 and *n*, where *n* is the number of *procedure-names* listed. If the value of the *data-name* or the arithmetic expression is 1, control goes to the first procedure in the series, and so on. If the value of the *data-name* is anything other than the positive and unsigned integers between 1 and *n*,

then no transfer occurs and control passes to the next statement in the normal sequence for execution.

GOBACK — Prime Extension

Marks the logical end of a called program.

Format

GOBACK

Chapter 13 discusses the GOBACK statement.

IF

Causes the evaluation of a condition, permitting the execution of specified statements depending on the value of the condition.

Format

```
IF CORRESPONDING condition-1 [THEN] { statement-1       }
                                    { NEXT SENTENCE     }

[ { ELSE {statement-2} ... [END-IF]                     } ]
  { OTHERWISE {statement-2} ... [END-IF]                }
  { ELSE NEXT SENTENCE                                  }
  { OTHERWISE NEXT SENTENCE                             }
  { END-IF                                              }
```

Syntax Rules

1. The conditions in the IF statement must conform to the rules for conditions specified in the section Conditional Expressions in Chapter 4, and in the section on Arithmetic Statements in the PROCEDURE DIVISION at the beginning of this chapter.
2. You can omit the ELSE and OTHERWISE clauses if you do not need them.
3. **Prime Extension:** OTHERWISE is a Prime extension.
4. THEN is always optional. OTHERWISE and ELSE are equivalent.
5. Use the CORRESPONDING phrase with a relation condition. Both operands of the relation must be group items.
6. If you specify the END-IF phrase, you must not specify the NEXT SENTENCE phrase.

General Rules

1. If the condition is true, either *statement-1* or NEXT SENTENCE is executed as follows:

   - If you specify *statement-1*, it is executed. Control then passes to the next executable sentence following the IF statement, unless *statement-1* contains a branch or conditional statement, in which case control is transferred according to the rules for that statement.
   - If you specify the NEXT SENTENCE phrase, control passes to the next executable sentence.
   - If you specify the ELSE/OTHERWISE clause, it is ignored.

   For example,

```
IF BALANCE = 0 GO TO NOT-FOUND
ELSE NEXT SENTENCE.

IF X = 1.74 THEN MOVE 'M' TO FLAG
     OTHERWISE MOVE 'N' TO SECOND-TIME.

IF ACCOUNT-FIELD = SPACES OR NAME = SPACES
     ADD 1 TO SKIP-COUNT
ELSE PERFORM BYPASS.
```

2. If the condition is false, *statement-1* or its replacement NEXT SENTENCE is bypassed, and control passes as follows:

   - If you specify *statement-2*, it is executed. Control then passes to the next executable sentence, unless *statement-2* contains a branch or conditional statement, in which case control is transferred according to the rules for that statement.
   - If you specify the NEXT SENTENCE phrase, control passes to the next executable sentence.
   - If you do not specify an ELSE/OTHERWISE clause, control passes to the next executable sentence.

3. The IF statement is **nested** whenever *statement-1* or *statement-2* contains another IF statement. In nested IF statements, ELSEs are paired with IFs in the following way. Any ELSE encountered applies to the last preceding IF that is not already paired with an ELSE. It is not required that the number of ELSEs in a sentence be the same as the number of IFs, but the number of ELSEs must not exceed the number of IFs. OTHERWISE follows the same pairing rule as ELSE with nested IFs.

   For example,

```
IF X = Y
THEN  IF A = B
      THEN MOVE "*" TO SWITCH
      ELSE MOVE "A" TO SWITCH
ELSE MOVE SPACE TO SWITCH.
```

   The tree structure in Figure 8-2 represents the flow of this sentence.

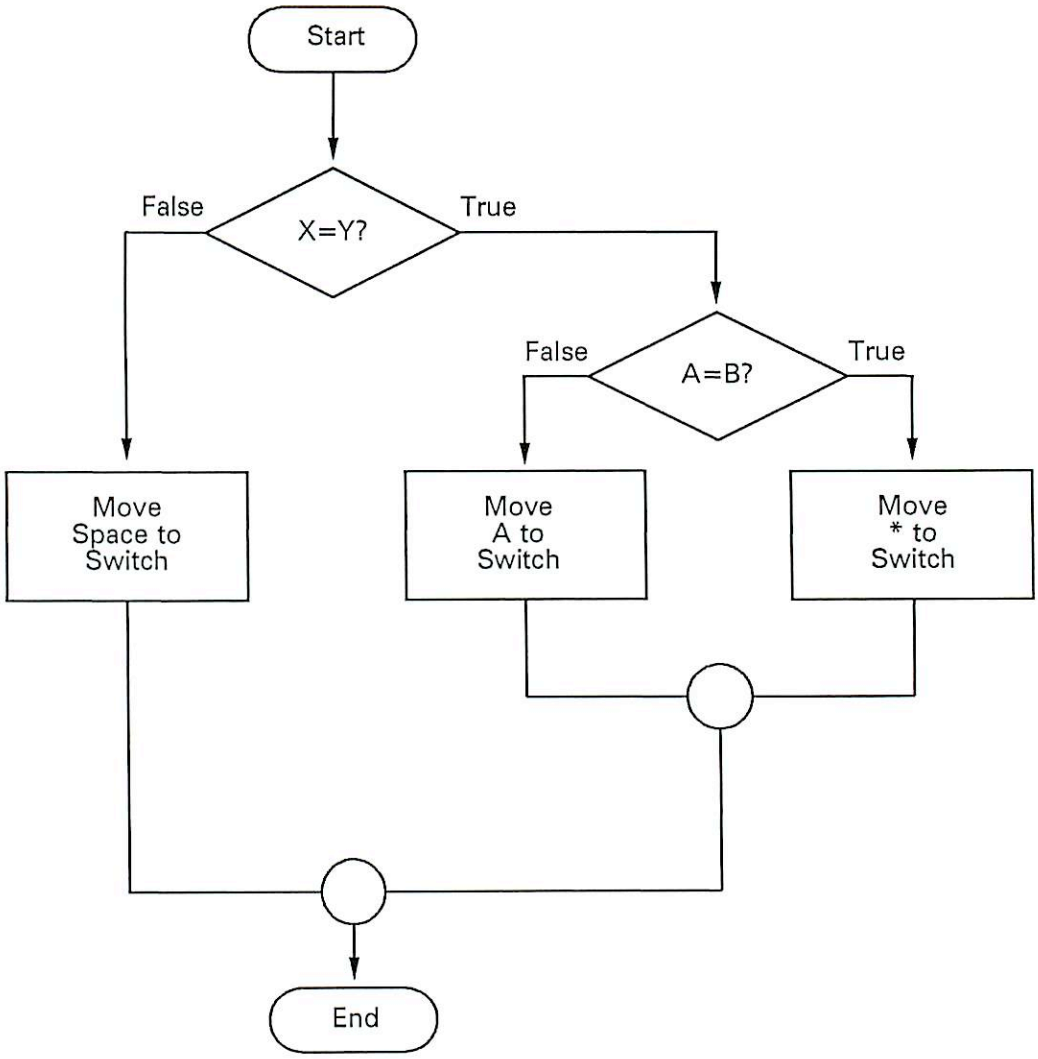


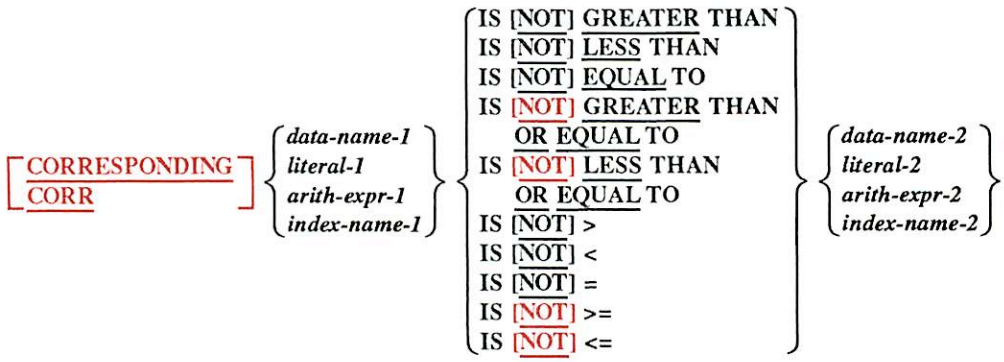
FIGURE 8-2
Nested IF Structure

4. You can terminate the scope of the IF statement by an END-IF phrase at the same level of nesting.
5. You can terminate the scope of the IF statement by a separator period for all levels of nesting.
6. You can terminate the scope of a nested IF statement that already contains an ELSE phrase by the ELSE of the containing IF statement.

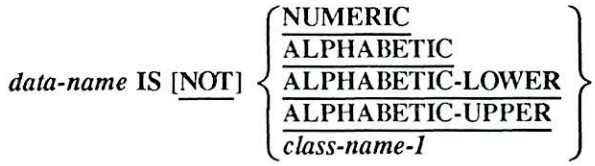
Note

Chapter 4 explains in detail the following condition types. See Chapter 4 also for status-name, negated conditions, and combined abbreviated conditions.

7. The relation condition causes a comparison of two operands. Its format is



8. The class condition determines whether an operand is numeric, alphabetic, lowercase alphabetic, uppercase alphabetic, or contains only the characters in the set of characters specified by the CLASS clause as defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. Its format is

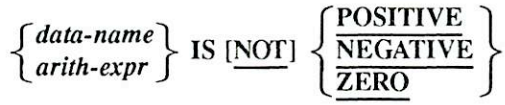


9. The condition-name condition tests the value of a conditional variable. Its format is

   [NOT] *condition-name*

   Define *condition-name* as a level-88 data item in a *record-description-entry* in the DATA DIVISION. The **conditional variable** is the data item immediately preceding the level-88 item or items. It can also be a switch-status name. (See Chapter 4.)

10. The sign condition tests an arithmetic expression to determine whether its value is greater than, less than, or equal to zero. The format is



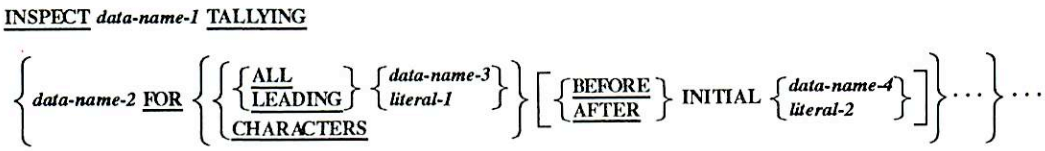
11. You can combine two or more conditions by the logical operators AND and OR. The format for a combined condition is

12. See the rules for CORRESPONDING at the beginning of this chapter.

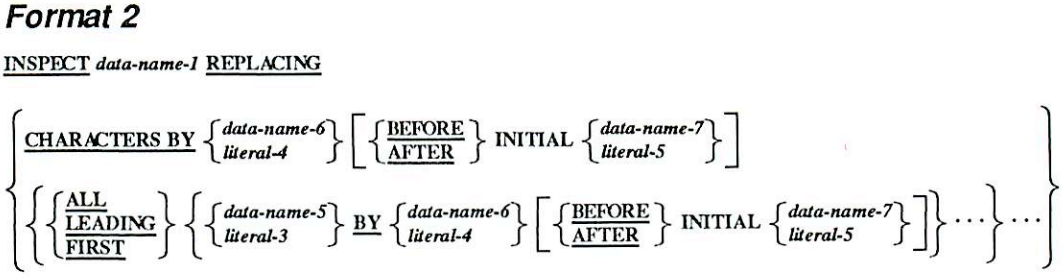
INSPECT

Enables you to examine a character-string data item and to tally, replace, or tally and replace occurrences of single characters or groups of characters in the data item.

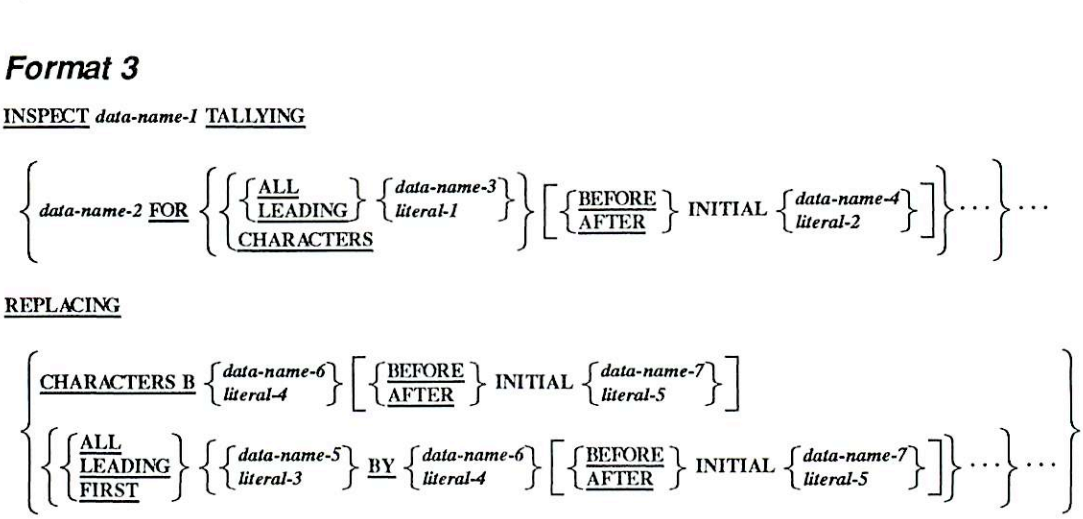
Format 1



Format 2



Format 3



Syntax Rules

1. The operand after INSPECT *(data-name-1)* must be a group item or an elementary item described (implicitly or explicitly) as USAGE IS DISPLAY.
2. The operands of all clauses except TALLYING can be either data items or literals. If they are data items, these operands must reference elementary alphabetic, alphanumeric or numeric items described (implicitly or explicitly) as USAGE IS DISPLAY.
3. If they are literals, each of these operands must be a nonnumeric literal and can be any figurative constant, except ALL.
4. *literal-1* through *literal-5* and *data-name-3* through *data-name-7* can be characters or groups of characters.
5. Operands of INSPECT can be no longer than 32767 bytes in length.

Syntax Rules for Formats 1 and 3

6. The operand of TALLYING *(data-name-2)* must be an elementary numeric data item.
7. If either *literal-1* or *literal-2* is a figurative constant, the figurative constant refers to an implicit one-character data item.

Syntax Rules for Formats 2 and 3

8. The size of *literal-4* or *data-name-6* must be equal to the size of *literal-3* or *data-name-5*. When you use a figurative constant as *literal-4*, the size of the figurative constant is equal to the size of *literal-3* or the size of *data-name-5*.
9. When you use the CHARACTERS phrase, *literal-4*, *literal-5*, or the size of *data-name-6* and *data-name-7* must be one character long.
10. When you use a figurative constant as *literal-3*, then *literal-4*, or *data-name-6* must be one character long.

General Rules

1. The INSPECT statement enables examination of a character-string item, permitting various combinations of the following actions:

   - Counting appearances of a specified character
   - Replacing a specified character or group of characters by an alternative
   - Qualifying and limiting the above actions according to the appearance of other specific characters

   Inspection includes the comparison cycle, the establishment of boundaries for the BEFORE or AFTER phrase, and the mechanism for tallying and/or replacing. It begins at the leftmost character position of *data-name-1* and proceeds from left to right to the rightmost character position, as described in General Rules 4 through 6.
2. In the INSPECT statement, COBOL85 treats the contents of *data-names 1, 3, 4, 5, 6,* and *7* as follows:

   - If you describe any of these *data-names* as alphanumeric, the INSPECT statement treats the contents of each such field as a character-string.
   - If you describe any of these *data-names* as alphanumeric edited, numeric edited, or unsigned numeric, COBOL85 inspects the data item as though you redefined it as alphanumeric and the INSPECT statement refers to the redefined data item.
   - If you describe any of these *data-names* as signed numeric, COBOL85 inspects the data item as though you moved it to an unsigned numeric data item of the same length.

3. All references to *literal-1, literal-2, literal-3, literal-4,* and *literal-5* apply equally to the contents of *data-names 3, 4, 5, 6,* and *7*, respectively.
4. During inspection of the contents of *data-name-1*, COBOL85 tallies each properly matched occurrence of *literal-1* (Formats 1 and 3) and/or replaces each properly matched occurrence of *literal-3* by *literal-4* (Formats 2 and 3).

5. The comparison operation to determine the occurrences of literals to be tallied and/or replaced (*literals 1* and *3*) occurs as follows:

   - COBOL85 considers the operands of the TALLYING and REPLACING phrases in the order in which you specify them in the INSPECT statement. The first literal is compared to an equal number of contiguous characters, starting with the leftmost character position in the data item referenced by *data-name-1*. The literal and that portion of *data-name-1* match if, and only if, they are equal character for character.
   - If no match occurs in the comparison of the first literal, the comparison is repeated with each successive literal, if any, until either a match is found or there is no next successive literal. When there is no next successive literal, the character position in *data-name-1* immediately to the right of the leftmost character position considered in the last comparison cycle is considered as the leftmost character position, and the comparison cycle begins again with the first literal.
   - Whenever a match occurs, tallying and/or replacing takes place as described in General Rules 8 and 9. The character position in *data-name-1* immediately to the right of the rightmost character position that participated in the match is now considered to be the leftmost character position of *data-name-1* and the comparison cycle starts again with the first literal.
   - The comparison operation continues until the rightmost character position of *data-name-1* has participated in a comparison or has been considered as the leftmost character position. When this occurs, inspection is terminated.

     This series of steps is represented in Figure 8-3, for the statements

     ```
     MOVE 0 TO TALLY-WORD.
     INSPECT TARGET-WORD TALLYING TALLY-WORD FOR ALL 'SS'.
     ```

     The braces mark the two characters being inspected for a match with 'SS' in each step.
   - If you specify the CHARACTERS phrase, an implied one-character operand participates in the cycle described in the preceding paragraphs, except that no comparison to the contents of *data-name-1* takes place. This implied character is considered always to match the leftmost character of *data-name-1* participating in the current comparison cycle.

6. If you do not specify the BEFORE or AFTER phrase, *literal-1*, *literal-3*, or the implied operand of the CHARACTERS phrase participates in the comparison operation.

   If you specify the BEFORE phrase, the associated *literal-1*, *literal-3*, or the implied operand of the CHARACTERS phrase participates only in those comparison cycles that involve that portion of *data-name-1* from its leftmost character position up to, but not including, the first occurrence of *literal-2* or *literal-5*. COBOL85 determines the position of this first occurrence before beginning the first cycle of the comparison operation. If, on any comparison cycle, *literal-1*, *literal-3*, or the implied operand of the CHARACTERS phrase is not eligible to participate, it is considered not to match the contents of *data-name-1*. If there is no occurrence of *literal-2*, *literal-5* within *data-name-1*, then the associated *literal-1*, *literal-3*, or the implied operand of the CHARACTERS phrase participates in the comparison operation as though you did not specify the BEFORE phrase.

   If you specify the AFTER phrase, the associated *literal-1*, *literal-3*, or the implied operand of the CHARACTERS phrase can participate only in those comparison cycles that involve

that portion of *data-name-1* from the character position immediately to the right of the rightmost character position of the first occurrence of *literal-2*, or *literal-5*. COBOL85 determines the position of this first occurrence before beginning the first cycle of the comparison operation. If, on any comparison cycle, *literal-1*, *literal-3*, or the implied operand of the CHARACTERS phrase is not eligible to participate, it is considered not to match the contents of *data-name-1*. If there is no occurrence of *literal-2*, *literal-5* within *data-name-1*, then the associated *literal-1*, *literal-3*, or the implied operand of the CHARACTERS phrase is never eligible to participate in the comparison operation.

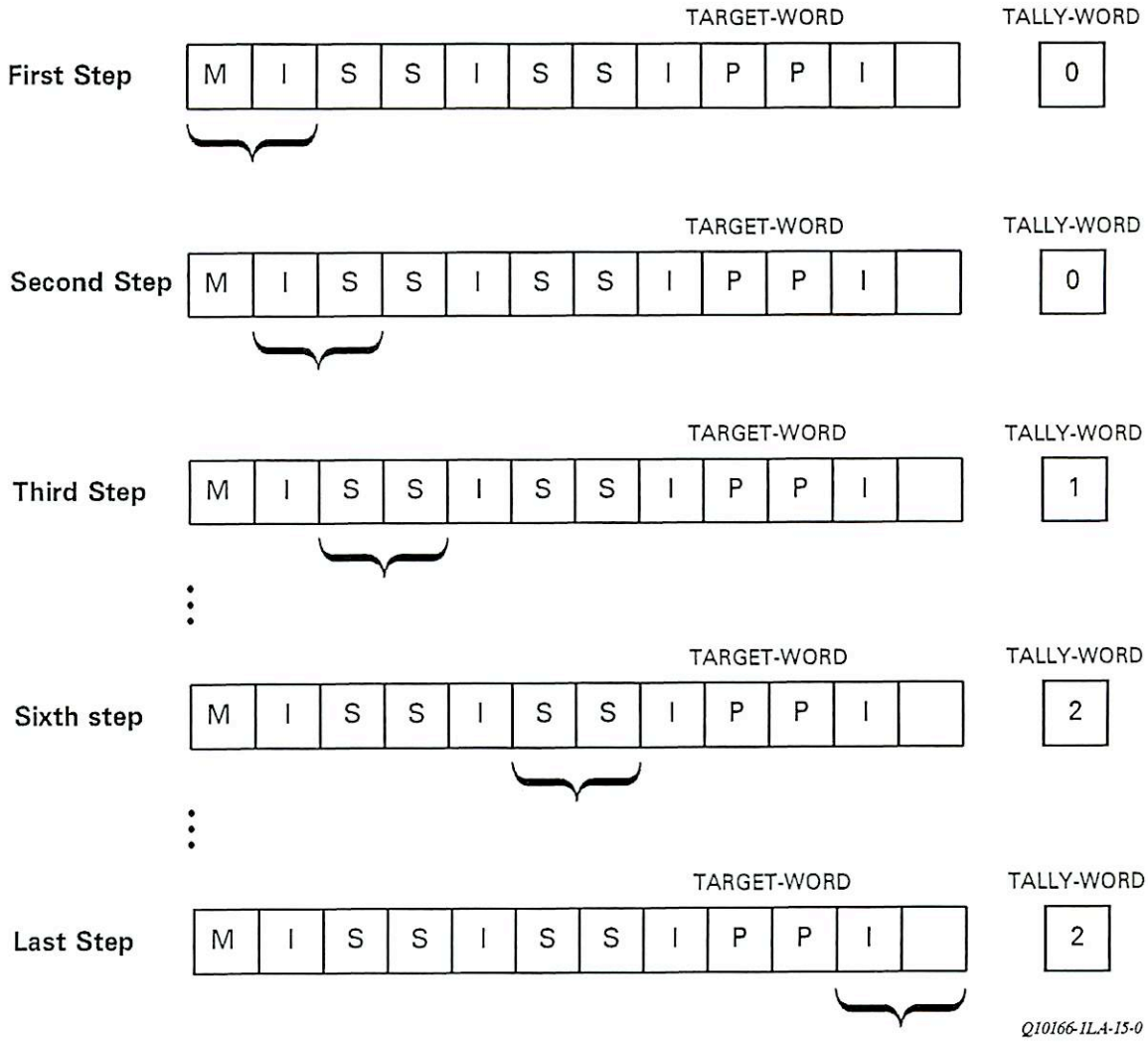


FIGURE 8-3
Steps in INSPECT... TALLYING

7. Execution of the INSPECT statement does not initialize the content of *data-name-2*.
8. The TALLYING clause causes character-by-character or character-group by character-group comparison, from left to right, of *data-name-1* with *data-name-3* or *literal-1*. The count is accumulated in *data-name-2*. See example 1 below.

- When you specify the AFTER INITIAL clause, the counting process begins only after detection of a character or character-group in *data-name-1* matching the operand following INITIAL. If you specify BEFORE INITIAL operand, the counting process terminates upon encountering a character in *data-name-1* that matches the operand following INITIAL. See examples 2 and 4 below.
- If you specify the ALL phrase, the content of *data-name-2* is incremented by one for each occurrence of the operand after FOR matched within the content of *data-name-1*.
- If you specify the LEADING phrase, the content of *data-name-2* is incremented by one for each contiguous occurrence of the operand after FOR matched within the content of *data-name-1*, provided that the leftmost such occurrence is at the point where the comparison began and where the operand after FOR was eligible to participate.
- If you specify the CHARACTERS phrase, the content of *data-name-2* is incremented by one for each character in *data-name-1*.

9. The reserved words ALL, LEADING, and FIRST apply to each succeeding BY phrase until the next adjective appears.
10. The REPLACING clause causes replacement of characters under specified conditions.

    - If you specify the BEFORE INITIAL operand, replacement does not continue after detection of a character in *data-name-1* matching the operand after INITIAL. If you specify AFTER INITIAL, replacement does not commence until detection of a character in *data-name-1* matching the operand after INITIAL.
    - If you specify the ALL phrase, each occurrence of the operand after REPLACING matched within the content of *data-name-1* is replaced by the operand after BY.
    - If you specify the LEADING phrase, each contiguous occurrence of the operand after REPLACING matched within the content of *data-name-1* is replaced by the operand after BY, provided that the leftmost occurrence is at the point where the comparison began and where the operand after REPLACING was eligible to participate. See example 6 below.
    - If you specify the FIRST phrase, the leftmost occurrence of the operand after REPLACING matched within the content of *data-name-1* is replaced by the operand after BY. See example 5 below.
    - When you specify the CHARACTERS phrase, each character in *data-name-1* is replaced by the operand after BY. See example 3 below.

11. A Format 3 INSPECT statement is executed as though you wrote two separate INSPECT statements. The first contains only a TALLYING clause, the second contains only a REPLACING clause. See example 4 below.

Examples

1. INSPECT *name* TALLYING *counter* FOR ALL 'L'.

Name Before	*Counter After*	*Name After*
LILLY	3	LILLY
SMALL	2	SMALL

2. INSPECT *name* TALLYING *counter* FOR LEADING 'B' AFTER INITIAL 'A' REPLACING CHARACTERS BY 'X'.

Name Before	*Counter After*	*Name After*
ABACK	1	XXXXXXX
CABBAGE	2	XXXXXXX

3. INSPECT *name* REPLACING CHARACTERS BY 'S' BEFORE INITIAL '.'.

Name Before	*Counter After*	*Name After*
AB D.99		$$$$$.99

4. INSPECT *name* TALLYING *counter* FOR CHARACTERS AFTER INITIAL 'E' REPLACING ALL 'B' BY 'A'.

Name Before	*Counter After*	*Name After*
DEBATE	4	DEAATE
IBEX	1	IAEX

5. INSPECT *name* REPLACING FIRST 'A' BY 'P' AFTER INITIAL 'M'.

Name Before	*Counter After*	*Name After*
LLAMAA		LLAMPA
LLOYD		LLOYD

6. INSPECT 'ABC/DEF' REPLACING LEADING 'ABC' BY '123'.

Name Before	*Counter After*	*Name After*
ABC/DEF		123/DEF

MERGE

Combines two or more sorted files on a set of specified keys, and during the process makes records available, in merge order, to an output procedure or file.

Format

```
MERGE file-name-1 ON { ASCENDING  } KEY data-name-1 [, data-name-2] ...
                     { DESCENDING }

[ ON { ASCENDING  } KEY data-name-3 [, data-name-4] ... ] ...
     { DESCENDING }

[COLLATING SEQUENCE IS alphabet-name]

USING file-name-2, file-name-3 [, file-name-4] ...

{ OUTPUT PROCEDURE IS procedure-name-1 [ { THROUGH } procedure-name-2 ] }
                                       { THRU    }
{ GIVING file-name-5 [, file-name-6] ... }
```

Chapter 14 discusses the MERGE statement.

MOVE

Transfers data from one area of main storage to another, performing conversion and editing as indicated.

Format 1

```
MOVE { data-name-1 } TO data-name-2 [ROUNDED] [, data-name-3 [ROUNDED]] ...
     { literal     }
     { arith-expr  }
```

Format 2

```
MOVE { CORRESPONDING } data-name-1 TO data-name-2 [ROUNDED]
     { CORR          }
```

Syntax Rules

1. The *data-name-1*, *arith-expr*, and the *literal* represent the **sending area**; *data-name-2*, *data-name-3*, and so on, represent the **receiving area**.
2. When you specify the CORRESPONDING phrase, both operands must be group items.

3. An index data item cannot be an operand of MOVE.
4. **Prime Extension:** The use of ROUNDED and of an arithmetic expression in a MOVE statement is a Prime extension.

General Rules

1. When either the sending or the receiving field is a group item, characters are moved without conversion and without editing.
2. During elementary moves, editing occurs and alignment is performed according to the alignment rules in the section Data Representation and Alignment in Chapter 4. Any necessary data conversion is done as described in the following rules.
3. Moves of numeric items to numeric or numeric edited fields follow these rules:

   - The items are aligned by decimal point, with generation of zeros or truncation on either end, as required.
   - When the types of the source field and receiving field differ, conversion to the type of the receiving field takes place.
   - If the receiving field is numeric edited, the item moved can have special editing performed on it such as suppression of zeros, insertion of a dollar sign, and decimal point alignment, as specified by the PICTURE clause of the receiving field.
   - Conversion of signs follows these rules. When the receiving item is signed, COBOL85 places the sign of the sending item in it. When the sending item is unsigned, a positive sign is generated. When the receiving item is unsigned, the absolute value of the sending item is moved.

4. When the sending item is alphanumeric and the receiving area is numeric or numeric edited, COBOL85 moves data as if the sending item were an unsigned integer. The discussion in General Rule 3 applies.
5. For moves to and from alphabetic and alphanumeric fields, the following rules apply:

   - COBOL85 places the characters in the receiving area from left to right (unless JUSTIFIED RIGHT applies).
   - If the receiving field is not completely filled by data being moved, the remaining positions are filled with spaces.
   - If the source field is longer than the receiving field, the move is terminated as soon as the receiving field is filled, resulting in right truncation.

6. If the sending item is signed numeric and the receiving field is alphanumeric, the sign is not moved. If the sign occupies a separate character position, that character is not moved. The discussion in General Rule 5 applies.
7. If the sending item is nonnumeric and the receiving item is numeric, the sending field is assumed to be a numeric integer, and a numeric move is generated.
8. When you use overlapping fields as operands, results are undefined.
9. When you use the CORRESPONDING phrase, *data-name-1* and *data-name-2* must be group items. The contents of elementary items subordinate to *data-name-1* are moved to elementary items subordinate to *data-name-2* that are defined by the same names. The

move is treated as a series of elementary moves. See the rules for CORRESPONDING at the beginning of this chapter.

10. You can specify the ROUNDED phrase only for numeric moves. The rules are the same as those listed for arithmetic statements at the beginning of this chapter.
11. A MOVE statement can specify any of the various types of moves summarized in Table 8-3.

TABLE 8-3
Moves by Category

	Receiving Field									
Sending Data Item	Alphabetic	Alphanumeric edited	Numeric integer	Numeric noninteger	Numeric edited	Alphanumeric	BINARY/COMP	COMP-1	COMP-2	PACKED-DEC/COMP-3
Alphabetic	Y	Y				Y				
Alphanumeric edited	Y	Y				Y				
Numeric integer		Y	Y	Y	Y	Y	Y	Y	Y	Y
Numeric noninteger		Y	Y	Y	Y	Y	Y	Y	Y	Y
Numeric edited		Y				Y				
Alphanumeric*	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
COMP and BINARY		Y	Y	Y	Y	Y	Y	Y	Y	Y
COMP-1		Y	Y	Y	Y	Y	Y	Y	Y	Y
COMP-2		Y	Y	Y	Y	Y	Y	Y	Y	Y
COMP-3 and PACKED-DEC		Y	Y	Y	Y	Y	Y	Y	Y	Y

Y = Permitted.

* Sending field must contain appropriate numeric data when moved to numeric fields.

MULTIPLY

Computes the product of two numeric data items and stores the result.

Format 1

```
MULTIPLY { data-name-1 }
         { literal-1   } BY data-name-2 [ROUNDED] [, data-name-3 [ROUNDED]] ...
         { arith-expr-1 }

[ON SIZE ERROR imperative-statement-1]

[NOT ON SIZE ERROR imperative-statement-2]

[END-MULTIPLY]
```

Format 2

```
MULTIPLY { data-name-1 } BY { data-name-2 }
         { literal-1   }    { literal-2   }
         { arith-expr-1 }   { arith-expr-2 }

GIVING data-name-3 [ROUNDED] [, data-name-4 [ROUNDED]] ...

[ON SIZE ERROR imperative-statement-1]

[NOT ON SIZE ERROR imperative-statement-2]

[END-MULTIPLY]
```

Format 3

```
MULTIPLY { CORRESPONDING } data-name-1 BY data-name-2 [ROUNDED]
         { CORR          }

[ON SIZE ERROR imperative-statement-1]

[NOT ON SIZE ERROR imperative-statement-2]

[END-MULTIPLY]
```

Syntax Rules

1. Each *data-name* must refer to an elementary numeric item, except that in Format 2 the operands after GIVING must be elementary numeric or numeric edited.
2. Each literal must be a numeric literal.
3. The maximum size of each operand is 18 decimal digits. The composite of operands in Format 1 must not contain more than 18 decimal digits.

4. In Format 3, *data-name-1* and *data-name-2* must be group items.
5. **Prime Extension**: The use of the CORRESPONDING phrase and the use of arithmetic expressions in MULTIPLY statements are Prime extensions.

General Rules

1. In Format 1, the product is stored in *data-name-2*.
2. When you use the GIVING phrase, the product is stored in *data-name-3*, *data-name-4*, and so on.
3. See the section titled Algebraic Signs in Chapter 4 for the rules for signs.
4. The MULTIPLY statement is governed by the rules for GIVING, ROUNDED, CORRESPONDING, ON SIZE ERROR, and NOT ON SIZE ERROR in the section titled Arithmetic Statements in the PROCEDURE DIVISION, at the beginning of this chapter, and by the rules for arithmetic statements discussed in Chapter 4.
5. In Format 3, elementary items subordinate to *data-name-1* are multiplied by elementary items subordinate to *data-name-2* that are defined with the same names. Each result is stored in the associated elementary item subordinate to *data-name-2*.
6. The END-MULTIPLY clause delimits the scope of the MULTIPLY statement. For more information, see the section Scope Terminators at the beginning of this chapter.

NOTE — Prime Extension

Allows comment entries in the PROCEDURE DIVISION.

Format

NOTE *comment-entry.*

Syntax Rules

1. An entry of any length can follow NOTE.
2. If NOTE is the first statement in a paragraph, the compiler treats the entire paragraph as a comment. If NOTE is not the first statement in a paragraph, only the text through the first period is treated as a comment.

General Rule

NOTE can appear only in the PROCEDURE DIVISION.

Example

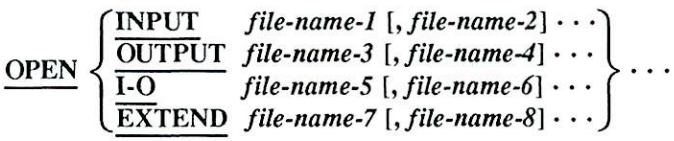
```
MOVE ALL 1 TO ITEM-1.

NOTE  THE FOLLOWING TEST ALLOWS OPENING OF A SECOND DISK
      INPUT FILE IF THE SERIAL NUMBER IS ALL 1's,
```

```
        OTHERWISE DISK-FILE-1 WILL BE CLOSED, REOPENED,
        AND READ AGAIN.

IF NAME = ITEM-1 PERFORM 150-SECOND-INPUT
        OTHERWISE PERFORM 180-REOPEN.
```

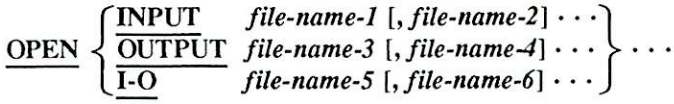
OPEN

Initiates the processing of files and enables other input/output operations, such as label checking, reading, and writing.

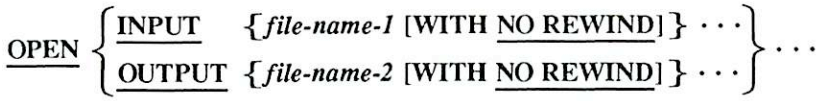
Format 1



Format 2



Format 3



Chapters 9, 10, 11, and 12 describe the use of the OPEN statement with sequential files, indexed files, relative files, and tape files, respectively.

PERFORM

Transfers control explicitly to one or more procedures, and returns control implicitly to the normal sequence after execution of the specified procedures.

Format 1

[*imperative-statement-1* END-PERFORM]

Format 2

```
PERFORM [ procedure-name-1 [ { THROUGH } procedure-name-2 ] ]
                             { THRU    }

{ integer     }
{ data-name-1 } TIMES
{ arith-expr-1}

[imperative-statement-1 END-PERFORM]
```

Format 3

```
PERFORM [ procedure-name-1 [ { THROUGH } procedure-name-2 ] ]
                             { THRU    }

UNTIL condition-1

[imperative-statement-1 END-PERFORM]
```

Format 4

```
PERFORM [ procedure-name-1 [ { THROUGH } procedure-name-2 ] ]
                             { THRU    }

VARYING { data-name-2  } FROM { data-name-3  }
        { index-name-1 }      { index-name-2 }
                              { literal-1    }
                              { arith-expr-1 }

    BY { data-name-4  } UNTIL condition-1
       { literal-2    }
       { arith-expr-2 }

[ AFTER { data-name-5  } FROM { data-name-6  }
        { index-name-3 }     { index-name-4 }
                             { literal-3    }
                             { arith-expr-2 }

    BY { data-name-7  } UNTIL condition-2 ] . . .
       { literal-4    }
       { arith-expr-4 }

[imperative-statement-1 END-PEFORM]
```

Syntax Rules

1. The words THROUGH and THRU are equivalent.
2. Each *data-name* represents an elementary numeric item described in the DATA DIVISION. In Format 2, *data-name-1* must represent a numeric integer.
3. Each literal represents a numeric integer.
4. A *procedure-name* can be either a *section-name* or a *paragraph-name*.
5. If you use an *index-name* in the VARYING or AFTER phrase, then
   - The *data-name* in the associated FROM and BY phrases must be an integer data item.
   - The *literal* in the associated FROM phrase must be a positive integer. The arithmetic expression must evaluate to a positive integer.
   - The *literal* in the associated BY phrase must be a nonzero integer. The arithmetic expression must evaluate to a positive integer.
6. If you specify an *index-name* in the FROM phrase, then
   - The *data-name* in the associated VARYING or AFTER phrase must be an integer data item.
   - The *data-name* in the associated BY phrase must be an integer data item.
   - The *literal* in the associated BY phrase must be an integer.
7. A *literal* in the BY phrase must not be zero.
8. *condition-1*, *condition-2*, and *condition-3* can be any conditional expression described in Chapter 4.
9. When you specify both *procedure-name-1* and *procedure-name-2* and either is the name of a procedure in a declarative section, then both must be *paragraph-names* in the same declarative section.
10. **Prime Extension**: The use of arithmetic expressions in FROM and BY clauses is a Prime extension.
11. If you omit the FROM and BY clauses, the value 1 is assumed.
12. If you omit *procedure-name-1*, you must specify *imperative-statement-1* and the END-PERFORM phrase; if you specify *procedure-name-1*, do not specify *imperative-statement-1* and the END-PERFORM phrase.
13. In Format 4, if you omit *procedure-name-1*, do not specify the AFTER phrase.
14. In Format 2, if you specify the END-PERFORM phrase, the use of *arith-expr-1* is not allowed.

General Rules

1. When you specify *procedure-name-1*, the PERFORM statement is an **out-of-line** PERFORM statement; when you omit *procedure-name-1*, the PERFORM statement is an **in-line** PERFORM statement.

Note

Unless specially qualified by the word in-line or out-of-line, all the general rules that apply to the out-of-line PERFORM statement also apply to the in-line PERFORM statement.

2. *data-name-4* and *data-name-7* must not have a zero value.
3. If you write the PERFORM statement with no options, control is transferred to the first statement of *procedure-name-1*. At the completion of *procedure-name-1*, control is returned to the next executable statement following the PERFORM statement. If *procedure-name-1* is a *paragraph-name*, control is returned after the last statement of *procedure-name-1*. If *procedure-name-1* is a *section-name*, control is returned after the last statement of the last paragraph in *procedure-name-1*.
4. If you specify *procedure-name-2* as a *paragraph-name*, control is returned to the statement following the PERFORM after the last statement of that paragraph is executed.
5. If you specify *procedure-name-2* as a *section-name*, control is returned to the statement following the PERFORM after the last statement of the last paragraph of that section is executed.
6. In Formats 1 and 2, if you use the THROUGH option, multiple paragraphs or sections can be executed before control is returned to the statement after PERFORM.
7. In Format 2, the following rules apply:

   - If you use the TIMES option, procedures are performed the number of times specified by *data-name-1*, *arith-expr-1*, or *integer*.
   - If *data-name-1*, *arith-expr-1*, or *integer* is initially zero or negative, the PERFORM statement is not executed; control passes to the statement following the PERFORM statement.
   - During execution of the PERFORM statement, if the value of *data-name-1* or *arith-expr-1* changes, the number of times the procedure is executed is, nevertheless, that of the initial value of *data-name-1*.

8. In Formats 3 and 4, the condition can be of any type, including the CORRESPONDING relation condition.
9. In Format 3, the following rules apply:

   - If you use the UNTIL option, successive execution of procedures occurs until *condition-1* is satisfied.
   - *condition-1* is tested prior to execution of the PERFORM statement. If the condition is not true, the specified procedures are performed until the condition is true. Control is then passed to the next statement after PERFORM. If the condition is true prior to execution of the PERFORM statement, *procedure-name-1* is not executed and control passes to the next statement after PERFORM.

10. In Format 4, the following rules apply:

    - If you vary one identifier, *data-name-2* is set to the current value of *data-name-3*, *arith-expr-1*, *index-name-1*, or *literal-1* at the point of initial execution of the PERFORM statement. If the condition is true, the procedures are not executed and control passes to the next statement after PERFORM. If the condition is false, *procedure-name-1* and *procedure-name-2* are executed once.

- The value of *data-name-2* is then incremented or decremented by the value in *data-name-4*, *arith-expr-2*, or *literal-2*. The condition is reevaluated. The cycle continues until the condition is true, at which point control is transferred to the next statement following the PERFORM statement. See Figure 8-4.
- At the termination of the PERFORM statement, *data-name-2* or *index-name-1* has a value that differs from the last used setting by the value of *data-name-4*, *arith-expr-2*, or *literal-2*. If the condition was true before initial execution of the PERFORM statement, *data-name-2* or *index-name-1* contains the current value of *data-name-3*, *literal-1*, *arith-expr-1*, or *index-name-2*.

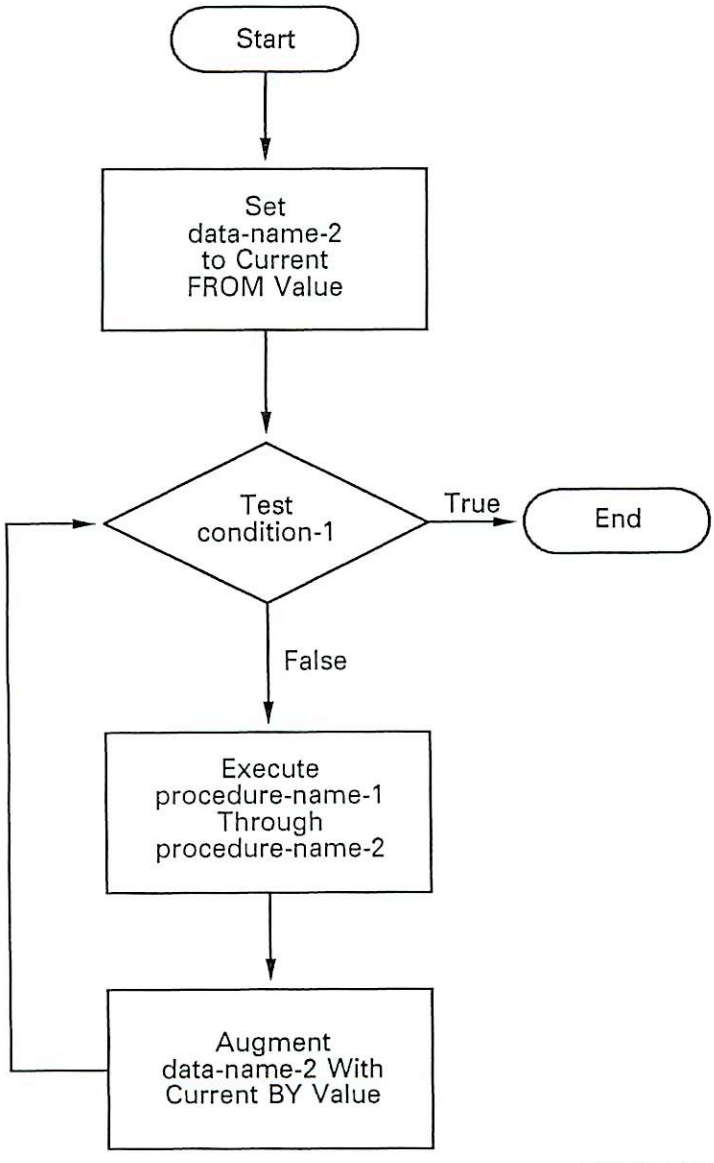


FIGURE 8-4
Logic of PERFORM Statement (One Identifier Varied)

- When you vary two identifiers, *data-name-2* and *data-name-5* are set to the current value of *data-name-3* and *data-name-6*, respectively. *condition-1* is then evaluated. If it is true, control is transferred to the next statement; if false, *condition-2* is evaluated. If *condition-2* is false, *procedure-name-1* and *procedure-name-2* are executed once, then *data-name-5* is augmented by *data-name-7*, *arith-expr-4*, or *literal-4*, and *condition-2* is evaluated again. This cycle of evaluation and augmentation continues until *condition-2* is true. When *condition-2* is true, *data-name-5* or *index-name-3* is set to the value of *literal-3*, *data-name-6*, *index-name-4* (if you vary *index-name-3*), or *arith-expr-3*.
- *data-name-2* is augmented by *data-name-4*, *literal-2*, or *arith-expr-2*, and *condition-1* is reevaluated. The PERFORM statement is completed if *condition-1* is true; if not, the cycles continue until *condition-1* is true.
- During the execution of the procedures associated with the PERFORM statement, any change to the VARYING *data-name*, the BY *data-name*, the AFTER *data-name*, or the FROM *data-name* affect the operation of the PERFORM statement.
- *data-name-5* goes through a complete cycle (FROM, BY, UNTIL) each time *data-name-2* is varied. See Figure 8-5.
- At the termination of the PERFORM statement, *data-name-5* contains the current value of *data-name-6*. *data-name-2* has a value that exceeds the last used setting by one increment or decrement value, unless *condition-1* was true when the PERFORM statement began, in which case *data-name-2* contains the current value of *data-name-3*.
- When you vary three or more identifiers, the mechanism is an extension of the one for two identifiers. See Figure 8-6.
- After the completion of the PERFORM statement, each data item varied by an AFTER phrase contains the current value of the *data-name* in the associated FROM phrase. *data-name-2* has a value that exceeds its last used setting by one increment or decrement value, unless *condition-1* is true when the PERFORM statement begins, in which case *data-name-2* contains the current value of *data-name-3*.

  The following example illustrates a Format 4 PERFORM statement:

```
                    .
                    .
                    .
START-PARA.
    PERFORM INT-PARA
        VARYING INDX1 FROM 1 BY 1
          UNTIL INDX1 > 2
        AFTER INDX2 FROM 1 BY 1
          UNTIL INDX2 > 12
        AFTER INDX3 FROM 1 BY 1
          UNTIL INDX3 > 7.
INT-PARA.
        MOVE ZEROS TO DEPT-TOTAL(INDX1, INDX2, INDX3).
                    .
                    .
                    .
```

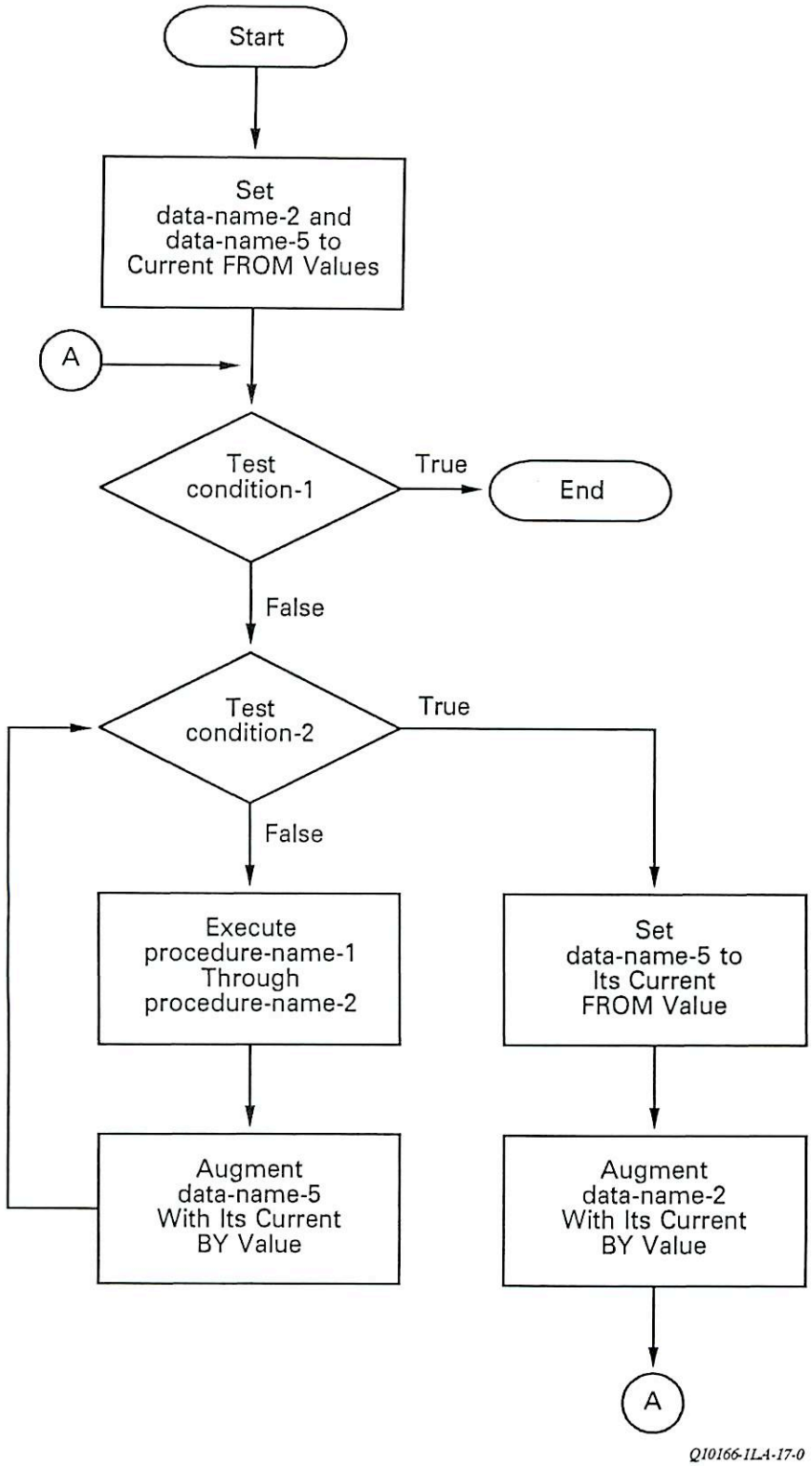


FIGURE 8-5
Logic of PERFORM Statement (Two Identifiers Varied)

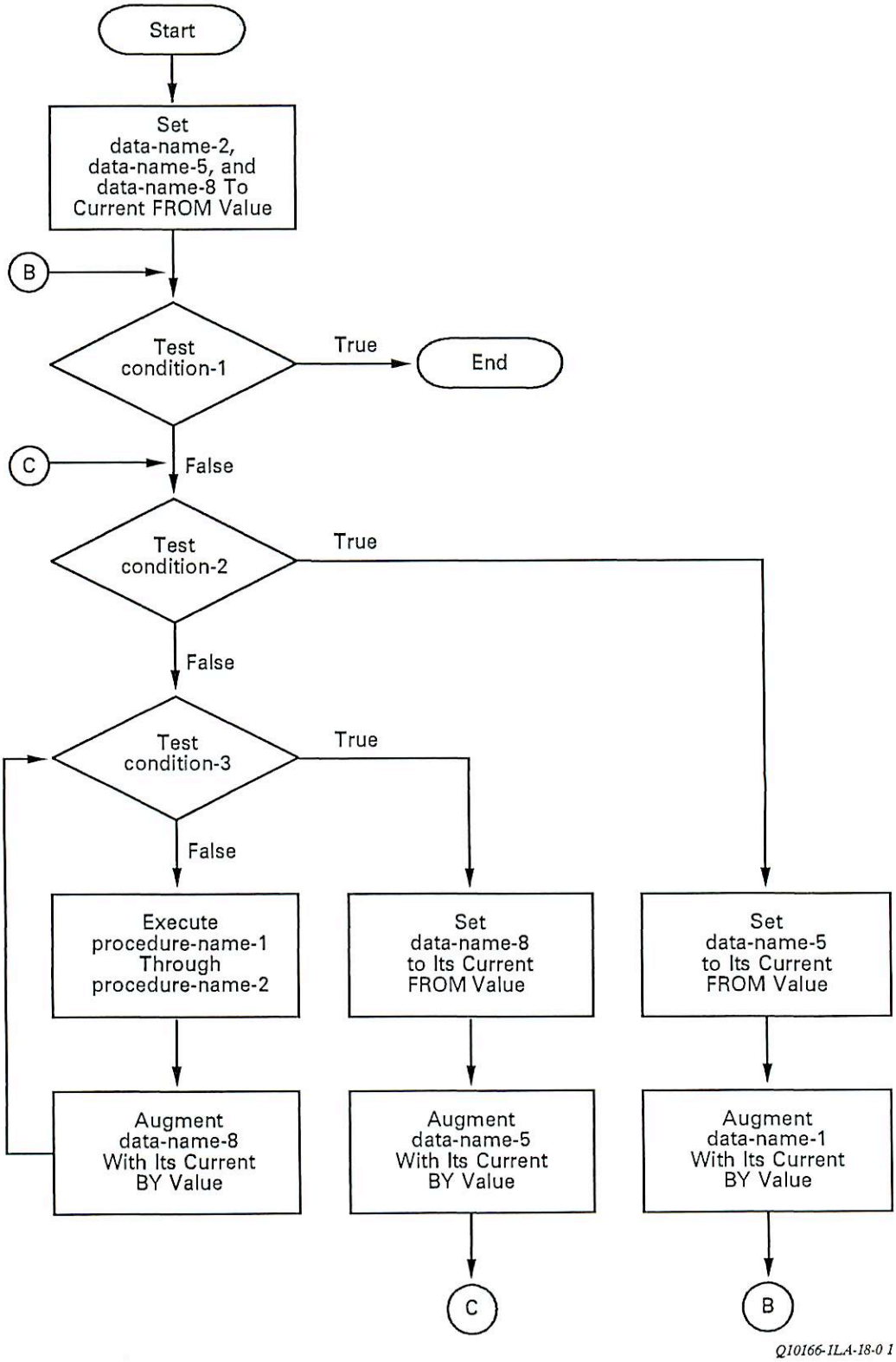


FIGURE 8-6
Logic of PERFORM Statement (Three Identifiers Varied)

11. If a sequence of statements referred to by a PERFORM statement includes another PERFORM statement, the sequence of procedures associated with the included PERFORM must itself be either totally included in, or totally excluded from the logical sequence encompassed by the first PERFORM. Thus, an active PERFORM statement, whose execution point begins within the range of another active PERFORM statement, must not allow control to pass to the exit of the other active PERFORM statement. Furthermore, two or more such active PERFORM statements cannot have a common exit. See Figure 8-7.
12. In the overlapping PERFORM sequence illustrated in Figure 8-7, the second PERFORM statement (statement d) must not be executed while the first PERFORM statement (statement x) is active. Otherwise, program-control code at statement m causes an erroneous return to the statement following statement x.
13. The statements contained within the range of *procedure-name-1* (through *procedure-name-2*, if you specify one) for an out-of-line PERFORM statement or contained within the PERFORM statement itself for an in-line PERFORM statement are called the specified set of statements.
14. The END-PERFORM phrase delimits the scope of the in-line PERFORM statement. See the section Scope Terminators at the beginning of this chapter.
15. If you specify an in-line PERFORM statement, an execution of the PERFORM statement is complete after the last statement contained within it is executed.

    The following example illustrates an in-line PERFORM statement:

```
IF A > B
   PERFORM VARYING INDX1 FROM 1 BY 1 UNTIL INDX1 > 7
      ADD 1 TO TABLE1(INDX1)
      DISPLAY TABLE1(INDX1)
   END-PERFORM
ELSE
   NEXT SENTENCE.
```

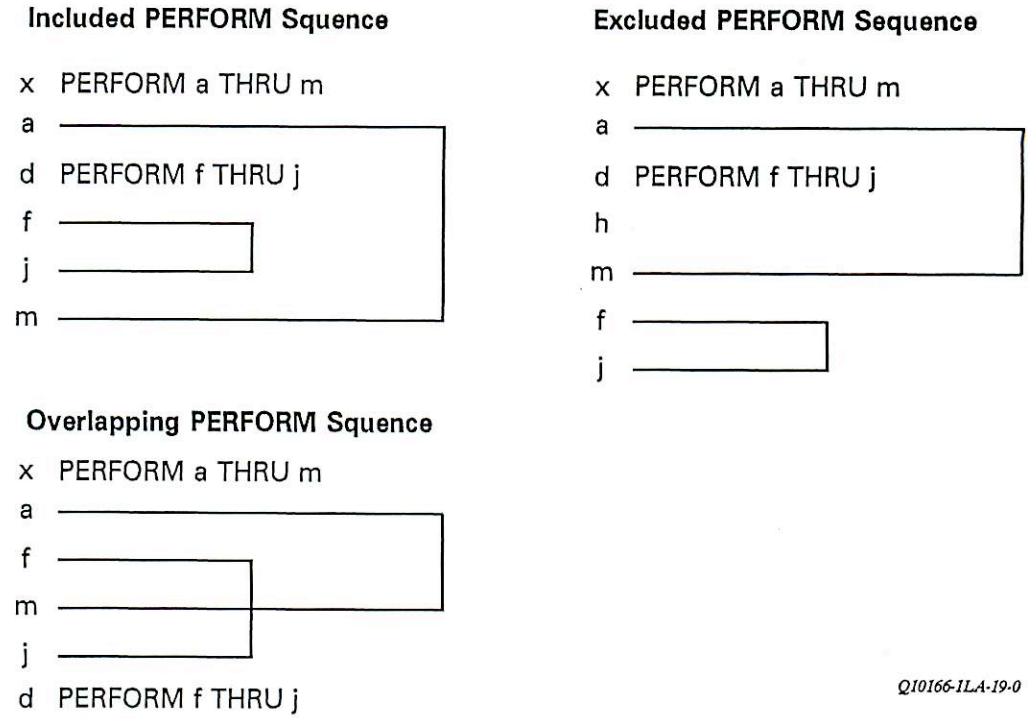


FIGURE 8-7
Permissible PERFORM Sequences

READ

Makes available a record from a file.

Format 1

READ *file-name* **RECORD** [**INTO** *data-name-1*]

[**AT END** *imperative-statement-1*]

[**NOT AT END** *imperative-statement-2*]

[**END-READ**]

Format 2

READ *file-name* [**NEXT**] **RECORD** [**INTO** *data-name-1*]

[**AT END** *imperative-statement-1*]

[**NOT AT END** *imperative-statement-2*]

[**END-READ**]

Format 3

READ *file-name* RECORD [INTO *data-name-1*]

[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-READ]

Format 4

READ *file-name* RECORD [INTO *data-name-1*]

[KEY IS *data-name-2*]

[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-READ]

Chapters 9, 10, 11, and 12 describe the use of the READ statement with sequential files, indexed files, relative files, and tape files, respectively.

READY TRACE — Prime Extension

Enables the display of trace information on the terminal.

Format

READY TRACE.

General Rules

1. After a READY TRACE statement is executed, each time a paragraph or section in the PROCEDURE DIVISION is encountered, that paragraph or section name is output to the terminal to provide debugging information.
2. Do not use READY TRACE before the first *paragraph-name* in the PROCEDURE DIVISION.
3. Terminate the display of trace information by coding the RESET TRACE statement.

Example

When you run the sample program at the end of this chapter with READY TRACE inserted at the beginning of the PROCEDURE DIVISION, the actual flow of program execution is displayed. The following example illustrates sample output.

```
OK, RESUME OLDCASH
trace:  010-GET-JOBINFO
ENTER MONTH(ALPHA)
NOVEMBER, 1987
ENTER JOB CODE
25
trace:  020-NEW-DETAIL-PAGE
trace:  150-NEW-PAGE
trace:  150-NEW-PAGE-EXIT
trace:  030-PROCESS-DETAIL
trace:  035-READ-AND-PRINT
trace:  040-EDIT
trace:  050-DEPT-TOTALS
trace:  035-READ-AND-PRINT
trace:  040-EDIT
trace:  050-DEPT-TOTALS
trace:  035-READ-AND-PRINT
trace:  040-EDIT
trace:  050-DEPT-TOTALS
trace:  035-READ-AND-PRINT
trace:  040-EDIT
trace:  050-DEPT-TOTALS
trace:  035-READ-AND-PRINT
trace:  040-EDIT
trace:  050-DEPT-TOTALS
trace:  035-READ-AND-PRINT
trace:  040-EDIT
trace:  050-DEPT-TOTALS
trace:  035-READ-AND-PRINT
trace:  040-EDIT
trace:  060-REJECTS
trace:  035-READ-AND-PRINT
trace:  040-EDIT
trace:  060-REJECTS
trace:  035-READ-AND-PRINT
trace:  040-EDIT
trace:  060-REJECTS
trace:  035-READ-AND-PRINT
trace:  040-EDIT
trace:  060-REJECTS
trace:  070-TOTALS
trace:  150-NEW-PAGE
trace:  150-NEW-PAGE-EXIT
trace:  080-BALANCE-TOTALS
trace:  150-NEW-PAGE
trace:  150-NEW-PAGE-EXIT
trace:  090-PROCESS-TAPE
IS TAPE OUTPUT DESIRED--ENTER YES OR NO
NO
NO TAPE
     END OF RUN
OK,
```

RELEASE

Transfers records to the initial phase of a SORT operation.

Format

RELEASE *record-name* [**FROM** *data-name*]

Chapter 14 discusses the RELEASE statement.

RESET TRACE — Prime Extension

Turns off the display of trace information.

Format

RESET TRACE.

General Rules

1. The RESET TRACE statement has meaning only after the execution of a READY TRACE statement.
2. You cannot use RESET TRACE before the first *paragraph-name* in the PROCEDURE DIVISION.

RETURN

Obtains sorted records from the final phase of a SORT operation.

Format

RETURN *file-name* **RECORD** [**INTO** *data-name-1*]

[**AT END** *imperative-statement-1*]

[**NOT AT END** *imperative-statement-2*]

[**END-RETURN**]

The RETURN statement is discussed in Chapter 14.

REWRITE

Logically replaces a record existing in a disk file.

Format 1

REWRITE *record-name* [**FROM** *data-name*]

[**END-REWRITE**]

Format 2

REWRITE *record-name* [**FROM** *data-name*]

[**INVALID KEY** *imperative-statement-1*]

[**NOT INVALID KEY** *imperative-statement-2*]

[**END-REWRITE**]

Chapters 9, 10, and 11 describe the use of the REWRITE statement with sequential files, indexed files, and relative files, respectively.

SEARCH

Searches a table for a table element that satisfies the specified condition, and adjusts the associated *index-name* to indicate that table element.

Format 1

SEARCH *data-name-1* [**VARYING** { *data-name-2* / *index-name-1* }]

[**AT END** *imperative-statement-1*]

{ **WHEN** *condition-1* { *imperative-statement-2* / **NEXT SENTENCE** } } ...

[**END-SEARCH**]

Format 2

```
SEARCH ALL data-name-1 [AT END imperative-statement-1]

WHEN { data-name-2 { IS EQUAL TO } { data-name-3 }
                   { IS =        } { literal-1   }
                                   { arith-expr-1 }
     { condition-name-1 }

[ AND { data-name-4 { IS EQUAL TO } { data-name-5 }
                    { IS =        } { literal-2   }
                                    { arith-expr-2 }
      { condition-name-2 } ] ...

{ imperative-statement-2 }
{ NEXT SENTENCE          }

[END-SEARCH]
```

Syntax Rules

1. In Formats 1 and 2, *data-name-1* must not be subscripted or indexed, but its description must contain an OCCURS clause.
2. Unless you use the VARYING clause, the description of *data-name-1* must contain an INDEXED BY clause.
3. When you specify *data-name-2*, you must describe it as USAGE IS INDEX, or as a numeric elementary data item without any positions to the right of the assumed decimal point.
4. In Format 1, *condition-1* or *condition-2* can be any condition as described in the section Conditional Expressions in Chapter 4.
5. In Format 2, you must define all referenced *condition-names* as having only a single value. The *data-name* associated with a *condition-name* must appear in the KEY clause of *data-name-1*. Each *data-name-2* or *data-name-4* can be qualified. Further, each *data-name-2* or *data-name-4* must be indexed by the first *index-name* associated with *data-name-1* along with other indexes or literals as required.
6. In Format 2, when a *data-name* in the KEY clause of *data-name-1* is referenced, or when a *condition-name* associated with a *data-name* in the KEY clause of *data-name-1* is referenced, all preceding *data-names* in the KEY clause of *data-name-1* or their associated *condition-names* must also be referenced.
7. The END-SEARCH clause delimits the scope of the SEARCH statement. For more information, see the section Scope Terminators at the beginning of this chapter.
8. If you specify the END-SEARCH phrase, you must not specify the NEXT SENTENCE phrase.

General Rules

1. The Format 1 SEARCH statement enables a serial search operation, starting with the current index setting.

   - If, at the start of execution of the SEARCH statement, the *index-name* associated with *data-name-1* contains a value greater than the highest permissible occurrence number for *data-name-1*, the specified *imperative-statement-1* is executed. If the AT END phrase is not specified, control passes to the next executable sentence.
   - If, at the start of execution of the SEARCH statement, the *index-name* associated with *data-name-1* contains a value not greater than the highest permissible occurrence number for *data-name-1*, the SEARCH statement operates by evaluating the conditions in the order you write them, making use of the index settings, wherever specified, to determine the occurrence of those items to be tested. If none of the conditions is satisfied, the *index-name* for *data-name-1* is incremented to obtain reference to the next occurrence.
   - The process is repeated, using the new *index-name* settings. If the new value of the *index-name* settings for *data-name-1* corresponds to a table element outside the permissible range of occurrence values, the search terminates as indicated in the paragraph above. If one of the conditions is satisfied upon its evaluation, the search terminates immediately and the imperative statement associated with that condition is executed; the *index-name* remains set at the occurrence that satisfied the condition.

   For example, the following code assumes that you define the table MONTH-TAB as shown in the example with OCCURS. The SEARCH statement causes a search of MONTH-TAB, changing the value of INDX until the element whose position is referenced by INDX has the value of MONTH-ACCEPT.

```
05  MONTH-TAB OCCURS 12 TIMES INDEXED BY INDX
    ASCENDING KEY MONTH-NO.
    10  MONTH-NO      PIC 99.
    10  MONTH-VALUE   PIC XXX.
      .
      .
      .

FIND-MONTH.
   SEARCH MONTH-TAB
      WHEN MONTH-NO(INDX) = MONTH-ACCEPT
      MOVE MONTH-VALUE(INDX)   TO PRINT-MONTH.
```

2. In Format 1, if you do not use the VARYING phrase, the *index-name* used for the search operation is the first (or only) *index-name* appearing in the INDEXED BY phrase of *data-name-1*. Any other *index-names* for *data-name-1* remain unchanged.
3. In Format 1, if you specify the VARYING *index-name-1* phrase, and if *index-name-1* appears in the INDEXED BY phrase of *data-name-1*, that *index-name* is used for this search. If this is not the case, or if you specify the VARYING *data-name-2* phrase, the first *index-name* given (if it exists) in the INDEXED BY phrase of *data-name-1* is used for the search. In addition, the following operations occur:

- If you use the VARYING *index-name-1* phrase, and if *index-name-1* appears in the INDEXED BY phrase of another table entry, *index-name-1* is incremented simultaneously by the same amount as the *index-name* associated with *data-name-1* is incremented.
- If you specify the VARYING *data-name-2* phrase, and *data-name-2* is an index data item, then *data-name-2* is incremented simultaneously by the same amount as the *index-name* associated with *data-name-1* is incremented. If *data-name-2* is not an index data item, *data-name-2* is incremented by the value 1 at the same time as the *index-name* associated with *data-name-1* is incremented.

4. In a Format 2 SEARCH statement, the results of the SEARCH ALL operation are predictable only when

   - The data in the table is ordered in the same manner as described in the ASCENDING/ DESCENDING KEY clause associated with the description of *data-name-1*.
   - The contents of the key(s) referenced in the WHEN clause are sufficient to identify a unique table element.

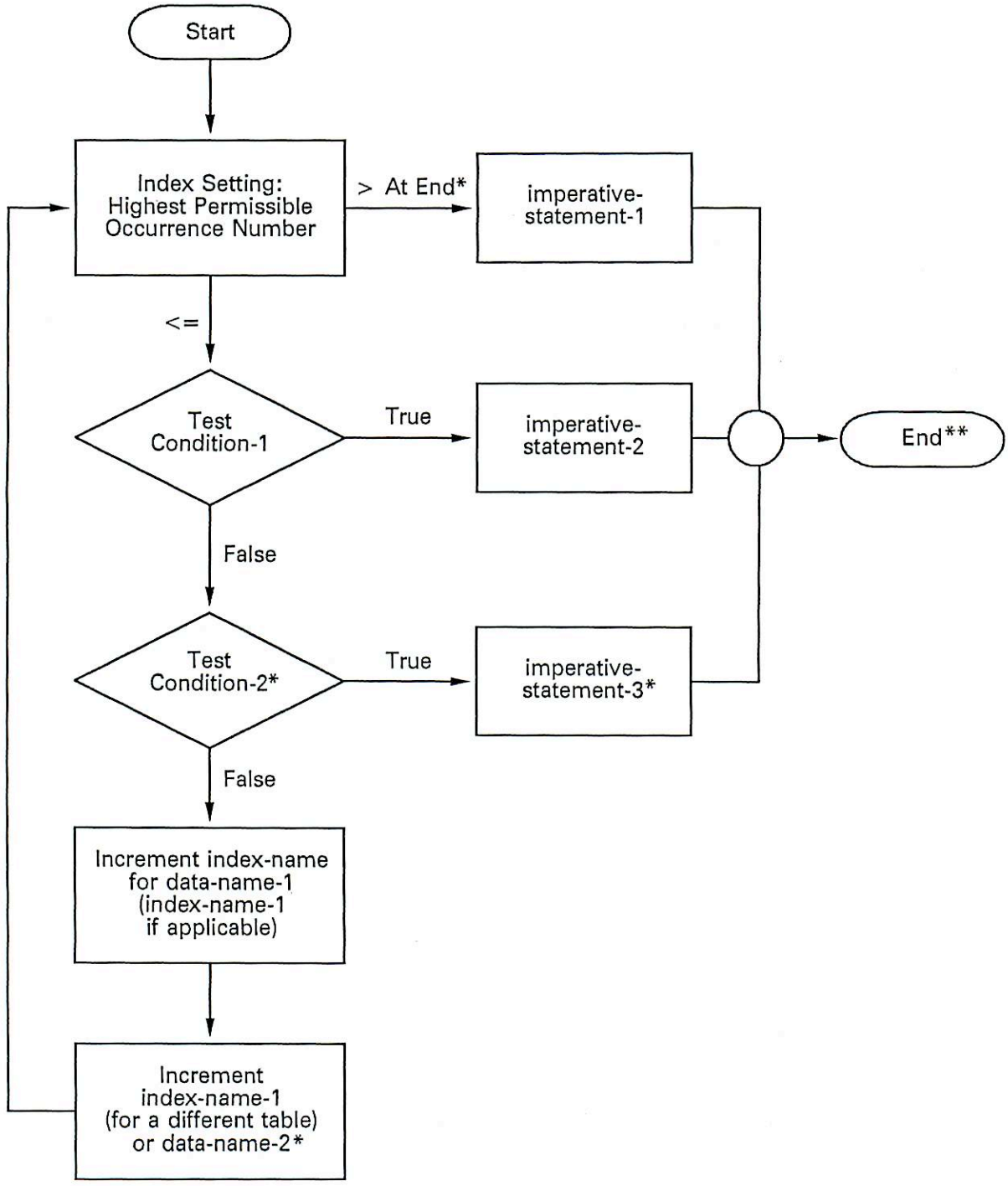
5. When you use a Format 2 SEARCH ALL, the initial setting of the *index-name* for *data-name-1* is ignored and its setting is varied during the search operation. However, at no time is the *index-name* set to a value that exceeds the number of elements in the table, or that is less than the value that corresponds to the first element of the table.

   If any of the conditions specified in the WHEN clause cannot be satisfied for any setting of the index within the permitted range, control passes to *imperative-statement-1* of the AT END phrase, when specified, or to the next executable sentence. In either case, the final setting of the index is not predictable. If all the conditions can be satisfied, the index indicates an occurrence that allows the conditions to be satisfied, and control passes to *imperative-statement-2*.

6. If *imperative-statement-1* or *imperative-statement-2* does not terminate with a GO TO statement, control passes to the next sentence.
7. In Format 2, COBOL85 uses as the *index-name* for the search operation the first (or only) *index-name* that appears in the INDEXED BY clause of *data-name-1*. Any other *index-names* for *data-name-1* remain unchanged.
8. If *data-name-1* is a data item subordinate to another data item containing an OCCURS clause (providing for a two-dimensional or three-dimensional table), you must use the INDEXED BY phrase to associate an *index-name* with each dimension of the table. Only the setting of the *index-name* associated with *data-name-1* (and *data-name-2* or *index-name-1*, if present) is modified by the execution of the SEARCH statement.

   To search an entire two-dimensional or three-dimensional table, you must execute a SEARCH statement several times. Prior to each execution of a SEARCH statement, you must execute SET statements to adjust *index-names* to appropriate settings.

9. A flowchart of the Format 1 SEARCH operation containing two WHEN phrases is presented in Figure 8-8.
10. You can terminate the scope of a SEARCH statement by an END-SEARCH phrase at the same level of nesting.
11. You can terminate the scope of a SEARCH statement by a separator period for all levels of nesting.
12. You can terminate the scope of a SEARCH statement by an ELSE or END-IF phrase associated with a preceding IF statement.



* These operations are options included only when specified in the SEARCH statement.

** Each of these control transfers is to the next sentence unless the imperative-statement ends with a GO TO statement.

Q10166-ILA-20-0

FIGURE 8-8
Format 1 SEARCH Flowchart

SEEK — Prime Extension

Specifies a disk record to be accessed.

Format

SEEK *file-name* RECORD

General Rules

1. SEEK is treated as documentation only in COBOL85. It is included only for compatibility with programs transported from other vendors' systems.
2. You must define the *file-name* with a *file-description-entry* in the DATA DIVISION.

SET

Establishes reference points for table-handling operations by setting *index-names* associated with table elements. Also use the SET statement to alter the status of external switches.

Format 1

SET { *index-name-1* [, *index-name-2*] ... / *data-name-1* [, *data-name-2*] ... } TO { *index-name-3* / *data-name-3* / *integer-1* / *arith-expr-1* }

Format 2

SET *index-name-4* [, *index-name-5*] ... { UP BY / DOWN BY } { *data-name-4* / *integer-2* / *arith-expr-2* }

Format 3

SET { {*mnemonic-name-1*} ... TO { ON / OFF } } ...

Syntax Rules

1. All references to *index-name-1*, *data-name-1*, and *index-name-4* apply equally to *index-name-2*, *data-name-2*, and *index-name-5*, respectively.
2. You must describe *data-name-4* as an elementary numeric integer.
3. The *data-name-1* and *data-name-3* must name either index data items or elementary integer items.

4. The *integer-1* and *integer-2* can be signed. However, *integer-1* must have a positive value. *arith-expr-1* and *arith-expr-2* must evaluate to positive integers.
5. **Prime Extension**: The use of arithmetic expressions in the SET statement is a Prime extension.
6. You must associate *mnemonic-name-1* with an external switch.

General Rules

1. *index-names* are related to a specific table. Define them with the INDEXED BY clause.
2. If you specify *index-name-3*, the value of the index before the execution of the SET statement must not exceed the maximum number of elements in the associated table.
3. In Format 1, the following actions occur:

   - The *index-name-1* is set to a value causing it to refer to the table element that corresponds in occurrence number to the value of the name after TO. If *data-name-3* is an index data item, or if *index-name-3* is related to the same table as *index-name-1*, no conversion takes place.
   - If *data-name-1* is an index data item, you can set it equal to the contents of either *index-name-3* or *data-name-3*, where *data-name-3* is also an index data item; no conversion takes place in either case.
   - If *data-name-1* is not an index data item, you can set it only to an occurrence number that corresponds to the value of *index-name-3*. In this case, you can use neither *data-name-3* nor *integer-1*.
   - The process is repeated for *index-name-2*, *data-name-2*, and so on. Each time, the value of *index-name-3* or *data-name-3* is used as it was at the beginning of the execution of the statement.

4. In Format 2, the index name or names following SET are incremented or decremented by the value after UP or DOWN, respectively. Each time, the value of *data-name-4* is used as it was at the beginning of the execution of the statement.
5. Table 8-4 represents the validity of various operand combinations in the SET statement.
6. In Format 3, the status of each external switch associated with the specified *mnemonic-name-1* is modified such that the truth value resultant from evaluation of a *condition-name* associated with that switch reflects the status of the phrase specified.

TABLE 8-4
Validity of Operand Combinations in the SET Statement

	Type of Receiving Item		
Sending Item	*Integer Data Item*	*Index-name*	*Index Data Item*
Integer arith-expr		Valid	
Integer literal		Valid	
Integer data item		Valid	
Index-name	Valid	Valid	Valid*
Index data item		Valid*	Valid*

*No conversion takes place.

SKIP — Prime Extension

Directs the compiler to place blank lines in the program listing.

Format

SKIP*n*

Syntax Rule

The letter *n* represents the number 1, 2, or 3 written as one word with SKIP.

General Rule

The number after SKIP causes one, two, or three blank lines to be inserted in the program listing, after the line containing SKIP.

Example

The following program contains SKIP1 in line 4 and SKIP3 in line 6 of the source.

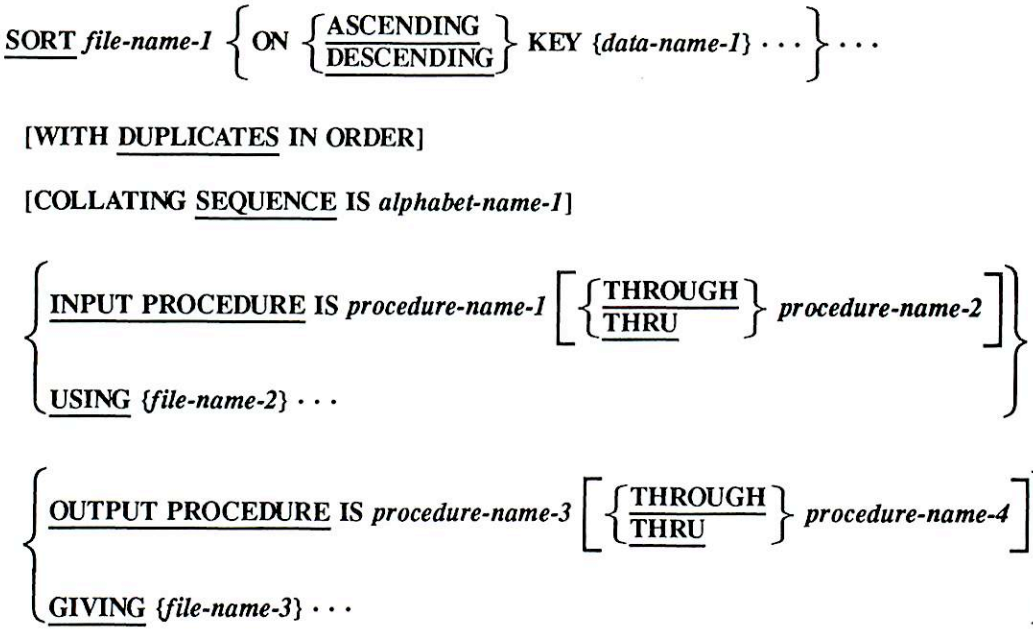
```
SOURCE FILE: <MYMFD>MYDIR>COBOL85>SKIP.COBOL85
COMPILED ON: WED, JUL 27 1988 AT: 14:31 BY: COBOL85 REV. 1.0-22.0 04/18/88.09:36
Options selected:  skip -listing
Optimization note: Currently "-OPTimize" means "-OPTimize 2",
Options used (* follows those that are not default):
     64V No_Ansi_Obsolete Big_Tables Binary CALCindex No_COMP No_CORrMap
     No_DeBuG No_Data_Rep_Opt No_ERRorFile ERRTty No_EXPlist No_File_Assign
     Formatted_DISplay No_HEXaddress Listing* No_MAp No_OFFset OPTimize(2)
     No_PRODuction No_RAnge No_SIGnalerrors SIlent(0) No_SLACKbytes TIME
     No_STANdard No_STATistics Store_Owner_Field SYNtaxmsg No_TRUNCdiags
     VARYing No_XRef

1            IDENTIFICATION DIVISION.
2            PROGRAM-ID. SIZECK.
3            AUTHOR.  ANNE.
```

```
 5          DATE-COMPILED. 880727.14:31:22


 7          ENVIRONMENT DIVISION.
 8          CONFIGURATION SECTION.
 9          SOURCE-COMPUTER. PRIME.
10          OBJECT-COMPUTER.
11          INPUT-OUTPUT SECTION.
12          FILE-CONTROL.
13          DATA DIVISION.
14          FILE SECTION.
15          PROCEDURE DIVISION.
```

SORT

Creates a sort-file by executing input procedures or by transferring records from another file. It sorts the records in the sort-file on a set of specified keys, and makes the sorted records available to output procedures or to an output file.

Format

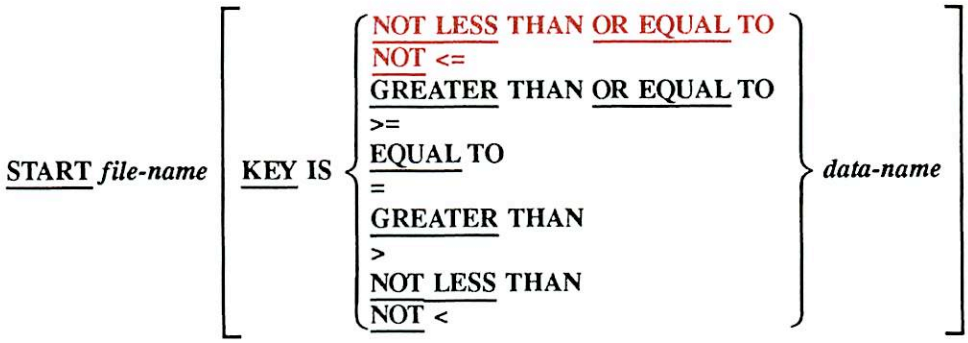


The SORT statement is discussed in Chapter 14.

START

Provides a basis for logical positioning within an indexed or relative file for subsequent sequential or dynamic retrieval of records.

Format



[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

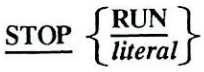
[END-START]

Chapters 10 and 11 describe the use of the START statement with indexed files and relative files, respectively.

STOP

Terminates or delays execution of the object program.

Format



Syntax Rules

1. The *literal* must be a numeric unsigned integer, nonnumeric literal, or any figurative constant without the keyword ALL.
2. If a STOP RUN statement appears in a consecutive sequence of imperative statements within a sentence, it must appear as the last statement in that sequence.

General Rules

1. STOP RUN terminates execution of a program, returning control to the operating system.
2. If STOP RUN appears in a called program, execution halts when the statement is encountered. Control is not returned to the calling program.
3. If you specify STOP *literal*, *literal* is displayed on the terminal, and execution is suspended. Execution resumes at the next executable statement in sequence after you

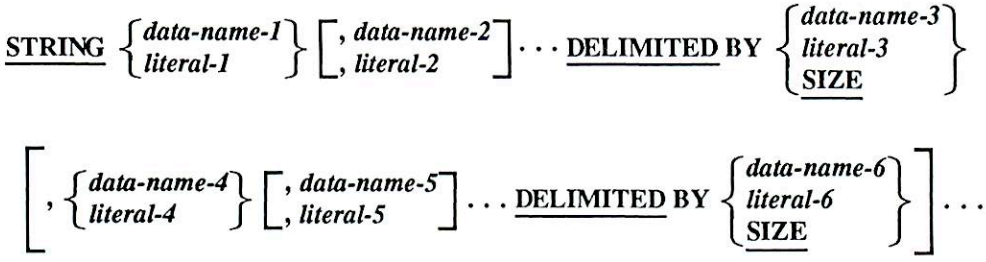
press the carriage return. Presumably, you perform a function suggested by the contents of *literal* before resuming program execution.

4. During the execution of a STOP RUN statement, an implicit CLOSE statement is executed for each open file. Any USE procedure associated with any of the files is not executed.

STRING

Concatenates the partial or complete contents of two or more data items.

Format



Syntax Rules

1. Each *literal* can be any figurative constant (without the optional word ALL).
2. You must describe all *literals* as nonnumeric literals. You must describe all *data-names*, except *data-name-8*, implicitly or explicitly as USAGE IS DISPLAY.
3. The operand after INTO, *data-name-7*, must represent an elementary alphanumeric data item without editing symbols or the JUSTIFIED clause.
4. The POINTER operand, *data-name-8*, must represent an elementary numeric integer data item big enough to contain a value equal to the size of *data-name-7* + 1. You cannot use the symbol P in the PICTURE *character-string* of *data-name-8*.
5. Where *data-name-1*, *data-name-2*, or *data-name-3* is an elementary numeric data item, you must describe it as an integer without the symbol P in its PICTURE *character-string*.
6. Operands of STRING have a maximum length of 32,766 bytes.

General Rules

1. All references to *data-name-1* through *data-name-3* and *literal-1* through *literal-3* apply equally to *data-name-4* through *data-name-6* and *literal-4* through *literal-6*, respectively.
2. The items referred to by *data-name-1*, *literal-1*, *data-name-2*, and *literal-2* are the **sending items**. *data-name-7* represents the **receiving item**.
3. The operands of DELIMITED (*literal-3*, *data-name-3*) indicate the character(s) delimiting the move. If the SIZE phrase is used, the complete sending item is moved. When a figurative constant is used as the delimiter, the constant stands for a single-character nonnumeric literal.
4. When you specify a figurative constant as *literal-1*, *literal-2*, *or literal-3*, the constant refers to an implicit one-character data item whose usage is DISPLAY.
5. When the STRING statement is executed, the transfer of data is governed by the following rules:

   - Characters from the sending items are transferred to *data-name-7* in accordance with the rules for alphanumeric to alphanumeric moves, except that no space-filling is provided.
   - If you specify the DELIMITED phrase without the SIZE phrase, the contents of the sending items are transferred to the receiving data item. This occurs in the sequence specified in the STRING statement, beginning with the leftmost character and continuing from left to right until the end of the data item is reached, or until the character(s) specified by *literal-3* or by the contents of *data-name-3* are encountered. The character(s) specified by *literal-3* or by the data item referenced by *data-name-3* are not transferred.
   - If you specify the DELIMITED phrase with the SIZE phrase, the entire contents of *literal-1*, *literal-2*, or *data-name-1*, *data-name-2*, are transferred. The transfer proceeds in the sequence specified in the STRING statement to *data-name-7*, until all data is transferred or the end of the data item referenced by *data-name-7* is reached.

6. If you specify the POINTER phrase, you must set the initial value of *data-name-8*. The initial value must not be less than 1.
7. If you do not specify the POINTER phrase, characters are transferred to the receiving item as if you specified *data-name-8* with an initial value of 1.
8. When COBOL85 transfers characters to the receiving item *(data-name-7)*, the transfer occurs as if characters are moved, one at a time, from the sending item to the character position of *data-name-7* designated by the value of *data-name-8*. *data-name-8* is increased by one prior to the move of the next character. This is the only way the value of *data-name-8* is changed during execution of the STRING statement.
9. When the STRING statement is executed, only the portion of *data-name-7* that is referenced during the execution of the STRING statement is changed. All other portions of *data-name-7* contain data that was present before this execution of the STRING statement.
10. Data transfer to *data-name-7* terminates when the value in *data-name-8* is either less than 1, or exceeds the number of character positions in *data-name-7*. Such termination can occur at any point at or after initialization of the STRING statement. If termination occurs as a result of such a condition, and if you specify the ON OVERFLOW phrase, control passes to *imperative-statement-1*.

11. If an overflow condition is not encountered, and you specify the NOT ON OVERFLOW phrase, the ON OVERFLOW phrase, if specified, is ignored, and control is transferred to *imperative-statement-2*. Execution then continues according to the rules in Chapter 4.
12. The END-STRING clause delimits the scope of the STRING statement. See the section Scope Terminators at the beginning of this chapter for more information.

Example

The following program concatenates two strings to form a single string.

```
OK, SLIST STRING.LIST
SOURCE FILE: <MYMFD>MYDIR>COBOL85>STRING.COBOL85
COMPILED ON: WED, JUL 27 1988 AT: 14:46  BY: COBOL85 REV. 1.0-22.0
Options selected:  STRING -L
Optimization note: Currently "-OPTimize" means "-OPTimize 2",
Options used (* follows those that are not default):
 64V No_Ansi_Obsolete Big_Tables Binary CALCindex No_COMP No_CORrMap
 No_DeBuG No_Data_Rep_Opt No_ERRorFile ERRTty No_EXPlist No_File_Assign
 Formatted_DISplay No_HEXaddress Listing* No_MAp No_OFFset OPTimize(2)
 No_PRODuction No_RAnge No_SIGnalerrors SIlent(0) No_SLACKbytes TIME
 No_STANdard No_STATistics Store_Owner_Field SYNtaxmsg No_TRUNCdiags
 VARYing No_XRef

    1            ID DIVISION.
    2            PROGRAM-ID. USTRING.
    3            ENVIRONMENT DIVISION.
    4            CONFIGURATION SECTION.
    5            SOURCE-COMPUTER. PRIME-850.
    6            OBJECT-COMPUTER. PRIME-850.
    7            DATA DIVISION.
    8            WORKING-STORAGE SECTION.
    9            77  USER-NUMBER-WS        PIC X(1).
   10            77  USER-PREFIX-WS        PIC X(7).
   11            77  USER-NAME-WS          PIC X(8).
   12
   13            PROCEDURE DIVISION.
   14            010-STRING.
   15                DISPLAY 'PREFIX: '  ACCEPT USER-PREFIX-WS.
   16                DISPLAY 'RUN NO: '  ACCEPT USER-NUMBER-WS.
   17                STRING USER-PREFIX-WS, USER-NUMBER-WS DELIMITED
   18                    BY ' ' INTO USER-NAME-WS
   19                    ON OVERFLOW DISPLAY 'FILENAME MAY ONLY BE 8
   20          -       'CHARACTERS LONG   PLEASE START AGAIN'.
   21                DISPLAY 'USER-NAME WILL BE ' USER-NAME-WS.
   22                STOP RUN.
```

When you execute this program (supposing a runfile named STRING.RUN), screen dialog approximates the following example:

```
OK, RESUME STRING
PREFIX:
EVELYN
RUN NO:
1
USER-NAME WILL BE EVELYN1
OK,
```

SUBTRACT

Subtracts one or more numeric data items from a specified item and stores the difference.

Format 1

```
SUBTRACT { data-name-1 } [ , data-name-2 ] ...
         { literal-1   }   , literal-2
         { arith-expr-1}   , arith-expr-2

  FROM data-name-3 [ROUNDED] [, data-name-n [ROUNDED]] ...

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-SUBTRACT]
```

Format 2

```
SUBTRACT { data-name-1 } [ , data-name-2 ] ...
         { literal-1   }   , literal-2
         { arith-expr-1}   , arith-expr-2

  FROM { data-name-3 } GIVING data-name-6 [ROUNDED] [, data-name-7 [ROUNDED]] ...
       { literal-3   }
       { arith-expr-3}

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-SUBTRACT]
```

Format 3

SUBTRACT { CORRESPONDING / CORR } *data-name-1*

FROM *data-name-2* [ROUNDED]

[ON SIZE ERROR *imperative-statement-1*]

[NOT ON SIZE ERROR *imperative-statement-2*]

[END-SUBTRACT]

Syntax Rules

1. Each *data-name* must refer to a numeric elementary item, except that *data-name-5*, *data-name-6*, and so on (following GIVING) can be elementary numeric-edited items.
2. Each *literal* must be a numeric literal.
3. The maximum size of each operand is 18 decimal digits. If all receiving data items were superimposed and aligned by their decimal points, their composite must not exceed 18 decimal digits in length. This rule is ignored if you use the GIVING phrase.
4. In Format 3, both *data-name-1* and *data-name-2* must be group items.
5. **Prime Extension**: The use of arithmetic expressions in SUBTRACT statements is a Prime extension.

General Rules

1. The SUBTRACT statement is governed by the rules for the GIVING, CORRESPONDING, ROUNDED, ON SIZE ERROR, and NOT ON SIZE ERROR phrases described at the beginning of this chapter, by the rules for arithmetic statements and algebraic signs in Chapter 4, and by the rules listed in the section Arithmetic Statements in the PROCEDURE DIVISION, also at the beginning of this chapter.
2. In Format 1, the effect of the SUBTRACT statement is to add the values of all the operands that precede FROM, and then subtract that sum from the value of each of the operands following FROM. The result is stored in each of the operands following FROM.
3. In Format 2, all operands preceding FROM are added, the resulting sum is subtracted from *data-name-3*, *arith-expr-3*, or *literal-3*, and the result is stored in each of the operands following GIVING. See the discussion of composite of operands in the section Arithmetic Statements in the PROCEDURE DIVISION, at the beginning of this chapter.
4. In Format 3, elementary items subordinate to *data-name-1* are subtracted from and stored in the matching elementary items subordinate to *data-name-2*.
5. The SIZE ERROR phrase is executed if the result is too large for its field.
6. The END-SUBTRACT clause delimits the scope of the SUBTRACT statement. For more information, see the section Scope Terminators at the beginning of this chapter.

UNSTRING

Separates contiguous data in a sending field and places the separated data into multiple receiving fields.

Format

UNSTRING *data-name-1*

[DELIMITED BY [ALL] { *data-name-2* / *literal-1* } [, OR [ALL] { *data-name-3* / *literal-2* }] ...]

INTO *data-name-4* [, DELIMITER IN *data-name-5*] [, COUNT IN *data-name-6*]

[, *data-name-7* [, DELIMITER IN *data-name-8*] [, COUNT IN *data-name-9*]] ...

[WITH POINTER *data-name-10*] [TALLYING IN *data-name-11*]

[ON OVERFLOW *imperative-statement-1*]

[NOT ON OVERFLOW *imperative-statement-2*]

[END-UNSTRING]

Syntax Rules

1. Each *literal* must be a nonnumeric literal. In addition, each literal can be any figurative constant without the optional word ALL. (The ALL phrase of the UNSTRING clause is not the figurative constant ALL.)
2. You must describe the items represented by *data-name-1*, *data-name-2*, *data-name-3*, *data-name-5*, and *data-name-8*, implicitly or explicitly, as alphanumeric.
3. You can describe the items represented by *data-name-4* and *data-name-7* as either alphabetic (except that you cannot use the symbol B in their PICTURE *character-strings*), alphanumeric, or numeric (except that you cannot use the symbol P in their PICTURE *character-strings*). You must describe them with USAGE IS DISPLAY.
4. You must describe the items represented by *data-name-6*, *data-name-9*, *data-name-10*, and *data-name-11* as elementary numeric integer data items (except that you cannot use the symbol P in their PICTURE *character-strings*).
5. No *data-name* can name a level-88 entry.
6. You can specify the DELIMITER IN phrase and the COUNT IN phrase only if you specify the DELIMITED BY phrase.
7. Operands of UNSTRING have a maximum length of 32,766 bytes.

General Rules

1. All references to *data-name-2*, *literal-1*, *data-name-4*, *data-name-5*, and *data-name-6* apply equally to *data-name-3*, *literal-2*, *data-name-7*, *data-name-8*, and *data-name-9*, respectively.
2. *data-name-1* represents the **sending area**.
3. *data-name-4* represents the **receiving area**. *data-name-5* represents the receiving area for delimiters.
4. *literal-1* or the data item referenced by *data-name-2* specifies a delimiter.
5. *data-name-6* represents the count of the number of characters within *data-name-1* isolated by the delimiters for the move to *data-name-4*. This value does not include a count of the delimiter character(s).
6. The data item referenced by *data-name-10* contains a value that indicates a relative character position within the area defined by *data-name-1*.
7. The data item referenced by *data-name-11* is a counter that records the number of data items acted upon during the execution of an UNSTRING statement.
8. When you use a figurative constant as the delimiter, it stands for a single-character nonnumeric literal.

   When you specify the ALL phrase, COBOL85 treats two or more contiguous occurrences of *literal-1* (figurative constant or not), or of the contents of *data-name-2*, as only one occurrence. This occurrence is moved to the receiving data item *(data-name-4)* according to the rules for the DELIMITER IN phrase in General Rule 13 below.
9. When an examination encounters two contiguous delimiters, the current receiving area is either space-filled or zero-filled, according to the description of the receiving area.
10. The *literal-1*, or the contents of the data item referenced by *data-name-2*, can contain any character in the computer's character set.
11. Each *literal-1* or *data-name-2* represents one delimiter. When a delimiter contains two or more characters, all the characters must be present in contiguous positions of the sending item and in the order given to be recognized as a delimiter.
12. When you specify two or more delimiters in the DELIMITED BY phrase, an OR condition exists between them. Each delimiter is compared to the sending field. If a match occurs, the character(s) in the sending field is a single delimiter. No character(s) in the sending field can be part of more than one delimiter.

    Each delimiter is applied to the sending field in the sequence specified in the UNSTRING statement.
13. When the UNSTRING statement is initiated, the **current receiving area** is the data item referenced by *data-name-4*. Data is transferred from *data-name-1* to *data-name-4* according to the following rules:

    - If you specify the POINTER phrase, the string of characters referenced by *data-name-1* is examined, beginning with the relative character position indicated by the contents of *data-name-10*. If you do not specify the POINTER phrase, the string of characters is examined beginning with the leftmost character position.
    - If you specify the DELIMITED BY phrase, the examination proceeds, left to right, until a delimiter specified by either *literal-1* or *data-name-2* is encountered. (See General Rule 11.) If you do not specify the DELIMITED BY phrase, the number of

characters examined is equal to the size of the receiving area. However, if the sign of the receiving area is defined as occupying a separate character position, the number of characters examined is one less than the size of the current receiving area.

If the end of *data-name-1* is encountered before the delimiting condition is met, the examination terminates with the last character examined.

- The characters thus examined (excluding any delimiting characters), are treated as an elementary alphanumeric data item, and are moved into the current receiving area according to the rules for the MOVE statement.
- If you specify the DELIMITER IN phrase, the delimiting character(s) is treated as an elementary alphanumeric data item and is moved into *data-name-5* according to the rules for the MOVE statement. If the delimiting condition is the end of *data-name-1*, then *data-name-5* is space-filled.
- If you specify the COUNT IN phrase, a value equal to the number of characters thus examined (excluding the delimiter character(s), if any) is moved into *data-name-6* according to the rules for an elementary move.
- If you specify the DELIMITED BY phrase, the string of characters is further examined, beginning with the first character to the right of the delimiter. If you do not specify the DELIMITED BY phrase, the string of characters is further examined, beginning with the character to the right of the last character transferred.
- After data is transferred to *data-name-4*, the current receiving area is *data-name-7*. The behavior described in the preceding four paragraphs is repeated until either all the characters are exhausted in *data-name-1*, or there are no more receiving areas.

14. You must initialize the contents of the data items associated with the POINTER phrase or the TALLYING phrase.
15. The contents of *data-name-10* are incremented by one for each character examined in *data-name-1*. When the execution of an UNSTRING statement with a POINTER phrase is complete, *data-name-10* contains a value equal to the initial value, plus the number of characters examined in *data-name-1*.
16. When the execution of an UNSTRING statement with a TALLYING phrase is complete, the contents of *data-name-11* is a value equal to its initial value, plus the number of data receiving items acted upon.
17. Either of the following situations causes an overflow condition:

    - An UNSTRING is initiated, and the value of *data-name-10* is less than 1 or greater than the value of *data-name-1*.
    - During execution of an UNSTRING statement, all data receiving areas are acted upon, and *data-name-1* contains characters that were not examined.

18. When an overflow condition exists, the UNSTRING operation is terminated. If you specify an ON OVERFLOW phrase, *imperative-statement-1* is executed.
19. If an overflow condition does not exist, and you specified the NOT ON OVERFLOW phrase, control is transferred to *imperative-statement-2*. If you specified the ON OVERFLOW phrase, it is ignored. Execution then continues according to the rules listed in Chapter 4.
20. The END-UNSTRING clause delimits the scope of the UNSTRING statement. See the section Scope Terminators at the beginning of this chapter for more information.

21. Subscript evaluation for the *data-names* proceeds as follows:

    - Any subscripting associated with *data-name-1*, *data-name-10*, or *data-name-11* is evaluated only once, immediately before any data is transferred as the result of executing the UNSTRING statement.
    - Any subscripting associated with *data-name-2* through *data-name-6* is evaluated immediately before the transfer of data into the respective data item.

Note

Prime Restriction: You can specify no more than five delimiters within an UNSTRING statement.

Example

```
ID DIVISION.
PROGRAM-ID.  INPUT1.
AUTHOR. GJK.
REMARKS.  THIS IS A SUBPROGRAM THAT ACCEPTS THREE FIELDS.  THE FIRST
   FIELD (CALL-INPUT) CONTAINS THE NUMERIC DATA (LEFT-JUSTIFIED) THAT
   WAS ACCEPTED FROM THE KEYBOARD.  THE DATA WILL BE RETURNED TO THE
   CALLING PROGRAM IN THE SECOND FIELD (CALL-RECEIVE), RIGHT-
   JUSTIFIED IF THE FIRST FIELD IS IN CORRECT FORMAT.  OTHERWISE
   AN ERROR CODE IS RETURNED IN THE THIRD FIELD (CALL-ERROR-CODE).
DATA DIVISION.
WORKING-STORAGE SECTION.
01  AMOUNT-BEFORE-UNSTRING             PIC X(20).
01  UNSTRING-FIELDS.
    05  UN-AMOUNT-1                    PIC 9(16).
    05  UN-AMOUNT-2                    PIC X(2).
01  AMOUNT-ALIGNED   REDEFINES UNSTRING-FIELDS  PIC 9(16)V99.
01  AMOUNT-TEST                        PIC X(20).
01  INS-TALLY                          PIC 99.
*
LINKAGE SECTION.
01  CALL-INPUT                         PIC X(20).
01  CALL-RECEIVE                       PIC 9(16)V99.
01  CALL-ERROR-CODE                    PIC 9.
*
PROCEDURE DIVISION USING CALL-INPUT, CALL-RECEIVE, CALL-ERROR-CODE.
050-MAIN.
    MOVE ZEROS TO UN-AMOUNT-1, INS-TALLY, CALL-ERROR-CODE.
    MOVE SPACES TO UN-AMOUNT-2.
    PERFORM 100-EDIT-AMOUNT.
    IF CALL-ERROR-CODE NOT EQUAL 0 NEXT SENTENCE
       ELSE PERFORM 200-PREPARE-FOR-UNSTRING,
            PERFORM 250-ALIGN-AMOUNT-WITH-UNSTRING,
            MOVE AMOUNT-ALIGNED TO CALL-RECEIVE.
    EXIT PROGRAM.
100-EDIT-AMOUNT.
    IF CALL-INPUT EQUAL SPACES
       MOVE 1 TO CALL-ERROR-CODE
    ELSE PERFORM 150-IS-AMOUNT-NUMERIC.
```

```
150-IS-AMOUNT-NUMERIC.
    MOVE CALL-INPUT TO AMOUNT-TEST.
    INSPECT AMOUNT-TEST TALLYING INS-TALLY FOR ALL '.'.
    IF INS-TALLY EQUAL 0  MOVE 2 TO CALL-ERROR-CODE
        ELSE INSPECT AMOUNT-TEST REPLACING ALL SPACES BY ZEROES,
             INSPECT AMOUNT-TEST REPLACING FIRST '.' BY ZERO
             IF AMOUNT-TEST IS NUMERIC NEXT SENTENCE,
                ELSE MOVE 1 TO CALL-ERROR-CODE.
200-PREPARE-FOR-UNSTRING.
    INSPECT CALL-INPUT REPLACING ALL SPACES BY ZEROES.
    MOVE CALL-INPUT TO AMOUNT-BEFORE-UNSTRING.
250-ALIGN-AMOUNT-WITH-UNSTRING.
    UNSTRING AMOUNT-BEFORE-UNSTRING DELIMITED BY '.'
        INTO UN-AMOUNT-1, UN-AMOUNT-2.
```

USE

Specifies procedures for input-output error handling.

Format

Syntax Rules

1. A USE statement, when present, must immediately follow a section header in the declaratives section, separated from it by a period and a space. The remainder of the section must consist of zero, one, or more paragraphs that define the procedures to be used.
2. The USE statement itself is never executed; rather, it defines the conditions for the execution of the following paragraphs.
3. The same *file-name* can appear in more than one USE statement, in a different specific arrangement of the format. Appearance of a *file-name* in a USE statement must not cause the simultaneous request for execution of more than one USE procedure.
4. The words EXCEPTION and ERROR are interchangeable.
5. The files implicitly or explicitly referenced in a USE statement need not all have the same organization or access.

General Rules

1. The paragraphs introduced by USE are executed after the standard I-O recovery for the following conditions:

    - After the end-of-file condition arises on a statement lacking the AT END clause

- After the invalid key condition arises on a statement lacking the INVALID KEY clause
- After a permanent error condition or logic error condition arises
- After a recoverable error condition arises that is not an invalid key or end-of-file condition

See Chapter 4 for a full discussion of I-O status codes and runtime error recovery.

2. If more than one USE procedure applies to a file because of an explicit reference to the *file-name* in one USE statement and an implicit reference to the OPEN mode in another USE statement, the USE procedure explicitly referencing the *file-name* is invoked when and if required.
3. For recoverable errors, after a USE procedure is executed, control returns to the statement following the I-O statement whose execution resulted in invoking the USE procedure. For fatal errors, after a USE procedure is executed, the program terminates.
4. A USE procedure must not contain any reference to a nondeclarative procedure. Conversely, the nondeclarative portion of a USE procedure must not contain any reference to *procedure-names* that appear in the declarative portion, except that PERFORM statements can refer to a USE statement or to the procedures associated with it.
5. Because tape files can only be opened in INPUT or OUTPUT mode, USE procedures for I-O or EXTEND are ignored for tape files.
6. During the execution of a USE procedure for a fatal error, no input-output operations on the file in error are allowed, except for a CLOSE statement.

   During the execution of a USE procedure for any type of error, no statement can be executed that would cause the execution of a USE procedure that had previously been invoked and had not yet returned control to the invoking routine.

   See Chapter 4 for a full discussion of I-O status codes and error recovery.

Example

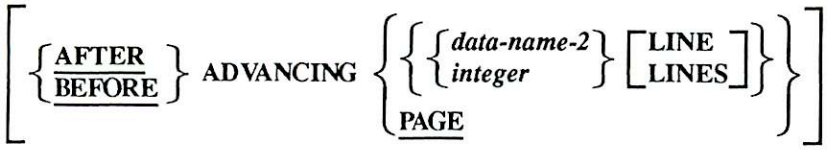
The sample program at the end of this chapter contains an example of USE in declarative sections for disk and tape files.

WRITE

Releases a logical record for an output or I-O file. Use it also for vertical positioning of lines within a logical page.

Format 1

WRITE *record-name* [FROM *data-name-1*]



[END-WRITE]

Format 2

WRITE *record-name* [FROM *data-name*]

[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-WRITE]

Chapters 9, 10, 11, and 12 describe the use of the WRITE statement with sequential files, indexed files, relative files, and tape files, respectively.

PROCEDURE DIVISION Example

This example, with the examples at the end of Chapters 5, 6, and 7, forms one program.

```
 PROCEDURE DIVISION.
*
 DECLARATIVES.
 INPUT-ERROR SECTION. USE AFTER ERROR PROCEDURE ON DISK-FILE.
 FIRST-PARAGRAPH.
     DISPLAY  '**** I-O ERROR ON ENTRY: ***', FILE STATUS.
     DISPLAY  ENTRY-DETAIL.
     CLOSE DISK-FILE, PRINT-FILE.
     STOP RUN.
 TAPE-ERROR SECTION. USE AFTER ERROR PROCEDURE ON TAPE-FILE.
 SECOND-PARAGRAPH.
 END DECLARATIVES.
*
 001-BEGIN.
     READY TRACE.
     OPEN INPUT DISK-FILE, OUTPUT PRINT-FILE.
     PERFORM 010-GET-JOBINFO.
     PERFORM 020-NEW-DETAIL-PAGE.
     PERFORM 030-PROCESS-DETAIL.
     PERFORM 070-TOTALS.
     PERFORM 080-BALANCE-TOTALS.
```

```
           PERFORM 090-PROCESS-TAPE.
           CLOSE DISK-FILE, PRINT-FILE.
           DISPLAY '     END OF RUN'.
           STOP RUN.
      *
       010-GET-JOBINFO.
           DISPLAY 'ENTER MONTH(ALPHA)'. ACCEPT VARIABLE-MONTH.
           DISPLAY 'ENTER JOB CODE'. ACCEPT JOB-CODE.
           IF NOT CORRECT-CODE DISPLAY 'WRONG CODE',
               CLOSE DISK-FILE, PRINT-FILE, STOP RUN.
      *
       020-NEW-DETAIL-PAGE.
           MOVE '      DETAIL LIST       ' TO VARIABLE-HEADING.
           PERFORM 150-NEW-PAGE THRU 150-NEW-PAGE-EXIT.
           MOVE SPACES TO PRINT-LINE.
           MOVE ' DATE             VENDOR                CHECK
      -    '      ACCOUNT       AMOUNT' TO PRINT-LINE.
           WRITE PRINT-LINE AFTER ADVANCING VARIABLE LINES.
           ADD 4 TO LINECOUNT.
           EJECT
       030-PROCESS-DETAIL.
           READ DISK-FILE AT END MOVE 'Y' TO NO-MORE-RECORDS,
               DISPLAY 'INPUT FILE WAS EMPTY',
               CLOSE PRINT-FILE, DISK-FILE, STOP RUN.
           PERFORM 035-READ-AND-PRINT UNTIL NO-MORE-RECORDS = 'Y'.
           EXIT.
          035-READ-AND-PRINT.
           MOVE 0 TO ERR-CODE.
           PERFORM 040-EDIT.
           IF ERR-CODE NOT = 0 PERFORM 060-REJECTS,
           ELSE PERFORM 050-DEPT-TOTALS.
           MOVE 1 TO VARIABLE.
           MOVE CORRESPONDING ENTRY-DETAIL TO PRINT-DETAIL.
           MOVE ENTRY-ACCT-NO TO PRINT-ACCT-NO.
           MOVE ENTRY-AMOUNT TO PRINT-AMOUNT.
           WRITE PRINT-LINE FROM PRINT-DETAIL AFTER ADVANCING
               1 LINE.
           ADD 1 TO LINECOUNT.
           IF LINECOUNT > 50 PERFORM 020-NEW-DETAIL-PAGE.
      ****************************************************************
      * READ ALL SUBSEQUENT ENTRIES.                                 *
      ****************************************************************
           READ DISK-FILE AT END MOVE 'Y' TO NO-MORE-RECORDS.
           EXIT.
      *
       040-EDIT.
      ****************************************************************
      * ONLY ONE ERROR IS FLAGGED FOR EACH REJECT                    *
      ****************************************************************
           MOVE 0 TO ERR-CODE.
           IF ENTRY-ACCT-NO NOT NUMERIC  MOVE 1 TO ERR-CODE.
```

```
      IF ENTRY-ACCT-NO LESS THAN 100 OR GREATER THAN 449,
          MOVE 2 TO ERR-CODE.
      IF ENTRY-AMOUNT NOT NUMERIC  MOVE 3 TO ERR-CODE.
      IF ENTRY-MONTH OF ENTRY-DETAIL NOT NUMERIC MOVE 4 TO
          ERR-CODE.
      IF ENTRY-CHECK-NO OF ENTRY-DETAIL NOT NUMERIC,
            MOVE 7 TO ERR-CODE.
*
 050-DEPT-TOTALS.
****************************************************************
* MAKE CROSS-TOTAL AS CHECK,                                   *
* FIND HOME-ACCOUNT FOR EACH ACCOUNT NUMBER.                   *
* ADD ENTRY-AMOUNT TO HOME DEPARTMENT TOTAL.                   *
****************************************************************
      ADD ENTRY-AMOUNT TO CROSS-TOTAL.
      IF ENTRY-ACCT-NO LESS THAN 200, ADD ENTRY-AMOUNT TO TOTAL1,
      ELSE  IF ENTRY-ACCT-NO LESS THAN 300 AND ENTRY-ACCT-NO >
                        199, ADD ENTRY-AMOUNT TO TOTAL2,
            ELSE IF ENTRY-ACCT-NO LESS THAN 420 AND ENTRY-ACCT-NO
                      > 300, ADD ENTRY-AMOUNT TO TOTAL3,
                 ELSE IF ENTRY-ACCT-NO LESS THAN 430 AND
                        ENTRY-ACCT-NO > 419, ADD ENTRY-AMOUNT TO
                                             TOTAL4,
                      ELSE IF ENTRY-ACCT-NO  LESS THAN 440 AND
                             ENTRY-ACCT-NO > 429, ADD
                             ENTRY-AMOUNT TO TOTAL5,
                           ELSE IF ENTRY-ACCT-NO > 439, ADD
                                ENTRY-AMOUNT TO TOTAL6.
*
 060-REJECTS.
****************************************************************
* MAKE CROSS-TOTAL FOR REJECTS.                                *
****************************************************************
      IF ENTRY-AMOUNT NUMERIC, ADD ENTRY-AMOUNT TO REJECT-TOTAL.
      MOVE  ' ** ERROR FOLLOWS **' TO MESSAGE.
      WRITE ERROR-LINE.
      EJECT
 070-TOTALS.
      MOVE 'TOTALS BY ACCOUNT NUMBER' TO VARIABLE-HEADING.
      PERFORM 150-NEW-PAGE THRU 150-NEW-PAGE-EXIT.
      MOVE '           ACCOUNT             TOTAL DISBURSEMENT
-     ' ' TO PRINT-LINE.
      WRITE PRINT-LINE AFTER ADVANCING VARIABLE LINES.
****************************************************************
* PRINT TOTALS FOR EACH HOME ACCOUNT                           *
****************************************************************
      MOVE '100' TO HOME-NUMBER.
      MOVE TOTAL1 TO HOME-TOTAL.
      WRITE PRINT-LINE FROM HOME-ACCT-LINE AFTER ADVANCING 1.
      MOVE '200' TO HOME-NUMBER.
      MOVE TOTAL2 TO HOME-TOTAL.
```

```
           WRITE PRINT-LINE FROM HOME-ACCT-LINE AFTER ADVANCING 1.
           MOVE '410' TO HOME-NUMBER.
           MOVE TOTAL3 TO HOME-TOTAL.
           WRITE PRINT-LINE FROM HOME-ACCT-LINE AFTER ADVANCING 1.
           MOVE '420' TO HOME-NUMBER.
           MOVE TOTAL4 TO HOME-TOTAL.
           WRITE PRINT-LINE FROM HOME-ACCT-LINE AFTER ADVANCING 1.
           MOVE '430' TO HOME-NUMBER.
           MOVE TOTAL5 TO HOME-TOTAL.
           WRITE PRINT-LINE FROM HOME-ACCT-LINE AFTER ADVANCING 1.
           MOVE '440' TO HOME-NUMBER.
           MOVE TOTAL6 TO HOME-TOTAL.
           WRITE PRINT-LINE FROM HOME-ACCT-LINE AFTER ADVANCING 1.
           MOVE 'REJ' TO HOME-NUMBER.
           MOVE REJECT-TOTAL TO HOME-TOTAL.
           WRITE PRINT-LINE FROM HOME-ACCT-LINE AFTER ADVANCING 1.
           ADD TOTAL1, TOTAL2, TOTAL3, TOTAL4, TOTAL5, TOTAL6,
             REJECT-TOTAL GIVING FINAL-TOTAL.
           MOVE 'FINAL TOTAL          ' TO HOME-NUMBER.
           MOVE FINAL-TOTAL TO HOME-TOTAL.
           WRITE PRINT-LINE FROM HOME-ACCT-LINE AFTER ADVANCING 2.
       080-BALANCE-TOTALS.
           MOVE '      BALANCE RUN       ' TO VARIABLE-HEADING.
           PERFORM 150-NEW-PAGE THRU 150-NEW-PAGE-EXIT.
      ****************************************************************
      * GOOD ITEMS AND REJECTS ARE ADDED FOR GRAND-TOTAL, WHICH      *
      * COMPARED WITH THE FINAL TOTAL (OBTAINED BY ADDING ACCOUNT    *
      * TOTALS AND REJECT TOTAL).                                    *
      ****************************************************************
           MOVE '            GOOD ITEMS         REJECT TOTAL
      -     '        GRAND-TOTAL  ' TO PRINT-LINE.
           WRITE PRINT-LINE AFTER ADVANCING VARIABLE LINES.
           ADD CROSS-TOTAL, REJECT-TOTAL GIVING GRAND-TOTAL.
           MOVE GRAND-TOTAL TO FIELD-DIFF.
           MOVE REJECT-TOTAL TO FIELD-REJECT.
           MOVE CROSS-TOTAL TO FIELD-TOTAL.
           WRITE PRINT-LINE FROM BALANCE-LINE AFTER ADVANCING 1.
           IF GRAND-TOTAL NOT EQUAL FINAL-TOTAL,
               MOVE '*** TOTALS DO NOT BALANCE ***' TO ERROR-LINE,
               WRITE ERROR-LINE AFTER ADVANCING 2 LINES.
       090-PROCESS-TAPE.
           DISPLAY 'IS TAPE OUTPUT DESIRED--ENTER YES OR NO '.
           ACCEPT TAPE-CHOICE.
           IF TAPE-CHOICE = 'yes' OR
              TAPE-CHOICE = 'YES' PERFORM 095-WRITE-TAPE,
           ELSE DISPLAY 'NO TAPE'.
      *
       095-WRITE-TAPE.
           OPEN OUTPUT TAPE-FILE.
           MOVE 1 TO VARIABLE.
           MOVE VARIABLE-MONTH TO TAPE-MONTH.
```

```
           WRITE TAPE-LINE FROM TAPE-HEADER.
           ACCEPT JOB-DATE FROM DATE.
           MOVE JOB-DATE TO SAVE-DATE-TAPE.
           MOVE '100' TO SAVE-ACCT-TAPE.
           MOVE TOTAL1 TO SAVE-TOTAL-TAPE.
           WRITE TAPE-LINE FROM SAVE-TAPE.
           MOVE '200' TO SAVE-TOTAL-TAPE.
           MOVE TOTAL2 TO SAVE-TOTAL-TAPE.
           WRITE TAPE-LINE FROM SAVE-TAPE.
           MOVE '410' TO SAVE-ACCT-TAPE.
           MOVE TOTAL3 TO SAVE-TOTAL-TAPE.
           WRITE TAPE-LINE FROM SAVE-TAPE.
           MOVE '420' TO SAVE-ACCT-TAPE.
           MOVE TOTAL4 TO SAVE-TOTAL-TAPE.
           WRITE TAPE-LINE FROM SAVE-TAPE.
           MOVE '430' TO SAVE-ACCT-TAPE.
           MOVE TOTAL5 TO SAVE-TOTAL-TAPE.
           WRITE TAPE-LINE FROM SAVE-TAPE.
           MOVE '440' TO SAVE-ACCT-TAPE.
           MOVE TOTAL6 TO SAVE-TOTAL-TAPE.
           WRITE TAPE-LINE FROM SAVE-TAPE.
           CLOSE TAPE-FILE.
           PERFORM 095-VERIFY-TAPE.
       095-VERIFY-TAPE.
           DISPLAY 'FIRST TAPE RECORD - VERIFICATION ONLY'.
           OPEN INPUT TAPE-FILE.
           READ TAPE-FILE INTO TAPE-HEADER.
           READ TAPE-FILE.
           DISPLAY TAPE-LINE.
           CLOSE TAPE-FILE.
           EXIT.
      *
       150-NEW-PAGE.
           MOVE PAGECOUNT TO HEADING-PAGE.
           MOVE 2 TO VARIABLE.
           WRITE PRINT-LINE FROM HEADING1 AFTER ADVANCING PAGE.
           WRITE PRINT-LINE FROM HEADING2 AFTER ADVANCING
              VARIABLE LINES.
           WRITE PRINT-LINE FROM HEADING3 AFTER ADVANCING
                 VARIABLE LINES.
           ADD 1 TO PAGECOUNT.
           MOVE SPACES TO PRINT-LINE.
           MOVE 8 TO LINECOUNT.
       150-NEW-PAGE-EXIT.
```

The following dialog compiles, links, and executes this program, stored as OLDCASH.COBOL85. (The example at the end of Chapter 12 executes the tape sections.)

```
OK, COBOL85 OLDCASH -LISTING
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]
[0 ERRORS IN PROGRAM: OLDCASH.COBOL85]

OK, BIND -LOAD OLDCASH -LI COBOL85LIB -LI
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
BIND COMPLETE

OK, RESUME OLDCASH
ENTER MONTH(ALPHA)
NOVEMBER, 1987
ENTER JOB CODE
25
 TAPE OUTPUT DESIRED--ENTER YES OR No
NO
NO TAPE
     END OF RUN
OK,
```

A sample input file (DISBURSE) is

```
408080185   ASHTABULA HDWE       4300035476
409080185   CAIRO CHEMICAL       4360002746
410080285   ST.BOTOLPHSTOWN SUPP4200005108
411080285   DOVER MUTUAL         4100034166
412080385   PARIS AUTO           4100015000
413090385   ROME BOATING         4150017982
C82080785   ODESSA SERVICES      4100004670
4500B0785   ANTIOCH SERVALL      4300002580
580080785   BETHLEHEM TAXI       RR00009840
680080785   ATHENS LUMBER        18500036BB
```

A sample output file (PRINT-FILE) is

```
                    MONTHLY CASH DISBURSEMENTS JOURNAL

                         FOR    NOVEMBER, 1987                  PAGE 1

                                DETAIL LIST

DATE            VENDOR                  CHECK      ACCOUNT     AMOUNT
080185      ASHTABULA HDWE               408          430       354.76
080185      CAIRO CHEMICAL               409          436        27.46
080285      ST.BOTOLPHSTOWN SUPP         410          420        51.08
080285      DOVER MUTUAL                 411          410       341.66
080385      PARIS AUTO                   412          410       150.00
090385      ROME BOATING                 413          415       179.82
** ERROR FOLLOWS **
080785      ODESSA SERVICES              C82          410        46.70
```

```
** ERROR FOLLOWS **
0B0785      ANTIOCH SERVALL          450        430       25.80
** ERROR FOLLOWS **
080785      BETHLEHEM TAXI           580        RR0       98.40
** ERROR FOLLOWS **
080785      ATHENS LUMBER            580        185       36.BB
```

MONTHLY CASH DISBURSEMENTS JOURNAL

FOR NOVEMBER, 1987 PAGE 2

TOTALS BY ACCOUNT NUMBER

ACCOUNT	TOTAL DISBURSEMENT
100	.00
200	.00
410	671.48
420	51.08
430	382.22
440	.00
REJ	170.90
FINAL TOTAL	1275.68

MONTHLY CASH DISBURSEMENTS JOURNAL

FOR NOVEMBER, 1987 PAGE 3

BALANCE RUN

GOOD ITEMS	REJECT TOTAL	GRAND-TOTAL
1104.78	170.90	1275.68

9 Sequential Files

COBOL85 allows you to access records of a disk file in an established sequence. The record sequence is established as a result of writing the records to the file.

This chapter discusses sequential file concepts, common operations on sequential files, and PROCEDURE DIVISION verbs as they pertain to sequential file processing.

Note

The discussions in this chapter pertain to sequential disk files. See Chapter 12 for a discussion of sequential tape files.

Sequential File Concepts

Organization

Sequential files are organized such that each record, except the last, has a unique successor record, and each record, except the first, has a unique predecessor record. The successor relationships are established by the order of execution of WRITE statements when the file is created. Once established, successor relationships do not change, except in the case of records being added to the end of a file.

A sequential disk file has the same logical structure as a sequential tape file. However, you can use the REWRITE statement to update a sequential disk file in place. When you use REWRITE, each updated record must be the same size as the original record. You cannot use the REWRITE statement to add new records to the file.

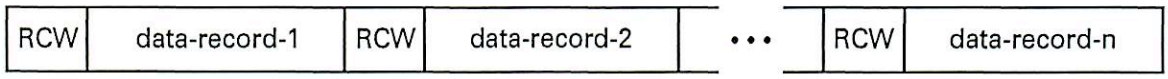
Access Mode

The order of sequential access is the order in which the records are originally written to the file.

File Formats

You can use variable-length record formatting or fixed-length record formatting for PRIMOS and PRISAM sequential files. For PRISAM files, the template for the file must define the file's format consistent with the COBOL85 program definition. For additional information, see the *PRISAM User's Guide*.

Figure 9-1 illustrates variable-length record format for PRIMOS sequential disk files.



Q10166-1LA-3-0

```
where RCW = Record Control Word
            (an octal value containing the size
             in words of each data record)

data record n = actual data record
```

FIGURE 9-1
PRIMOS Sequential Variable-length Record Format

Notes

This file format is consistent with the FTNBIN format associated with variable-length record files in FTN. Such a file cannot be accessed by ED, EMACS, SLIST, and so on.

Odd-length records contained in PRIMOS sequential disk files contain an extra byte of data to fill the record out to the word boundary. This extra byte of data is undefined. For PRISAM sequential files, odd-length records are padded in such a way that this extra byte is not visible to the user.

Current Record Pointer

The **current record pointer** is a conceptual entity used to indicate the next record to be accessed within a given file. The setting of the current record pointer is affected only by the OPEN, CLOSE, and READ statements. The concept of the current record pointer has no meaning for a file opened in OUTPUT or EXTEND mode.

File Status

To determine the success or failure of an I-O operation, code a file status check in the program. You then can use the result of the file status check to control the next program action. If you specify the FILE STATUS clause in a *file-control-entry*, a value is automatically placed into the FILE STATUS data item during the execution of an OPEN, CLOSE, READ, WRITE, or REWRITE statement to indicate the status of that I-O operation. The FILE STATUS clause is discussed in Chapter 6.

For a complete discussion of COBOL85 file status codes, see Chapter 4.

The AT END Condition

The AT END condition can occur as a result of a READ statement when no next logical record exists in the file. When the AT END condition is recognized, these actions occur in the following order:

1. A value indicating an AT END condition is placed into the FILE STATUS data item, if you specify one for the file.
2. If you specify the AT END phrase in the statement that causes the condition, control is transferred to the imperative statement in the AT END phrase. Any USE procedure that you specify for the file is not executed. After the execution of the imperative statement, control is transferred to the end of the I-O statement. The NOT AT END phrase, if you specify one, is ignored.
3. If you do not specify the AT END phrase in the statement that causes the condition, but you do specify a USE procedure, either explicitly or implicitly, that procedure is executed.

The NOT AT END Condition

The NOT AT END condition can occur as a result of a successfully completed READ statement. When the NOT AT END condition is recognized, these actions occur in the following order:

1. The FILE STATUS data item, if you specify one, is updated to indicate a successful completion.
2. If you specify the NOT AT END phrase, control is transferred to the imperative statement associated with the NOT AT END phrase. After the execution of the imperative statement in the NOT AT END phrase, control is transferred to the end of the I-O statement. The AT END phrase, if you specify one, is ignored.
3. If you do not specify the NOT AT END phrase, control is transferred to the end of the I-O statement. The AT END phrase, if you specify one, is ignored.

Exception Conditions

Exception conditions can occur as a result of a Permanent Error condition or a Logic Error condition. When an exception condition that is not an AT END condition occurs, these actions occur in the following order:

1. A value indicating the exception condition is placed into the FILE STATUS data item, if you specify one for the file.
2. If you specify a USE procedure for the file, that procedure is executed. Any AT END and NOT AT END phrases are ignored.
3. Program execution terminates.

Chapter 4 includes a complete discussion of exception conditions and error recovery.

Common Operations on Sequential Files

This section discusses the following common operations on sequential files:

- Opening and closing a file
- Reading a file
- Updating (changing) a record
- Creating (adding) records
- Handling I-O errors

Opening and Closing a File

You must open files with the OPEN statement before any other I-O statements are executed, and you must close files with the CLOSE statement before the program ends. You must also close files before reopening them in another mode of operation.

Reading a File

If a file is open, no special operation other than READ is necessary to read from the start of the file in sequential order.

Updating (Changing) a Record

You must open the file for I-O. Use the READ statement to read the file and the REWRITE statement to update records in place.

Creating (Adding) Records

Use the WRITE statement to add new records to the file.

Handling I-O Errors

The FILE STATUS data item, the AT END phrase in the READ statement, and USE procedures in the declaratives section provide a means of handling I-O errors.

PROCEDURE DIVISION

This section contains information that pertains to sequential disk files. Chapter 8 contains information that applies to all file organizations. Chapters 10, 11, and 12 contain information that pertains to indexed files, relative files, and sequential tape files, respectively.

CLOSE

Terminates the processing of files.

Format

CLOSE *file-name-1* [,*file-name-2*] . . .

Syntax Rule

The files referenced in the CLOSE statement need not all have the same access or organization.

General Rules

1. Once a CLOSE statement is executed for a file, no other statement can be executed for that file unless an intervening OPEN statement for that file is executed.
2. A CLOSE statement can be executed only for a file that is open. If a CLOSE statement is attempted on a file that is not open, the FILE STATUS data item, if you specify one, is set to status code 42. After execution of any applicable declarative procedure, the program terminates. For this particular error, the only applicable declarative procedure is a procedure for *file-name*.
3. If any other error occurs during a CLOSE operation, the FILE STATUS data item, if you specify one, is set to status code 30. Execution continues according to the rules specified in Chapter 4.

CLOSE Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a CLOSE statement: 00, 07, 30, 42, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

OPEN

Initiates the processing of files and enables other I-O operations, such as reading and writing.

Format

OPEN { INPUT *file-name-1* [, *file-name-2*] . . .
OUTPUT *file-name-3* [, *file-name-4*] . . .
I-O *file-name-5* [, *file-name-6*] . . .
EXTEND *file-name-7* [, *file-name-8*] . . . } . . .

Syntax Rules

1. The files referred to in the OPEN statement need not all have the same organization or access.
2. You can use the EXTEND phrase only for sequential files assigned to PRIMOS, PRISAM, PRINTER, or PFMS.
3. You can open files assigned to PRINTER only in OUTPUT or EXTEND mode.

4. You can open files assigned to OFFLINE-PRINT only in OUTPUT mode.
5. You can open files assigned to TERMINAL only in INPUT or OUTPUT mode.

General Rules

1. For each file, an OPEN statement must be executed prior to a READ, WRITE, REWRITE, or CLOSE statement for that file.
2. An OPEN statement can be executed only for a file that is closed.
3. A file opened as INPUT can be accessed only by a READ statement.
4. A file opened as OUTPUT can be accessed only by a WRITE statement.
5. A file opened as I-O can be accessed by a READ, WRITE, or REWRITE statement.
6. A file opened as EXTEND can be accessed only by a WRITE statement.
7. If permitted for the device, a file can be opened with the INPUT, OUTPUT, EXTEND, and I-O phrases in the same program. Following the initial OPEN, you must precede each subsequent OPEN statement for the file by a CLOSE statement for the file.
8. OPEN OUTPUT causes PRIMOS to create a file if one does not exist. The ASSIGN clause associated with the file must not state PRISAM as the file type. If the ASSIGN clause does state PRISAM, the OPEN statement is unsuccessful. All other sequential file OPEN statements cause the file to be created as a PRIMOS sequential file. The created file contains no records.

   If a PRIMOS or PFMS sequential file contains records when it is opened for OUTPUT, the file is truncated. All other file types generate an error if the file contains records when it is opened for OUTPUT.
9. For an optional file that is unavailable, the successful execution of an OPEN statement with an EXTEND or I-O phrase creates the file. The ASSIGN clause associated with the file must not state PRISAM as the file type.

   For an optional file that is unavailable and is opened in INPUT mode, the first READ statement to the file returns an end-of-file status code.

   For an optional file that is available, file attributes are checked and normal OPEN processing continues.
10. The *file-description-entry* for files opened with INPUT, I-O, or EXTEND must be equivalent to that used when the file was created.
11. The current record pointer is a conceptual flag pointing to the next record to be accessed. For files opened with the INPUT or I-O phrase, the OPEN statement sets the current record pointer to the first record in the file. If no records exist in the file, the current record pointer is set such that the next executed READ statement for the file results in an AT END condition.
12. When you specify the EXTEND phrase, the OPEN statement opens the file, and moves the current record pointer to the bottom of the file (immediately following the last logical record). Subsequent WRITE statements to the file add records as if the file were opened with the OUTPUT phrase, with the current record pointer at the end of the file.
13. The LABEL RECORDS ARE STANDARD clause is ignored for non-magtape files. No label processing takes place on OPEN statements.

14. If any system error occurs during the OPEN processing, a status code of 30 is returned. An exception condition occurs, and the FILE STATUS field is updated, if one exists. Exception conditions and file status are discussed in the section, Sequential File Concepts, earlier in this chapter.
15. During the execution of the OPEN statement, COBOL85 checks file attributes against the attributes described for the file in the program. These attributes are organization, minimum and maximum logical record sizes, and the file type (fixed or variable).

    If any of these attributes conflict, the OPEN statement is unsuccessful, and a status code of 39 is returned. An exception condition occurs, and the FILE STATUS field is updated, if one exists. Exception conditions and file status are discussed in the section, Sequential File Concepts, earlier in this chapter.
16. An OPEN statement must be successfully executed for a file before any statement that references the file (except SORT and MERGE) can be executed.
17. If an error occurs during an OPEN operation, the FILE STATUS data item, if you specify one, is updated. Then control is transferred to the declarative procedure, if you specify one. If a declarative procedure exists for the *file-name* being opened, it is invoked; otherwise, if a declarative procedure exists for the open mode being attempted, it is invoked. After execution of the declarative procedure, the program terminates.
18. A file referenced in a SORT or MERGE statement must not be open at the time of execution of the SORT or MERGE statement. Once a file is opened under control of SORT or MERGE, no I-O operation other than those under control of SORT or MERGE can be executed until the sort or merge is completed.

OPEN Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of an OPEN statement: 00, 05, 07, 30, 35, 37, 39, 41, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

READ

Makes available a record from a file.

Format

READ *file-name* **RECORD** [**INTO** *data-name-1*]

[**AT END** *imperative-statement-1*]

[**NOT AT END** *imperative-statement-2*]

[**END-READ**]

Syntax Rule

You must specify the AT END phrase if you do not specify an applicable USE procedure for *file-name*.

General Rules

1. A file must be open for INPUT or I-O when a READ statement for the file is executed.
2. Execution of the READ statement makes a record available to the program, provided AT END is not invoked. The READ statement uses the current record pointer, a conceptual entity that points to the next record to be accessed. The record made available by the READ statement is determined as follows:

   - If the current record pointer is positioned by the execution of an OPEN statement, the record indicated by the current record pointer is made available.
   - If the current record pointer is positioned by the execution of a previous READ statement, the current record pointer is set to point to the next record in the file, and that record is made available.

3. The execution of the READ statement updates the value of any FILE STATUS data item associated with *file-name*.
4. When you describe the logical records of a file with more than one record description, these records automatically share the same storage area; this is equivalent to an implicit redefinition of the area. The contents of any data items that lie beyond the range of the current data record are undefined at the completion of execution of the READ statement.
5. For variable-length records, the following rules apply:

   - If the number of character positions in the record that is read is less than the minimum size specified in the *record-description-entry*, the portion of the record area to the right of the last valid character read is undefined.
   - If the number of character positions in the record that is read is larger than the maximum size specified in the *record-description-entry*, the record is truncated on the right to the maximum size.

   In either of these cases, the READ statement is successful and the FILE STATUS data item, if you specify one, is set to 04 to indicate a record length conflict.
6. If you specify the INTO phrase, the record being read is moved from the record area to the area specified by *data-name-1*, according to the rules for the MOVE statement without the CORRESPONDING phrase. The implied MOVE does not occur if the execution of the READ statement is unsuccessful. Any subscripting or indexing associated with *data-name-1* is evaluated after the record is read and immediately before it is moved to the data item. *data-name-1* must not be defined in the FD entry for the file.
7. When you use the INTO phrase, the record being read is available in both the input record area and the data area associated with *data-name-1*.
8. If, at the time of execution of a READ statement, the position of the current record pointer for the file is undefined, the execution of the READ statement is unsuccessful.
9. If, at the time of execution of a READ statement, no next logical record exists in the file, or an optional input file does not exist, the AT END condition occurs, and the execution of the READ statement is unsuccessful.

10. When the AT END condition is recognized, these actions occur in the following order:

    - A value indicating an AT END condition is placed into the FILE STATUS data item, if you specify one for the file.
    - If you specify the AT END phrase in the statement causing the condition, control is transferred to *imperative-statement-1*. *imperative-statement-2*, if you specify one, is ignored. Any USE procedure that you specify for the file is not executed.
    - If you do not specify the AT END phrase, then you must specify a USE procedure for the file. That procedure is now executed.

    When the AT END condition occurs, execution of the I-O statement that caused the condition is unsuccessful.
11. If an exception condition other than an AT END condition occurs, the FILE STATUS data item, if you specify one, is set to indicate the error condition. After execution of any applicable declarative procedure, the program terminates.
12. Following the unsuccessful execution of any READ statement, the contents of the associated record area and the position of the current record pointer are undefined.
13. If no exception condition occurs and the AT END condition does not occur, the FILE STATUS data item, if you specify one, is set to indicate a successful READ statement. Control is transferred to the end of the READ statement, or to *imperative-statement-2*, if you specify one.
14. The END-READ clause delimits the scope of the READ statement. For more information, see the section Scope Terminators, in Chapter 8.

READ Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a READ statement: 00, 04, 10, 30, 46, 47, 93, 97, 99 (PRISAM only). For a complete discussion of COBOL85 file status codes, see Chapter 4.

REWRITE

Logically replaces a record existing in a disk file.

Format

REWRITE *record-name* [**FROM** *data-name*]

[**END-REWRITE**]

Syntax Rules

1. The *record-name* and the *data-name* can refer to the same storage area.
2. The *record-name* is the name of a logical record in the FILE SECTION and can be qualified.

General Rules

1. The file containing *record-name* must be a disk file and must be open for I-O prior to execution of a REWRITE statement.
2. The last I-O statement executed for the associated file prior to the execution of the REWRITE statement must be a successfully executed READ statement. REWRITE logically replaces the record that is accessed by the READ statement.
3. The number of character positions in the record referenced by *record-name* must be equal to the number of character positions in the record being replaced.
4. **Prime Extension:** The logical record released by a successful execution of the REWRITE statement is still available in the record area.
5. If you use the FROM phrase, the information in *data-name* is moved to the record area prior to the REWRITE.
6. The current record pointer (the conceptual entity that determines the next record to be accessed) is not affected by the execution of a REWRITE statement.
7. The execution of the REWRITE statement updates the value of any FILE STATUS data item associated with the file.
8. A sequential file used with REWRITE must be either a nonprinter file created by COBOL85, or any uncompressed file. Compressed files cannot be rewritten. See Chapter 7 for more information about compressed files.
9. The END-REWRITE clause delimits the scope of the REWRITE statement. For more information, see the section Scope Terminators, in Chapter 8.

REWRITE Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a REWRITE statement: 00, 37, 43, 44, 49, 99 (PRISAM only). For a complete discussion of COBOL85 file status codes, see Chapter 4.

WRITE

Releases a logical record for an output or I-O file. Use it also to vertically position lines within a logical page.

Format

```
WRITE record-name [FROM data-name-1]

[ { AFTER  } ADVANCING { { { data-name-2 } [ LINE  ] } }
  { BEFORE }           { { { integer     } [ LINES ] } }
                       { PAGE                        }  ]

[END-WRITE]
```

Syntax Rules

1. The *record-name* and *data-name-1* can refer to the same storage area.
2. The *record-name* is the 01-level *record-name* of a logical record, described in a *record-description-entry* in the FILE SECTION of the DATA DIVISION. It can be qualified.
3. When you use *data-name-2* in the ADVANCING phrase, it must be the name of an elementary integer data item.
4. The integer or the value of the data item referenced by *data-name-2* must be in the range 0 through 62.

General Rules

1. The file associated with *record-name* must be open as OUTPUT or EXTEND.
2. **Prime Extension:** The logical record released by the execution of the WRITE statement is still available in the record area.
3. If you name the associated file in a SAME RECORD AREA clause, the logical record is available to the program as a record of other files referenced in the same SAME RECORD AREA clause, as well as to the file associated with *record-name*.
4. If you use the FROM phrase, the information is moved to the record area prior to execution of the WRITE statement. If the data being moved is longer than the receiving field, the data is truncated to the size of the receiving field. If the receiving field is longer than the sending field, the remaining area is space-filled.

   After execution of the WRITE statement, the information in the area referenced by *data-name-1* is still available.
5. The current record pointer (the conceptual entity that determines the next record to be accessed) is unaffected by the execution of a WRITE statement.
6. The execution of the WRITE statement causes the value of any FILE STATUS data item associated with the file to be updated.
7. You establish the maximum record size for a file at the time you create the file, and you must not subsequently change it.
8. The number of character positions on a disk required to store a logical record in a file may or may not be equal to the number of character positions defined by the logical description of that record in the program.
9. The execution of the WRITE statement releases a logical record to the file system.
10. The ADVANCING phrase is meaningful only if the file is assigned to PRINTER.
11. If you use the ADVANCING phrase, COBOL85 reserves the print control character internally. Do not reserve the first position in the record as FILLER for the print control character.

    - If you use the BEFORE phrase, a line is written before advancing.
    - If you use the AFTER phrase, spacing occurs first, and then the line is written.
    - *data-name-2* LINE(S) is the number of spacing lines required between data lines. The value of *data-name-2* must be in the range 0 through 62.

    If you do not use the ADVANCING phrase, the default is one line.

12. Table 9-1 lists the significance of the integer values in the ADVANCING phrase.
13. If you specify PAGE, the record is presented on the logical page before or after (depending on the phrase used) the device is repositioned to the next logical page.
14. If the number of character positions in *record-name* is larger than the largest or smaller than the smallest number of character positions allowed for the file, an exception condition exists and the contents of the record area are unaffected. The FILE STATUS data item, if you specify one, is set to indicate the error condition, and execution continues as specified in the section, I-O Status Codes and Error Recovery, in Chapter 4.

TABLE 9-1
Carriage Control Integer Values

Integer	*Carriage Control Actions*
0	Overprinting
1	Single spacing
2	Double spacing
3	Triple spacing
4	4-line spacing
5	5-line spacing
6	6-line spacing
.	.
.	.
.	.
62	62-line spacing
PAGE	Skips to top of new page

15. When an attempt is made to write beyond the externally defined boundaries of a sequential file, an exception condition exists and the contents of the record area are unaffected. The FILE STATUS data item, if you specify one, is set to indicate the error condition, and execution continues as specified in the section, I-O Status Codes and Error Recovery, in Chapter 4.
16. The END-WRITE clause delimits the scope of the WRITE statement. For more information, see the section, Scope Terminators, in Chapter 8.

WRITE Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a WRITE statement: 00, 30, 34, 44, 48, 97, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

Example

See the examples at the end of Chapters 5, 6, 7, and 8.

10 Indexed Files

COBOL85 allows random or sequential access of records in an indexed disk file. Each record in an indexed file is uniquely identified by the value of one or more keys within that record.

COBOL85 processes indexed files created with the MIDASPLUS or PRISAM interface. Each COBOL85 record **key** corresponds to a MIDASPLUS or PRISAM **index**. See the *MIDASPLUS User's Guide* for information on preparing files for COBOL85 access with MIDASPLUS. Throughout this chapter references are made to MIDASPLUS utilities. If you use PRISAM, see the *PRISAM User's Guide* for information about creating indexed files.

This chapter discusses indexed file concepts, common operations on indexed files, and elements of the ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION as they pertain to indexed file processing. The chapter concludes with an example.

Indexed File Concepts

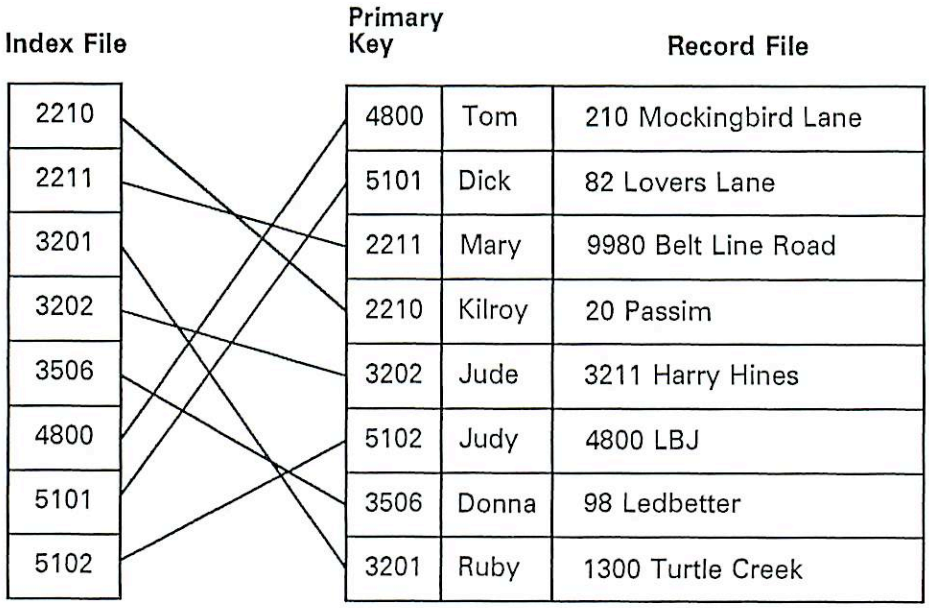
This section discusses the following indexed file concepts:

- Organization
- Primary and secondary keys
- Access modes
- File formats
- Current record pointer
- File status
- INVALID KEY condition
- NOT INVALID KEY condition
- AT END condition
- NOT AT END condition
- Exception conditions

Organization

An indexed file is a disk file in which data records can be accessed by the value of a key. A record description must include one or more data items used as keys, each of which is associated with a MIDASPLUS or PRISAM index.

MIDASPLUS or PRISAM creates the records of an indexed file in any order on a disk, and also constructs one or more files of indexes. The value of the key field in the program, as related to the file's indexes, controls all access to the records in the indexed file. Figure 10-1 represents an indexed file and a file of indexes.



Q10166-1LA-21-0

FIGURE 10-1
Indexed Record File and File of Indexes

Primary and Secondary Keys

For inserting, updating, and deleting records in a file, the value of a record key identifies each record. The data item named in the RECORD KEY clause of a *file-control-entry* for a file is the primary record key for that file. The ALTERNATE RECORD KEY clause designates secondary keys.

A secondary or alternate record key corresponds to a MIDASPLUS or PRISAM secondary index.

Access Modes

COBOL85 supports three access modes for indexed files. Specify them in the SELECT clause as follows:

- In sequential access mode, COBOL85 accesses records in ascending order of record key values. In the case of duplicate secondary key values, COBOL85 retrieves the records in the order in which they occur in the file.
- In random access mode, the program controls the sequence in which it accesses records. To access a particular record, the program places the value of the record's key in the corresponding field of the *record-description-entry*.
- In dynamic access mode, you can change at will from sequential access to random access for reading the file. The section Common Operations on Indexed Files, later in this chapter, discusses access mode requirements for each I-O statement.

File Formats

You can use variable-length record formatting or fixed-length record formatting for MIDASPLUS and PRISAM indexed files. The template for the file must define the file's format consistent with the COBOL85 program definition. For additional information, see the *MIDASPLUS User's Guide* and the *PRISAM User's Guide*.

Note

Odd-length records contained in MIDASPLUS indexed files contain an extra byte of data to fill the record out to the word boundary. This extra byte of data is undefined. For PRISAM indexed files, odd-length records are padded in such a way that this extra byte is not visible to the user.

Current Record Pointer

The current record pointer is a conceptual entity used to indicate the next record to be accessed within a given file. Only the OPEN, START, DELETE, and READ statements affect the setting of the current record pointer.

File Status

Code a file status check in the program to determine the success or failure of an I-O operation. Then use the result of the file status check to control the next program action. If you specify the FILE STATUS clause in a *file-control-entry*, COBOL85 places a value into the specified data item during the execution of a READ, WRITE, REWRITE, DELETE, OPEN, CLOSE, or START statement to indicate the status of that input-output operation. The section ENVIRONMENT DIVISION, later in this chapter, discusses the FILE STATUS clause. For a complete discussion of COBOL85 file status codes, see Chapter 4.

The INVALID KEY Condition

The INVALID KEY condition can occur as a result of the execution of a START, READ, WRITE, REWRITE, or DELETE statement. For details of the causes of the condition, see the relevant statement.

When the INVALID KEY condition occurs, COBOL85 performs the following steps:

1. A value is placed into the FILE STATUS data item, if you specify one for the file, to indicate an INVALID KEY condition.

2. If you specify the INVALID KEY phrase in the statement causing the condition, control is transferred to the INVALID KEY imperative statement. Any USE procedure that you specify for the file is not executed. After the execution of the INVALID KEY imperative statement, control is transferred to the end of the I-O statement and the NOT INVALID KEY phrase, if you specify one, is ignored.
3. If you do not specify the INVALID KEY phrase for the file, but you specify a USE procedure, either explicitly or implicitly, that procedure is executed.

When the INVALID KEY condition occurs, execution of the input-output statement that caused the condition is unsuccessful and the file is not affected.

The NOT INVALID KEY Condition

If the I-O operation is successful, COBOL85 performs the following steps:

1. The FILE STATUS data item, if you specify one, is updated to indicate a successful completion.
2. If you specify the NOT INVALID KEY phrase, control is transferred to the imperative statement associated with the NOT INVALID KEY phrase. After the execution of this imperative statement, control is transferred to the end of the I-O statement and the INVALID KEY phrase, if you specify one, is ignored.
3. If you do not specify the NOT INVALID KEY phrase, control is transferred to the end of the I-O statement and the INVALID KEY phrase, if you specify one, is ignored.

The AT END Condition

The AT END condition can occur as a result of a sequential READ statement when no next logical record exists in the file. When the AT END condition occurs, COBOL85 performs the following steps:

1. A value is placed into the FILE STATUS data item, if you specify one for the file, to indicate an AT END condition.
2. If you specify the AT END phrase in the statement causing the condition, control is transferred to the AT END imperative statement. Any USE procedure that you specify for the file is not executed. After the execution of the AT END imperative statement, control is transferred to the end of the I-O statement and the NOT AT END phrase, if you specify one, is ignored.
3. If you do not specify the AT END phrase for the file, but you specify a USE procedure, either explicitly or implicitly, that procedure is executed.

The NOT AT END Condition

The NOT AT END condition can occur as a result of a successfully completed sequential READ statement. When the NOT AT END condition occurs, COBOL85 performs the following steps:

1. The FILE STATUS data item, if you specify one, is updated to indicate a successful completion.

2. If you specify the NOT AT END phrase, control is transferred to the imperative statement associated with the NOT AT END phrase. After the execution of the NOT AT END imperative statement, control is transferred to the end of the I-O statement and the AT END phrase, if you specify one, is ignored.
3. If you do not specify the NOT AT END phrase, control is transferred to the end of the I-O statement and the AT END phrase, if you specify one, is ignored.

Exception Conditions

Exception conditions can occur as a result of a Permanent Error condition or a Logic Error condition. (See Chapter 4.) When an exception condition that is not an INVALID KEY or AT END condition occurs, COBOL85 performs the following steps:

1. A value is placed into the FILE STATUS data item, if you specify one for the file, to indicate the exception condition.
2. If you specify a USE procedure for the file, that procedure is executed. Any INVALID KEY, NOT INVALID KEY, AT END, NOT AT END phrases are ignored.
3. Program execution terminates.

Common Operations on Indexed Files

This section discusses the following common operations on indexed files:

- Creating a file
- Positioning the file to a certain record
- Reading a certain record
- Establishing a key
- Deleting a certain record
- Updating (changing) a certain record
- Creating (adding) records
- Handling errors

The concept of record key is essential for most of the following operations on indexed files, because only the keys or indexes allow access to the data records themselves.

Creating a File

To create a MIDASPLUS file, use the MIDASPLUS CREATK command to create an index template. Then build a file on that template either with the MIDASPLUS KBUILD command and an existing data file, or with a COBOL85 program. In some cases, the COBOL85 program creates an indexed file. See the OPEN statement for details.

The COBOL85 program must describe the new file's organization as INDEXED, and open it in one of the three access modes discussed earlier.

For more information on using CREATK and KBUILD, see the *MIDASPLUS User's Guide.*

For information on creating PRISAM files, see the *PRISAM User's Guide.*

Positioning the File to a Certain Record

In sequential or dynamic access mode, use START to position the file if you do not want to access the first record. If you want to sequentially read groups of records within the file, use additional START statements.

In random access mode, all READs use the primary key unless you use the KEY IS clause. Do not use START in random access mode.

Reading a Certain Record

If you open a file in sequential access mode, no special operations other than READ are necessary to read from the start of the file in sequential order of the primary key. If you do not want to read the first record, or if you want to read the file in secondary key order, precede the first READ by a START to specify the key of the first record you want to read. (START is required to establish a secondary key of reference.) All subsequent READs access each next record in key order until the program executes another START or closes the file, or an error occurs.

In random access mode all READs use the primary key unless you include the KEY IS clause.

If you want to access the file both sequentially and randomly, specify dynamic access mode. To change from random to sequential reading, use a series of Format 1 READs (READ NEXT).

Establishing a Key

The default key for any I-O verb is the primary key. To establish a secondary key, use START KEY IS ... *data-name* or READ ... KEY IS *data-name*.

Deleting a Certain Record

The file must be open in I-O mode. In sequential access mode, first read the record to be deleted; then use the DELETE verb. In random or dynamic access mode, place the proper value in the key field before using DELETE.

Updating (Changing) a Certain Record

The file must be open in I-O mode. In any access mode, first read the record to ensure that another program running concurrently cannot update it; then use the REWRITE verb. See the REWRITE statement, later in this chapter, for more information on reading the record in advance of rewriting it.

Creating (Adding) Records

Add new records with the WRITE statement. In sequential access mode, insert records in ascending order. Before writing each record, place a unique value in the primary key field.

Handling Errors in All of These Statements

The INVALID KEY clause and the USE statement in the declaratives section provide error handling. One of these elements is required for each I-O statement. Use the AT END phrase only for a sequential READ. Use USE statements in the declaratives section to provide error handling for non-end-of-file errors for a sequential READ, and for non-invalid-key errors.

ENVIRONMENT DIVISION

This section contains information that is unique to indexed files. Chapter 6 contains information that applies to all file organizations. Chapters 11 and 12 contain information that applies to relative files and sequential tape files, respectively.

INPUT-OUTPUT SECTION — FILE-CONTROL

Use the INPUT-OUTPUT SECTION to specify peripheral devices and information needed to transmit and handle data between the devices and the program.

Use the FILE-CONTROL paragraph to name each file and to specify other file-related information. Each file requires one *file-control-entry*.

Format

```
SELECT [OPTIONAL] file-name-1
  ASSIGN TO { device-name }
            { literal-1   }
[ RESERVE integer-1 [ AREA  ] ]
                    [ AREAS ]
[ORGANIZATION IS] INDEXED
[ ACCESS MODE IS { SEQUENTIAL } ]
                 { RANDOM     }
                 { DYNAMIC    }
RECORD KEY IS data-name-1
[ ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLICATES]] . . .
[ FILE STATUS IS data-name-3].
```

General Rules

1. The SELECT clause specifies the name of the indexed file. You must specify the SELECT clause first in the *file-control-entry*. The remaining clauses can appear in any order. COBOL85 manages all indexed files with either the MIDASPLUS interface or the PRISAM interface. Use ASSIGN TO MIDASPLUS to access MIDASPLUS files or ASSIGN TO PRISAM to access PRISAM files. You can also specify ASSIGN TO PFMS for compatibility with previous Prime COBOL compilers, but access is not as efficient.
2. The ORGANIZATION IS INDEXED clause specifies that the file named in the SELECT clause contains data organized by indexes. You establish file organization at the time you create the file, and you cannot subsequently change it.
3. The ACCESS MODE clause specifies the manner in which the program is to read or write to an indexed file. If you omit this clause, the default is sequential access. The three access modes are described below:

   - When you specify SEQUENTIAL, COBOL85 writes and retrieves records in the order of ascending values for a given key field.
   - When you specify RANDOM, COBOL85 writes and retrieves records randomly only, based on the value placed in the RECORD KEY field prior to a READ or WRITE. You must place the complete RECORD KEY value in *data-name-1* prior to every access operation; otherwise, the record is not found. Random mode precludes a sequential READ NEXT.
   - When you specify DYNAMIC, a program can read randomly or sequentially. Other operations follow the rules for random access, except that you can use START in dynamic access.

4. The RECORD KEY clause specifies the data item within each record that is used as the primary index.

   - *data-name-1* must be within the *record-description-entry*.
   - *data-name-1* must not be of variable size.
   - **Prime Extension:** *data-name-1* can be either alphanumeric or numeric.
   - Multiple *record-description-entries* must have the same data description in the same relative position for the record key.
   - Do not specify *data-name-1* with an OCCURS clause, or include it within a group subordinate to an OCCURS clause. This means it must not be subscripted or indexed, but it can be qualified. If *data-name-1* occurs more than once in the program, then it must be fully qualified.
   - **Prime Restriction:** Do not specify *data-name-1* with a P character or a separator sign in its PICTURE clause. It cannot exceed 64 characters.
   - *data-name-1* must have the same description as the primary index in the file template created with CREATK or DDL.
   - The value contained within *data-name-1* must be unique for each record in a file.

5. The ALTERNATE RECORD KEY clause specifies a data item within each record that is used as a secondary index. You can specify a maximum of 17 alternate record keys. The number of alternate record keys cannot be greater than the number of secondary indexes specified in the file template created with CREATK.

Alternate record keys must be part of the *record-description-entry*, but they must neither be embedded within nor overlap the primary record key. Alternate keys must not start in the same position as any other key. They follow the rules for RECORD KEY above, except that an alternate key can have duplicates.

The WITH DUPLICATES clause specifies that records in the file can contain secondary keys having the same value. Specify WITH DUPLICATES only if duplicates are allowed for the corresponding secondary index in the MIDASPLUS template. If you do not specify WITH DUPLICATES, the secondary key value in each record must be unique.

Define secondary keys in the same order as the corresponding MIDASPLUS indexes. The first secondary key that you specify is equivalent to MIDASPLUS index number 1, the second key is equivalent to MIDASPLUS index number 2, and so forth.

Alternate record keys can be nonnumeric.

6. In the FILE STATUS clause, *data-name-3* must be a two-character field described in the DATA DIVISION. The file control system moves a value into *data-name-3* following the execution of every statement that explicitly or implicitly references the file. This value indicates the execution status of the statement. Following a successful I-O operation, *data-name-3* contains '00'. For a complete discussion of COBOL85 file status codes, see Chapter 4.

   Prime Extension: *data-name-3* can be either alphanumeric or numeric (PIC XX or PIC 99).

7. The FILE STATUS item (*data-name-3*) must not be part of the record description for the file. It must be in the WORKING-STORAGE or LINKAGE SECTION. It can be qualified but must not be subscripted or indexed.

   If the *file-description-entry* specifies EXTERNAL, the FILE STATUS item (*data-name-3*) is also EXTERNAL; therefore, do not define it in the LINKAGE SECTION.

DATA DIVISION

The elements of the DATA DIVISION are the same for indexed files as those described in Chapter 7, except for the *record-description-entry*.

record-description-entry

Each record must have a unique value for the primary key. Rules for primary and secondary keys are given in the section ENVIRONMENT DIVISION, above.

COBOL85 record lengths and MIDASPLUS or PRISAM record lengths must be the same.

PROCEDURE DIVISION

This section contains information that pertains to indexed files. Chapter 8 contains information that applies to all file organizations. Chapters 9, 11, and 12 contain information that pertains to sequential disk files, relative files, and sequential tape files, respectively.

CLOSE

Terminates the processing of files.

Format

CLOSE *file-name-1* [,*file-name-2*] . . .

Syntax Rule

The files named in the CLOSE statement need not all have the same organization or access.

General Rules

1. Once a program executes a CLOSE statement for a file, it must execute an OPEN statement for that file before executing any other statements for that file.
2. A program can execute a CLOSE statement only for a file that is open. If a CLOSE statement is attempted on a file that is not open, the file status data item, if you specify one, is set to indicate status code 42. After execution of any applicable declarative procedure, the program terminates. For this particular error, the only applicable declarative procedure is a procedure for *file-name*.
3. If any other error occurs during a CLOSE operation, the file status data item, if you specify one, is updated to indicate status code 30. Execution continues according to the rules specified in Chapter 4.

DELETE

Logically removes a record from a file.

Format

DELETE *file-name* RECORD

[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-DELETE]

Syntax Rules

1. Do not specify the INVALID KEY and the NOT INVALID KEY phrases for a DELETE statement that references a file that is in sequential access mode.
2. You must specify the INVALID KEY phrase for a DELETE statement on a file that is not in sequential access mode and for which you specify no USE procedure.

General Rules

1. The DELETE statement logically removes a data record from the indexed file together with all the associated index entries. The file must be open for I-O. Execution of DELETE updates the value of the FILE STATUS data item, if you specify one. It does not affect the contents of the record area associated with *file-name*.
2. Transfer of control following the successful or unsuccessful execution of the DELETE operation depends on the presence or absence of the optional INVALID KEY and NOT INVALID KEY phrases, and on the presence or absence of USE procedures associated with the file. See the section Indexed File Concepts, earlier in this chapter, for more information.
3. The END-DELETE clause delimits the scope of the DELETE statement. For more information, see the section Scope Terminators, in Chapter 8.

Rules for Sequential Access

1. In sequential access, the program must successfully read a record before deleting it. Otherwise, an exception condition occurs, and the DELETE statement is unsuccessful. A status code of 43 is placed in the FILE STATUS field, if you specify one. Exception conditions and file status are discussed in the section Indexed File Concepts, earlier in this chapter.
2. You must not change the primary record key between the READ and DELETE statements.

Rules for Random and Dynamic Access

1. Random and dynamic access modes require that you place the primary key of the record to be deleted in the RECORD KEY field.
2. If that record does not exist in the file, the INVALID KEY statement is executed, and a status code of 23 is placed in the FILE STATUS field, if one exists. INVALID KEY and FILE STATUS are discussed in the section Indexed File Concepts, earlier in this chapter.

DELETE Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a DELETE statement: 00, 23, 43, 49, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

OPEN

Initiates the processing of files, and performs other input-output operations.

Format

```
OPEN { INPUT  file-name-1 [, file-name-2] ... }
     { OUTPUT file-name-3 [, file-name-4] ... } ...
     { I-O    file-name-5 [, file-name-6] ... }
```

General Rules

1. A file opened as INPUT can be accessed only in a READ or START statement.
2. A file opened as OUTPUT can be accessed only in a WRITE statement.
3. A file opened as I-O can be accessed by a READ, WRITE, REWRITE, START, or DELETE statement.

   Note

   You cannot use all I-O statements in all access modes. Table B-8 in Appendix B specifies the types of I-O statements permissible with the different access modes.

4. Following the initial execution of an OPEN statement for a file, the program must close the file before it can open the file again.
5. Execution of the OPEN statement does not obtain or release the first data record. However, for files opened in INPUT or I-O mode, the OPEN statement positions the file to the first record.
6. The *file-description-entry* for an opened file must be equivalent to the file's template created by the FAU or CREATK utility. For MIDASPLUS variable-length files, supply the minimum and maximum sizes during CREATK. See the *MIDASPLUS User's Guide* for more information.
7. For PRISAM files opened in OUTPUT mode or an OPTIONAL file opened in I-O mode, OPEN does not create an indexed file. It merely opens an existing file for writing. You must create the file template with the PRISAM FAU utility. If the file is not present, the OPEN does not occur. A status code of 37 is returned, declaratives are invoked, if present, and the program terminates.
8. For MIDASPLUS and PFMS files opened in OUTPUT mode or an OPTIONAL file opened in I-O mode, if the file is not present, OPEN creates the file. COBOL85 builds a template using runtime calls to CREATK and using information in the program to describe the file to MIDASPLUS.
9. For MIDASPLUS and PRISAM files opened in OUTPUT mode, if the file is present and contains data, the OPEN statement is unsuccessful, and a status code of 37 is returned. An exception condition occurs, and the FILE STATUS field is updated, if one exists. Exception conditions and file status are discussed in the section Indexed File Concepts, earlier in this chapter.
10. If any system error occurs during the OPEN processing, a status code of 30 is returned. An exception condition occurs, and the FILE STATUS field is updated, if one exists. Exception conditions and file status are discussed in the section Indexed File Concepts, earlier in this chapter.
11. During the execution of the OPEN statement, COBOL85 checks file attributes against the attributes described for the file in the program. These attributes are organization, primary key, secondary keys, minimum and maximum logical record sizes, and the file type (fixed or variable, MIDASPLUS or PRISAM).

If any of these attributes conflict, the OPEN statement is unsuccessful, and a status code of 39 is returned. An exception condition occurs, and the FILE STATUS field is updated, if one exists. Exception conditions and file status are discussed in the section Indexed File Concepts, earlier in this chapter. If the SELECT statement associated with the file specifies MIDASPLUS as the assigned device, MIDASPLUS is used to open the file.

12. If the SELECT statement associated with the file specifies PRISAM as the assigned device, PRISAM is used to open the file. If the SELECT statement associated with the file specifies PFMS as the assigned device, an internal check is made to see if the file is a MIDASPLUS or a PRISAM file. All subsequent I-O operations are performed using the appropriate data management interface.
13. If an unavailable optional file is opened with the INPUT phrase, the OPEN statement sets the current record pointer to indicate the AT END condition. The OPEN statement is successful. The FILE STATUS data item is set to the informational status code 05.

OPEN Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of an OPEN statement: 00, 05, 07, 30, 35, 37, 39, 41, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

READ

For sequential access, READ makes available the next logical record from a file; for random access, READ makes available a record with a specific key.

Format 1 (Sequential or Dynamic)

READ *file-name* [NEXT] RECORD [INTO *data-name-1*]

[AT END *imperative-statement-1*]

[NOT AT END *imperative-statement-2*]

[END-READ]

Format 2 (Random or Dynamic)

READ *file-name* RECORD [INTO *data-name-1*]

[KEY IS *data-name-2*]

[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-READ]

Syntax Rules

1. The *file-name* must be the name of an indexed file with sequential or dynamic access.
2. The key name (*data-name-2*) must be the name of a data item that you specify as a record key associated with *file-name*.
3. The key name can be qualified. It must not be subscripted or indexed.
4. You must specify the INVALID KEY phrase (Format 2) or the AT END phrase (Format 1) if you do not specify an applicable USE procedure for *file-name*.

General Rules

1. You can use the INTO phrase when the input file contains logical records of various sizes.

   Prime Extension: The storage area associated with *data-name-1* and the record area associated with *file-name* can be the same storage area.
2. The associated file must be open in the INPUT or I-O mode at the time this statement is executed.
3. The execution of the READ statement updates the value of any FILE STATUS data item associated with *file-name*. File status is discussed in the section Indexed File Concepts, earlier in this chapter.
4. If you specify the INTO phrase, the record being read is moved from the record area to *data-name-1* according to the rules specified for the MOVE statement without the CORRESPONDING phrase. The implied MOVE does not occur if the execution of the READ statement is unsuccessful. Any subscripting or indexing associated with *data-name-1* is evaluated after the record is read and immediately before it is moved to the data item.

   When you use the INTO phrase, the record being read is available in both the input record area and *data-name-1*.
5. For variable-length records, the following rules apply:

   - If the number of character positions in the record that is read is less than the minimum size specified in the *record-description-entry*, the portion of the record area to the right of the last valid character read is undefined.
   - If the number of character positions in the record that is read is greater than the maximum size specified in the *record-description-entry*, the record is truncated on the right to the maximum size.

   In either of these cases, the READ statement is successful, and the FILE STATUS data item, if you specify one, is set to 04 to indicate a record length conflict.
6. When the key value for the current key of reference is equal to the value of the same key in the next logical record within the current key of reference, the FILE STATUS item, if you specify one, is set to an informational status code, 02, to indicate that a duplicate record exists.
7. Following the unsuccessful execution of any READ statement, the contents of the associated record area and the position of the current record pointer are undefined. For indexed files the key of reference is also undefined.

8. Transfer of control following the successful or unsuccessful execution of the READ operation depends on the presence or absence of the optional INVALID KEY, NOT INVALID KEY, AT END, and NOT AT END phrases, and on the presence or absence of USE procedures associated with the READ statement. See the section Indexed File Concepts, at the beginning of this chapter, for more information.
9. The END-READ clause delimits the scope of the READ statement. For more information, see the section Scope Terminators, in Chapter 8.

Rules for Format 1 (Sequential and Dynamic Access)

1. Use Format 1 for all files in sequential access mode. The NEXT phrase is optional and has no effect for sequential access.
2. The NEXT phrase is required for files in dynamic access mode, when you want to retrieve records sequentially.
3. The record made available by a Format 1 READ statement is determined as follows:

   - If the current record pointer was positioned by the START or OPEN statement, the record to which it points is made available, provided that it is still accessible. If the record is no longer accessible, perhaps because the record is deleted or an alternate record key is changed, the current record pointer is updated to point to the next existing record within the established key of reference. That record is then made available.
   - If the current record pointer was positioned by the execution of a previous READ statement, the current record pointer is updated to point to the next existing record in the file with the established key of reference and that record is made available.
   - For an indexed file in the dynamic access mode, the execution of an OPEN I-O statement followed by one or more WRITE statements and then a READ NEXT statement causes the READ NEXT statement to access the first record in the file. However, if the WRITE statement inserted records with a key value lower than that of any records previously existing in the file, these records are returned by the first READ NEXT statement.
   - If an alternate key is the key of reference, and the alternate key is changed by a REWRITE statement to a value between the current value and the next value in the file, a subsequent READ NEXT statement obtains the record just rewritten.
   - If the current record pointer was established by a previous READ statement, and the current key of reference does not allow duplicates, the first existing record in the file whose key value is greater than the current record pointer is selected. In other words, without the WITH DUPLICATES phrase, a sequential READ statement reads only the first duplicate and skips to the next nonduplicated key.

4. If, at the time of execution of a Format 1 READ statement, the position of the current record pointer for that file is undefined, the execution of that READ statement is unsuccessful. The FILE STATUS data item, if you specify one, is set to 46, and an exception condition occurs.
5. If, at the time of the execution of a Format 1 READ statement, no next logical record exists in the file or an optional input file is unavailable, the AT END condition occurs, and the execution of the READ statement is unsuccessful.

6. When the AT END condition occurs,

   a. A value of 10 is placed into the FILE STATUS data item, if you specify one for the file, to indicate an AT END condition.

   b. If you specify the AT END phrase, control is transferred to the associated *imperative-statement*. Any USE procedure you specify for this file is not executed.

   c. If you do not specify the AT END phrase, then the USE procedure you specify for this file is executed. Otherwise, execution is aborted.

7. When the AT END condition occurs, the program must not execute a Format 1 READ statement for that file without first executing one of the following:

   - A successful CLOSE statement followed by the execution of a successful OPEN statement for that file
   - A successful START statement for that file
   - A successful Format 2 READ statement for that file

8. If an AT END condition does not occur during the execution of a READ statement and the READ statement is successful, the AT END phrase, if you specify one, is ignored, and the following actions occur:

   a. The current record pointer is set and the FILE STATUS data item, if you specify one for this file, is updated.

   b. The record is made available in the record area and any implicit move resulting from the presence of an INTO phrase is executed. Control is transferred to the end of the READ statement or to the *imperative-statement* specified by the NOT AT END phrase, if you specify one for the file.

9. If an exception condition that is not an AT END conditon exists, the FILE STATUS item is updated and control is transferred according to the rules for the USE statement.
10. In dynamic access mode, a Format 1 READ NEXT statement retrieves the next logical record from the file as described in General Rule 3 for Format 1.
11. If an alternate record key is the key of reference and duplicates are allowed, records having the same value in that key are read in the same order in which they were written.
12. For a Format 1 READ statement, include a USE procedure in the program to handle any non-end-of-file errors that may occur. This action may be necessary to handle errors that may occur when multiple users are accessing a MIDASPLUS or PRISAM file that has been opened in I-O mode. This USE procedure also handles other unexpected conditions that may be generated during sequential file access.

Rules for Format 2 (Random and Dynamic Access)

1. Use Format 2 for files in random access mode or for files in dynamic access mode when you want to retrieve records randomly.
2. If an optional input file is unavailable, the INVALID KEY condition exists and the READ statement is unsuccessful.

3. For an indexed file, if you specify the KEY phrase in a Format 2 READ statement, *data-name-2* is established as the key of reference for this retrieval.
4. If you do not specify the KEY phrase in a Format 2 READ statement, the primary record key is established as the key of reference for this retrieval. If you specify dynamic access mode, this key of reference is also used for retrievals by any subsequent Format 1 READ statements for the file until the program establishes a different key of reference for the file.
5. Execution of a Format 2 READ statement compares the value of the key of reference with the value contained in the corresponding index until the record having an equal value is found or, for a secondary key, the first record having the value is found. The current record pointer is positioned to this record, which is then made available. If no record can be so identified, the INVALID KEY condition exists and execution of the READ statement is unsuccessful.

READ Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a READ statement: 00, 02, 04, 10, 23, 30, 46, 47, 90, 97, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

REWRITE

Logically replaces a record on a disk file.

Format

REWRITE *record-name* [**FROM** *data-name*]

[**INVALID KEY** *imperative-statement-1*]

[**NOT INVALID KEY** *imperative-statement-2*]

[**END-REWRITE**]

Syntax Rules

1. The *record-name* and the *data-name* can refer to the same storage area.
2. The *record-name* is the name of a logical record in the FILE SECTION of the DATA DIVISION and can be qualified.
3. In random or dynamic access mode, you must specify the INVALID KEY phrase in the REWRITE statement for files for which you do not specify an appropriate USE procedure.
4. Do not specify the INVALID KEY and the NOT INVALID KEY phrases for a REWRITE statement that references a file that is in sequential access mode.

General Rules

1. In sequential access mode, the program must successfully read a record prior to the REWRITE statement to ensure that the record is locked and cannot be updated by another program running concurrently.
2. In dynamic or random access mode, a READ statement is not required prior to a REWRITE, unless the current key of reference is a secondary key. When a READ statement is not required and has not been executed, the primary key specifies the record to be replaced.
3. If a required READ statement is not executed prior to the REWRITE, the REWRITE statement is unsuccessful and status code 43 is returned. An exception condition exists and the FILE STATUS field is updated, if one exists.
4. The REWRITE statement can change any or all data fields in the record except the primary record key. If a prior READ is required, and if the program changes the primary key last read, the REWRITE statement is unsuccessful, and a status code 21 is returned. The INVALID KEY condition occurs and the FILE STATUS field is updated, if one exists. INVALID KEY conditions and file status are discussed in the section Indexed File Concepts, earlier in this chapter.
5. The file must be open for I-O for all access methods.
6. The FROM option allows you to create the record in another area. It is equivalent to

   MOVE *data-name* TO *record-name*

   prior to the execution of the REWRITE statement. The primary key value must equal the key from the previous READ, or the INVALID KEY condition occurs.
7. The number of character positions in the record referenced by *record-name* must be equal to the number of character positions in the record being replaced.
8. **Prime Extension:** The logical record released by a successful execution of the REWRITE statement is still available in the record area.

   If you name the associated file in a SAME RECORD AREA clause, the logical record is also available to the program as a record of other files appearing in the same clause.
9. The execution of the REWRITE statement updates the value of any FILE STATUS data item associated with the file.
10. During a REWRITE operation, the contents of the alternate record key data item of the record being rewritten can differ from the value being replaced. If the new value is a duplicate of another record in the file and if you specify WITH DUPLICATES, an informational status code 02 is placed in the FILE STATUS data item, if one exists. The record is logically positioned last within the set of duplicates. If you do not specify WITH DUPLICATES, the INVALID KEY condition exists, and a status code 22 is placed in the FILE STATUS data item, if one exists.

Note

For PRISAM files, if you specify the WITH DUPLICATES phrase in the program, but the DDL specification does not allow duplicates, COBOL85 returns status code 22. However, during sequential access of such a file with a secondary key of reference, the current file position is undefined.

11. Transfer of control following the successful or unsuccessful execution of the REWRITE operation depends on the presence or absence of the optional INVALID KEY and NOT INVALID KEY phrases, and on the presence or absence of USE procedures associated with the REWRITE statement. See the section Indexed File Concepts, earlier in this chapter, for more information.
12. The END-REWRITE clause delimits the scope of the REWRITE statement. For more information, see the section Scope Terminators, in Chapter 8.
13. A REWRITE statement does not affect the current file position.

REWRITE Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a REWRITE statement: 00, 02, 21, 22, 30, 43, 44, 49, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

SEEK — Prime Extension

Is supported syntactically only for compatibility with other COBOL implementations.

Format

SEEK *file-name* RECORD

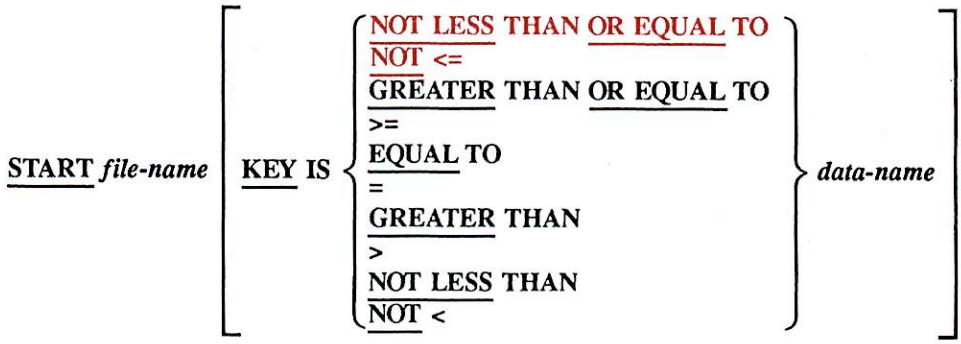
General Rules

1. SEEK is treated as documentation in COBOL85.
2. You must define the *file-name* by a *file-description-entry* in the DATA DIVISION.

START

Establishes a position in the file for subsequent READs.

Format



[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-START]

Syntax Rules

1. The *file-name* must be the name of an indexed file with sequential or dynamic access.
2. The *data-name* can be qualified but not indexed or subscripted.
3. You must specify the INVALID KEY phrase if you do not specify an applicable USE procedure for *file-name*.
4. The *data-name* must reference one of the following:

   - A data item that is a record key
   - A data item that is an alternate record key
   - An alphanumeric data item that is subordinate to the *data-name* specified as the record key or alternate record key of *file-name*, and whose leftmost character position corresponds to the leftmost character position of the record key or alternate record key (a partial key)

General Rules

1. *file-name* must be open in the INPUT or I-O mode at the time that the START statement is executed.
2. If you do not specify the KEY phrase, the relational operator IS EQUAL TO and the primary record key are the defaults.
3. The current record pointer is positioned to the first logical record in the file whose key satisfies the comparison.

If no record in the file satisfies the comparison, or an optional input file is unavailable, an INVALID KEY condition exists, the execution of the START statement is unsuccessful, and the position of the current record pointer is undefined.

4. The execution of the START statement updates the value of any FILE STATUS data item associated with *file-name*. File status is explained earlier in this chapter.
5. If you specify the KEY phrase, the comparison uses the data item referenced by *data-name*.
6. Successful execution of the START statement establishes a key of reference, which is used in subsequent Format 1 READ statements, as follows:

   - If you do not specify the KEY phrase, the primary record key specified for *file-name* becomes the key of reference.
   - If you specify the KEY phrase, and you specify *data-name* as a record key for *file-name*, that record key becomes the key of reference.

7. START does not retrieve a record, but only positions the file to a specific record.
8. If execution of START is unsuccessful, the key of reference is undefined.
9. Transfer of control following the successful or unsuccessful execution of the START operation depends on the presence or absence of the optional INVALID KEY and NOT INVALID KEY phrases, and on the presence or absence of USE procedures associated with the START statement. See the section Indexed File Concepts, earlier in this chapter, for more information.
10. The END-START clause delimits the scope of the START statement. For more information, see the section Scope Terminators, in Chapter 8.

START Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a START statement: 00, 23, 47, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

Example

In the following indexed file, each record contains a NAME field that serves as the primary key and a COMPANY field:

data-name	*NAME*	*COMPANY*
Picture	PIC X(10)	PIC X(25)
Values	BLYE	REPORTCO
	CLAPP	MERGANTHALER
	FIELDS	SERVICE
	GRIER	AUTOMATION
	HARPER	DESIGNERS
	KEANE	REPORTCO

The following example shows source coding to describe the file:

```
ENVIRONMENT DIVISION.
     .
     .
     .
      SELECT FILE-1 ASSIGN TO MIDASPLUS
           ORGANIZATION IS INDEXED
           ACCESS IS DYNAMIC
           RECORD KEY IS NAME.
     .
     .
     .
DATA DIVISION.
FILE SECTION.
FD  FILE-1
            VALUE OF FILE-ID IS 'FILE-1'.
01  FILE-1-RECORD.
    05  NAME       PIC X(10).
    05  COMPANY    PIC X(25).
```

To display records of people whose names begin with the characters F, G, H, and I, program actions must include a START statement to position the file to the first name beginning with one of these letters, and a series of executions of sequential READ statements.

To position with the START statement, first initialize the key field (NAME), as in the next example.

```
MOVE 'F          ' to NAME.                    Initialize key field.

START FILE-1 KEY IS NOT LESS THAN NAME         Find the first record
    INVALID KEY DISPLAY 'EOF'.                 whose key is not less
    .                                          than 'F'.  This posi-
    .                                          tions  the  file  to
    .                                          this record (FIELDS).

READ FILE-1 NEXT RECORD,                       Retrieve  the  first
    AT END DISPLAY 'EOF'.                      record (FIELDS).
    .
    .
PERFORM 120-READ-NEXT UNTIL NAME NOT           This  action  retrieves
  LESS THAN 'K'.                               the  records from
    .                                          GRIER  through KEANE,
    .                                          and prints  all except
    .                                          KEANE.
    .

120-READ-NEXT.

WRITE PRINT-LINE FROM FILE-1-RECORD.
READ FILE-1 NEXT RECORD
AT END DISPLAY 'EOF'.
```

WRITE

Releases a logical record for an output or input-output file.

Format

WRITE *record-name* [**FROM** *data-name*]

[**INVALID KEY** *imperative-statement-1*]

[**NOT INVALID KEY** *imperative-statement-2*]

[**END-WRITE**]

Syntax Rules

1. *record-name* and *data-name-1* can name the same storage area.
2. *record-name* is the name of a logical record in the FILE SECTION of the DATA DIVISION and can be qualified.
3. You must specify the INVALID KEY phrase if you do not specify an applicable USE procedure for the associated file.

General Rules

1. The associated file must be open in the OUTPUT or I-O mode.
2. Prime Extension: After a successful WRITE, the information is still available in *record-name*.

   The logical record released by the WRITE statement is also available to the program as a record of other files referenced in the same SAME RECORD AREA clause as the associated output file.
3. Execution of the WRITE statement with the FROM phrase is equivalent to the statement

   MOVE *data-name-1* **TO** *record-name*

   followed by a WRITE statement.

   After execution of the WRITE statement is complete, the information in the area referenced by *data-name-1* is available.
4. If the number of character positions in the record is less than the minimum size or larger than the maximum size allowed for the file, the WRITE statement is unsuccessful, and an exception condition exists.
5. The execution of the WRITE statement updates the value of any FILE STATUS data item associated with the file. File status is explained in the section Indexed File Concepts, earlier in this chapter.

Rules for Record Keys

1. MIDASPLUS stores the written record in such a way that you can use any record key to access it.
2. The value of the primary record key must be unique within the records in the file. Otherwise, an INVALID KEY condition occurs, and the WRITE statement is unsuccessful. A status code of 22 is placed in the FILE STATUS field, if one exists.
3. The program must set the data item specified as the primary record key to the desired value prior to the execution of the WRITE statement.
4. In sequential access mode, the program must write records in ascending order. Otherwise, an INVALID KEY condition occurs, and the WRITE statement is unsuccessful. A status code of 21 is placed in the FILE STATUS field, if one exists.
5. The value of any alternate record key must be unique unless you specify the WITH DUPLICATES phrase. In this case, an informational status code 02 is placed in the FILE STATUS data item, if one exists. The record is logically positioned last within the set of duplicates. If you do not specify WITH DUPLICATES, the INVALID KEY condition exists, and a status code of 22 is placed in the FILE STATUS field, if one exists.
6. Transfer of control following the successful or unsuccessful execution of the WRITE operation depends on the presence or absence of the optional INVALID KEY and NOT INVALID KEY phrases, and on the presence or absence of USE procedures associated with the WRITE statement. See the section Indexed File Concepts, earlier in this chapter, for more information.
7. The END-WRITE clause delimits the scope of the WRITE statement. For more information, see the section Scope Terminators, in Chapter 8.

WRITE Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a WRITE statement: 00, 02, 21, 22, 30, 44, 48, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

Example

This sample program illustrates the use of the SAME RECORD AREA clause as well as indexed concepts. Because TRANS-FILE and MASTER-FILE share the same record area, no MOVE or WRITE FROM is necessary to add or delete a record from MASTER-FILE.

```
ID DIVISION.
PROGRAM-ID.     RANDOM1.
*****************************************************************
REMARKS. THIS PROGRAM ILLUSTRATES READ, WRITE, REWRITE, AND
    DELETE IN RANDOM ACCESS MODE, AND CREATION OF AN INDEXED
    FILE.
```

```
     IN ALL CASES, THE KEY IS IN MASTER-RECORD AS SOON AS
     TRANS-FILE IS READ BECAUSE OF THE SAME AREA CLAUSE.
     EXCEPT IN THE CASE OF AN UPDATE TRANSACTION,
     NO MOVE OF ACCT-ENTRY TO ACCT-MS IS NECESSARY.
*****************************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PRIME.
 OBJECT-COMPUTER. PRIME.
 INPUT-OUTPUT SECTION.
*
 FILE-CONTROL.
*
     SELECT MASTER-FILE ASSIGN TO MIDASPLUS
         ORGANIZATION IS INDEXED,
         ACCESS MODE IS RANDOM,
         RECORD KEY IS ACCT-MS
         FILE STATUS IS FS-MS.
*
     SELECT TRANS-FILE ASSIGN TO MIDASPLUS,
         ORGANIZATION IS SEQUENTIAL,
         FILE STATUS IS FS-TR.
*
     SELECT NEW-FILE ASSIGN TO MIDASPLUS,
         ORGANIZATION IS INDEXED,
         ACCESS IS RANDOM,
         RECORD KEY IS ACCT-NEW,
         FILE STATUS IS FS-NEW.
*
     SELECT PRINT-FILE ASSIGN TO PRINTER.
*
 I-O-CONTROL.
     SAME RECORD AREA FOR TRANS-FILE, MASTER-FILE.
*****************************************************************
 DATA DIVISION.
 FILE SECTION.
*
 FD  MASTER-FILE,
     VALUE OF FILE-ID IS KDISBURS,
     RECORD CONTAINS 42,
     DATA RECORD IS MASTER-RECORD.
 01  MASTER-RECORD.
     05 ACCT-MS                  PIC X(3).
     05 DATE-MS                  PIC 9(6).
     05 FILLER                   PIC X(3).
     05 VENDOR-MS                PIC X(20).
     05 CHECK-MS                 PIC X(3).
     05 AMT-MS                   PIC 9(7).
*
```

```
 FD  TRANS-FILE COMPRESSED,
     VALUE OF FILE-ID IS TRANSFL,
     RECORD CONTAINS 43,
     DATA RECORD IS TRANS-RECORD.
 01  TRANS-RECORD.
     05 TRANS-ENTRY.
         10 ACCT-ENTRY          PIC X(3).
         10 FILLER              PIC X(39).
     05  ENTRY-CODE             PIC X.
*
 FD  NEW-FILE COMPRESSED,
     VALUE OF FILE-ID IS NEWFILE,
     RECORD CONTAINS 42,
     DATA RECORD IS NEW-RECORD.
 01  NEW-RECORD.
     05 NEW-ENTRY.
        10 ACCT-NEW             PIC X(3).
        10 FILLER               PIC X(38).
     05 NEW-CODE                PIC X.
*
 FD  PRINT-FILE,
     LABEL RECORDS ARE OMITTED,
     RECORD CONTAINS 42,
     DATA RECORD IS PRINT-LINE.
 01  PRINT-LINE                 PIC X(42).
*
 WORKING-STORAGE SECTION.
 77  FS-NEW                     PIC XX VALUE '00'.
 77  FS-MS                      PIC 99 VALUE 00.
 77  FS-TR                      PIC XX VALUE '00'.
 77  KDISBURS                   PIC X(28)
                 VALUE 'KDISBURS'.
 77  NEWFILE                    PIC X(27)
                 VALUE 'NEWFILE'.
 77  NO-MORE-INPUT              PIC X VALUE 'N'.
 77  PRINT-COUNT                PIC 99 VALUE 00.
 77  TRANSFL                    PIC X(28)
                 VALUE 'TRANSFL'.
 01  TRANS-RECORD-HOLD.
     05 TRANS-ENTRY-HOLD.
         10 ACCT-ENTRY-HOLD     PIC X(3).
         10 FILLER              PIC X(39).
     05  ENTRY-CODE-HOLD        PIC X.
**************************************************
 PROCEDURE DIVISION.
 000-MAINLINE.
     OPEN INPUT TRANS-FILE,
          I-O MASTER-FILE,
          I-O NEW-FILE,
          OUTPUT PRINT-FILE.
```

```
      PERFORM 010-PRINT-HEADINGS.
      READ TRANS-FILE AT END
           DISPLAY 'INPUT FILE IS EMPTY',
           CLOSE TRANS-FILE, MASTER-FILE, NEW-FILE, PRINT-FILE,
           STOP RUN.
      PERFORM 020-PROCESS-TRANS UNTIL NO-MORE-INPUT = 'Y'.
      CLOSE TRANS-FILE,
          MASTER-FILE,
          NEW-FILE,
          PRINT-FILE.
      STOP RUN.
*
 010-PRINT-HEADINGS.
*NOT INCLUDED.
*
 020-PROCESS-TRANS.
      IF ENTRY-CODE = 'U' PERFORM 100-UPDATE
         ELSE IF ENTRY-CODE = 'A' PERFORM 110-ADD
              ELSE IF ENTRY-CODE = 'D' PERFORM 120-DELETE
                   ELSE PERFORM 200-CREATE-ERROR-FILE.
      READ TRANS-FILE AT END
         MOVE 'Y' TO NO-MORE-INPUT
         DISPLAY 'END OF FILE',
         IF PRINT-COUNT = 0 MOVE 'NO PRINT RECORDS' TO PRINT-LINE,
            WRITE PRINT-LINE AFTER ADVANCING 2
         END-IF.
*
 100-UPDATE.
      MOVE TRANS-RECORD TO TRANS-RECORD-HOLD.
      READ MASTER-FILE INVALID KEY
          MOVE 'N' TO ENTRY-CODE
          PERFORM 200-CREATE-ERROR-FILE.
      MOVE TRANS-RECORD-HOLD TO MASTER-RECORD.
      REWRITE MASTER-RECORD INVALID KEY
          DISPLAY 'INVALID REWRITE'.
*
 110-ADD.
      WRITE MASTER-RECORD INVALID KEY
          MOVE 'D' TO ENTRY-CODE,
          PERFORM 200-CREATE-ERROR-FILE.
*
 120-DELETE.
      DELETE MASTER-FILE RECORD, INVALID KEY
          MOVE 'N' TO ENTRY-CODE,
          PERFORM 200-CREATE-ERROR-FILE.
*
 200-CREATE-ERROR-FILE.
*         ERRONEOUS INPUT RECORDS ARE WRITTEN TO THE INDEXED
*         NEW-FILE UNLESS A KEY IS DUPLICATED WITHIN NEW FILE.
*         IN THAT CASE, THE ERROR RECORD IS PRINTED INSTEAD.
```

```
        MOVE ENTRY-CODE TO NEW-CODE.
        MOVE TRANS-ENTRY TO NEW-ENTRY.
        WRITE NEW-RECORD, INVALID KEY
            MOVE NEW-RECORD TO PRINT-LINE,
            WRITE PRINT-LINE AFTER ADVANCING 1,
            ADD 1 TO PRINT-COUNT.
```

To compile, link, and execute this file, stored as RANDOM.COBOL85, use the following dialog:

```
OK, COBOL85 RANDOM -LISTING
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]
[0 ERRORS IN PROGRAM: RANDOM.COBOL85]

OK, BIND
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
: LOAD RANDOM
: LI COBOL85LIB
: LI
BIND COMPLETE
: FILE
OK, RESUME RANDOM
END OF FILE
OK,
```

Input Files

The master file, KDISBURS, contains the following records before the program is run:

```
408080185   ASHTABULA HDWE       4300035476
409080185   CAIRO CHEMICAL       4360002746
410080285   ST.BOTOLPHSTOWN SUPP4200005108
411080285   DOVER MUTUAL         4100034166
412080385   PARIS AUTO           4100015000
413090385   ROME BOATING         4150017982
C82080785   ODESSA SERVICES      4100004670
4500B0785   ANTIOCH SERVALL      4300002580
580080785   BETHLEHEM TAXI       RR00009840
681080785   ATHENS LUMBER        18500036BB
```

The transaction file, TRANSFL, contains the following records:

```
408111386   ASHTABULA HARDWARE   4300099999U
414061185   PRIME COMPUTER       4360000123A
410080285   ST.BOTOLPHSTOWN SUPP4200005108D
411080285   DOVER MUTUAL         4100034166
```

Output Files

The master file, KDISBURS, contains the following records after the program is run:

```
408111386   ASHTABULA HARDWARE  4300099999
409080185   CAIRO CHEMICAL      4360002746
411080285   DOVER MUTUAL        4100034166
412080385   PARIS AUTO          4100015000
413090385   ROME BOATING        4150017982
414061185   PRIME COMPUTER      4360000123
C82080785   ODESSA SERVICES     4100004670
4500B0785   ANTIOCH SERVALL     4300002580
580080785   BETHLEHEM TAXI      RR00009840
681080785   ATHENS LUMBER       18500036BB
```

The PRINT-FILE contains the following message:

```
NO PRINT RECORDS
```

NEWFILE contains the following record:

```
411080285   DOVER MUTUAL        410003416
```

For subsequent executions, enter

```
OK, RESUME RANDOM
```

11 Relative Files

COBOL85 allows access to records of a disk file in either a random or a sequential manner. Each record in a relative file is uniquely identified by its **relative key**, an integer value that specifies the record's position in the file.

COBOL85 processes direct access (relative) files created with the MIDASPLUS or PRISAM interface. The COBOL85 relative key corresponds to the MIDASPLUS direct access primary index. The relative key is part of the MIDASPLUS record description but is not part of the COBOL85 record description. The *MIDASPLUS User's Guide* discusses in detail how to prepare files for COBOL85 access with MIDASPLUS. Throughout this chapter, references are made to MIDASPLUS utilities. If you use PRISAM, see the *PRISAM User's Guide* for information on creating relative files.

This chapter discusses relative file concepts, common operations on relative files, and elements of the ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION as they pertain to relative file processing. The chapter concludes with an example.

Relative File Concepts

This section discusses the following relative file concepts:

- Organization
- Relative key
- Access modes
- File formats
- Current record pointer
- File status
- INVALID KEY condition
- NOT INVALID KEY condition
- AT END condition
- NOT AT END condition
- Exception conditions

Organization

Relative file organization is permitted only on disk. A relative file consists of records that are identified by relative record numbers. Think of the file as a serial string or array of areas, each capable of holding a logical record. A relative record number identifies each area. Records are stored and retrieved based on this number. For example, in the representation of a relative file in Figure 11-1, the tenth record is the one addressed by relative record number 10 and is in the tenth record area, whether or not records are written in record areas 1 through 9.

No.	Record
1	JAN 40102
2	FEB 29800
3	MAR 45895
4	
5	
6	
7	
8	
9	SEPT 7921
10	OCT 7580
11	NOV 8400
12	DEC 10298

Q10166-1LA-22-0

FIGURE 11-1
A Relative File

In most cases, a file used as a relative file in a program must have a template created as a direct access file with MIDASPLUS or PRISAM. Set the maximum number of records and declare all fields when you create the file template; you cannot change file attributes merely by changing their descriptions in the COBOL85 program.

In some cases, the COBOL85 program creates a relative file. See the OPEN statement for details.

Relative Key

The data item named in the RELATIVE KEY clause of the *file-control-entry* for a file contains the current record number for that file. For inserting, updating, and deleting records in a file, each record is identified solely by the value of its relative key. This value must, therefore, be unique and must not be changed when updating the record. The RELATIVE KEY data item is not part of the file's COBOL85 record description.

Access Modes

COBOL85 allows three access modes for relative files. Specify them in the SELECT clause as follows:

- In **sequential access mode**, COBOL85 accesses records in ascending order of relative key values.
- In **random access mode**, the program controls the sequence in which it accesses records. To access a particular record the program places the record's relative record number in the relative key data item.
- In **dynamic access mode**, you can change at will from sequential access to random access for reading the file. The section Common Operations on Relative Files, later in this chapter, discusses access mode requirements for each I-O statement.

File Formats

MIDASPLUS always uses fixed-length record formatting for relative files. PRISAM uses variable-length and fixed-length record formatting for relative files. For variable-length record formatting with relative files, all records are preallocated as maximum length records. The template for the file must define the file's format consistent with the COBOL85 program definition. For additional information, see the *MIDASPLUS User's Guide* and the *PRISAM User's Guide.*

Note

Odd-length records contained in MIDASPLUS relative files contain an extra byte of data to fill the record out to the word boundary. This extra byte of data is undefined. For PRISAM relative files, odd-length records are padded in such a way that this extra byte is not visible to the user.

Current Record Pointer

The current record pointer is a conceptual entity used to indicate the next record to be accessed within a given file. The concept of the current record pointer has no meaning for a file opened in the output mode. Only the OPEN, START, and READ statements affect the setting of the current record pointer.

File Status

Code a file status check in the program to determine the success or failure of an I-O operation. Then use the result of the file status check to control the next program action. If you specify the FILE STATUS clause in a *file-control-entry*, COBOL85 places a value into

the specified data item during the execution of a READ, WRITE, REWRITE, DELETE, OPEN, CLOSE, or START statement to indicate the status of that input-output operation. The section ENVIRONMENT DIVISION, later in this chapter, discusses the FILE STATUS clause. For a complete discussion of COBOL85 file status codes, see Chapter 4.

The INVALID KEY Condition

The INVALID KEY condition can occur as a result of the execution of a START, READ, WRITE, REWRITE, or DELETE statement. For details of the causes of the condition, see the relevant statement.

When the INVALID KEY condition occurs, COBOL85 performs the following steps:

1. A value is placed into the FILE STATUS data item, if you specify one for the file, to indicate an INVALID KEY condition.
2. If you specify the INVALID KEY phrase in the statement causing the condition, control is transferred to the INVALID KEY imperative statement. Any USE procedure that you specify for the file is not executed. After the execution of the INVALID KEY imperative statement, control is transferred to the end of the I-O statement and the NOT INVALID KEY phrase, if you specify one, is ignored.
3. If you do not specify the INVALID KEY phrase for the file, but you specify a USE procedure, either explicitly or implicitly, that procedure is executed.

When the INVALID KEY condition occurs, execution of the input-output statement that caused the condition is unsuccessful and the file is not affected.

The NOT INVALID KEY Condition

If the I-O operation is successful, COBOL85 performs the following steps:

1. The FILE STATUS data item, if you specify one, is updated to indicate a successful completion.
2. If you specify the NOT INVALID KEY phrase, control is transferred to the imperative statement associated with the NOT INVALID KEY phrase. After the execution of this imperative statement, control is transferred to the end of the I-O statement and the INVALID KEY phrase, if you specify one, is ignored.
3. If you do not specify the NOT INVALID KEY phrase, control is transferred to the end of the I-O statement and the INVALID KEY phrase, if you specify one, is ignored.

The AT END Condition

The AT END condition can occur as a result of a sequential READ statement when no next logical record exists in the file. When the AT END condition occurs, COBOL85 performs the following steps:

1. A value is placed into the FILE STATUS data item, if you specify one for the file, to indicate an AT END condition.

2. If you specify the AT END phrase in the statement causing the condition, control is transferred to the AT END imperative statement. Any USE procedure that you specify for the file is not executed. After the execution of the AT END imperative statement, control is transferred to the end of the I-O statement and the NOT AT END phrase, if you specify one, is ignored.
3. If you do not specify the AT END phrase for the file, but you specify a USE procedure, either explicitly or implicitly, that procedure is executed.

The NOT AT END Condition

The NOT AT END condition can occur as a result of a successfully completed sequential READ statement. When the NOT AT END condition occurs, COBOL85 performs the following steps:

1. The FILE STATUS data item, if you specify one, is updated to indicate a successful completion.
2. If you specify the NOT AT END phrase, control is transferred to the imperative statement associated with the NOT AT END phrase. After the execution of the NOT AT END imperative statement, control is transferred to the end of the I-O statement and the AT END phrase, if you specify one, is ignored.
3. If you do not specify the NOT AT END phrase, control is transferred to the end of the I-O statement and the AT END phrase, if you specify one, is ignored.

Exception Conditions

Exception conditions can occur as a result of a Permanent Error condition or a Logic Error condition. (See Chapter 4.) When an exception condition that is not an INVALID KEY or AT END condition occurs, COBOL85 performs the following steps:

1. A value is placed into the FILE STATUS data item, if you specify one for the file, to indicate the exception condition.
2. If you specify a USE procedure for the file, that procedure is executed. Any INVALID KEY, NOT INVALID KEY, AT END, NOT AT END phrases are ignored.
3. Program execution terminates.

Common Operations on Relative Files

This section discusses the following common operations on relative files:

- Creating a file
- Creating (adding) records
- Positioning the file to a certain record
- Reading a certain record
- Deleting a certain record
- Updating (changing) a certain record
- Handling errors

The concept of the relative key is essential for most of the following operations on relative files, because only the keys allow access to the data records themselves.

Creating a File

To create a MIDASPLUS file, use the MIDASPLUS CREATK command to create a direct access file template. Then build a file on that template either with the MIDASPLUS KBUILD command and an existing data file, or with a COBOL85 program. In some cases, the COBOL85 program creates a relative file. See the OPEN statement for details.

The COBOL85 program must describe the new file's organization as RELATIVE, and open it in one of the three access modes discussed earlier.

For more information on using CREATK and KBUILD, see the *MIDASPLUS User's Guide.*

For information on creating PRISAM files, see the *PRISAM User's Guide.*

Creating (Adding) Records

Use WRITE to add new records. In sequential access mode, COBOL85 writes records in ascending order regardless of any value in the key field. In random or dynamic access mode, you must place a unique value in the relative key field before writing each record. In random or dynamic access mode, you can write records in any order and insert them anywhere in an existing file.

Positioning the File to a Certain Record

In sequential or dynamic access mode, use START to position the file if you do not want to access the first record. If you want to sequentially read groups of records within the file, use additional START statements.

In random access mode, you must place the position of the record to be READ into the *data-name* in the KEY IS phrase. Do not use START in random access mode.

Reading a Certain Record

If you open a file in sequential access mode, no special operations other than READ are necessary to read from the start of the file in sequential order of the key. If you do not want to read the first record, precede the first READ by a START to specify the key of the first record you want to read. All subsequent READs access each next record in key order until the program executes another START or closes the file, or an error occurs.

In random access mode, you must specify a key before each READ by moving a value into the KEY field or by using the START verb.

If you want to access the file both sequentially and randomly, specify dynamic access mode. To change from random to sequential reading, use a series of Format 1 READs (READ NEXT).

Deleting a Certain Record

Use the DELETE statement to delete records. In all access modes, place the proper value in the key field before using DELETE. In sequential access mode, first read the record to be deleted; then use the DELETE verb.

Updating (Changing) a Certain Record

Use the REWRITE statement to update records. The file must be open in I-O mode. In all access modes, place the proper value in the key field before using REWRITE. In sequential access mode, first read the record to be rewritten; then use the REWRITE verb.

Handling Errors in All of These Statements

The INVALID KEY clause and the USE statement in the declaratives section provide error handling. One of these elements is required for each I-O statement. Use the AT END phrase only for a sequential READ. Use USE statements in the declaratives section to provide error handling for non-end-of-file errors for a sequential READ, and for non-invalid-key errors.

ENVIRONMENT DIVISION

This section contains information that is unique to relative files. Chapter 6 contains information that applies to all file organizations. Chapters 10 and 12 contain information that applies to indexed files and sequential tape files, respectively.

INPUT-OUTPUT SECTION — FILE-CONTROL

Use the INPUT-OUTPUT SECTION to specify peripheral devices and information needed to transmit and handle data between the devices and the program.

Use the FILE-CONTROL paragraph to name each file and to specify other file-related information. Each file requires one *file-control-entry*.

Format

```
SELECT [OPTIONAL] file-name-1
  ASSIGN TO { device-name }
            { literal-1   }

[ RESERVE integer-1 [ AREA  ] ]
                    [ AREAS ]

[ORGANIZATION IS] RELATIVE

[ ACCESS MODE IS { SEQUENTIAL [, RELATIVE KEY IS data-name-1] }
                 { { RANDOM  } , RELATIVE KEY IS data-name-1 }
                 { { DYNAMIC }                               } ]

[ FILE STATUS IS data-name-2].
```

General Rules

1. The SELECT clause specifies the name of the relative file. You must specify the SELECT clause first in the *file-control-entry*. The remaining clauses can appear in any order. COBOL85 manages all relative files with either the MIDASPLUS interface or the PRISAM interface. Use ASSIGN TO MIDASPLUS to access MIDASPLUS files or ASSIGN TO PRISAM to access PRISAM files. You can also specify ASSIGN TO PFMS for compatibility with previous Prime COBOL compilers, but access is not as efficient.
2. The ORGANIZATION IS RELATIVE clause specifies that the file named in the SELECT clause contains data organized by relative keys. You establish file organization at the time you create the file, and you cannot subsequently change it.
3. The ACCESS MODE clause specifies the manner in which the program is to read or write to a relative file. If you omit this clause, the default is sequential access. The three access modes are described below:

   - When you specify SEQUENTIAL, COBOL85 writes and retrieves records in the order of ascending record number.
   - When you specify RANDOM, COBOL85 writes and retrieves records randomly, based on the value placed in the RELATIVE KEY field prior to a READ or WRITE. Random mode precludes a sequential READ NEXT.

- When you specify DYNAMIC, a program can read randomly or sequentially, and you can use the START statement. Other operations follow the rules for random access.

4. The RELATIVE KEY clause specifies the data item used to communicate a relative record number between the COBOL85 program and the file management system. You need not specify RELATIVE KEY for sequential access.

   - Do not define *data-name-1* in the record description associated with *file-name*. Rather, define it in the WORKING-STORAGE or LINKAGE SECTION. It must refer to an unsigned integer.
   - If the *file-description-entry* specifies EXTERNAL, the relative key is also EXTERNAL; therefore, do not define it in the LINKAGE SECTION.
   - Do not specify *data-name-1* with an OCCURS clause, or include it within a group subordinate to an OCCURS clause. This means it must not be subscripted or indexed, but it can be qualified.
   - Do not specify *data-name-1* with a P character or a separate sign in its PICTURE clause.

5. All records stored in a relative file are uniquely identified by relative record numbers. The relative record number of a given record specifies the record's logical ordinal position in the file. The first logical record has a relative record number of 1, and subsequent logical records have relative record numbers of 2, 3, 4, and so on.
6. In the FILE STATUS clause, *data-name-2* must be a two-character field described in the DATA DIVISION. The file control system moves a value into *data-name-2* following the execution of every statement that explicitly or implicitly references the file. This value indicates the execution status of the statement to the program. Following a successful I-O operation, *data-name-3* contains '00'. For a complete discussion of COBOL85 file status codes, see Chapter 4.

   The FILE STATUS data item (*data-name-2*) must not be part of the record description for its file. It must be in the WORKING-STORAGE or LINKAGE SECTION. It can be qualified but must not be subscripted or indexed.

   If the *file-description-entry* specifies EXTERNAL, the FILE STATUS item (*data-name-2*) is also EXTERNAL; therefore, do not define it in the LINKAGE SECTION.

   Prime Extension: *data-name-2* can be either alphanumeric or numeric (PIC XX or PIC 99).

DATA DIVISION

The elements of the DATA DIVISION for relative files are the same as those described in Chapter 7 except for the *record-description-entry* and the relative key.

record-description-entry

COBOL85 record lengths and MIDASPLUS or PRISAM record lengths must be the same.

Relative Key

The relative key data item must not be part of the record description for its file. See the section INPUT-OUTPUT SECTION — FILE-CONTROL, earlier in this chapter, for a discussion.

PROCEDURE DIVISION

This section contains information that pertains to relative files. Chapter 8 contains information that applies to all file organizations. Chapters 9, 10, and 12 contain information that pertains to sequential disk files, indexed files, and sequential tape files, respectively.

CLOSE

Terminates the processing of files.

Format

CLOSE *file-name-1* [,*file-name-2*] . . .

Syntax Rule

The files named in the CLOSE statement need not all have the same organization or access.

General Rules

1. Once a program executes a CLOSE statement for a file, it must execute an OPEN statement for that file before executing any other statements for that file.
2. A program can execute a CLOSE statement only for a file that is open. If a CLOSE statement is attempted on a file that is not open, the file status data item, if you specify one, is set to indicate status code 42. After execution of any applicable declarative procedure, the program terminates. For this particular error, the only applicable declarative procedure is a procedure for *file-name*.
3. If any other error occurs during a CLOSE operation, the file status data item, if you specify one, is updated to indicate status code 30. Execution continues according to the rules specified in Chapter 4.

DELETE

Removes a record from a disk file.

Format

DELETE *file-name* RECORD

[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-DELETE]

Syntax Rules

1. Do not specify the INVALID KEY and the NOT INVALID KEY phrases for a DELETE statement that references a file that is in sequential access mode.
2. You must specify the INVALID KEY phrase for a DELETE statement on a file open in random or dynamic access mode and for which you specify no USE procedure.

General Rules

1. The DELETE statement logically removes a data record from the file. The file must be open in I-O mode. Execution of DELETE updates the value of the FILE STATUS data item, if you specify one. It does not affect the contents of the record area associated with *file-name*.
2. Transfer of control following the successful or unsuccessful execution of the DELETE operation depends on the presence or absence of the optional INVALID KEY and NOT INVALID KEY phrases, and on the presence or absence of USE procedures associated with the file. See the section Relative File Concepts, earlier in this chapter, for more information.
3. The END-DELETE clause delimits the scope of the DELETE statement. For more information, see the section Scope Terminators, in Chapter 8.

Rules for Sequential Access

1. In sequential access, the program must successfully read the record to be deleted before it can delete it. Otherwise, an exception condition occurs, and the DELETE statement is unsuccessful. A status code of 43 is placed in the FILE STATUS field, if you specify one. Exception conditions and file status are discussed in the section Relative File Concepts, earlier in this chapter.
2. You must not change the RELATIVE KEY between the READ and DELETE statements.

Rules for Random and Dynamic Access

1. Random and dynamic access modes do not require that you first read the record. COBOL85 uses the current key value to select the record to delete.
2. If that record does not exist in the file, the INVALID KEY statement is executed, and a status code of 23 is placed in the FILE STATUS field, if one exists. INVALID KEY and FILE STATUS are discussed in the section Relative File Concepts, earlier in this chapter.

DELETE Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a DELETE statement: 00, 23, 43, 49, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

OPEN

Initiates the processing of files and performs other input-output operations.

Format

```
OPEN { INPUT  file-name-1 [, file-name-2] ... }
     { OUTPUT file-name-3 [, file-name-4] ... } ...
     { I-O    file-name-5 [, file-name-6] ... }
```

General Rules

1. A file opened as INPUT can be accessed only in a READ or START statement.
2. A file opened as OUTPUT can be accessed only in a WRITE statement.
3. A file opened as I-O can be accessed by a READ, WRITE, REWRITE, START, or DELETE statement.

Note

You cannot use all I-O statements in all access modes. Table B-8 in Appendix B specifies the types of I-O statements permissible with the different access modes.

4. Following the initial execution of an OPEN statement for a file, the program must close the file before it can open the file again.
5. Execution of the OPEN statement does not obtain or release the first data record. However, for files opened in INPUT or I-O mode, the OPEN statement positions the file to the first record.
6. The *file-description-entry* for an opened file must be equivalent to the file's template created by the FAU or CREATK utility.

7. For PRISAM files opened in OUTPUT mode or an OPTIONAL file opened in I-O mode, OPEN does not create a relative file. It merely opens an existing file for writing. You must create the file template with the PRISAM FAU utility. If the file is not present, the OPEN does not occur. A status code of 37 is returned, declaratives are invoked, if present, and the program terminates.
8. For MIDASPLUS and PFMS files opened in OUTPUT mode or an OPTIONAL file opened in I-O mode, if the file is not present, OPEN creates the file. COBOL85 builds a template using runtime calls to CREATK and using information in the program to describe the file to MIDASPLUS. For relative files, COBOL85 cannot determine from any information in the program the number of records to allocate. Therefore, the number defaults to 500. Use CREATK with the DATA function to change this number.
9. For MIDASPLUS and PRISAM files opened in OUTPUT mode, if the file is present and contains data, the OPEN statement is unsuccessful and a status code of 37 is returned. An exception condition occurs and the FILE STATUS field is updated, if one exists. Exception conditions and file status are discussed in the section Relative File Concepts, earlier in this chapter.
10. If any system error occurs during the OPEN processing, a status code of 30 is returned. An exception condition occurs and the FILE STATUS field is updated, if one exists. Exception conditions and file status are discussed in the section Relative File Concepts, earlier in this chapter.
11. During the execution of the OPEN statement, COBOL85 checks file attributes against the attributes described for the file in the program. These attributes are organization, relative key, minimum and maximum logical record sizes, and the record type (fixed or variable, MIDASPLUS or PRISAM).

    If any of these attributes conflict, the OPEN statement is unsuccessful, and a status code of 39 is returned. An exception condition occurs, and the FILE STATUS field is updated, if one exists. Exception conditions and file status are discussed in the section Relative File Concepts, earlier in this chapter.
12. If the SELECT statement associated with the file specifies MIDASPLUS as the assigned device, MIDASPLUS is used to open the file. If the SELECT statement associated with the file specifies PRISAM as the assigned device, PRISAM is used to open the file. If the SELECT statement associated with the file specifies PFMS as the assigned device, an internal check is made to see if the file is a MIDASPLUS or a PRISAM file. All subsequent I-O operations are performed using the appropriate data management interface.
13. If an unavailable optional file is opened with the INPUT phrase, the OPEN statement sets the current record pointer to indicate the AT END condition. The OPEN statement is successful. The FILE STATUS data item is set to the informational status code 05.

OPEN Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of an OPEN statement: 00, 05, 07, 30, 35, 37, 39, 41, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

READ

For sequential access, makes available the next logical record from a file. For random access, makes available a record with a specific key value.

Format 1 (Sequential or Dynamic)

READ *file-name* [**NEXT**] **RECORD** [**INTO** *data-name-1*]

[**AT END** *imperative-statement-1*]

[**NOT AT END** *imperative-statement-2*]

[**END-READ**]

Format 2 (Random or Dynamic)

READ *file-name* **RECORD** [**INTO** *data-name-1*]

[**INVALID KEY** *imperative-statement-1*]

[**NOT INVALID KEY** *imperative-statement-2*]

[**END-READ**]

General Rules

1. Do not use the INTO phrase when the input file contains logical records of various sizes as indicated by their record descriptions.
2. You must specify the INVALID KEY phrase (Format 2) or the AT END phrase (Format 1) if you do not specify an applicable USE procedure for *file-name*.
3. The associated file must be open in the INPUT or I-O mode at the time this statement is executed.
4. The execution of the READ statement updates the value of any FILE STATUS data item associated with *file-name*. File status is discussed in the section Relative File Concepts, earlier in this chapter.
5. If you specify the INTO phrase, the record being read is moved from the record area to *data-name-1* according to the rules specified for the MOVE statement without the CORRESPONDING phrase. The implied MOVE does not occur if the execution of the READ statement is unsuccessful. Any subscripting or indexing associated with *data-name-1* is evaluated after the record is read and immediately before it is moved to the data item.

   When you use the INTO phrase, the record being read is available in both the input record area and *data-name-1*.
6. For variable-length records, the following rules apply:

   - If the number of character positions in the record that is read is less than the minimum size specified in the *record-description-entry*, the portion of the record area to the right of the last valid character read is undefined.

- If the number of character positions in the record that is read is greater than the maximum size specified in the *record-description-entry*, the record is truncated on the right to the maximum size.

In either of these cases, the READ statement is successful, and the FILE STATUS data item, if you specify one, is set to 04 to indicate a record length conflict.

7. Following the unsuccessful execution of any READ statement, the contents of the associated record area and the position of the current record pointer are undefined.
8. Transfer of control following the successful or unsuccessful execution of the READ operation depends on the presence or absence of the optional INVALID KEY, NOT INVALID KEY, AT END, and NOT AT END phrases, and on the presence or absence of USE procedures associated with the READ statement. See the section Relative File Concepts, earlier in this chapter, for more information.
9. The END-READ clause delimits the scope of the READ statement. For more information, see the section Scope Terminators, in Chapter 8.

Rules for Format 1 (Sequential and Dynamic Access)

1. Use Format 1 for all files in sequential access mode. The NEXT phrase is optional and has no effect for sequential access.
2. The NEXT phrase is required for files in dynamic access mode, when you want to retrieve records sequentially. If you do not use NEXT, you must place the relative record number of the record to be retrieved in the key field.
3. The record made available by a Format 1 READ statement is determined as follows:

   - If the current record pointer was positioned by the START or OPEN statement, the record to which it points is made available, provided that it is still accessible. If the record is no longer accessible, perhaps because the record is deleted, the current record pointer is updated to point to the next existing record. That record is then made available.
   - If the current record pointer was positioned by the execution of a previous READ statement, the current record pointer is updated to point to the next existing record in the file. Then that record is made available.
   - For a relative file in the dynamic access mode, the execution of an OPEN I-O statement followed by one or more WRITE statements and then a READ NEXT statement causes the READ NEXT statement to access the first record in the file. However, if the WRITE statement inserted records with a key value lower than that of any records previously existing in the file, these records are returned by the first READ NEXT statement.

4. If, at the time of execution of a Format 1 READ statement, the position of the current record pointer for that file is undefined, the execution of that READ statement is unsuccessful. The FILE STATUS data item, if you specify one, is set to 46, and an exception condition occurs.
5. If, at the time of the execution of a Format 1 READ statement, no next logical record exists in the file, an optional input file is unavailable, or the relative record number is larger than the relative key data item, the AT END condition occurs, and the execution of the READ statement is unsuccessful.

6. When the AT END condition occurs, COBOL85 performs the following steps:

   a. A value is placed into the FILE STATUS data item, if you specify one for the file, to indicate an AT END condition.

   b. If you specify the AT END phrase, control is transferred to the associated imperative statement. Any USE procedure you specify for this file is not executed.

   c. If you do not specify the AT END phrase, then the USE procedure you specify for this file is executed. Otherwise, execution is aborted.

7. When the AT END condition occurs, the program must not execute a Format 1 READ statement for that file without first executing one of the following:

   - A successful CLOSE statement followed by the execution of a successful OPEN statement for that file
   - A successful START statement for that file
   - A successful Format 2 READ statement for that file

8. If an AT END condition does not occur during the execution of a READ statement and the READ statement is successful, the AT END phrase, if you specify one, is ignored, and the following actions occur:

   - The current record pointer is set and the FILE STATUS data item, if you specify one for this file, is updated.
   - The record is made available in the record area and any implicit move resulting from the presence of an INTO phrase is executed. Control is transferred to the end of the READ statement or to the *imperative-statement* specified by the NOT AT END phrase, if you specify one for the file.

9. If an exception condition that is not an AT END condition exists, the FILE STATUS item is updated and control is transferred according to the rules for the USE statement.

10. In dynamic access mode, a Format 1 READ NEXT statement retrieves the next logical record from the file as described in General Rule 3 for Format 1.

11. If you specify the RELATIVE KEY phrase, the execution of a Format 1 READ statement updates the contents of the RELATIVE KEY data item with the relative record number of the record made available.

12. If the number of significant digits in the relative record number to be read is larger than the size of the relative key data item, the READ statement is unsuccessful, and the AT END condition occurs. The FILE STATUS data item, if you specify one, is updated to status code 14.

13. For a Format 1 READ statement, include a USE procedure in the program to handle any non-end-of-file errors that may occur. This action may be necessary to handle errors that may occur when multiple users are accessing a MIDASPLUS or PRISAM file that has been opened in I-O mode. This USE procedure also handles other unexpected conditions that may be generated during sequential file access.

Rules for Format 2 (Random and Dynamic Access)

1. Use Format 2 for files in random access mode or for files in dynamic access mode when you want to retrieve records randomly.
2. If an optional input file is unavailable, the INVALID KEY condition exists and the READ statement is unsuccessful.
3. Execution of a Format 2 READ statement compares the value of the relative key with the relative position of the stored records in the file, until the record having the corresponding record number is found. The current record pointer is positioned to this record, which is then made available. If no record can be so identified, the INVALID KEY condition exists and execution of the READ statement is unsuccessful.

READ Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a READ statement: 00, 02, 04, 10, 14, 23, 30, 46, 47, 90, 97, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

REWRITE

Logically replaces a record on a disk file.

Format

REWRITE *record-name* [FROM *data-name*]

[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-REWRITE]

Syntax Rules

1. The *record-name* is the name of a logical record in the FILE SECTION of the DATA DIVISION and can be qualified.
2. In random or dynamic access mode, you must specify the INVALID KEY phrase in the REWRITE statement for files for which you do not specify an appropriate USE procedure.
3. Do not specify the INVALID KEY and the NOT INVALID KEY phrases for a REWRITE statement that references a file that is in sequential access mode.

General Rules

1. The REWRITE statement can change all data fields in the record.
2. The file must be opened for I-O for all access methods.

3. In sequential access mode, the program must successfully read a record prior to the REWRITE statement.
4. For a file accessed in either random or dynamic access mode, REWRITE replaces the record specified by the contents of the RELATIVE KEY data item. If the file does not contain the record specified by the key, the INVALID KEY condition exists, the updating operation does not take place, and the data in the record area is unaffected.
5. The REWRITE statement does not affect the current file position.
6. The FROM option allows you to create the record in another area. It is equivalent to

   MOVE ***data-name*** **TO** ***record-name***

   prior to the execution of the REWRITE statement.
7. The number of character positions in the record referenced by *record-name* must be equal to the number of character positions in the record being replaced.
8. **Prime Extension:** The logical record released by a successful execution of the REWRITE statement is still available in the record area.

   If you name the associated file in a SAME RECORD AREA clause, the logical record is also available to the program as a record of other files appearing in the same clause.
9. The execution of the REWRITE statement updates the value of any FILE STATUS data item associated with the file. File status is explained in the section Relative File Concepts, earlier in this chapter.
10. Transfer of control following the successful or unsuccessful execution of the REWRITE operation depends on the presence or absence of the optional INVALID KEY and NOT INVALID KEY phrases, and on the presence or absence of USE procedures associated with the REWRITE statement. See the section Relative File Concepts, earlier in this chapter, for more information.
11. The END-REWRITE clause delimits the scope of the REWRITE statement. For more information, see the section Scope Terminators, in Chapter 8.

REWRITE Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a REWRITE statement: 00, 22, 30, 43, 44, 49, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

SEEK — Prime Extension

Is supported syntactically only for compatibility with other COBOL implementations.

Format

SEEK *file-name* RECORD

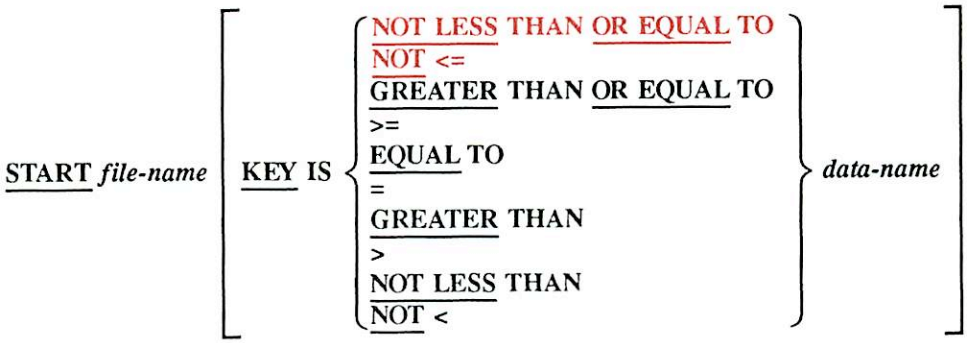
General Rules

1. SEEK is treated as documentation in COBOL85.
2. You must define the *file-name* in a *file-description-entry* in the DATA DIVISION.

START

Establishes a position in the file for subsequent READs.

Format



[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-START]

Syntax Rules

1. The *file-name* must be the name of a relative file with sequential or dynamic access.
2. The *data-name* can be qualified but not indexed or subscripted.
3. You must specify the INVALID KEY phrase if you do not specify an applicable USE procedure for *file-name*.
4. The *data-name*, if you use it, must be the name in the RELATIVE KEY clause for this file.

General Rules

1. The *file-name* must be open in the INPUT or I-O mode when the START statement is executed.
2. If you do not specify the KEY phrase, the relational operator IS EQUAL TO is the default.
3. The current record pointer is positioned to the first logical record in the file whose key satisfies the comparison.

If no record in the file satisfies the comparison, or an optional input file is unavailable, an INVALID KEY condition exists, the execution of the START statement is unsuccessful, and the position of the current record pointer is undefined.

4. The execution of the START statement updates the value of any FILE STATUS data item associated with *file-name*. File status is explained earlier in this chapter.
5. Whether or not you specify the KEY phrase, the comparison uses the data item referenced by the RELATIVE KEY *data-name*.
6. START does not retrieve a record, but only positions the file to a specific record.
7. Transfer of control following the successful or unsuccessful execution of the START operation depends on the presence or absence of the optional INVALID KEY and NOT INVALID KEY phrases, and on the presence or absence of USE procedures associated with the START statement. See the section Relative File Concepts, earlier in this chapter, for more information.
8. The END-START clause delimits the scope of the START statement. For more information, see the section Scope Terminators, in Chapter 8.

START Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a START statement: 00, 23, 47, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

WRITE

Releases a logical record for an output file.

Format

WRITE *record-name* [**FROM** *data-name*]

[**INVALID KEY** *imperative-statement-1*]

[**NOT INVALID KEY** *imperative-statement-2*]

[**END-WRITE**]

Syntax Rules

1. The *record-name* and *data-name-1* can name the same storage area.
2. The *record-name* is the name of a logical record in the FILE SECTION of the DATA DIVISION and can be qualified.
3. You must specify the INVALID KEY phrase if you do not specify an applicable USE procedure for the associated file.

General Rules

1. The associated file must be open in the OUTPUT or I-O mode.
2. **Prime Extension:** The logical record released by the WRITE statement is still available in the record area.

   If you name the associated file in a SAME RECORD AREA clause, the logical record is also available as a record of other files referenced in the clause.
3. Execution of the WRITE statement with the FROM phrase is equivalent to the statement

   MOVE ***data-name-1*** **TO** ***record-name***

   followed by a WRITE statement.
4. If the number of character positions in the record is less than the minimum size or larger than the maximum size allowed for the file, the WRITE statement is unsuccessful, and an exception condition exists.
5. The execution of the WRITE statement updates the value of any FILE STATUS data item associated with the file. File status is explained in the section Relative File Concepts, earlier in this chapter.

Rules for Record Keys

1. When you open a file in output mode, you can insert records into the file in one of the following ways:

   - In sequential access mode, the WRITE statement releases a record. The first record is given a relative record number of 1. Records released subsequently are given relative record numbers of 2, 3, 4, and so on. If you specify the RELATIVE KEY data item in the *file-control-entry* for the associated file, COBOL85 places the relative record number of the record just released into the RELATIVE KEY data item.
   - In random or dynamic access mode, prior to the execution of the WRITE statement, you must initialize the value of the RELATIVE KEY data item with the relative record number. That record is then released.

2. The INVALID KEY condition exists under any of the following circumstances:

   - The access mode is random or dynamic, and the RELATIVE KEY data item specifies a record that already exists.
   - The program attempts to write beyond the externally defined boundaries of the file when, for example, the relative key value is larger than the number of records allocated by MIDASPLUS for that file.
   - The program attempts a sequential WRITE statement, and the number of significant digits in the relative record number is larger than the size of the relative key data item.

3. Transfer of control following the successful or unsuccessful execution of the WRITE operation depends on the presence or absence of the optional INVALID KEY and NOT INVALID KEY phrases, and on the presence or absence of USE procedures associated with the WRITE statement. See the section Relative File Concepts, earlier in this chapter, for more information.
4. The END-WRITE clause delimits the scope of the WRITE statement. For more information, see the section Scope Terminators, in Chapter 8.

WRITE Status Codes

One of the following status codes is placed in the FILE STATUS data item, if one exists, at the completion of a WRITE statement: 00, 22, 24, 30, 48, 99. For a complete discussion of COBOL85 file status codes, see Chapter 4.

Example

This program illustrates operations on a relative file in random access mode.

```
 ID DIVISION.
 PROGRAM-ID.    RANDOM2.
*****************************************************************
 REMARKS. THIS PROGRAM ILLUSTRATES READ, WRITE, REWRITE, AND
     DELETE FOR A RELATIVE FILE IN RANDOM ACCESS MODE.  IT READS A
     TRANSACTION FILE CONTAINING UPDATES FOR MONTHLY BUDGETS,
     AND UPDATES A MONTH-FILE WHOSE RELATIVE KEY IS THE NUMBER
     OF THE MONTH.
*****************************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PRIME.
 OBJECT-COMPUTER. PRIME.
 INPUT-OUTPUT SECTION.
*
 FILE-CONTROL.
*
     SELECT MASTER-FILE ASSIGN TO MIDASPLUS
         ORGANIZATION IS RELATIVE,
         ACCESS MODE IS RANDOM,
         RELATIVE KEY IS KEY-MS
         FILE STATUS IS FS-MS.
*
     SELECT TRANS-FILE ASSIGN TO PRIMOS,
         ORGANIZATION IS SEQUENTIAL,
         FILE STATUS IS FS-TR.
*
     SELECT PRINT-FILE ASSIGN TO PRINTER.
*********************************************************
 DATA DIVISION.
 FILE SECTION.
*
 FD  MASTER-FILE,
     VALUE OF FILE-ID IS KMONTHF.
```

```
 01  MASTER-RECORD.
     05 MASTER-KEY                    PIC XX.
     05 INFORMATION                   PIC X(30).
     05 FILLER                        PIC X(6).
*
 FD  TRANS-FILE    COMPRESSED
     VALUE OF FILE-ID IS TMONTHF.
 01  TRANS-RECORD.
     05 TRANS-CODE               PIC X.
     05 TRANS-ENTRY              PIC X(30).
     05 KEY-CODE                 PIC 99.
*
 FD  PRINT-FILE,
     LABEL RECORDS ARE OMITTED.
 01  PRINT-LINE                  PIC X(33).
*
 WORKING-STORAGE SECTION.
 77  FS-MS                       PIC XX VALUE '00'.
 77  FS-TR                       PIC XX VALUE '00'.
 77  KEY-MS                      PIC 99 VALUE ZEROES.
 77  NO-MORE-INPUT               PIC X VALUE 'N'.
 77  KMONTHF                     PIC X(30)  VALUE 'KMONTH'.
 77  TMONTHF                     PIC X(30)  VALUE 'TMONTH'.
**************************************************
 PROCEDURE DIVISION.
*
 DECLARATIVES.
 INPUT-ERROR SECTION. USE AFTER STANDARD ERROR PROCEDURE ON INPUT.
 FIRST-PARAGRAPH.
     DISPLAY '**ERROR: **'.
     EXHIBIT FS-MS, FS-TR.
     CLOSE TRANS-FILE, MASTER-FILE, PRINT-FILE.
     STOP RUN.
 OUTPUT-ERROR SECTION. USE AFTER STANDARD ERROR PROCEDURE
                       ON OUTPUT.
 SECOND-PARAGRAPH.
     DISPLAY '**ERROR: **'.
     EXHIBIT FS-MS, FS-TR.
     CLOSE TRANS-FILE, MASTER-FILE, PRINT-FILE.
     STOP RUN.
 END DECLARATIVES.
 000-MAINLINE.
     READY TRACE.
     PERFORM 005-ACCEPT-FILE-NAMES.
     OPEN INPUT TRANS-FILE,
          I-O MASTER-FILE,
          OUTPUT PRINT-FILE.
     PERFORM 010-UPDATE-MONTHLY-BUDGETS.
     CLOSE TRANS-FILE, MASTER-FILE, PRINT-FILE.
     STOP RUN.
*
 005-ACCEPT-FILE-NAMES.
     DISPLAY 'ENTER MASTER-FILE -- KMONTH OR OTHER'.
     ACCEPT KMONTHF.
     DISPLAY 'ENTER TRANSACTION FILE -- TMONTH OR OTHER'.
     ACCEPT TMONTHF.
```

```
*
 010-UPDATE-MONTHLY-BUDGETS.
     READ TRANS-FILE AT END
          DISPLAY 'INPUT FILE IS EMPTY',
          CLOSE TRANS-FILE, MASTER-FILE, PRINT-FILE,
          STOP RUN.
     PERFORM 020-PROCESS-TRANS UNTIL NO-MORE-INPUT = 'Y'.
*
 020-PROCESS-TRANS.
     MOVE KEY-CODE TO KEY-MS.
     IF TRANS-CODE = 'U' PERFORM 100-UPDATE
        ELSE IF TRANS-CODE = 'A' PERFORM 110-INSERT
             ELSE IF TRANS-CODE = 'D' PERFORM 120-DELETE
                  ELSE EXHIBIT TRANS-RECORD
                       PERFORM 200-CREATE-ERROR-FILE.
     READ TRANS-FILE AT END
         MOVE 'Y' TO NO-MORE-INPUT
         DISPLAY 'END OF FILE'.
*
 100-UPDATE.
     READ MASTER-FILE INVALID KEY
         PERFORM 200-CREATE-ERROR-FILE.
     IF FS-MS = '00',
     MOVE TRANS-ENTRY TO INFORMATION.
     REWRITE MASTER-RECORD,
     INVALID KEY DISPLAY 'INVALID KEY'.
*
 110-INSERT.
     MOVE KEY-CODE TO KEY-MS MASTER-KEY.
     MOVE TRANS-ENTRY TO INFORMATION.
     WRITE MASTER-RECORD,
         INVALID KEY PERFORM 200-CREATE-ERROR-FILE.
*
 120-DELETE.
     READ MASTER-FILE INVALID KEY DISPLAY 'INVALID READ'.
     IF FS-MS = '00',
     DELETE MASTER-FILE RECORD, INVALID KEY
         PERFORM 200-CREATE-ERROR-FILE.
*
 200-CREATE-ERROR-FILE.
         WRITE PRINT-LINE FROM TRANS-RECORD.
```

This program, stored as RELATIVE.COBOL85, can be compiled, linked, and executed with the following dialog. The first and fifth input records of the file TMONTH cause the program to perform the error routine.

```
OK, COBOL85 RELATIVE -L
[COBOL85 Rev. 1.0-22.0 Copyright (c) 1988, Prime Computer, Inc.]
[0 ERRORS IN PROGRAM: <MYMFD>MYDIR>COBOL85>RELATIVE.COBOL85]

OK, BIND -LOAD RELATIVE -LI COBOL85LIB -LI
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
BIND COMPLETE

OK, RESUME RELATIVE
```

```
trace:  050-ACCEPT-FILE-NAMES
ENTER MASTER-FILE -- KMONTH OR OTHER
KMONTH
ENTER TRANSACTION FILE -- TMONTH OR OTHER
TMONTH
trace:  010-UPDATE-MONTHLY-BUDGETS
trace:  020-PROCESS-TRANS
TRANS-RECORD = XThis is wrong                  01
trace:  200-CREATE-ERROR-FILE
trace:  020-PROCESS-TRANS
trace:  120-DELETE
trace:  020-PROCESS-TRANS
trace:  100-UPDATE
trace:  020-PROCESS-TRANS
trace:  110-INSERT
trace:  020-PROCESS-TRANS
trace:  100-UPDATE
trace:  200-CREATE-ERROR-FILE
END OF FILE
OK,
```

Input Files

The master file, KMONTH, contains the following records before the program is run:

```
01This is the January Record          01
03This is the March Record            03
05This is the May Record              05
06This is the June Record             06
08This is the Aug. Record             08
09This is the Sept. Record            09
10This is the October Record          10
```

The transaction file, TMONTH, contains the following records:

```
XThis is wrong                 01
D                              08
UThis is the September Record  09
AThis is the February Record   02
UInvalid Key                   99
```

Output Files

The master file, KMONTH, contains the following records after the program is run:

```
01This is the January Record          01
02This is the February Record         02
03This is the March Record            03
05This is the May Record              05
06This is the June Record             06
09This is the September Record        09
10This is the October Record          10
```

The PRINT-FILE contains the following records:

```
XThis is wrong                  01
UInvalid Key                    99
```

For subsequent executions, enter

```
OK, RESUME RELATIVE
```

12 Tape Files

This chapter discusses the following topics related to tape file processing:

- Tape structure
- Blocking strategy
- Record structure
- Multivolume tape files
- Multiple file tapes
- Tape labels
- Compiling, linking, and executing programs that use tape
- Tape error messages

The chapter also discusses elements of the ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION as they pertain to tape file processing. The chapter concludes with an example.

Tape Structure

Prime computers support nine-track tape with parity checking. Multiple tape files on one volume and tape files that span multiple volumes are supported.

The amount of data that can be put on a tape depends on the following factors:

- Length of the tape in inches.
- Tape density in bytes per inch (bpi). Prime supports 800, 1600, and 6250 bpi.
- The blocking factor. The COBOL85 program's BLOCK CONTAINS clause of the *file-description-entry* allows grouping of more than one record or character into a block. The block is then read from or written to the tape at once, saving space. If records are not blocked, each record is followed by an Interrecord Gap (IRG) or Interblock Gap (IBG) of 1/2 inch minimum. (Exceptions are discussed in the *Magnetic Tape User's Guide*.) Blocking reduces the proportion of tape used by the IRGs. The size of the block that may be created in blocking is limited only by the maximum size of the tape buffer.

- The tape buffer size. For unblocked records, buffer size is the size of one record. Blocked records may occupy a buffer up to the maximum size. The maximum buffer or block size is listed in Appendix I.

Blocking Strategy

Blocking records in the largest groups possible makes the most efficient use of magnetic tape. Blocking saves space on the tape because, instead of a 1/2-inch gap after each record, a 1/2-inch gap occurs only after each block of records. Blocking also saves time because only one I-O operation is done per block of records, instead of one operation per record.

Figure 12-1 illustrates the saving of space and, therefore, time.

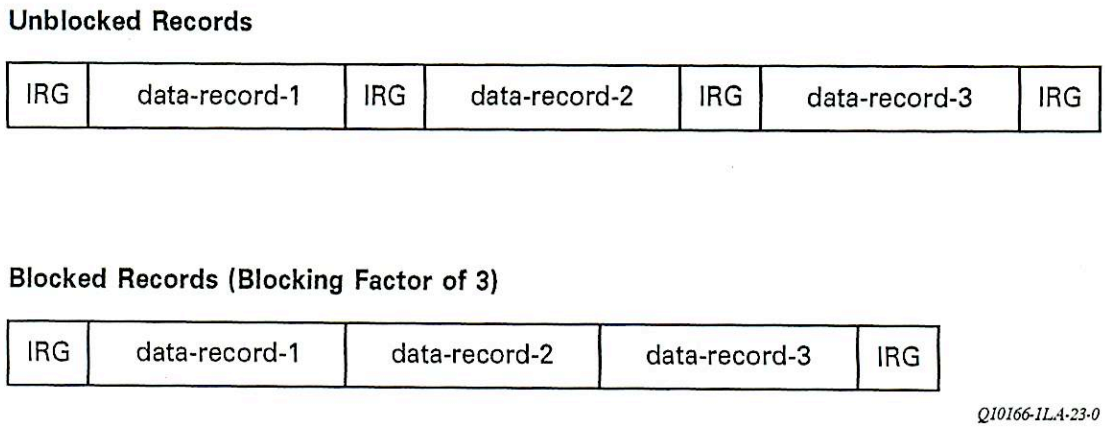


FIGURE 12-1
Blocking Strategy (IRG = Interrecord Gap)

Internal Structure of Fixed-length Records

Figure 12-2 illustrates fixed-length records of size *n* stored on magnetic tape, where *n* represents an even number of characters.

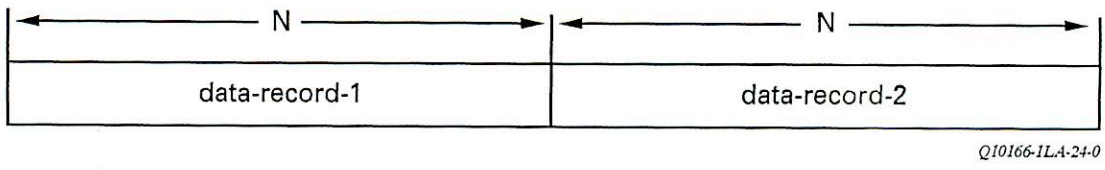


FIGURE 12-2
Internal Structure of Fixed-length Records (Even Length)

Fixed-length tape records are word-aligned. That is, if the record contains an even number of characters (*n*), then the records align themselves alongside one another.

Figure 12-3 illustrates fixed-length records of size *n*-1 stored on magnetic tape, where *n*-1 represents an odd number of characters and @ represents a pad byte.

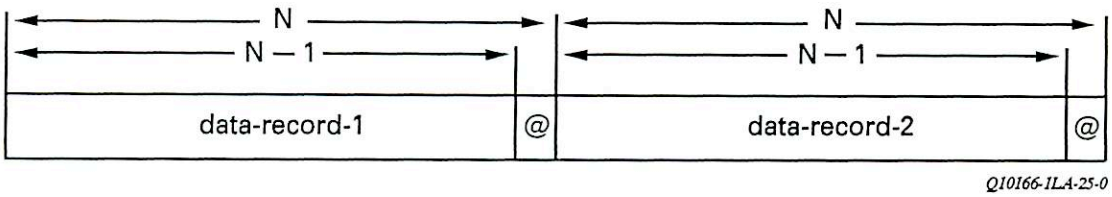


FIGURE 12-3
Internal Structure of Fixed-length Records (Odd Length)

If the record contains an odd number of characters (*n*-1), then the *n*th character is a pad byte. This pad byte, used to fill the record out to the word boundary, contains an undefined value. The record size passed to the MAGLIB interface routines is specified in words (*n*/2).

The maximum number of characters that can be written to tape at one time is 12K bytes (12288 characters). The pad byte within fixed odd-length records is considered part of the actual record when the record is written to tape.

Internal Structure of Variable-length Records

Variable-length records are written to tape according to the format specified in the ANSI Magtape Standard (ANSI X3.27-1978) illustrated in Figure 12-4 below.

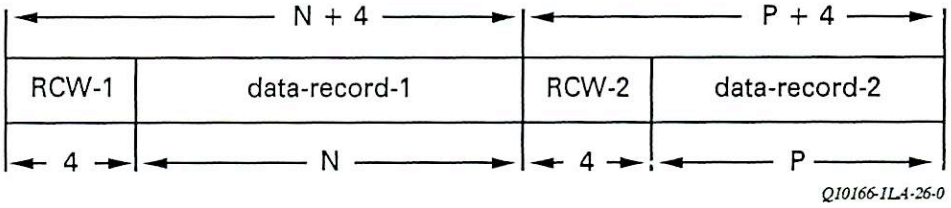
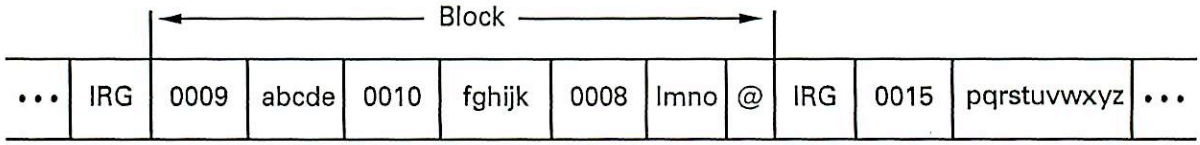


FIGURE 12-4
Internal Structure of Variable-length Records

A variable-length record consists of the Record Control Word (RCW) followed by the actual data record. The RCW is four characters long and contains the character value of the record length in bytes. The record length is the sum of the length of the data record and the four characters of the RCW. For example, RCW-1 above contains the value of $n + 4$, and RCW-2 contains the value of $p + 4$. Therefore, the maximum size of the data portion of a variable-length tape record is restricted to 9995 characters (9999 minus the four-character RCW).

To facilitate tape portability, the RCW is written in ANSI Standard 8-bit ASCII, otherwise referred to as Latin Alphabet No. 1 (ANSI X3.4-1977). The actual data records, however, are written in the native character set, defined in Appendix B, unless otherwise specified in the CODE-SET clause.

Unlike the word-aligned format of fixed-length tape records, the variable-length record format allows records to align themselves alongside one another without concern for interrecord word alignment. If a block ends on an odd byte, a pad byte is inserted at the end of the last record of the block to fill the block out to the word boundary. If a block ends on an even byte, no pad byte is necessary. For example, Figure 12-5 illustrates a tape containing blocked variable-length records and having a blocking factor of 3.



Q10166-1LA-27-1

```
where IRG = interrecord gap
        @  = block trailing pad byte
```

FIGURE 12-5
Blocked Variable-length Records

A pad byte is appended to unblocked variable-length records that contain an odd number of characters. This pad byte, however, is not considered part of the actual record when the record is written to tape.

Figure 12-6 illustrates a tape containing unblocked variable-length records.

Q10166-1LA-28-1

```
where  IRG = interrecord gap
         @ = pad byte
```

FIGURE 12-6
Unblocked Variable-length Records

Multivolume Tape Files

If a tape file is stored on more than one reel of tape, when the end of the first reel is detected, this message is displayed:

```
END OF VOLUME - MOUNT NEXT VOLUME
WHEN A NEW VOLUME HAS BEEN MOUNTED, HIT RETURN TO CONTINUE,
OTHERWISE, TYPE "A" TO ABORT PROCESSING
                                          (Recoverable error)
```

Put the tape drive offline, rewind the current tape, mount and load the next tape in the series, and then put the drive online again. Press the carriage return to continue execution.

If the wrong reel is mounted, the following message is displayed:

```
FILE SECTION-ID NUMBERS DO NOT MATCH
TYPE 'A' TO ABORT, ELSE CORRECT THE PROBLEM AND TYPE RETURN
TO CONTINUE:
```

To correct the problem, mount and load the correct tape volume in the series, put the drive online, and press the RETURN key to continue execution.

If an unlabeled tape reel is mounted, and the end of the tape is reached, a fatal error occurs. See the section Magnetic Tape Error Reporting, later in this chapter.

Multiple File Tapes

The MULTIPLE FILE TAPE clause, the expiration date parameter of the VALUE OF FILE-ID clause, and the OPEN WITH NO REWIND and CLOSE WITH NO REWIND statements allow you to process multiple files on a single tape volume, and multiple files on multiple tape volumes. See the discussions of each of these elements later in this chapter.

The following sections discuss positioning a multiple file tape for output and for input.

Positioning a Multiple File Tape for Output

The expiration dates of files already written on a multiple file tape control the positioning of the tape for output. Positioning for output is independent of the position specified in the POSITION phrase of the MULTIPLE FILE TAPE clause, and of the *tape-file-id*, *owner-id*, and *volume-id* parameters in the VALUE OF FILE-ID clause.

When the program executes an OPEN OUTPUT statement, COBOL85 positions the tape as follows:

- If the tape is at the end of data, it is left that way. The new file is appended.
- If the tape is not at the end of data, it is rewound. It is then positioned to its first expired file. The new file overwrites the expired file. All subsequent data on the tape is lost.
- If no expired file exists on the tape, the tape is positioned to the end of data, and the new file is appended.

Notes

A file expires on the date that is equal to or later than the expiration date. The default expiration date is the current date. Therefore, in order to create multiple file tape reels to which subsequently you can write selectively, specify expiration dates that lie in the future.

You cannot selectively overwrite a particular unexpired tape file.

Use OPEN WITH NO REWIND and CLOSE WITH NO REWIND when you write multiple files to a tape. Use CLOSE WITH NO REWIND to close a file just written, and OPEN

WITH NO REWIND to open the next file to be written. When you CLOSE a file WITH NO REWIND, you not only avoid a redundant rewind, but you also ensure that the next file is appended, and does not overwrite any existing files.

Caution

When writing several files with different expiration dates to the same tape, write them in the order of decreasing expiration date. This action prevents the loss of unexpired files when a preceding expired file is overwritten.

Positioning a Multiple File Tape for Input

The position specified in the POSITION phrase of the MULTIPLE FILE TAPE clause, and the *tape-file-id*, *owner-id*, and *volume-id* parameters in the VALUE OF FILE-ID clause control the positioning of a multiple file tape for input.

When the program executes an OPEN INPUT statement, COBOL85 positions the tape as follows:

- If you specify the POSITION phrase, the tape is positioned according to the position number specified in the phrase.
- If you omit the POSITION phrase, the tape is positioned to the first file whose *tape-file-id*, *volume-id*, and *owner-id* match the parameters you specify in the VALUE OF FILE-ID clause of the *file-description-entry*.

If the program attempts to position the tape beyond the end of data, a runtime error occurs.

Caution

Correct tape positioning requires that the tape be positioned at its load point when the program executes the OPEN INPUT statement. Correct access of the specified file is not guaranteed if the tape is positioned elsewhere, and in particular if the OPEN INPUT statement includes the WITH NO REWIND phrase.

Overview of the LABEL Command

The PRIMOS LABEL command writes tape labels on magnetic tapes and verifies existing tape labels. These labels can be in IBM format (9-track EBCDIC or 7-track BCD), ANSI format (9-track ASCII), or Prime format (nonstandard Level 1 volume labels followed by a dummy HDR1 label and two file markers). You can also use LABEL to read existing VOL1 and HDR1 labels.

ANSI labels are written in accordance with the American National Standards Institute standard ANSI X3.27-1978. IBM labels are written in accordance with IBM's specifications (IBM manual GC28-6680-5).

Any nonstandard labels such as 7-track ASCII or user-defined labels *cannot* be read or written.

If you use LABEL without the –VOLUME option on a tape that is already labeled, the command reads the existing label. If you want to relabel a previously labeled tape, you must use the –INIT option.

To ensure proper initialization of a magnetic tape, use the LABEL command to initialize the tape prior to processing. Always initialize a tape regardless of whether it was previously initialized by another program.

Using LABEL

The LABEL command has the following format:

```
                  ┌ –TYPE type                  ┐
                  │ –VOLUME volume-id           │
                  │ –OWNER owner                │
                  │ –ACCESS access              │
LABEL MTn         │ –HELP                       │
                  │ –INIT                       │
                  │ –OVERWRITE                  │
                  │ –PARITY { EVEN }            │
                  └       { ODD  }              ┘
```

The elements in the LABEL format have the following meanings:

Element	*Meaning*
MT*n*	Specifies the tape drive on which the tape to be labeled is mounted. *n*, an integer in the range 0 through 7 inclusive, is the tape drive's number. This argument must be present and must be the first option on the command line. You must previously have assigned the tape drive to yourself.
–INIT	Tells LABEL that this tape is being written for the first time. This option must be used on unformatted tapes or on tapes whose labels should be overwritten.
–OVERWRITE	Tells LABEL to overwrite a BRMS tape. If you try to overwrite an ARCHIVE, BACKUP, or TRANSPORT BRMS tape, you must use the –OVERWRITE option.
–OWNER *owner*	Identifies the owner of the tape. *owner* is a string which contains, for ANSI labels, 1 to 14 characters; for IBM labels, 1 through 10 characters. If you specify a label that is shorter than the allowed maximum length, it is blank-padded on the right to the maximum length. If you omit *owner*, LABEL uses your login name as the default value. –OWN is a synonym for –OWNER.
–PARITY	Specifies **EVEN** or **ODD** parity for the label. Use this option only with the –TYPE B option.
–TYPE *type*	Specifies what sort of label you want written. The legal *types* are

Type	*Description*
ANSI87	ANSI X3.27-1987 standard label
BCD	IBM label for 7-track BCD tapes

EBCDIC	IBM label for 9-track EBCDIC tapes
PRIME	Prime ASCII label. This is the default. P, ANSI, and A are synonyms for label type PRIME.
STANDARD_1	ANSI X3.27-1978 standard label. S1 is a synonym for STANDARD_1.

–VOLUME ***volume-id*** The volume number that uniquely identifies this tape reel. *volume-id* must be in the range 1 through 6 characters long; if it has fewer than 6 characters, it is blank-padded on the right to make six characters. –VOL, –VOLSER, and –VOLID are synonyms for –VOLUME. If this option is not present, LABEL attempts to read an existing label from the tape; if this option is present, LABEL writes a new label to the tape.

Errors Using LABEL

Improper use of the LABEL command results in an error message. These errors are the result of bad syntax in the LABEL command itself or of a system magnetic tape I-O error. See the *Magnetic Tape User's Guide* for a complete list of these error messages.

The LABEL Help Facility

The command LABEL –HELP causes LABEL to display at the terminal an abbreviated description of the command.

For a complete description of tape labels and their use, refer to the IBM publication GC28-6680, *OS Tape Labels* and the ANSI publication X3.27-1978, *American National Standard Magnetic Tape Labels for Information Interchange.*

Format of Magnetic Tape Labels

Tape labels are tape records that identify and provide processing information about tape reels and the files they contain. COBOL85 writes tape labels in ANSI Standard 8-bit ASCII, otherwise referred to as Latin Alphabet No. 1 (ANSI X3.4-1977). COBOL85 writes them in the format specified in the ANSI Magtape Standard (ANSI X3.27-1978).

COBOL85 supports multiple files on one tape reel, as well as files that span multiple tape reels.

COBOL85 recognizes the following types of tape label records:

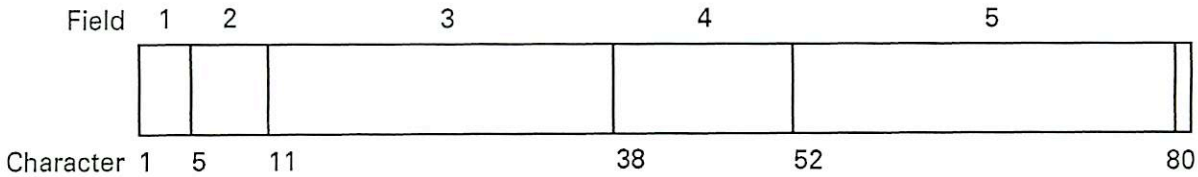
- **Volume 1 Label Records (VOL1)** identify the reel of tape and the owner of the reel. If the tape has any standard labels at all, it must have a VOL1 label as its first record.
- **Header 1 Label Records (HDR1)** identify and provide information about the tape file. Such information includes the file-identifier, volume serial number, file section number, file sequence number, generation number, generation version number, creation and expiration dates, and the access field. A labeled tape must include a single HDR1 label record before the file.
- **Header 2 Label Records (HDR2)** contain additional information about the tape file, such as the record format, block length, and record length. A single HDR2 label follows the HDR1 label record.

- **End-of-file and End-of-volume Label Records (EOF1/EOV1, EOF2/EOV2)** mark the end of a tape file or volume and include identical information to that found on the corresponding HDR1 and HDR2 label records. In addition, EOF1 and EOV1 contain the block count field.

The record formats and COBOL85 validation rules pertaining to these tape label records are presented below. For more detailed information, see the *Magnetic Tape User's Guide*.

Volume 1 Label Record

Figure 12-7 illustrates the format of the VOL1 label record.



Q101661-1LA-29-1

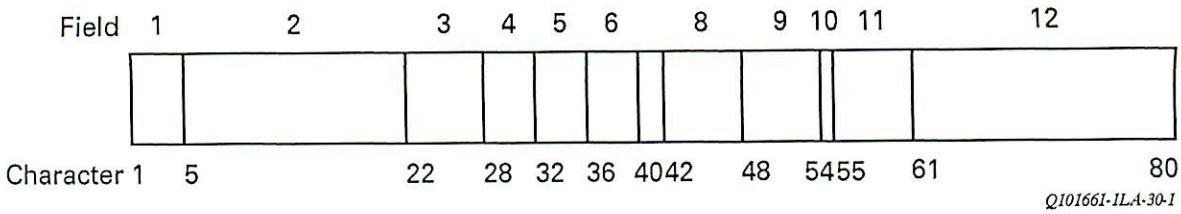
Field #	Description	# of chars.	Validated?
1	VOL1 label identifier	4	yes
2	Volume identifier number (volume serial number)	6	yes
3	Reserved for future use	27	no
4	Owner identifier	14	no
5	Reserved for future use	28	no
6	Label-standard version	1	no

FIGURE 12-7
Volume 1 Label Record Format

COBOL85 validates the VOL1 label identifier and the volume identifier number. In field 1, the VOL1 label identifier field must contain the value `VOL1`. The volume identifier, also referred to as the volume serial number, must contain the *volume-id*, as defined by the LABEL command.

Header 1 Label Record and EOF1/EOV1 Label Records

Figure 12-8 illustrates the format of the HDR1 label record as well as the End-of-File (EOF1) and End-of-Volume (EOV1) label records.



Field #	Description	# of chars.	Validated?
1	HDR1 (or EOF1 or EOV1) label identifier	4	yes
2	Tape file identifier	17	yes
3	File set-identifier number (volume serial number)	6	yes
4	File section number	4	yes
5	File sequence number	4	no
6	Generation number	4	no
7	Generation version number	2	no
8	Creation date	6	no
9	Expiration date	6	yes
10	Access	1	no
11	Block count (written for EOF1 and EOV1 only)	6	no
12	Reserved for future use	20	no

FIGURE 12-8
Header 1 Label Record and EOF1/EOV1 Label Records Format

COBOL85 validates

- The HDR1 label identifier
- The tape file identifier
- The volume serial number (file set-identifier number)
- The file section number
- The expiration date (multiple file tapes only)

The HDR1 label identifier must contain `HDR1`; the tape file identifier field must contain the tape *file-id*; the volume serial number (file set-identifier number) must be the same as the volume identifier number in the Volume 1 label record; and, in the case of a file that spans multiple tape reels, the file section number must indicate the tape's sequential position among the other tape reels of the file.

As indicated in the HDR1 table, several additional fields within the HDR1 label record are not currently validated by COBOL85. These fields are

- File sequence number
- Generation number
- Generation version number fields
- Creation date
- Access fields

However, when created by COBOL85, these fields contain the values specified by the ANSI Magtape Standard. No error or warning message is issued if any mismatch is found on these fields, and processing continues.

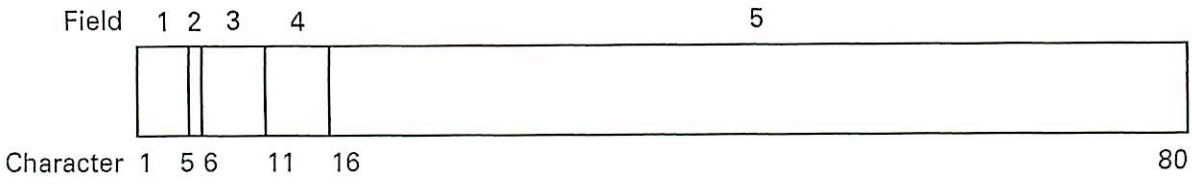
For the End-of-file 1 (EOF1) and End-of-volume 1 (EOV1) label records, the block count field is also written. The block count is the number of blocks that have been transferred to tape for a given magtape file since the beginning-of-file group label sequence.

Note

Although COBOL85 does not validate any of the EOF1 and EOV1 label fields, all label fields that are written for the HDR1 label are also written for the EOF1 and EOV1 labels.

Header 2 Label Record and EOF2/EOV2 Label Records

Figure 12-9 below illustrates the basic format of the HDR2 label record as well as the End-of-file (EOF2) and End-of-volume (EOV2) label records:



Q101661-1LA-31-1

Field #	Description	# of chars.	Validated?
1	HDR2 (or EOF2 or EOV2) label identifier	4	yes
2	Record format	1	yes
3	Block length	5	yes
4	Record length	5	yes
5	Reserved for future use	65	no

FIGURE 12-9
Header 2 Label Record and EOF2/EOV2 Label Records Format

The HDR2 label identifier must contain the value `HDR2`. The record format field must contain an `F` for fixed-length records or a `D` for variable-length records. Any other formats are not currently supported by the COBOL85 runtime libraries. The record length field must contain the maximum record length. The block length field must contain the block size. The block size is determined by multiplying the record length by the blocking factor.

Note

Although COBOL85 does not validate any of the EOF2 and EOV2 label fields, all label fields that are written for the HDR2 label are also written for the EOF2 and EOV2 labels, except for the label identifier.

Unlabeled Magnetic Tapes

Use caution when writing to or reading from unlabeled magnetic tape. COBOL85 cannot determine the record format (fixed or variable) of a tape for which no tape labels are present to identify the physical tape file attributes. Also, unpredictable results occur if a program attempts to read a fixed-length record tape file using a variable-length record description and vice versa.

For information on unlabeled magnetic tapes, see the *Magnetic Tape User's Guide*.

Compiling, Linking, and Executing Programs That Use Tape

Chapters 2 and 3 discuss compiling, linking, and executing programs. You need no special compiler options to compile a program that processes tape files. No special libraries or subroutines are required to link tape processing programs. However, before you execute a program that uses tape files, you must assign the tape drives. You may also need to assign tape filenames at runtime, if you have not already assigned them within the program.

Tape Drive Assignments at Execution Time

Assign tape drives with the following command at PRIMOS level:

```
ASSIGN  MTx [–ALIAS MTy]
AS
```

The value *x* must represent a physical tape drive from 0 through 7, and the value *y* must represent a logical tape drive from 0 through 7. The drivename in the file assignments presented below must correspond to the number after –ALIAS, if it is used, or else to the number after AS. See the *Magnetic Tape User's Guide*, and the example at the end of this chapter.

Note

The tape drive number in the file assignment and in the PRIMOS command ASSIGN must be the same. It is, however, independent of the *device-name* (MT9) in the SELECT statement.

Tape File Assignments Within the Program

Usually file assignments are handled within the COBOL85 program. Use a *literal* or a *data-name* in the VALUE OF FILE-ID clause to give the full tape assignment. If the clause contains a *data-name*, this field can then be given a value interactively with the ACCEPT statement. This technique is particularly helpful for drive numbers that are not known at compile time.

See the section VALUE OF FILE-ID, later in this chapter, for more information.

File Assignments With –FILE_ASSIGN

If you compiled your program with the –FILE_ASSIGN compiler option, execution includes a request for file assignments, in this format:

```
ENTER FILE ASSIGNMENTS:
>
```

Make one entry for each FD whose FILE-ID you wish to assign. Syntax errors are generated during file assignment for improper formats. When no file assignments remain to be entered, use a slash mark (/) to conclude the session.

Examples and format are given in the section VALUE OF FILE-ID, later in this chapter.

For additional information on the –FILE_ASSIGN option, see Chapter 2 and Appendix N.

Assignment Error Messages

The file assignment routine may output the following error messages.

BAD DELIMITER
No equal sign is found, or the equal sign is in an unexpected position.

ILLEGAL SPECIFICATION
The name to the right of the equal sign begins with $ but does not have the form $MT*x*.

LABEL SPECIFICATION EXPECTED
For magnetic tape, you must specify S or N.

MT*n* # OUT OF RANGE
The magnetic tape unit number must be in the range 0 through 7.

NAME BUFFER OVERFLOW
The buffer used to store the pathname of the file is full.

NAME REQUIRED
You must specify a name to the right of the equal sign.

NAME TOO LONG
The name to the right of the equal sign is greater than 17 characters for *tape-file-id* or greater than six characters for *volume-id*.

TAPE FILE-ID EXPECTED
No tape *file-id* was specified in a tape assignment.

VSN EXPECTED
The volume serial number (*volume-id*) is missing in a tape assignment.

ILLEGAL EXPIRATION DATE
The expiration date for a multiple file tape is invalid.

ILLEGAL EXPIRATION DATE FOR NON-LEAP YEAR
The expiration date for a multiple file tape is greater than 365 for a non-leap year.

ENVIRONMENT DIVISION

This section contains information that is unique to tape files. Chapter 6 contains information that applies to all file organizations. Chapters 10 and 11 contain information that applies to indexed files and relative files, respectively.

INPUT-OUTPUT SECTION — I-O-CONTROL

Use the INPUT-OUTPUT SECTION to specify peripheral devices and information needed to transmit and handle data between the devices and the program.

Use the MULTIPLE FILE TAPE clause in the I-O-CONTROL paragraph to specify the location of files on a multiple file reel.

Format

MULTIPLE FILE TAPE CONTAINS { *file-name-1* [POSITION *integer-1*]} · · ·

Syntax Rules

1. *integer-1* must be greater than zero.
2. *file-name-1* must be a sequential file assigned to MT9 that references a labeled tape.
3. *file-name-1* must not appear in more than one MULTIPLE FILE TAPE clause, nor more than once in the same MULTIPLE FILE TAPE clause.

General Rules

1. If a file on a multiple file tape is referenced in the program, you must specify the file in a MULTIPLE FILE TAPE clause. Optionally, you may specify in the appropriate MULTIPLE FILE TAPE clause files on multiple file tapes that are not referenced in the program.
2. A separate MULTIPLE FILE TAPE clause is required for each multiple file tape accessed by the program.
3. No file can appear in more than one MULTIPLE FILE TAPE clause, nor can a file appear more than once in the same MULTIPLE FILE TAPE clause.
4. POSITION refers to the sequential position of the file on the reel. The first file on a reel has position 1, the second position 2, and so on. The POSITION phrase associates a file with a specific position on the reel. The position specified in this phrase is significant when the file is opened for input. It is ignored when the file is opened for output.
5. Specify the MULTIPLE FILE TAPE clause only for labeled tape. Multiple file/multiple volume unlabeled tapes are not supported.
6. No more than one file on a tape reel can be opened at one time.
7. You can use the MULTIPLE FILE TAPE clause in conjunction with the VALUE OF FILE-ID, CLOSE WITH NO REWIND and OPEN WITH NO REWIND clauses. For more information on their interaction, see the section Multiple File Tapes, earlier in this chapter.

DATA DIVISION

This section contains information that pertains to tape files. Chapter 7 contains information that applies to all file organizations. Chapters 9, 10, and 11 contain information that pertains to sequential disk files, indexed files, and relative files, respectively.

BLOCK CONTAINS

Specifies the size of a physical record.

Format

BLOCK CONTAINS [*integer-1* TO] *integer-2* { RECORDS / CHARACTERS }

Syntax Rules

1. The BLOCK CONTAINS clause is optional.
2. Use the clause only with tape files. PRIMOS disk files do not require explicit blocking for efficient access.

General Rules

1. For an input file, the BLOCK CONTAINS clause must describe the blocking of the records when they were created.
2. The clause may be omitted if the physical record (block) contains one, and only one, complete logical record.
3. If this clause is omitted, records are treated as unblocked.
4. When the RECORDS option is used, the compiler assumes that the block size provides for *integer-2* records of the maximum size shown for the file and then provides additional space for any required control words.
5. When the word CHARACTERS is used, the physical record size is specified in terms of the number of character positions required to store the physical record, regardless of the types of characters used to represent the items within the physical record.
6. When neither the CHARACTERS nor the RECORDS option is specified, CHARACTERS is the default.
7. When both *integer-1* and *integer-2* are used, *integer-1* is for documentation purposes only.
8. The maximum size of a block is listed in Appendix I.

CODE-SET

Specifies the character code set used to represent data on the tape.

Format

CODE-SET IS *alphabet-name*

Syntax Rules

1. When you specify the CODE-SET clause for a file, you must describe all data in the file as USAGE IS DISPLAY and any signed numeric data with the SIGN IS SEPARATE clause.

2. The *alphabet-name* is a programmer-defined name that you specify in the ALPHABET clause of the SPECIAL-NAMES paragraph.
3. Specify the CODE-SET clause only for tape files.

General Rule

You can specify the *alphabet-name* of the CODE-SET clause as STANDARD-1, STANDARD-2, EBCDIC, or NATIVE.

LABEL RECORDS

The LABEL RECORDS clause specifies whether labels exist for the file.

Format

$$\underline{\text{LABEL}} \left\{ \begin{matrix} \underline{\text{RECORD}} \text{ IS} \\ \underline{\text{RECORDS}} \text{ ARE} \end{matrix} \right\} \left\{ \begin{matrix} \underline{\text{STANDARD}} \\ \underline{\text{OMITTED}} \end{matrix} \right\}$$

General Rules

1. If you do not specify the LABEL RECORDS clause, LABEL RECORDS OMITTED is the default.
2. OMITTED specifies that no explicit labels exist for the file or device to which the file is assigned.
3. STANDARD specifies that a label exists for the file and that the label conforms to system specifications.
4. Tape files can be labeled or unlabeled. However, the following restrictions apply to unlabeled tapes:

   - Multiple file tape reels cannot be created or accessed
   - Multivolume tape reels cannot be created or accessed
   - File attribute checking is not performed
   - A CLOSE statement does not rewind the file
   - OPTIONAL files are not supported

VALUE OF FILE-ID

Associates the internal filename with an external file, thus allowing for the linkage of internal and external filenames.

Format

VALUE OF FILE-ID IS $\left\{\begin{matrix} data\text{-}name\text{-}3 \\ literal\text{-}2 \end{matrix}\right\}$

General Rules

1. The *literal* is an alphanumeric literal that may not exceed 128 characters.
2. The *data-name* must be in the WORKING-STORAGE SECTION. It may be qualified, but it must not be subscripted, indexed, or described with USAGE IS INDEX. The value of the *data-name* must not exceed 128 characters.
3. You can override the VALUE OF FILE-ID clause at runtime, if you use the –FILE_ASSIGN compiler option. For information on the –FILE_ASSIGN option, see Chapter 2 and Appendix N.
4. Tape filenames in *data-name-3* and *literal-2* must have one of the following formats:

   - For unlabeled tapes,

     drivename, label-type

   - For multiple file tapes,

     drivename, label-type, tape-file-id, volume-id, owner-id, [expire-date]

   - For single file tapes,

     drivename, label-type, tape-file-id, volume-id

   The parameters in these formats contain the following information:

Parameter	*Contents*
drivename	$MT(*x*), where *x* is a drive number from 0 through 7.
label-type	N: Unlabeled tape S: ANSI standard labels
tape-file-id	A 1-character to 17-character field containing the *tape-file-id*. Use this parameter to distinguish files on a multiple file tape.
volume-id	A 1-character to 6-character field containing the *volume-id* number. (Different *volume-ids* can appear on files on the same tape, implying support for multiple volume tapes.)
owner-id	A 1-character to 14-character field containing the *owner-id*.
expire-date	A 5-character optional field specifying the expiration date of the file being referenced. Use this parameter to write files with expiration dates. Because *expire-date* is ignored on file input, you can omit it for tape files opened only for input. The expiration date must be in the format *yyddd*, where *yy* is the year and *ddd* is the Julian day of the year. The default is the current date (always expired).

Excess parameters are ignored at runtime. For more information on the interaction of these parameters and the MULTIPLE FILE TAPE clause, see the section Multiple File Tapes, earlier in this chapter.

5. If you do not specify the VALUE OF FILE-ID clause for a file assigned to MT9, COBOL85 supplies a default *file-id* as follows:

   - For single file tapes, if you specify the LABEL RECORDS ARE STANDARD clause, the compiler uses the FD entry for the *tape-file-id* and a value of 1 for the *volume-id*. In this case, the default file assignment is $MT0, S, FILENAME, 000001.
   - For single file tapes, if you specify the LABEL RECORDS ARE OMITTED clause (either explicitly or implicitly), the default file assignment is $MT0, N.
   - For multiple file tapes, the compiler uses the FD entry for *tape-file-id*, a value of 1 for the *volume-id*, a value of DEFAULT for the *owner-id*, and no expiration date. In this case, the default file assignment is $MT0, S, FILENAME, 000001, DEFAULT.

6. The implementor-name FILE-ID can be used as a programmer-defined word elsewhere in the program.

Examples

A tape file named FILEX can be associated with a logical COBOL85 file named TEST-FILE in any of the following ways.

1. Value is *literal*:

```
FD TEST-FILE
    LABEL RECORDS STANDARD
    VALUE OF FILE-ID '$MT0, S, FILEX, VOL1'.
```

2. Value is *data-name*:

```
FD TEST-FILE
    LABEL RECORDS STANDARD
    VALUE OF FILE-ID IS TFILE-NAME.
     .
     .
WORKING-STORAGE SECTION.
77 TFILE-NAME              PIC X(20).
```

An actual filename can be associated with the logical *file-name* TEST-FILE by executing COBOL85 statements. For example,

```
IF NEW-FILE = 1
             MOVE '$MT0, S, FILEX, VOL1' TO TFILE-NAME,
ELSE
             MOVE '$MT1, S, FILEY, VOL1' TO TFILE-NAME.
```

Another way to do it is

```
MOVE SPACES TO TFILE-NAME
DISPLAY "ENTER TEST-FILE NAME."
ACCEPT TFILE-NAME.
```

Then, when the request ENTER TEST-FILE NAME is displayed, enter a name such as $MT0, S, FILEX, VOL1.

3. With –FILE_ASSIGN, you enter a file assignment at execution time. For example, suppose that in a COBOL85 program the following statements exist:

```
FD  TEST-FILE
    LABEL RECORDS ARE STANDARD,
    VALUE OF FILE-ID IS 'FILE1'.
```

Then, an appropriate runtime dialog is

```
ENTER FILE ASSIGNMENTS:
>FILE1 = $MT0, S, FILEX, VOL1
>/
```

For additional information on the –FILE_ASSIGN option, see Chapter 2 and Appendix N.

If labels are not needed, you can enter a literal value, such as the following:

```
VALUE OF FILE-ID IS '$MT0, N'
```

PROCEDURE DIVISION

This section contains information that pertains to tape files. Chapter 8 contains information that applies to all file organizations. Chapters 9, 10, and 11 contain information that pertains to sequential disk files, indexed files, and relative files, respectively.

Tape files may be processed only with sequential I-O. Therefore, DELETE, START, REWRITE, and any clauses that are appropriate only for indexed or relative files cannot be used with tape files.

CLOSE

Terminates the processing of files.

Format

Syntax Rule

The files referenced in the CLOSE statement need not all have the same organization or access.

General Rules

1. A CLOSE statement implies a preceding OPEN on the same file.
2. If a CLOSE statement is executed for a file, no other statement can be executed that references that file, unless an intervening OPEN statement for that file is executed.
3. Following the successful execution of a CLOSE statement the record area associated with *file-name* is no longer available. The unsuccessful execution of such a CLOSE statement leaves the availability of the record area undefined.
4. The optional WITH NO REWIND phrase specifies that the file reel not be rewound upon execution of the CLOSE statement.

   The WITH NO REWIND phrase is ignored at compile time if it does not apply to a tape file. However, a status code of 07 is returned at runtime if the file referenced in the CLOSE WITH NO REWIND statement is not a tape file.

   If you do not specify the WITH NO REWIND phrase for a tape file, the execution of the CLOSE statement automatically positions the file to the beginning of the tape reel, except for unlabeled tapes. Unlabeled tapes always default to WITH NO REWIND.

   If you specify the WITH NO REWIND phrase for a tape file, the execution of the CLOSE statement suppresses the automatic positioning of the file to the beginning of the tape reel.

   For more information on the use of this phrase in conjunction with multiple file tape processing, see the section Multiple File Tapes, earlier in this chapter.
5. If REEL or UNIT is specified, the CLOSE statement is ignored and the file status data item, if specified, is set to indicate status code 07. This rule applies to tape and non-tape media. REEL and UNIT have no meaning under Prime's tape I-O system.

OPEN

Initiates the processing of files and performs checking and writing of labels.

Format

```
OPEN { INPUT  {file-name-1 [WITH NO REWIND]} ... } ...
     { OUTPUT {file-name-2 [WITH NO REWIND]} ... }
```

Syntax Rule

The files referenced in the OPEN statement need not all have the same organization or access.

General Rules

1. The successful execution of an OPEN statement determines the availability of the file, puts the file in open mode, and makes the associated record area available to the program. Prior to the successful execution of an OPEN statement for a given file, no statement that references that file can be executed.
2. A file may be opened with the INPUT and OUTPUT phrases in the same program. Following the initial execution of an OPEN statement for a file, each subsequent OPEN statement for that same file must be preceded by the execution of a CLOSE statement for that file.
3. Execution of the OPEN statement does not obtain or release the first data record.
4. If label records are specified for the file, the beginning labels are processed as follows.

   - When you specify the INPUT phrase, the execution of the OPEN statement causes the labels to be checked against the file assignments.
   - When you specify the OUTPUT phrase, the execution of the OPEN statement causes the labels to be written as specified in the file assignments.

5. The *file-description-entry* for *file-name-1*, *file-name-2*, *file-name-3*, or *file-name-4* must be equivalent to the entry used when the file was created, including blocking of records.
6. For files being opened with the INPUT phrase, the OPEN statement sets the current record pointer to the first record currently existing within the file. If no records exist in the file, the current record pointer is set so that the next executed READ statement for the file results in an AT END condition.
7. Upon successful execution of an OPEN statement with the OUTPUT phrase specified, a file is created. At that time the associated file contains no data records.
8. The optional WITH NO REWIND phrase specifies that the file reel not be rewound upon file processing initiation.

   The WITH NO REWIND phrase is ignored at compile time if it does not apply to a tape file. However, a status code of 07 is returned at runtime if the file referenced in the OPEN WITH NO REWIND statement is not a tape file.

   If the WITH NO REWIND phrase is not specified for a tape file, the execution of the OPEN statement causes the file to be positioned at the beginning of the tape reel.

   If the WITH NO REWIND phrase is specified for a tape file, the execution of the OPEN statement does not cause the file to be repositioned.

   For more information on the use of this phrase in conjunction with multiple file tape processing, see the section Multiple File Tapes, earlier in this chapter.

READ

Releases a record from the tape buffer to the program.

Format

READ *file-name* RECORD [INTO *data-name-1*]

[AT END *imperative-statement-1*]

[NOT AT END *imperative-statement-2*]

[END-READ]

The results of a READ from tape are the same as those of a READ from disk. However, you must know how the records were blocked when they were written to tape, and use the BLOCK CONTAINS clause, if necessary, to specify the same blocking factor.

The READ statement appears to read one record at a time from the tape file. However, a whole block is read from tape at once and each subsequent READ statement releases one record to the program.

Include a USE procedure within a declaratives section to handle any I-O errors encountered during a READ operation. A file-status of 98 may be returned to indicate a hardware I-O failure that may be recoverable. If a USE procedure is not included and an I-O operation is unsuccessful, the results are undefined and cause the program to terminate. A permanent error condition may arise for any error encountered that is not recoverable. If so, a file-status of 30 is returned.

WRITE

Releases a record to the tape buffer.

Format

WRITE *record-name* [FROM *data-name-1*]

[END-WRITE]

The results of a WRITE to tape appear to be the same as those of a WRITE to disk. However, a WRITE statement actually releases a record to the tape buffer until a block is filled. The whole block is then written to the tape.

Include a USE procedure within a declaratives section to handle any I-O errors encountered during a WRITE operation. A file-status of 98 may be returned to indicate a hardware I-O failure that may be recoverable. If a USE procedure is not included and an I-O operation is unsuccessful, the results are undefined and cause the program to terminate. A permanent error condition may arise for any error encountered that is not recoverable. If so, a file-status of 30 is returned.

Magnetic Tape Error Reporting

Depending on the operation in progress at the time of error detection, magnetic tape error diagnostics are available for the following types of operations:

- General
- General OPEN operations
- OPEN OUTPUT operations
- OPEN INPUT operations
- WRITE operations
- READ operations
- CLOSE operations

Magnetic tape error diagnostics of each of these types are listed below. Except where otherwise noted, fatal errors generate a status code 30. For all fatal errors, COBOL85 executes any USE procedure that you specify, and terminates the program.

General Tape Error Diagnostics

```
TAPE DRIVE NOT ASSIGNED
WHEN THE TAPE DRIVE HAS BEEN ASSIGNED, TYPE "S" TO CONTINUE
(Recoverable error)
```

The desired tape drive has not been assigned by the AS MT*n* command (where *n* is the tape drive number).

The COBOL85 library enters a PRIMOS command level interlude during which you can assign the desired magtape device. Subsequently, type S to resume the aborted process.

```
TAPE DRIVE NOT READY OR OFFLINE (Recoverable error)
```

A tape must be mounted and loaded on the desired tape drive; the ONLINE button on the tape drive must be pushed.

Check the physical status of the specified tape drive and ensure that the tape drive is online and ready.

```
UNRECOVERABLE TAPE CONTROLLER ERROR ENCOUNTERED
(Fatal error)
```

An invalid internal operation was attempted on the tape controller in use. This error is most likely to occur when using cartridge tape controllers, which have limited functionality in terms of tape movement and positioning.

```
TYPE 'A' TO ABORT, ELSE CORRECT THE PROBLEM AND TYPE RETURN
TO CONTINUE
```

This message accompanies recoverable errors, such as write-protected tape, tape drive not ready or offline, or label mismatch errors. When this message appears, take one of the following actions:

- If the problem can be corrected without aborting, correct the problem (for example, put the tape drive online) and press the Return key. Processing resumes from the point at which it was interrupted.

- Suspend or abort the process by typing A. The system enters a PRIMOS command level interlude, during which you may correct the problem by issuing PRIMOS commands. For example, you may invoke LABEL to format the tape if the VOL1 label record was missing for a program that specified labeled tape, or you may abort the process and continue with other activities. If you have corrected the problem, type S to restart the process from the point at which it was interrupted. This action returns process control to the COBOL85 library.
- Type Q to abort with no restart capability.

Tape Error Diagnostics for General OPEN Operations

`OPEN MAGTAPE ERROR - FILE IN USE (Fatal error)`
The file specified is currently in use and cannot be opened by the tape unit.

`OPEN OPERATION WAS NOT SUCCESSFUL (Fatal error)`
The specified file could not be opened successfully. This error is of an undetermined nature. The problem may be caused by internal vector or pointer setup problems, inability to successfully forward or back space on tape, inability to find file marks on tape, or the presence of badspots on tape.

`PROGRAM SPECIFIED MAXIMUM TOTAL PHYSICAL BLOCK SIZE EXCEEDS SYSTEM MAXIMUM OF 12288 CHARACTERS (Fatal error)`
The maximum block size specified in the user application exceeds the currently supported system maximum of 12288 (12K) characters. Check the computed maximum block size (maximum record size multiplied by block size) in the program.

`MAXIMUM RECORD SIZE SPECIFIED FOR VARIABLE LENGTH RECORD FILE EXCEEDS 9995 CHARACTERS (Fatal error)`
The maximum record size specified in the user application exceeds the ANSI specified maximum of 9995 characters.

Tape Error Diagnostics for OPEN OUTPUT Operations

`ERROR WRITING HDR1 LABEL RECORD (Fatal error)`
An error occurred while attempting to write the HDR1 label record.

`ERROR WRITING HDR2 LABEL RECORD (Fatal error)`
An error occurred while attempting to write the HDR2 label record.

`VOL1 LABEL RECORD MISSING (Recoverable error)`
A VOL1 label record was expected but not found.

Use the LABEL command to initialize the tape prior to writing to it.

`ERROR READING VOL1 LABEL RECORD (Recoverable error)`
An error occurred while the VOL1 label record was being read.

Use the LABEL command to reinitialize the tape.

`VOLUME-ID MISMATCH ON VOL1 LABEL RECORD (Recoverable error)`
The *volume-id* specified within the user application is not the same as the *volume-id* on the tape.

Check that the *volume-id* specified in the program is the same as the *volume-id* generated by the LABEL command.

Tape Error Diagnostics for OPEN INPUT Operations

VOL1 LABEL RECORD MISSING (Recoverable error)
A VOL1 label record was expected but was not found.

Check that the correct labeled tape has been mounted.

ERROR READING VOL1 LABEL RECORD (Recoverable error)
An error occurred while the VOL1 label was being read.

The system could not successfully read the VOL1 label record. Either the system had difficulty in correctly positioning and reading the tape, or the expected tape reel was not mounted.

VOLUME-ID MISMATCH ON VOL1 LABEL RECORD (Recoverable error)
The *volume-id* on the tape is not the same as the *volume-id* that was supplied by the user.

Check the *volume-ids* specified on the tape and in the program to ensure that the correct volume has been mounted and specified by the application program.

ERROR READING HDR1 LABEL RECORD (Recoverable error)
An error occurred while the HDR1 label record was being read.

The system could not successfully read the HDR1 label record. Either the system had difficulty in correctly positioning and reading the tape, or the expected tape reel was not mounted. In most cases consider this error to be fatal.

ERROR READING HDR2 LABEL RECORD (Recoverable error)
An error occurred while the HDR2 label record was being read.

The system could not successfully read the HDR2 label record. Either the system had difficulty in correctly positioning and reading the tape, or the expected tape reel was not mounted. In most cases consider this error to be fatal.

HDR1 LABEL RECORD MISSING (Recoverable error)
An HDR1 label record was expected but was not found.

Check that the correct tape volume has been mounted. In most cases consider this error to be fatal.

VOLUME SERIAL NUMBERS MISMATCH (Recoverable error)
The volume serial number (file-set identifier) on the HDR1 tape label is not the same as the *volume-id.*

Check that the volume serial number on the tape coincides with the *volume-id* specified in the program.

FILE SECTION-ID NUMBERS DO NOT MATCH (Recoverable error)
The file *section-id* number (also referred to as the file section number) on the tape is not the same as the file *section-id* number supplied by the user.

Check that the correct tape volume has been mounted.

HDR2 LABEL RECORD MISSING (Recoverable error)
An HDR2 label record was expected but was not found.

Check that the correct tape volume has been mounted. In most cases consider this error to be fatal.

TAPE FILE-IDS DO NOT MATCH (Fatal error)
The tape *file-id* field on the tape is not the same as the tape *file-id* field supplied by the user.

After invoking any DECLARATIVES for status code 35, the program terminates.

INVALID RECORD FORMAT FOUND ON HDR2 LABEL RECORD
ONLY FIXED AND VARIABLE FORMATS ARE SUPPORTED
(Fatal error)

An invalid tape format (such as spanned) has been found on the tape being read. No formats other than F (fixed) or D (variable) are currently supported by COBOL85. This error is a file attribute error; COBOL85 returns status code 39.

INVALID RECORD FORMAT - VARIABLE SPECIFIED/FIXED FOUND
(Fatal error)

The user specified variable-length records in the program, but the record format on the HDR2 label record indicates fixed-length records exist on the tape file. This error is a file attribute error; COBOL85 returns status code 39.

INVALID RECORD FORMAT - FIXED SPECIFIED/VARIABLE FOUND
(Fatal error)

The user specified fixed-length records in the program, but the record format on the HDR2 label record indicates variable-length records exist on the tape file. This error is a file attribute error; COBOL85 returns status code 39.

THE SIZE SPECIFIED DOES NOT MATCH THE HDR2 LABEL RECORD

The block size and/or record size specified in the program is larger than the size specified when the file was created. This error is a file attribute error; COBOL85 returns status code 39.

Tape Error Diagnostics for WRITE Operations

END OF TAPE - MULTIPLE-VOLUME UNLABELED TAPE IS UNSUPPORTED
(Fatal error)

The physical end of tape has been detected for an unlabeled tape. Multiple reel tape is not allowed for unlabeled tape; therefore, the process is terminated. This error generates status code 82.

END OF VOLUME - MOUNT NEXT VOLUME
WHEN A NEW VOLUME HAS BEEN MOUNTED, HIT RETURN TO CONTINUE,
OTHERWISE, TYPE "A" TO ABORT PROCESSING
(Recoverable error)

The physical end of tape has been detected for labeled tape. This message appears on the terminal to alert the operator to mount the next volume. When a new volume has been mounted, press the Return key to continue processing from the point at which it was interrupted. Otherwise, invoke the abort option at this point.

UNRECOVERABLE MAGTAPE WRITE ERROR (Fatal error)

The system was unable to successfully write to tape. The problem is of an undetermined nature. The problem could be caused by the tape drive's inability to position itself correctly to write the record or block, or by the presence of badspots on the tape.

This error is not fatal if a file status of 30 is returned through the file status word and a USE procedure within a declaratives section is present in the program.

TAPE DRIVE IS WRITE PROTECTED - WRITE ACCESS REQUIRED
(Recoverable error)

A write-enable-ring has not been inserted on the tape reel. Therefore, the tape is write-protected.

To write to the specified tape, insert a write-enable-ring before processing continues. Otherwise, modify the program to access the file for INPUT only.

TOTAL BLOCK SIZE OF BLOCK CURRENTLY BEING WRITTEN
EXCEEDS SYSTEM MAXIMUM OF 12288 CHARACTERS
(Fatal error)

The maximum block size currently being written to tape exceeds the presently supported system maximum of 12288 (12K) characters. Check the computed maximum block size (maximum record size multiplied by block size) in the program.

MAXIMUM RECORD SIZE SPECIFIED FOR VARIABLE
LENGTH RECORD FILE EXCEEDS 9995 CHARACTERS
(Fatal error)

The maximum record size specified in the user application exceeds the ANSI specified maximum of 9995 characters.

FILE IS NOT OPEN FOR OUTPUT (Fatal error)

The file specified is not currently open for writing.

Tape Error Diagnostics for READ Operations

Unexpected end of file (No message)

The end of the file was reached unexpectedly. A file status of 10 is returned in the file status word, and the AT END path is taken.

END OF TAPE - MULTIPLE-VOLUME UNLABELED TAPE IS UNSUPPORTED
(Fatal error)

The physical end of the tape has been detected for an unlabeled tape. Multiple reel tape is not allowed for unlabeled tape; therefore, the process is terminated. This error generates status code 82.

END OF VOLUME - MOUNT NEXT VOLUME
WHEN A NEW VOLUME HAS BEEN MOUNTED, HIT RETURN TO CONTINUE,
OTHERWISE, TYPE "A" TO ABORT PROCESSING
(Recoverable error)

The physical end of tape has been detected for labeled tape. This message appears on the terminal to alert the operator to mount the next volume. When a new volume has been mounted, press the Return key to continue processing from the point at which it was interrupted. Otherwise, invoke the abort option at this point.

UNRECOVERABLE MAGTAPE READ ERROR (Fatal error)

The system was unable to successfully read the tape. The problem is of an undetermined nature. The problem could be caused by the tape drive's inability to position itself correctly to read the record or block, or by the presence of badspots on the tape.

This error is not fatal if a file status of 98 is returned through the file status word and a USE procedure within a declaratives section is present in the program. A retry may be possible.

TOTAL BLOCK SIZE OF BLOCK CURRENTLY BEING READ
EXCEEDS SYSTEM MAXIMUM OF 12288 CHARACTERS
(Fatal error)

The maximum block size currently being read from the tape exceeds the presently supported system maximum of 12288 (12K) characters. Check the computed maximum block size (maximum record size multiplied by blocking factor) in the program. This error is a file attribute error; COBOL85 returns status code 39.

FILE IS NOT OPEN FOR INPUT (Fatal error)

The file specified is not currently open for reading.

Tape Error Diagnostics for CLOSE Operations

CLOSE OPERATION WAS UNSUCCESSFUL (Fatal error)
The system was unable to successfully close the tape file. The problem is of an undetermined nature. The problem could be caused by the tape drive's inability to position itself correctly to write or read the EOF or EOV labels, or by the presence of badspots on the tape.

ERROR WRITING EOF1 LABEL RECORD (Fatal error)
An error occurred while attempting to write the EOF1 label record.

ERROR WRITING EOV1 LABEL RECORD (Fatal error)
An error occurred while attempting to write the EOV1 label record.

ERROR WRITING EOF2 LABEL RECORD (Fatal error)
An error occurred while attempting to write the EOF2 label record.

ERROR WRITING EOV2 LABEL RECORD (Fatal error)
An error occurred while attempting to write the EOV2 label record.

ERROR WRITING TRAILER FILEMARKS (Fatal error)
An error occurred while attempting to write the trailer filemarks.

Example

The following example runs with normal I-O. This program represents the tape portion of the program OLDCASH given at the end of Chapters 5, 6, 7, and 8.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.      DISBURSE.
 AUTHOR.          ANNE
 INSTALLATION.    PRIME.
 DATE-WRITTEN.    SEPTEMBER 20, 1987.
 DATE-COMPILED. 870612.15:29:20.
 REMARKS.  THIS PART OF THE PROGRAM WRITES TOTALS
     BY DEPARTMENT TO TAPE.
*
     TO WRITE TAPE RECORDS, ENTER YES FOR TAPE REQUEST.
******************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  PRIME.
 OBJECT-COMPUTER. PRIME.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT TAPE-FILE, ASSIGN TO MT9.
******************************************************
 DATA DIVISION.
*
 FILE SECTION.
*
```

```
 FD  TAPE-FILE,
     LABEL RECORD IS STANDARD,
     BLOCK CONTAINS 4 RECORDS,
     VALUE OF FILE-ID IS TAPENAME,
     DATA RECORD IS TAPE-LINE.
*
 01  TAPE-LINE            PIC X(20).
*
 WORKING-STORAGE SECTION.
 77  TAPE-CHOICE      PIC XXX VALUE 'NO '.
 77  TAPENAME         PIC X(20)
              VALUE '$MT0, S, EVELYN, T1'.
 77  TOTAL1           PIC S9(9)V99    COMP-3 VALUE ZERO.
 77  TOTAL2           PIC S9(9)V99    COMP-3 VALUE ZERO.
 77  TOTAL3           PIC S9(9)V99    COMP-3 VALUE ZERO.
 77  TOTAL4           PIC S9(9)V99    COMP-3 VALUE ZERO.
 77  TOTAL5           PIC S9(9)V99    COMP-3 VALUE ZERO.
 77  TOTAL6           PIC S9(9)V99    COMP-3 VALUE ZERO.
 77  VARIABLE-MONTH   PIC X(15) VALUE 'THIS MONTH     '.
*********************************************************
*       TAPE OUTPUT
*********************************************************
 01  TAPE-HEADER.
     05 TAPE-MONTH         PIC X(15)       VALUE SPACES.
     05 FILLER             PIC X(5)        VALUE SPACES.
 01  SAVE-TAPE.
     05 SAVE-DATE-TAPE     PIC 9(6).
     05 SAVE-ACCT-TAPE     PIC XXX.
     05 SAVE-TOTAL-TAPE    PIC S9(9)V99     COMP-3.
*********************************************************
 PROCEDURE DIVISION.
*
 DECLARATIVES.
 TAPE-ERROR SECTION. USE AFTER ERROR PROCEDURE ON TAPE-FILE.
 FIRST-PARAGRAPH.
     DISPLAY '**** I-0 ERROR ON TAPE OUTPUT ***'.
 END DECLARATIVES.
*
 001-BEGIN.
     READY TRACE.
     PERFORM 010-GET-JOBINFO.
     PERFORM 030-PROCESS-DETAIL.
     PERFORM 050-DEPT-TOTALS.
     PERFORM 090-PROCESS-TAPE.
     DISPLAY '     END OF RUN'.
     STOP RUN.
*
 010-GET-JOBINFO.
*NOT INCLUDED.
*
 030-PROCESS-DETAIL.
```

```
*NOT INCLUDED
*
 050-DEPT-TOTALS.
*********************************************************
*  MOVE ARBITRARY NUMBERS TO TOTAL1, TOTAL2, ETC.
*********************************************************
     MOVE 11111111 TO TOTAL1.
     MOVE 22222222 TO TOTAL2.
     MOVE 33333333 TO TOTAL3.
     MOVE 44444444 TO TOTAL4.
     MOVE 55555555 TO TOTAL5.
     MOVE 66666666 TO TOTAL6.
*
 090-PROCESS-TAPE.
     DISPLAY 'IS TAPE OUTPUT DESIRED--ENTER YES OR NO'.
     ACCEPT TAPE-CHOICE.
     IF TAPE-CHOICE = 'yes' OR
        TAPE-CHOICE = 'YES' PERFORM 095-WRITE-TAPE THRU
                  097-VERIFY-TAPE,
     ELSE DISPLAY 'NO TAPE'.
*
 095-WRITE-TAPE.
     OPEN OUTPUT TAPE-FILE.
     MOVE VARIABLE-MONTH TO TAPE-MONTH.
     WRITE TAPE-LINE FROM TAPE-HEADER.
     ACCEPT SAVE-DATE-TAPE FROM DATE.
     MOVE '100' TO SAVE-ACCT-TAPE.
     MOVE TOTAL1 TO SAVE-TOTAL-TAPE.
     WRITE TAPE-LINE FROM SAVE-TAPE.
     MOVE '200' TO SAVE-TOTAL-TAPE.
     MOVE TOTAL2 TO SAVE-TOTAL-TAPE.
     WRITE TAPE-LINE FROM SAVE-TAPE.
     MOVE '410' TO SAVE-ACCT-TAPE.
     MOVE TOTAL3 TO SAVE-TOTAL-TAPE.
     WRITE TAPE-LINE FROM SAVE-TAPE.
     MOVE '420' TO SAVE-ACCT-TAPE.
     MOVE TOTAL4 TO SAVE-TOTAL-TAPE.
     WRITE TAPE-LINE FROM SAVE-TAPE.
     MOVE '430' TO SAVE-ACCT-TAPE.
     MOVE TOTAL5 TO SAVE-TOTAL-TAPE.
     WRITE TAPE-LINE FROM SAVE-TAPE.
     MOVE '440' TO SAVE-ACCT-TAPE.
     MOVE TOTAL6 TO SAVE-TOTAL-TAPE.
     WRITE TAPE-LINE FROM SAVE-TAPE.
     CLOSE TAPE-FILE.
*
 097-VERIFY-TAPE.
     DISPLAY 'FIRST TAPE RECORD - VERIFICATION ONLY'.
     OPEN INPUT TAPE-FILE.
     MOVE SPACES TO TAPE-HEADER, SAVE-TAPE.
     READ TAPE-FILE INTO TAPE-HEADER.
```

```
           READ TAPE-FILE INTO SAVE-TAPE.
           EXHIBIT SAVE-TOTAL-TAPE.
           EXHIBIT SAVE-TAPE.
           CLOSE TAPE-FILE.
           EXIT.
*
```

The following dialog compiles, loads, and executes the program, stored as TAPECASH.COBOL85. When the tape record SAVE-TAPE is displayed for verification, the part represented by SAVE-TOTAL-TAPE is not displayed because it is declared as COMP-3.

```
OK, COBOL85 TAPECASH -LISTING
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]
OK, ASSIGN MT0
Device MT0 assigned.
OK, LABEL MT0 -TYPE A -VOLID T1 -OWNER EVELYN -INIT
OK, BIND
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
: LO TAPECASH
: LI COBOL85LIB
: LI
BIND COMPLETE
FILE
OK, RESUME TAPECASH
trace:  010-GET-JOBINFO
trace:  030-PROCESS-DETAIL
trace:  050-DEPT-TOTALS
trace:  090-PROCESS-TAPE
IS TAPE OUTPUT DESIRED--ENTER YES OR NO
YES
trace:  095-WRITE-TAPE
trace:  097-VERIFY-TAPE
FIRST TAPE RECORD - VERIFICATION ONLY
SAVE-TOTAL-TAPE =    11111111.00
SAVE-TAPE = 820611100
     END OF RUN
OK,
```

13 Interprogram Communication

Interprogram communication is the transfer of control and data from one program to another within a runfile.

The **calling program**, which must contain a CALL statement, transfers control to the **called program**. If you also wish to transfer data, the called program must contain a LINKAGE SECTION that describes the data, and a USING clause in its PROCEDURE DIVISION header. The called program can also contain an EXIT PROGRAM or GOBACK statement. If you do not include a GOBACK or EXIT PROGRAM statement, control returns to the calling program after execution of the last statement in the called program.

This chapter discusses the LINKAGE SECTION clauses and PROCEDURE DIVISION verbs that control interprogram communication. It also explains how to link and execute runfiles that contain more than one program. Finally, the chapter emphasizes the need for data type compatibilty when a COBOL85 program calls a program written in another Prime high-level language.

LINKAGE SECTION

The LINKAGE SECTION describes data, defined in a calling program, that is available to a called program.

Format

LINKAGE SECTION.

[*level-77-description-entry*]
[*record-description-entry*] ...

Syntax Rules

1. The LINKAGE SECTION in a program is meaningful if, and only if, the program is to function under the control of a CALL statement, and the CALL statement in the calling program contains a USING phrase.

2. The LINKAGE SECTION describes data made available in memory from another program module, but which is to be referred to in both the calling and the called programs.
3. COBOL85 does not allocate space in a program for data items described in its LINKAGE SECTION. PROCEDURE DIVISION references to such items are resolved at load time, by equating the references in the called program to the locations used in the calling program.
4. Data items defined in the LINKAGE SECTION of the called program can be referred to in the PROCEDURE DIVISION of that program only if

   - They are specified as operands of the USING phrase of the PROCEDURE DIVISION header or are subordinate to such operands.
   - The called program is under the control of a CALL statement with a USING phrase. (See the example at the end of this chapter.)

5. You can use any *record-description* clause in Chapter 7 to describe items in the LINKAGE SECTION, with the following exceptions:

   - Specify the VALUE clause only for level-88 items.
   - Do not specify the EXTERNAL clause for items defined in the LINKAGE SECTION.
   - Each *record-name* and *level-77* name in the LINKAGE SECTION must be unique (cannot be qualified).
   - Arguments in a CALL statement must correspond to the *data-names* in the USING list of the PROCEDURE DIVISION header in the called program.
   - You need not group into records noncontiguous elementary items (items that bear no hierarchical relationship to one another). You can define such items in separate *level-77* entries.

     Such entries must include a *level-number* 77, a *data-name*, and a PICTURE clause or the USAGE IS INDEX, BINARY, COMP, COMP-1, or COMP-2 clause.

PROCEDURE DIVISION

Format

PROCEDURE DIVISION [**USING** *data-name-1* [, *data-name-2*] . . . [*data-name-64*]].

Syntax Rules

1. You must define each operand in the USING phrase of the PROCEDURE DIVISION as a data item in the LINKAGE SECTION of the same program.
2. The data item must have an 01 or 77 *level-number*.
3. Addresses are passed from an external CALL in one-to-one correspondence to the operands in the USING list of the PROCEDURE DIVISION header so that data in the calling program can be manipulated in the called program.
4. Corresponding operands must have identical definitions in the DATA DIVISION of the calling and called programs.

5. You must ensure that the size, structure, and alignment of each passed operand are the same as those of the LINKAGE SECTION operand to which it corresponds. This equivalence is not checked at execution time.

The following sections, arranged in alphabetical order, discuss PROCEDURE DIVISION verbs that control interprogram communication.

CALL

Allows one program to communicate with another. It causes control to be transferred from one object program to another within a runfile.

Format

CALL { *data-name-1* / *literal-1* } [USING *data-name-2* [, *data-name-3*] · · ·]

[ON OVERFLOW *imperative-statement-1*]

[END-CALL]

Syntax Rules

1. The CALL statement appears in the calling program.
2. *literal-1* must be a nonnumeric literal; *data-name-1* must be defined as an alphanumeric data item.
3. *literal-1*, or the value of *data-name-1*, must be the *program-name* given in the PROGRAM-ID statement of the called program, not the PRIMOS filename of the called program.

   Note

   SEG recognizes only the first 8 characters of *program-name*. BIND recognizes the first 32 characters.

4. Include the USING phrase in the CALL statement only if a USING phrase appears in the PROCEDURE DIVISION header of the called program. Corresponding USING phrases in the calling and the called programs must have the same number of operands. The maximum number of *data-names* allowed after USING is listed in Appendix I.
5. Each operand in the USING phrase must have been defined as a data item in the FILE SECTION, WORKING-STORAGE SECTION, or LINKAGE SECTION of the calling program. These *data-names* can be qualified or subscripted.
6. Arguments in a CALL statement can have any *level-number* except 66 or 88. They can be subscripted or qualified.
7. The ON OVERFLOW phrase is checked for syntax only.
8. The END-CALL clause delimits the scope of the CALL statement. For more information, see the section Scope Terminators, in Chapter 8.

General Rules

1. The program whose name is specified by *literal-1*, or by the value of *data-name-1*, is the **called program**; the program in which the CALL statement appears is the **calling program**. The execution of a CALL statement transfers control from the calling program to the called program.
2. A program is in its initial state the first time it is called within a runfile. On all other entries into the called program, the state of the program remains the same as when control last passed from its EXIT statement back to the calling program. This includes all data fields and the status and positioning of all files.
3. Called programs can contain CALL statements. However, a called program must not contain a CALL statement that directly or indirectly calls the calling program.
4. The *data-names* specified by the USING phrase of the CALL statement represent those data items in a calling program that can be referred to in the called program.

   The order in which the *data-names* appear in the USING phrases of the two programs is critical; correspondence is positional, not by name. Corresponding operands in the called and calling programs must have the same number of character positions. Corresponding *data-names* refer to a single set of data that is available to the called and calling programs.

   Any *index-names* in the calling and called programs always refer to separate indexes.
5. The called program can be written in any language available on a Prime computer. However, ensure that the alignment of the calling operand and called definitions are compatible.

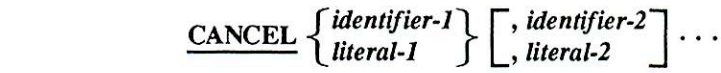
Note

If you wish to pass arguments whose size exceeds one segment to a called program written in a language other than COBOL85, and you do not specify the size of the arguments in the called program, compile the called program with the –BIG option, if available.

CANCEL

Releases the memory areas occupied by the referenced program.

Format



Syntax Rules

1. *literal-1*, *literal-2*, and so on, must each be a nonnumeric literal.
2. *identifier-1*, *identifier-2*, and so on, must each be defined as an alphanumeric data item such that its value can be a program name.

General Rule

CANCEL is syntax-checked only.

ENTER

Used for documentation only. This statement has no effect on the compiler or the compiled program.

Format

ENTER ***language-name*** **[*routine-name*].**

Syntax Rule

You can use the *language-name* and *routine-name* following ENTER as programmer-defined words elsewhere in the program. Each word must contain at least one alphabetic character.

EXIT PROGRAM

Marks the logical end of a called program.

Format

EXIT PROGRAM.

Syntax Rules

1. The EXIT PROGRAM statement must appear in a sentence by itself.
2. The EXIT PROGRAM sentence can be the only sentence in a paragraph. If used, it must be the last sentence in the paragraph.

General Rules

1. The execution of an EXIT PROGRAM statement in a called program returns control to the calling program.
2. An EXIT PROGRAM statement in a program that is not invoked by a CALL statement functions as an EXIT statement.
3. When you use the –FILE_ASSIGN compile option, EXIT PROGRAM suppresses the request for interactive file assignments. (See Chapter 3, Linking and Executing Programs.)

GOBACK — Prime Extension

Marks the logical end of a called program.

Format

GOBACK

General Rules

1. In a called program, execution of GOBACK returns control to the calling program.
2. A GOBACK statement in a program that is not called functions as an EXIT statement.
3. GOBACK is a synonym for EXIT PROGRAM.

Linking and Executing More Than One Program

To create a runfile containing interdependent programs, use BIND to link the program object files according to the steps listed in Chapter 3. You must link the main program first, followed by the called programs. Chapter 3 explains naming conventions for object files and runfiles. See also the example later in this section.

Note

For information on using the SEG loading utility, see Appendix M.

Error Messages

If the BIND utility does not return the message `BIND COMPLETE` after you enter the subcommand LI alone, you probably did not link all required subprograms or libraries. At this point you can use the MAP –UNDEFINED subcommand to locate the unresolved references.

If you attempt to execute a runfile with unresolved references, the system may return a message such as `LINKAGE_FAULT$` or `POINTER_FAULT$`, or may appear to run the program. For more information on BIND, see the *Programmer's Guide to BIND and EPFs.*

Example

This example presents two programs. The first, CALLER, exhibits some values, then calls the second, CALLED, which changes those values. CALLER then exhibits the changed values. Finally, the example shows how to load and execute the two programs together.

Prime Extension: The argument passed between the two programs is defined with level 02 in CALLER, but level 01 in CALLED.

Calling Program:

```
SOURCE FILE: <MYMFD>MYDIR>COBOL85>CALLER.COBOL85
COMPILED ON: THU, JUL 28 1988  AT: 13:22  BY: COBOL85 REV. 1.0-22.0  04/18/88.09:36
Options selected:  CALLER -LISTING
Optimization note: Currently "-OPTimize" means "-OPTimize 2",
Options used (* follows those that are not default):
     64V No_Ansi_Obsolete Big_Tables Binary CALCindex No_COMP No_CORrMap
     No_DeBuG No_Data_Rep_Opt No_ERRorFile ERRTty No_EXPlist No_File_Assign
     Formatted_DISplay No_HEXaddress Listing* No_MAp No_OFFset OPTimize(2)
     No_PRODuction No_RAnge No_SIGnalerrors SIlent(0) No_SLACKbytes TIME
     No_STANdard No_STATistics Store_Owner_Field SYNtaxmsg No_TRUNCdiags
     VARYing No_XRef
```

```
 1            IDENTIFICATION DIVISION.
 2            PROGRAM-ID. CALLER.
 3           *
 4            DATA DIVISION.
 5           *
 6            WORKING-STORAGE SECTION.
 7            01 A1.
 8                02 A2              PIC X VALUE 'A'.
 9                02 A3.
10                   03 A4           PIC X VALUE 'B'.
11                   03 A5           PIC X VALUE 'C'.
12                   03 A6.
13                      04 A7        PIC X VALUE 'D'.
14                      04 A8        COMP-2 VALUE -31415.9E-4.
15                   03 A9           PIC X(10) VALUE 'COBOL85'.
16
17           *
18            PROCEDURE DIVISION.
19                EXHIBIT NAMED A4.
20                EXHIBIT NAMED A5.
21                EXHIBIT NAMED A7.
22                EXHIBIT NAMED A8.
23                EXHIBIT NAMED A9.
24                CALL 'CALLED' USING A3.
25                EXHIBIT NAMED A4.
26                EXHIBIT NAMED A5.
27                EXHIBIT NAMED A7.
28                EXHIBIT NAMED A8.
29                EXHIBIT NAMED A9.
30                STOP RUN.
```

Called Program:

```
SOURCE FILE: <MYMFD>MYDIR>COBOL85>CALLED.COBOL85
COMPILED ON: THU, JUL 28 1988  AT: 13:21  BY: COBOL85 REV. 1.0-22.0  04/18/88.09:36
Options selected:  CALLED -LISTING
Optimization note: Currently "-OPTimize" means "-OPTimize 2",
Options used (* follows those that are not default):
     64V No_Ansi_Obsolete Big_Tables Binary CALCindex No_COMP No_CORrMap
     No_DeBuG No_Data_Rep_Opt No_ERRorFile ERRTty No_EXPlist No_File_Assign
     Formatted_DISplay No_HEXaddress Listing* No_MAp No_OFFset OPTimize(2)
     No_PRODuction No_RAnge No_SIGnalerrors SIlent(0) No_SLACKbytes TIME
     No_STANdard No_STATistics Store_Owner_Field SYNtaxmsg No_TRUNCdiags
     VARYing No_XRef

 1            IDENTIFICATION DIVISION.
 2            PROGRAM-ID. CALLED.
 3           *
 4            DATA DIVISION.
 5           *
 6            LINKAGE SECTION.
 7            01  ARG1.
 8                03 A4                PIC X.
 9                03 A5                PIC X.
10                03 A6.
11                   04 A7             PIC X.
12                   04 A8             COMP-2.
13                04 A9                PIC X(10).
14            PROCEDURE DIVISION USING ARG1.
15                DISPLAY 'ENTERING CALLED'.
```

```
16              MOVE 'X' TO A4.
17              MOVE 'Y' TO A5.
18              MOVE 'Z' TO A7.
19              MOVE 0 TO A8.
20              MOVE 'NEWCOBOL85' TO A9.
21              GOBACK.
```

Compilation, Linking, and Execution: To run CALLER and CALLED together, follow these steps:

1. Compile CALLER.COBOL85.

```
OK, COBOL85 CALLER -LISTING
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]

[0 ERRORS IN PROGRAM: CALLER.COBOL85]
```

2. Compile CALLED.COBOL85.

```
OK, COBOL85 CALLED -LISTING
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]
[0 ERRORS IN PROGRAM: CALLED.COBOL85]
```

3. Use BIND to create the runfile.

```
OK, BIND -LOAD CALLER -LOAD CALLED -LI COBOL85LIB -LI
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
BIND COMPLETE
OK,
```

4. Execute the runfile.

```
OK, RESUME CALLER
A4 = B
A5 = C
A7 = D
A8 = -3.1415900000000E+0000
A9 = COBOL85
ENTERING CALLED
A4 = X
A5 = Y
A7 = Z
A8 =  0.0000000000000E+0000
A9 = NEWCOBOL85
OK,
```

Calling a Program From an EPF Library: If a calling program of a runfile contains CALL statements of the form CALL *data-name-1*, you must take the following additional steps before linking and executing the runfile:

- Build the called programs into a program-class library EPF.
- Declare the called programs as entrypoints.
- Add the library EPF name to your ENTRY$ search rules.

For example, to call the program CALLED.COBOL85 from an EPF library, follow these steps:

1. Compile CALLED.COBOL85 as in the previous example; then build CALLED.BIN into a program-class library EPF as follows:

```
OK, BIND
[BIND Rev. 22.0 Copyright (c) 1988, Prime Computer, Inc.]
: LIBMODE -PROGRAM
Library is per program type
: LOAD CALLED                          /* name of binary file
: ENTRYNAME CALLED                     /* PROGRAM-ID
: LI COBOL85LIB
: LI
BIND COMPLETE
: FILE MY.SUBROUTINES.RUN              /* program-class library
EPF
OK,
```

2. Add the library EPF name, MY.SUBROUTINES.RUN, to your ENTRY$ search rules.
3. Use the SET_SEARCH_RULES command to enable your updated ENTRY$ search rules.
4. Modify CALLER.COBOL85 from the previous example, as follows:

```
WORKING-STORAGE SECTION.
01 WORK-FIELDS.
   05  SUBROUTINE-NAME       PIC X(32).
                .
                .
PROCEDURE DIVISION.
                .
                .
    DISPLAY 'ENTER SUBROUTINE NAME ...'.
    ACCEPT SUBROUTINE-NAME.
    CALL SUBROUTINE-NAME USING A3.
                .
                .
```

5. Compile CALLER.COBOL85 as in the previous example; then link CALLER.BIN, as follows:

```
OK, BIND
[BIND Rev. 22.0 Copyright (c) 1988, Prime Computer, Inc.]
: LOAD CALLER
: LI COBOL85LIB
: LI
BIND COMPLETE
: FILE
OK,
```

6. Now when you execute CALLER, it prompts you for the name of the called program.

```
OK, RESUME CALLER
A4 = B
A5 = C
A7 = D
A8 = -3.1415900000000E+0000
A9 = COBOL85
ENTER SUBROUTINE NAME ...
CALLED
ENTERING CALLED
A4 = X
A5 = Y
A7 = Z
A8 =  0.0000000000000E+0000
A9 = NEWCOBOL85
OK,
```

If CALLER cannot find CALLED at runtime, the system displays the following error message:

```
Error: condition "LINKAGE_FAULT$" raised.
Entry name "CALLED" not found while attempting to resolve
dynamic link from procedure "CALLER".
ER!
```

If this happens, then either your ENTRY$ search rules list does not include MY.SUBROUTINES.RUN, or MY.SUBROUTINES.RUN does not include the entryname CALLED.

For more information on library EPFs, entrypoints, and search lists, see the *Advanced Programmer's Guide.*

Language Interfaces

Because all Prime high-level languages are alike at the object-code level, COBOL85 object files can call and be called by object files produced by the BASICV, CC, CBL, FTN, F77, PASCAL, PMA, PL1, and PL1G compilers, provided that you observe the following restrictions:

- Write all I-O routines in the same language.
- If you wish to pass arguments whose size exceeds one segment to a called program written in a language other than COBOL85, and you do not specify the size of the arguments in the called program, compile the called program with the –BIG option, if available.
- Ensure that data types for variables being passed as arguments do not conflict.

Table 13-1 summarizes compatible data types. For more information on language interfaces, see the *Subroutines Reference I.*

TABLE 13-1
Data Type Compatibility in Prime Languages

Generic Unit	*BASIC/VM SUB FORTRAN*	*C*	*COBOL 74*	*COBOL85*	*FORTRAN IV*	*FORTRAN 77*	*Pascal*	*PL/I*
16-bit integer	INT	short enum	COMP PIC S9(1)- PIC S9(4)	COMP/BINARY PIC S9(1)- PIC S9(4)	INTEGER INTEGER*2 LOGICAL	INTEGER*2 LOGICAL*2	INTEGER Enumerated	FIXED BIN FIXED BIN(15)
32-bit integer	INT*4	int long	COMP PIC S9(5)- PIC S9(9)	COMP/BINARY PIC S9(5)- PIC S9(9)	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	LONGINTEGER	FIXED BIN(31)
64-bit integer			COMP PIC S9(10)- PIC S9(18)	COMP/BINARY PIC S9(10)- PIC S9(18)				
32-bit float single precision	REAL	float	COMP-1	COMP-1	REAL REAL*4	REAL REAL*4	REAL	FLOAT BIN FLOAT BIN(23)
64-bit float double precision	REAL*8	double	COMP-2	COMP-2	REAL*8	REAL*8	LONGREAL	FLOAT BIN(47)
128-bit float quad precision						REAL*16		
1 bit		short						BIT BIT(1)
1 left-aligned bit		short					BOOLEAN	BIT(1) ALIGNED
Bit string		unsigned int					SET	BIT(n)

TABLE 13-1
Data Type Compatibility in Prime Languages - Continued

Generic Unit	*BASIC/VM SUB FORTRAN*	*C*	*COBOL 74*	*COBOL85*	*FORTRAN IV*	*FORTRAN 77*	*Pascal*	*PL/I*
Fixed-length character string	INT	char NAME[n] char NAME	DISPLAY PIC A(n) PIC X(n) FILLER	DISPLAY PIC A(n) PIC X(n) FILLER		CHARACTER *n	CHAR PACKED ARRAY[1..n] OF CHAR	CHAR(n)
Fixed-length digit string			DISPLAY PIC 9(n)	DISPLAY PIC 9(n)				PICTURE
Fixed-length digit string, 2 digits per byte			COMP-3	COMP-3 PACKED- DECIMAL				FIXED DECIMAL
Varying-length character string		struct {short LENGTH; char DATA[n]; } CVAR					STRING[n]	CHAR(n) VARYING
32-bit pointer		Pointer (32IX-mode)			LOC( )	LOC( )		POINTER OPTIONS (SHORT)
48-bit pointer		Pointer (64V-mode)					Pointer	POINTER
Index			INDEX	INDEX				

Notes

For a discussion of possible workarounds for some of the empty boxes in this table as well as a description of generic units for PMA, refer to the appropriate language chapter in the *Subroutines Reference I*.

14 The SORT and MERGE Verbs

The SORT verb orders one or more data files. The MERGE verb combines two or more identically ordered files. Each verb manipulates its files according to a set of user-specified keys contained within each record.

This chapter discusses sort and merge operations, and the elements of the ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION that pertain to sort and merge operations. The discussions of the MERGE and SORT statements include program examples.

Sort and Merge Operations

To accomplish sort or merge operations, the program must use the SELECT clause in the ENVIRONMENT DIVISION, the sort or merge file description (SD) entry in the DATA DIVISION, and the SORT or MERGE statement in the PROCEDURE DIVISION.

A sort or merge operation ordinarily names

- One or more source files containing data to be sorted or merged. (Merge operations require two source files.) List these files in a GIVING clause.
- One or more destination files to contain resulting sorted or merged data. List these files in the USING clause.
- A work file (usually referred to as the sort file or merge file).

A program can substitute special processing for the preprocessing and postprocessing provided by the SORT and MERGE verbs. For example, the program can create or select certain records to be sorted, and process sorted or merged records in memory. Input and output procedures that you name in the SORT statement, and output procedures that you name in the MERGE statement contain this special processing. Input procedures must contain RELEASE statements, and output procedures must contain RETURN statements.

Note

The SORT verb invokes either PRIMOS SORT or the separately priced SyncSort/PRIME, whichever is installed on the system. A COBOL85 program can also invoke either PRIMOS SORT or SyncSort/PRIME through a subroutine call. For information on PRIMOS SORT, see the *PRIMOS User's Guide*. For information on SyncSort/PRIME, see the *SyncSort/PRIME Reference Manual*.

Strategy

If your application requires special operations, such as selecting records to be sorted or selecting records to be written to an output file, use input or output procedures, possibly combined with USING and GIVING clauses, to minimize the number of records processed.

For example, if you want to produce a sorted list of customers whose account numbers are greater than a specific value, specify an input procedure and a GIVING file. Within the input procedure RELEASE only those records whose account numbers qualify for sorting.

Likewise, you can produce a sorted list of selected customers by specifying a USING file and an output procedure. In this case, COBOL85 sorts all the records in the file. In the output procedure, code RETURN statements to retrieve each sorted record, but WRITE only selected records to the output file. This method is not as efficient as the first, especially for files containing many records, because COBOL85 processes each record more often.

Note

When you specify the USING and GIVING clauses, COBOL85 may simulate input and output procedures if additional file processing is required. For example, tape files, indexed files, relative files, multiple GIVING files, USE procedures, and files of different sizes need special processing from the COBOL85 runtime libraries. COBOL85 provides the required processing by simulating input and output procedures.

Linking Sort and Merge Programs

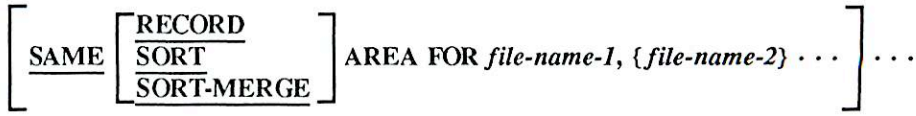
When you create a runfile that includes a SORT or MERGE statement, include the sort library (VSRTLI) in the linking sequence. An example of linking is given at the end of this chapter.

ENVIRONMENT DIVISION — I-O-CONTROL

You can use the SAME clause in the I-O-CONTROL paragraph to specify the memory area to be shared by different files.

Format

I-O-CONTROL.



Syntax Rules

1. In the SAME clause, SORT and SORT-MERGE are equivalent.
2. If you use the SAME SORT AREA or SAME SORT-MERGE AREA clause, at least one of the *file-names* must represent a sort or merge file. You may also name files that do not represent sort or merge files in the clause.
3. More than one SAME clause may be included in a program. However,

   - A *file-name* must not appear in more than one SAME RECORD AREA clause.
   - A *file-name* that represents a sort or merge file must not appear in more than one SAME SORT AREA or SAME SORT-MERGE AREA clause.
   - If a *file-name* that does not represent a sort or merge file appears in a SAME AREA clause and in one or more SAME SORT AREA or SAME SORT-MERGE AREA clauses, all the files named in the first clause must be named in the second clause(s).

4. The files referenced in the SAME SORT AREA, SAME SORT-MERGE AREA, or SAME RECORD AREA clause need not all have the same organization or access.

General Rules

1. The SAME RECORD AREA clause specifies that two or more files are to use the same memory area for processing the current logical record. All the files may be open at the same time. A logical record in the SAME RECORD AREA is considered as a logical record of each opened output file whose *file-name* appears in this SAME RECORD AREA clause and of the most recently read input file whose *file-name* appears in this SAME RECORD AREA clause. This is equivalent to implicit redefinition of the area; records are aligned on the leftmost character position.
2. If you use the SAME SORT AREA or SAME SORT-MERGE AREA clause, at least one of the *file-names* must represent a sort or merge file. You may also name files that do not represent sort or merge files in the clause. This clause specifies that storage is shared as follows:

   - The SAME SORT AREA or SAME SORT-MERGE AREA clause specifies a memory area that will be made available for use in sorting or merging each sort or merge file named. Thus, any memory area allocated for sorting or merging a sort or merge file is available for reuse in sorting or merging any other sort or merge file.
   - In addition, storage areas assigned to files that do not represent sort or merge files may be allocated as needed for sorting or merging the sort or merge files named in the SAME SORT AREA or SAME SORT-MERGE AREA clause.
   - Files other than sort or merge files do not share the same storage area with each other. If you wish these files to share the same storage area with each other, the program must also include a SAME AREA or SAME RECORD AREA clause naming these files.
   - During the execution of a SORT or MERGE statement that refers to a sort or merge file named in this clause, any non-sort-merge files named in the same clause must be closed.

DATA DIVISION — FILE SECTION

A sort-merge file description (SD) in the FILE SECTION furnishes information concerning the physical structure, identification, and record names of the sort or merge file. There are no label procedures that you can control, and the rules for blocking and internal storage are peculiar to the SORT statement.

Format

```
SD file-name-1

[ RECORD { CONTAINS integer-1 CHARACTERS                                      }
         { IS VARYING IN SIZE [[FROM integer-2] [TO integer-3] CHARACTERS]    }
         { CONTAINS integer-4 TO integer-5 CHARACTERS                         }
         { IS NOT VARYING IN SIZE                                             } ]

[ DATA { RECORD IS  } {data-name-1} ... ] .
       { RECORDS ARE }
```

Syntax Rules

1. The level indicator SD identifies the beginning of the sort-merge file description.
2. The clauses that follow *file-name*, and their order of appearance, are optional.
3. One or more *record-description-entries* must follow the file description; however, no I-O statements may be executed for this file.
4. The file must be specified in a SELECT clause and assigned to PFMS or PRIMOS.
5. A sort-merge file description may appear anywhere in the FILE SECTION.
6. The *record-description-entry* must specify all the data items listed as keys (at least one) in the SORT or MERGE statements that reference the sort or merge file.

PROCEDURE DIVISION

The following sections discuss these COBOL85 statements:

- MERGE
- RELEASE
- RETURN
- SORT

The discussions of MERGE and SORT each include a program example.

MERGE

Combines two or more identically sequenced files on a set of specified keys, and during the process makes records available, in merged order, to an output procedure or to one or more output files.

Format

```
MERGE file-name-1 ON { ASCENDING  } KEY data-name-1 [, data-name-2] ...
                     { DESCENDING }

[ ON { ASCENDING  } KEY data-name-3 [, data-name-4] ... ] ...
     { DESCENDING }

[COLLATING SEQUENCE IS alphabet-name]

USING file-name-2, file-name-3 [, file-name-4] ...

{ OUTPUT PROCEDURE IS procedure-name-1 [ { THROUGH } procedure-name-2 ] }
                                       { THRU    }
{ GIVING file-name-5 [, file-name-6] ... }
```

Syntax Rules

1. Each *file-name-1* must be described in a sort-merge *file-description-entry* in the DATA DIVISION.
2. *procedure-name-1* specifies the first section or paragraph in an output procedure. *procedure-name-2*, if specified, identifies the last section or paragraph of an output procedure.
3. Each *file-name-2*, *file-name-3*, *file-name-4*, *file-name-5*, and *file-name-6* must be described in a *file-description-entry*, not in a sort-merge *file-description-entry*, in the DATA DIVISION.
4. The actual size(s) of the logical record(s) described for *file-name-2*, *file-name-3*, and *file-name-4* (the USING files) may be different from the actual size of the logical record(s) described for *file-name-1*. The restrictions are as follows:

   - If *file-name-1* contains variable-length records, the size(s) of the records contained in the USING files must not be less than the smallest record nor larger than the largest record described for *file-name-1*.
   - If *file-name-1* contains fixed-length records, the size(s) of the records contained in the USING files must not be larger than the largest record described for *file-name-1*. If the record(s) described for the USING files are smaller than the records in *file-name-1*, the records are left justified, and any unused character positions at the right end of the record are filled with blanks when the record is released to the merge utility.

5. The actual size(s) of the logical record(s) described for *file-name-5* and *file-name-6* (the GIVING files) may be different from the actual size of the logical record(s) described for *file-name-1*. The restrictions are as follows:

   - If *file-name-5* contains variable-length records, the size(s) of the records contained in *file-name-1* must not be less than the smallest record nor larger than the largest record described for *file-name-5*.
   - If *file-name-5* contains fixed-length records, the size(s) of the records contained in *file-name-1* must not be larger than the largest record described for *file-name-5*. If the record(s) described for *file-name-1* are less than the record size specified for the GIVING file, the records are left justified, and any unused character positions at the right end of the record are filled with blanks when the record is returned from the merge utility.

6. The logical record size(s) associated with *file-name-5* and *file-name-6* may be larger than the largest record size described for *file-name-2*, *file-name-3*, and *file-name-4*. The smaller record is left justified, and any unused character positions at the right end of the record are filled with blanks.
7. The words THRU and THROUGH are equivalent.
8. *data-name-1*, *data-name-2*, *data-name-3*, and *data-name-4* are **KEY data-names** and are subject to the following rules:

   - The data items identified by KEY *data-names* must be described in records associated with *file-name-1*.
   - KEY *data-names* may be qualified.
   - The data items identified by KEY *data-names* must not be variable-length items. KEY *data-names* for variable-length records must be within the fixed portion of the variable-length record.
   - If *file-name-1* has more than one record description, then the data items identified by KEY *data-names* need be described in only one of the record descriptions.
   - None of the KEY *data-names* can be described by an entry that either contains an OCCURS clause or is subordinate to an entry that contains an OCCURS clause.

9. MERGE statements may not appear in the declaratives portion of the PROCEDURE DIVISION or in an input or output procedure associated with a SORT or MERGE statement. MERGE statements, wherever they occur, must not be executed under the control of an input or output procedure.
10. You can specify a maximum of twenty files in the GIVING clause of the MERGE statement. Exceeding this maximum causes a fatal error. These GIVING files may all be of different types and record formats.
11. No more than one GIVING file may specify a tape file on the same reel.
12. You can specify a maximum of eleven files in the USING clause of the MERGE statement. Exceeding this maximum causes a fatal error. These USING files may be compressed or uncompressed, fixed-length or variable-length.
13. Full tape support for the MERGE statement is provided. This includes support for single-reel single-volume, single-reel multi-volume, multi-reel single-volume, and multi-reel multi-volume tape files used in the USING and GIVING clauses as well as specified within the corresponding OUTPUT PROCEDURE.

No two tape files specified in any one MERGE statement can reside on the same multiple-file reel.

No pair of *file-names* in a MERGE statement may be specified in the same SAME AREA, SAME SORT AREA or SAME SORT-MERGE AREA clause. Files named in the GIVING phrase may be specified in a SAME RECORD AREA clause.

14. The *alphabet-name* is a programmer-defined word and is defined in the SPECIAL-NAMES paragraph in the CONFIGURATION SECTION. It may be specified as NATIVE, STANDARD-1, STANDARD-2, or EBCDIC. More discussion of these collating sequences is given with SORT below.

General Rules

1. Files referenced in a MERGE statement must be closed prior to execution of the MERGE and may not be opened until the merge operation is complete. An output file in an output procedure must be opened by an explicit OPEN statement, written, probably in the output procedure, and then explicitly CLOSED.

   The MERGE statement merges all records contained in *file-name-2*, *file-name-3*, and *file-name-4*. These files are automatically opened and closed by the merge operation with all implicit functions performed, such as the execution of any associated USE procedures, and the setting of any associated file status codes. On termination, status of all files is as if a CLOSE statement were executed for each file.

2. The execution of any USE procedure must not cause the execution of any statement that manipulates the files referenced by either the USING or GIVING clauses.

3. The *data-names* following the word KEY are listed from left to right in the MERGE statement in order of decreasing significance without regard to how they are divided into KEY phrases. In the format, *data-name-1* is the major key, *data-name-2* is the next most significant key, and so on.

   - When the ASCENDING phrase is specified, the merged sequence is from the lowest value of the KEY *data-names* to the highest value.
   - When the DESCENDING phrase is specified, the merged sequence is from the highest value of the KEY *data-names* to the lowest value.
   - The key values are compared according to the rules for comparison of operands in a relation condition. (See the section titled Conditional Expressions, in Chapter 4.)

4. If the COLLATING SEQUENCE is not specified, the MERGE statement uses the collating sequence specified in the PROGRAM COLLATING SEQUENCE clause of the OBJECT COMPUTER paragraph. The collating sequence is either NATIVE, STANDARD-1, STANDARD-2, or EBCDIC. (The default is NATIVE.)

5. The output procedure must consist of one or more sections or paragraphs that appear contiguously in a source program and do not form part of any other procedure. In order to make merged records available for processing, the output procedure must include the execution of at least one RETURN statement.

   The output procedure's range includes all statements that are executed as a result of a transfer of control by the CALL, EXIT, GO TO, and PERFORM statements, as well as all statements in declarative procedures that are executed as a result of the execution of statements in the range of the output procedure.

Control must not be passed to the output procedure except when a related SORT or MERGE statement is being executed. The output procedure may consist of any procedures needed to select, modify, or copy the records that are being returned one at a time in merged order, from *file-name-1*. The restrictions on the statements within the output procedure are as follows:

- The range of the output procedure must not cause the execution of any MERGE, RELEASE, or SORT statements.
- The statements within the output procedure must not manipulate the files referenced by the USING clause.
- The remainder of the PROCEDURE DIVISION must not contain any transfers of control to points within the output procedures; ALTER, GO TO, and PERFORM statements in the remainder of the PROCEDURE DIVISION are not permitted to refer to *procedure-names* within the output procedures.

6. If an output procedure is specified, control passes to it during execution of the MERGE statement. When control passes the last statement in the output procedure, the merge terminates, and control passes to the next executable statement after the MERGE statement. Before entering the output procedure, the merge operation reaches a point at which it can select the next record in merged order when requested. The RETURN statements in the output procedure are the requests for the next record.
7. If the GIVING phrase is specified, all the merged records in *file-name-1* are automatically written on *file-name-5*, *file-name-6*, and so on, as the implied output procedure for this MERGE statement.
8. In the case of identical key fields between records from two or more input files, the records are written on *file-name-5*, *file-name-6*, and so on, or returned to the output procedure, in the order that the associated input files are specified in the MERGE statement.
9. The results of the merge operation are predictable only when the records in the files referenced by *file-name-2*, *file-name-3*, and so on, are ordered as described in the ASCENDING or DESCENDING KEY clause associated with the MERGE statement.
10. All file types are supported for both the USING and GIVING clauses of the MERGE statement. This includes INDEXED and RELATIVE (MIDASPLUS or PRISAM), as well as SEQUENTIAL (PRIMOS, MT9, PRISAM).
11. If an indexed file is specified in the GIVING clause, the first specification of the *data-name-1* must be associated with an ASCENDING phrase. In addition, the data item referenced by *data-name-1* must occupy the same character positions in its record as the associated primary key for that file.
12. If a relative file is specified in the GIVING clause, the relative key data item for the first record returned contains the value '1'; for the second record returned, the value '2', and so on. After the execution of the MERGE statement, the content of the relative key data item indicates the last record returned to the file.
13. PRISAM sequential, indexed, and relative nontransactional files are supported for the GIVING clause of the MERGE statement. MERGE cannot START and END a PRISAM transaction. Therefore, PRISAM transactional files cannot be referenced in the GIVING clause. However, the user can specify a PRISAM transactional file for the output file of the merge operation by using an OUTPUT PROCEDURE and writing each returned record to the PRISAM file.

14. For PRISAM and MIDASPLUS files, an empty, but already created file is expected for the output file. Use the corresponding FAU and CREATK utilities to create the file prior to program execution.

MERGE Example

The following program merges two files, MRGFIL1 and MRGFIL2, using an output procedure that sends them to the print file YEARLY. An example of linking and execution, and the sample files follow the program.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. MERGSAMP.
 AUTHOR. W. T. C.
 INSTALLATION. PRIME.
 DATE-WRITTEN. 14 APRIL 82.
 DATE-COMPILED.
 REMARKS. TESTING THE MERGE VERB. FILES MUST BE SORTED FIRST!
*****************************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PRIME.
 OBJECT-COMPUTER. PRIME.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT FIRST-HALF ASSIGN TO PRIMOS.
     SELECT SECOND-HALF ASSIGN TO PRIMOS.
     SELECT YEARLY ASSIGN TO PRINTER.
     SELECT MERGE-FILE ASSIGN TO PRIMOS.
*
 DATA DIVISION.
 FILE SECTION.
 FD  FIRST-HALF
     VALUE OF FILE-ID IS 'MRGFIL1'
     DATA RECORD IS SALES-HISTORY-1.
 01  SALES-HISTORY-1.
     05 DEPT-NO          PIC 999.
     05 PROD-NO          PIC 9(5).
 FD  SECOND-HALF
     VALUE OF FILE-ID IS 'MRGFIL2'
     DATA RECORD IS SALES-HISTORY-2.
 01  SALES-HISTORY-2.
     05 DEPT-NO          PIC 999.
     05 PROD-NO          PIC 9(5).
 FD  YEARLY
     DATA RECORD IS CUMULATIVE-SALES.
 01  CUMULATIVE-SALES.
     05  DEPT            PIC 9(3).
     05  FILLER-1        PIC X(3).
     05  PROD            PIC 9(5).
 SD  MERGE-FILE
     DATA RECORD IS MERGE-RECORD.
```

```
 01  MERGE-RECORD.
     05 DEPARTMENT        PIC 999.
     05 PRODUCT           PIC 9(5).
*
 WORKING-STORAGE SECTION.
 01 END-OF-DATA           PIC XXX     VALUE 'NO '.
 01  PROOF-LIST.
     05 DEPT-NO-REPORT    PIC 999.
     05 PROD-NO-REPORT    PIC 9(5).
*
 PROCEDURE DIVISION.
 START-PARA.
     MERGE MERGE-FILE ON ASCENDING KEY DEPARTMENT
         USING FIRST-HALF, SECOND-HALF
         OUTPUT PROCEDURE IS OUTPUT-PROCEDURE.
     STOP RUN.
*
 OUTPUT-PROCEDURE SECTION.
 CREATE-PROOF-LIST.
     OPEN OUTPUT YEARLY.
     PERFORM RETURN-DATA.
     PERFORM WRITE-DATA UNTIL END-OF-DATA = 'YES'.
     CLOSE YEARLY.
     GO TO END-MERGE.

  RETURN-DATA.
     RETURN MERGE-FILE INTO PROOF-LIST
       AT END
         MOVE 'YES' TO END-OF-DATA.
*
 WRITE-DATA.
     MOVE SPACES TO FILLER-1.
     MOVE DEPARTMENT TO DEPT.
     MOVE PRODUCT TO PROD.
     WRITE CUMULATIVE-SALES.
     PERFORM RETURN-DATA.

  END-MERGE.
     EXIT.
```

First Input File:

```
00123576
00376231
00592862
```

Second Input File:

```
00263550
00443651
00640166
```

Compiling, Linking, and Executing: Compile, link, and run the program with the following dialog:

```
OK, COBOL85 MERGE -LISTING
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]
[0 ERRORS IN PROGRAM: MERGE.COBOL85]

OK, BIND
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
: LOAD MERGE
: LI COBOL85LIB
: LI VSRTLI
: LI
BIND COMPLETE
: FILE
OK, RESUME MERGE
OK,
```

Output file (YEARLY):

```
001   23576
002   63550
003   76231
004   43651
005   92862
006   40166
```

RELEASE

Transfers records to the initial phase of a sort operation, allowing processing of the record content.

Format

RELEASE *record-name* [FROM *data-name*]

Syntax Rules

1. Specify a RELEASE statement only within an input procedure associated with a SORT statement. Input procedures are described with the SORT statement below.
2. The *record-name* must be the name of a logical record in the SD entry for the sort file named in the associated SORT statement. The *record-name* may be qualified. The *data-name* must specify a field capable of holding a record read from an input file (FD entry). It may be in WORKING-STORAGE.

General Rules

1. The execution of a RELEASE statement causes *record-name* to be released to the initial phase of a sort operation.

   A RELEASE statement must be executed for each record to be sent to a sort or merge operation.
2. If the FROM phrase is used, the contents of *data-name* are moved to *record-name*, then the contents of *record-name* are released to the sort file. Moving takes place according to the rules for the MOVE statement without the CORRESPONDING phrase.
3. After the execution of the RELEASE statement, the logical record is still available as a record of other files referenced in the SAME AREA clause, as well as being available to the file associated with *record-name*. When control passes from the input procedure, the sort file consists of all those records placed in it by the execution of RELEASE statements.
4. If a RELEASE statement releases a record associated with an input file, the input file must have been opened and read.

RETURN

Obtains sorted records from the final phase of a sort operation, or merged records during a merge.

Format

RETURN *file-name* **RECORD** [**INTO** *data-name-1*]

AT END *imperative-statement-1*

[**NOT AT END** *imperative-statement-2*]

[**END-RETURN**]

Syntax Rules

1. *file-name* must be described by an SD entry in the FILE SECTION of the DATA DIVISION.
2. *data-name* must be able to contain a record to be written to an output file.
3. A RETURN statement may be specified only within an output procedure associated with a SORT or MERGE statement for *file-name*. Output procedures are defined in SORT and MERGE statements.
4. The areas associated with *data-name* and *file-name* must not be the same storage area.

General Rules

1. If more than one record description is associated with *file-name*, these records automatically share the same storage area; that is, the area is implicitly redefined. After the execution of the RETURN statement, any data items that lie beyond the range of the current record are undefined.
2. When the RETURN statement is executed, the next record from *file-name* (in the order of the key) is made available for processing in the record areas associated with the sort or merge file.

   A RETURN statement must be executed for each record to be retrieved from the sort or merge operation.
3. If the INTO phrase is specified, the current record is moved from the input (file) area to the area specified by *data-name* according to the rules for the MOVE statement without the CORRESPONDING phrase. The implied MOVE does not occur if there is an AT END condition. Any subscripting or indexing associated with *data-name* is evaluated after the record is returned and immediately before it is moved to *data-name*.
4. When the INTO phrase is used, the data is available in both the input record area and the data area associated with *data-name*.
5. If no next logical record exists at the execution of a RETURN statement, the AT END condition occurs. The contents of the record areas associated with the file are undefined when that condition occurs. After the execution of the *imperative-statement* in the AT END phrase, no RETURN statement may be executed as part of the current output procedure. Control is transferred to the end of the RETURN statement and the NOT AT END phrase is ignored, if specified.
6. If an AT END condition does not occur, then after the record is made available and after processing an INTO phrase, control is transferred to *imperative-statement-2*, if specified. Otherwise, control is transferred to the end of the RETURN statement.
7. The END-RETURN clause delimits the scope of the RETURN statement. For more information, see the section titled Scope Terminators, in Chapter 8.

SORT

Creates a sort file by executing an input procedure or by transferring records from another file or files; sorts the records in the sort file on a set of specified keys; and, in the final phase of the sort operation, makes available each record from the sort file, in sorted order, to an output procedure or to an output file or files.

Format

```
SORT file-name-1 { ON { ASCENDING } KEY {data-name-1} ... } ...
                   { DESCENDING }

[WITH DUPLICATES IN ORDER]

[COLLATING SEQUENCE IS alphabet-name-1]

{ INPUT PROCEDURE IS procedure-name-1 [ { THROUGH } procedure-name-2 ] }
{                                     [ { THRU    }                  ] }
{ USING {file-name-2} ...                                              }

{ OUTPUT PROCEDURE IS procedure-name-3 [ { THROUGH } procedure-name-4 ] }
{                                      [ { THRU    }                  ] }
{ GIVING {file-name-3} ...                                              }
```

Syntax Rules

1. SORT statements must not appear in the DECLARATIVES portion of the PROCEDURE DIVISION or in an input or output procedure associated with a SORT or MERGE statement. SORT statements, wherever they occur, must not be executed under the control of an input or output procedure.
2. *file-name-1* must be described in an SD entry in the DATA DIVISION. Each *file-name-2* and *file-name-3* must be described in a *file-description-entry*, not in a sort-merge *file-description-entry*, in the DATA DIVISION.
3. *data-name-1*, *data-name-2*, and so on (**KEY data-names**) are subject to the following rules:

   - The data items identified by KEY *data-names* must be described in records associated with *file-name-1*.
   - KEY *data-names* may be qualified.
   - KEY *data-names* may not describe variable-length data items, nor may they name group items that contain variable-occurrence data items. KEY *data-names* for variable-length records must be within the fixed portion of the variable-length record.

- If *file-name-1* has more than one record description, then the data items identified by KEY *data-names* need be described in only one of the record descriptions. In other words, the same character positions referenced by a KEY *data-name* in one *record-description-entry* are taken as the KEY in all records of *file-name-1*.
- The data items identified by KEY *data-names* may not contain an OCCURS clause or be subordinate to an item that contains an OCCURS clause.

4. The *procedure-name-1* specifies the first section or paragraph in an input procedure. The *procedure-name-2*, if specified, identifies the last section or paragraph of an input procedure.

   Similarly, *procedure-name-3* and *procedure-name-4* specify an output procedure.
5. The words THRU and THROUGH are equivalent.
6. *file-name-2* and *file-name-3* may be the same *file-name*. However, for PRISAM and MIDASPLUS files, the same physical file may not be used as both input and output files for the same SORT statement.
7. The logical record size associated with *file-name-2* must not be larger than the record size described for *file-name-3*.
8. The actual size of the logical record(s) described for *file-name-2* (the USING file) may be different from the actual size of the logical record(s) described for *file-name-1*. The restrictions are as follows:

   - If *file-name-1* contains variable-length records, the size of the records contained in the USING files must not be less than the smallest record nor larger than the largest record described for *file-name-1*.
   - If *file-name-1* contains fixed-length records, the size of the records contained in the USING files must not be larger than the largest record described for *file-name-1*. If the record(s) described for the USING files are smaller than the records in *file-name-1*, the records are left justified, and any unused character positions at the right end of the record are filled with blanks when the record is released to the sort utility.

9. You can specify a maximum of twenty files in the GIVING clause of the SORT statement. Exceeding this maximum causes a fatal error. These GIVING files may all be of different types and record formats.
10. You can specify a maximum of twenty files in the USING clause of the SORT statement. Exceeding this maximum causes a fatal error. These USING files may be compressed or uncompressed, fixed-length or variable-length.
11. The actual size of the logical record(s) described for *file-name-3* (the GIVING file) may be different from the actual size of the logical record(s) described for *file-name-1*. The restrictions are as follows:

    - If *file-name-3* contains variable-length records, the size of the records contained in *file-name-1* must not be less than the smallest record nor larger than the largest record described for *file-name-3*.
    - If *file-name-3* contains fixed-length records, the size of the records contained in *file-name-1* must not be larger than the largest record described for *file-name-3*. If the record(s) described for *file-name-1* are less than the record size specified for the GIVING file, the records are left justified, and any unused character positions at the right end of the record are filled with blanks when the record is returned from the sort utility.

12. The logical record size associated with *file-name-3* may be larger than the largest record size described for *file-name-2*. The smaller record is left justified, and any unused character positions at the right end of the record are filled with blanks.
13. The *alphabet-name* is a programmer-defined word; define it in the SPECIAL-NAMES paragraph in the CONFIGURATION SECTION. Specify it as NATIVE, STANDARD-1, STANDARD-2, or EBCDIC.

General Rules

1. Files referenced in a SORT statement must be closed prior to execution of the sort operation. They may not be opened except through an input or output procedure, until the sort is complete.

   The SORT statement sorts all records contained in *file-name-2*. These files are automatically opened and closed by the sort operation with all implicit functions performed, such as the execution of any associated USE procedures, and the setting of any associated file status codes. The terminating function for all files is performed as if a CLOSE statement were executed for each file.
2. The execution of any USE procedure must not cause the execution of any statement that manipulates the files referenced by either the USING or GIVING clauses.
3. If *file-name-1* contains only fixed-length records, any record in *file-name-2* released to *file-name-1* is left justified, and any unused character positions at the right end of the record are filled with blanks.
4. The *data-names* following the word KEY are listed in order of decreasing significance no matter how they are divided into KEY phrases. For example, *data-name-1* is the major key, *data-name-2* is the next most significant key.

   - When the ASCENDING phrase is specified, the sorted sequence is from the lowest key value to the highest key value.
   - When the DESCENDING phrase is specified, the sorted sequence is from the highest key value to the lowest key value.
   - The key values are compared according to the rules for comparison of operands in a relation condition. (See the section titled Conditional Expressions, in Chapter 4.)

5. If the DUPLICATES phrase is specified and the contents of all the key data items associated with one data record are equal to the contents of the corresponding key data items associated with one or more other data records, then the order of return of these records is

   - The order of the associated input files as specified in the SORT statement. Within a given input file the order is that in which the records are accessed from that file.
   - The order in which these records are released by an input procedure, when an input procedure is specified.

   If the DUPLICATES phrase is not specified and the contents of all the key data items associated with one data record are equal to the contents of the corresponding key data items associated with one or more other data records, then the order of return of these records is undefined.

6. The COLLATING SEQUENCE IS clause may be used to specify the collating sequence to be used in the sort. If it is not specified, the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph is used, if present. The following examples show the difference that the COLLATING sequence can make.

```
OK,    SLIST COLLATING.DATA
BABC010132780300200
AABC000123456700000
200C020043298765400
```

   If this file is sorted with COLLATING SEQUENCE IS NATIVE or no COLLATING SEQUENCE clause, the output file is the following:

```
OK,    SLIST F2.NATIVE
200C020043298765400
AABC000123456700000
BABC010132780300200
```

   However, if the COLLATING SEQUENCE clause specifies *alphabet-name* as EBCDIC, the output file is the following:

```
OK,    SLIST F2.EBCDIC
AABC000123456700000
BABC010132780300200
200C020043298765400
```

7. If the files referenced by the USING and GIVING clauses are magnetic tape files, they may reside on the same multiple-file reel.

   USING and GIVING tape files with variable-length record descriptions are supported.

8. No more than one GIVING file may specify a tape file on the same reel.

9. MIDASPLUS and PRISAM indexed files are fully supported for the SORT statement. This includes support for USING and GIVING files.

   If an indexed file is specified in the GIVING clause, the first specification of the *data-name-1* must be associated with an ASCENDING phrase. In addition, the data item referenced by *data-name-1* must occupy the same character positions in its record as the associated primary key for that file.

10. MIDASPLUS and PRISAM relative files are fully supported for the SORT statement. This includes support for USING and GIVING files.

    If a relative file is specified in the USING phrase, the content of the relative key data item is undefined after the execution of the SORT statement.

    If a relative file is specified in the GIVING clause, the relative key data item for the first record returned contains the value '1'; for the second record returned, the value '2', and so on. After the execution of the SORT statement, the content of the relative key data item indicates the last record returned to the file.

11. If the USING or GIVING clauses specify PRISAM sequential, indexed, or relative files, these files must be PRISAM nontransactional files.

SORT cannot START and END a PRISAM transaction. However, if you wish to sort a PRISAM transactional file, you must use an INPUT PROCEDURE to RELEASE the records to the sort utility. Additionally, if you wish the sorted records to be written to a PRISAM transactional file, you must use an OUTPUT PROCEDURE to RETURN the records from the sort utility.

12. For PRISAM and MIDASPLUS files, an empty, but already created file is expected for the output file. Use the corresponding FAU and CREATK utilities to create the file prior to program execution.

Rules for Input Procedures and USING

1. The input procedure must consist of one or more sections or paragraphs that are written consecutively and do not form a part of any output procedure. In order to transfer records to *file-name-1*, the input procedure must include at least one RELEASE statement. Control must not be passed to the input procedure except when a related SORT statement is being executed. An example is given at the end of this chapter.

   The input procedure can include any procedures needed to select, create, or modify records, including a READ for the input file, which must first be opened. The range of the input procedure includes all statements that are executed as a result of a transfer of control by the CALL, EXIT, GO TO, and PERFORM statements, as well as all statements in declarative procedures that are executed as a result of the execution of statements in the range of the input procedure currently being executed. The statements within the input procedure have two restrictions:

   - The range of the input procedure must not cause the execution of any MERGE, RETURN, or SORT statement.
   - The remainder of the PROCEDURE DIVISION must not contain any transfers of control to points inside the input procedure; GO TO and PERFORM statements in the remainder of the PROCEDURE DIVISION must not refer to *procedure-names* within the input procedure.

2. If an input procedure is specified, control is passed to the input procedure before *file-name-1* is sorted by the SORT statement. When control passes the last statement in the input procedure, the records that have been released to *file-name-1* are sorted.
3. If the USING phrase is specified, all the records in the USING file list (*file-name-2*, and so on) are automatically transferred to *file-name-1*. At the time of execution of the SORT statement, files in the USING list must not be open. For files in the USING list, the execution of the SORT statement performs the following actions:

   - The processing of the file is initiated as if an OPEN statement with the INPUT phrase were executed.
   - The file references are passed to the sort routine, which puts all the records in a single file. Each record is obtained as if a READ statement with the NEXT and the AT END phrase were executed.
   - The processing of the file is terminated as if a CLOSE statement were executed.

Rules for Output Procedures and GIVING

1. The output procedure must consist of one or more sections or paragraphs that are written consecutively and do not form a part of any input procedure. In order to make sorted records available for processing, the output procedure must include at least one RETURN statement. Control must not be passed to the output procedure except when a related SORT statement is being executed. An example is given with the discussion of MERGE above.

   The output procedure may consist of any procedures needed to select, modify, or copy the records that are being returned, one at a time in sorted order, from the sort file. The output procedure may include a WRITE statement for the output file, which must first be opened. The range of the output procedure includes all statements that are executed as a result of a transfer of control by the CALL, EXIT, GO TO, and PERFORM statements, as well as all statements in declarative procedures that are executed as a result of the execution of statements in the range of the output procedure currently being executed. The procedural statements within the output procedure have two restrictions:

   - The range of the output procedure must not cause the execution of any MERGE, RELEASE, or SORT statement.
   - The remainder of the PROCEDURE DIVISION must not contain any transfers of control to points within the output procedure; GO TO and PERFORM statements in the remainder of the PROCEDURE DIVISION must not refer to *procedure-names* within the output procedure.

2. If an output procedure is specified, control passes to it after *file-name-1* has been sorted by the SORT statement. When control passes the last statement in the output procedure, control returns to the next executable statement after the SORT statement. Before entering the output procedure, the sort operation reaches a point at which it can select the next record in sorted order, when requested. The RETURN statements in the output procedure are the requests for the next record.
3. If the GIVING phrase is specified, all the sorted records are automatically written to *file-name-3* as the implied output procedure for the SORT statement. At the time of the execution of the SORT statement, *file-name-3* must not be open. For *file-name-3*, the execution of the SORT statement performs the following actions:

   - Initiates the processing of the file. The initiation is performed as if an OPEN statement with the OUTPUT phrase were executed.
   - Returns the sorted logical records and writes them into the file. The records are written as if a WRITE statement without any optional phrases were executed.
   - Terminates the processing of the file. The termination is performed as if a CLOSE statement were executed.

4. If *file-name-3* contains only fixed-length records, any record in *file-name-1* containing fewer character positions is left justified and padded with blanks at the right end of the record when the record is returned to *file-name-3*.

SORT Example

The following example is a source file for a sample program SAMPLE.SORT.COBOL85. This example uses a SORT statement with an input procedure. The input procedure edits records for errors before releasing them to the sort file SORT-WK.

Below the example are

- A sample input file (SEFILE) including one erroneous entry
- A sample dialog for compiling, linking, and running the program
- The resulting output file (OUT-SORT)

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.                        SRTBUDGT.
 AUTHOR.                            PEGGY PECK.
 INSTALLATION.                      PRIME.
 DATE-COMPILED.
*****************************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PRIME.
 OBJECT-COMPUTER. PRIME.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT IN-FILE ASSIGN TO PRIMOS,
         FILE STATUS IS FILE-STAT.
     SELECT OUT-SORT ASSIGN TO PRIMOS.
     SELECT SORT-WK ASSIGN TO PRIMOS.
*****************************************************************
 DATA DIVISION.
 FILE SECTION.
*
 FD  IN-FILE,COMPRESSED,
     VALUE OF FILE-ID IS 'SEFILE'.
 01  ENTRY.
     05 CODE-IN                    PIC X.
     05 ACCT-IN                    PIC X(3).
     05 FILLER                     PIC X(76).
*
 FD  OUT-SORT,
     RECORD CONTAINS 80 CHARACTERS.
 01  SORTOUT                       PIC X(80).
*
 SD  SORT-WK,
     RECORD CONTAINS 80 CHARACTERS.
 01  SORT-REC.
     05 CODE-SD                    PIC X.
     05 ACCT-SD                    PIC X(3).
     05 FILLER                     PIC X(19).
     05 CAT-SD                     PIC XX.
     05 FILLER                     PIC X(55).
```

```
*
 WORKING-STORAGE SECTION.
 77  FILE-STAT                    PIC XX.
 77  NO-MORE-RECORDS              PIC X  VALUE 'N'.
*****************************************************************
 PROCEDURE DIVISION.
 MAINLINE SECTION.
 000-MAINLINE.
     PERFORM 020-SORT-TRANSACTIONS.
     STOP RUN.
*
 020-SORT-TRANSACTIONS.
     SORT SORT-WK ASCENDING KEY CODE-SD,
         DESCENDING KEY ACCT-SD,
         ASCENDING KEY CAT-SD,
         INPUT PROCEDURE IS 030-INPUT-PROC,
         GIVING OUT-SORT.
*
 030-INPUT-PROC SECTION.
 030-BEGIN.
      OPEN INPUT IN-FILE.
      READ IN-FILE INTO ENTRY,
         AT END DISPLAY 'EMPTY FILE'
         MOVE 'Y' TO NO-MORE-RECORDS.
      PERFORM 035-ERROR-CHECK UNTIL NO-MORE-RECORDS
          = 'Y'.
      CLOSE IN-FILE.
      GO TO 030-END.
*
 035-ERROR-CHECK.
     IF ACCT-IN NOT NUMERIC,
          DISPLAY '**ERROR: ***',
          DISPLAY ENTRY,
     ELSE RELEASE SORT-REC FROM ENTRY.
     READ IN-FILE INTO ENTRY,
          AT END DISPLAY 'END OF FILE' MOVE 'Y'
          TO NO-MORE-RECORDS.
  030-END.
     EXIT.
```

Input file:

```
2350JOSEPHINE BLOW      00 12345678
2400JOSEF BLO           03 12345678
1090JOSEPHINE BLOUGH    06 12345678
1090JOSEPH BLOUGH       07 12345678
1091JOSEPH BLOW         10 12345678
4A75JOSIP BLOUGH        33 12345678
1090JOSE BLOUGH         06 12345678
1091JOSEF BLOUGH        14 12345678
2400JOE BLOW            02 12345678
1090JOSEPH BLOUGH       05 12345678
```

Compiling, Linking, and Execution:

```
OK, COBOL85 SAMPLE.SORT -L
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]
[0 ERRORS IN PROGRAM: SAMPLE.SORT.COBOL85]

OK, BIND -LO SAMPLE.SORT -LI COBOL85LIB -LI VSRTLI -LI
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
BIND COMPLETE

OK, RESUME SAMPLE.SORT
**ERROR: ***
4A75JOSIP BLOUGH        33 12345678
END OF FILE
OK,
```

Sorted Output File (OUT-SORT):

```
OK, SLIST OUT-SORT
1091JOSEPH BLOW         10 12345678
1091JOSEF BLOUGH        14 12345678
1090JOSEPH BLOUGH       05 12345678
1090JOSEPHINE BLOUGH    06 12345678
1090JOSE BLOUGH         06 12345678
1090JOSEPH BLOUGH       07 12345678
2400JOE BLOW            02 12345678
2400JOSEF BLO           03 12345678
2350JOSEPHINE BLOW      00 12345678
```

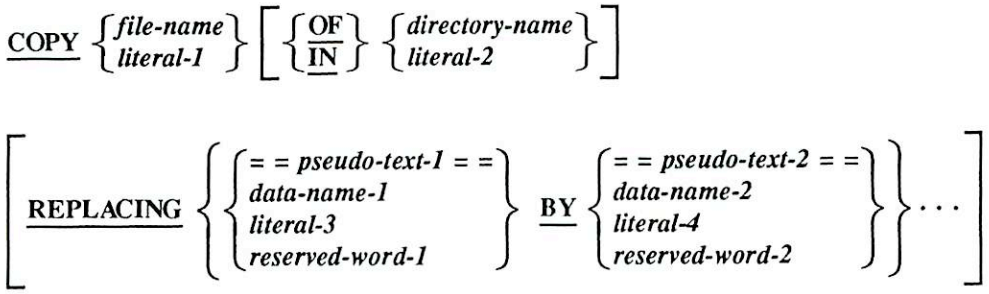
15 Source Text Manipulation

This chapter discusses the COPY source text manipulation statement. You can use this statement to insert and replace source program text during compilation.

COPY

Incorporates COBOL85 source coding from another file into a source program at compile time. This is a compiler-directing statement.

Format



Syntax Rules

1. A COPY statement can occur anywhere in the source program, in any division where a character-string or a separator (other than the closing quotation mark) can usually occur, except within the object of another COPY statement. It is not executed, however, if found in a comment-entry.
2. The COPY statement must be preceded by a space and terminated by a separator period.
3. OF and IN are interchangeable and mutually exclusive.
4. *file-name* must be the name of a PRIMOS file containing COBOL85 source code.

5. *literal-1* must contain a PRIMOS filename or a fully qualified pathname. *literal-1* can be partially qualified if used in conjunction with *literal-2*. *literal-2* must be a fully or partially qualified *directory-name*, so that concatenating it with *literal-1* forms a fully qualified pathname.
6. If you specify *directory-name*, it must be the name of the directory that contains *filename*.
7. You can specify passwords in *literal-1* or *literal-2*. For example,

```
COPY 'MYDIR>SUB PSWD>MYFILE'.
COPY MYFILE OF 'MYDIR>SUB PSWD'.
```

8. Files referenced in COPY statements can be located by the COBOL85 compiler based upon the INCLUDE$ search rules in effect for your environment. Files that are referenced as pathnames or with the OF or IN clause are located by those pathnames; files referenced as simple filenames are located using the PRIMOS search rules facility to provide the full pathnames. The system default is the current attach point. For more information see the section, COPY Files and the Search Rules Facility, later in this chapter.
9. *pseudo-text* is a literal string with no quotation marks, unless these quotation marks are to appear in the text. *pseudo-text* allows quotation marks to be copied.
10. *pseudo-text-1* must include at least one text word; *pseudo-text-2* can be null.
11. *pseudo-text-1* cannot consist entirely of a separator comma or a separator semicolon.
12. Character-strings within *pseudo-text-1* and *pseudo-text-2* can be continued.
13. *reserved-word-1* and *reserved-word-2* can be any single COBOL85 word except COPY.

General Rules

1. During compilation, COBOL85 processes all COPY statements before processing the resultant source program.
2. The COPY statement copies everything in the text file into the COBOL85 source program.
3. If you do not specify the REPLACING phrase, COBOL85 copies the text unchanged.

   If you specify the REPLACING phrase, COBOL85 copies the text and replaces each properly matched occurrence of *pseudo-text-1*, *data-name-1*, *literal-3*, and *reserved-word-1* in the text by the corresponding *pseudo-text-2*, *data-name-2*, *literal-4*, and *reserved-word-2*.
4. For purposes of matching, COBOL85 treats *data-name-1*, *literal-3*, and *reserved-word-1* as pseudo-text containing only *data-name-1*, *literal-3*, and *reserved-word-1*, respectively.
5. The comparison operation to determine text replacement occurs in the following manner:

   - The leftmost text word in the text file that is not a separator comma or a separator semicolon is the first text word used for comparison. Any text word or space preceding this text word is copied into the source program. Starting with the first text word for comparison and the first *pseudo-text-1*, *data-name-1*, *literal-3*, or *reserved-word-1* that you specify in the REPLACING phrase, COBOL85 compares the entire *pseudo-text-1*, *data-name-1*, *literal-3*, or *reserved-word-1* to an equivalent number of contiguous text words.

- *pseudo-text-1*, *data-name-1*, *literal-3*, or *reserved-word-1* matches the text if, and only if, the ordered sequence of text words that forms *pseudo-text-1*, *data-name-1*, *literal-3*, or *reserved-word-1* is equal, character for character, to the ordered sequence of text words. For purposes of matching, each occurrence of a separator comma, semicolon, or space in *pseudo-text-1*, *data-name-1*, *literal-3*, *reserved-word-1*, or in the text is considered a single space. Each sequence of one or more space separators is considered a single space.
- If no match occurs, the comparison is repeated with each next successive occurrence of *pseudo-text-1*, *data-name-1*, *literal-3*, or *reserved-word-1*, if any, in the REPLACING phrase until either a match is found or there is no successive occurrence of *pseudo-text-1*, *data-name-1*, *literal-3*, or *reserved-word-1*.
- When all occurrences of *pseudo-text-1*, *data-name-1*, *literal-3*, or *reserved-word-1* have been compared and no match has occurred, the leftmost text word is copied into the source program. The next successive text word is then considered as the leftmost text word, and the comparison cycle starts again with the first occurrence of *pseudo-text-1*, *data-name-1*, *literal-3*, or *reserved-word-1*.
- Whenever a match occurs between *pseudo-text-1*, *data-name-1*, *literal-3*, or *reserved-word-1* and the text, the corresponding *pseudo-text-2*, *data-name-2*, *literal-4*, or *reserved-word-2* is placed into the source program. The text word immediately following the rightmost text word that participated in the match is then considered as the leftmost text word. The comparison cycle starts again with the first occurrence of *pseudo-text-1*, *data-name-1*, *literal-3*, or *reserved-word-1*.
- The comparison operation continues until the rightmost text word in the text file has either participated in a match or been considered as a leftmost text word and participated in a complete comparison cycle.

6. Comment lines and blank lines occurring in the text file and in *pseudo-text-1* are ignored for purposes of matching; and the sequence of text words in the text file, if any, and in *pseudo-text-1* is determined by the coding rules defined in Chapter 4. Comment lines or blank lines in *pseudo-text-2* are copied into the resultant program unchanged whenever *pseudo-text-2* is placed into the source program as a result of text replacement. A comment line or blank line in the text file is not copied into the resultant program unchanged if that comment line or blank line appears within the sequence of text words that match *pseudo-text-1*.
7. Debugging lines are permitted within the text file and pseudo-text. Text words within a debugging line participate in the matching rules as if the 'D' or 'd' did not appear in the indicator area.
8. COBOL85 cannot independently determine the syntactic correctness of the text file. Except for COPY statements, COBOL85 cannot determine the syntactic correctness of the program until it first processes all COPY statements.
9. Each text word copied from the text file but not replaced is copied so as to start in the same area of the line in the resultant program as it begins in the line within the text file. However, if a text word copied from the text file begins in Area A but follows another text word that also begins in Area A of the same line, and if replacement of a preceding text word in the line by replacement text of greater length occurs, the following text word begins in Area B if it cannot begin in Area A. Each text word in *pseudo-text-2* that is to be placed into the resultant program begins in the same area of the resultant

program as it appears in *pseudo-text-2*. Each *data-name-2*, *literal-4*, or *reserved-word-2* that is to be placed into the resultant program begins in the same area of the resultant program as the leftmost text word that participated in the match would appear if it had not been replaced.

The text in the text file must conform to the coding rules defined in Chapter 4.

If additional lines are introduced into the source program as a result of a COPY statement, each text word introduced appears on a debugging line if the COPY statement begins on a debugging line or if the text word being introduced appears on a debugging line in the text file. In these cases, only those text words that are specified on debugging lines within *pseudo-text-2* appear on debugging lines in the resultant program. If any literal specified as *literal-2* or within *pseudo-text-2* or the text file is too long to be accommodated on a single line without continuation to another line in the resultant program, and the literal is not being placed on a debugging line, additional continuation lines are introduced, which contain the remainder of the literal. If replacement requires the continued literal to be continued on a debugging line, the program is in error.

10. Text words inserted into the source program as a result of the REPLACING phrase are placed in the source program according to the coding rules defined in Chapter 4. When copying text words of *pseudo-text-2* into the source program, you can introduce additional spaces only between text words that already have an existing space between them (including the assumed space between source lines).
11. If additional lines are introduced into the source program as a result of the processing of COPY statements, the indicator area of the introduced lines contains the same character as the line on which the text being replaced begins, unless that line contains a hyphen, in which case the introduced line contains a space. If a literal is continued onto an introduced line that is not a debugging line, a hyphen is placed in the indicator area.

Examples

In the following example, the first and second COPY statements copy files that are referenced by simple filenames. The third and fourth COPY statements copy files contained in a directory named MYDIR. The fifth COPY statement uses a complete pathname. The last two COPY statements contain a directory name and a filename with a period. Because the period has special significance in COBOL85, enclose the entry in quotation marks to avoid errors.

```
FILE-CONTROL.  COPY file-name-1.
DATA DIVISION.
FILE SECTION.
COPY file-name-2.
FD MASTER-FILE COPY file-name-3 OF MYDIR.
01 MASTER-RECORD.  COPY 'my.file' IN MYDIR.
01 HEADER-RECORD.  COPY "MYDIR>COPYBOOKS>file-name-4".
PARAGRAPH-NAME.
COPY file-name-5 IN 'ANNE.F'.
COPY "init.ins.COBOL85".
```

The following example is an excerpt from DATA DIVISION coding in a source program.

```
01  MASTER-DESCRIPTION.  COPY MASDES
                         REPLACING ==03== BY ==05==
                                ==PIC X(15)== BY ==PIC X(20)==.
01  EMPLOYMENT-HISTORY.
```

The file MASDES must not contain the 01 MASTER-DESCRIPTION entry; it can have the following format:

```
03  BADGE-NO  PIC 9(5).
03  NAME.
    10  LAST-NAME  PIC X(15).
    10  FIRST-NAME  PIC X(15).
```

After compilation, the listing file includes the following:

```
    60          .
    61          .
    62    01  MASTER-DESCRIPTION.  COPY MASDES
    63                             REPLACING ==03== BY ==05==
    64                                   ==PIC X(15)== BY ==PIC X(20)==
<      1>     05  BADGE-NO  PIC 9(5).
<      2>     05  NAME.
<      3>         10  LAST-NAME  PIC X(20).
<      4>         10  FIRST-NAME  PIC X(20).
    65    01  EMPLOYMENT-HISTORY.
    66          .
    67          .
```

In this example, the COPY statement part of lines 62-64 is a comment only. Line numbering of the inserted text is independent of the line numbers of the source.

COPY Files and the Search Rules Facility

The PRIMOS search rules facility enables you to establish an **INCLUDE$ search list**. An INCLUDE$ search list is a list of directories that are to be searched for a file whenever a COPY statement is executed. (Although there are several kinds of search lists, this section explains only the INCLUDE$ search list. For complete information about the PRIMOS search rules facility, see the *Advanced Programmer's Guide, Volume II*.)

When you specify a file in a COPY statement, you must ordinarily give as much of the file pathname as PRIMOS needs to locate the file. If you use COPY files often, and if the files are kept in a number of different directories, keeping track of the file pathnames can be difficult. Now, however, you can locate COPY files by supplying only a filename and using the search rules facility to provide the full pathname.

Establishing Search Rules: To establish search rules for COPY files, perform the following steps:

1. Create a **template** file called

   [yourchoice.]INCLUDE$.SR

This file should contain a list of the pathnames of the directories that contain your COPY files. For example, you might create a file called MY.INCLUDE$.SR that contains the following directory names:

```
<SYS1>MASTER_DIR>INSERT_FILES
<SYS2>MYUFD>COPYBOOKS
```

2. Activate the template file by using the SET_SEARCH_RULES command. For example, if your file is named MY.INCLUDE$.SR, type

```
OK, SSR MY.INCLUDE$
```

This command sets the INCLUDE$ search list for your process. This search list may contain system search rules and administrator search rules in addition to the rules you specified in MY.INCLUDE$.SR.

When you give the SSR command shown in step 2, PRIMOS copies the contents of MY.INCLUDE$.SR into your INCLUDE$ search list. If you have no special system or administrator search rules, your INCLUDE$ search list appears as follows when you give the LIST_SEARCH_RULES command:

```
List: INCLUDE$
Pathname of template: <MYSYS>MYUFD>COBOL>MY.INCLUDE$.SR

    [home_dir]
    <SYS1>MASTER_DIR>INSERT_FILES
    <SYS2>MYUFD>COPYBOOKS
```

[home_dir], your current attach point, is the system default. It is always the first directory searched, unless you remove it from the list or change the order of evaluation by using the –NO_SYSTEM option of the SSR command. Additional search rules, established as system-wide defaults by your System Administrator, may also appear at the beginning of your INCLUDE$ search list.

The SET_SEARCH_RULES and LIST_SEARCH_RULES commands are described in the *PRIMOS Commands Reference Guide*. For more information about establishing search rules, see the *Advanced Programmer's Guide, Volume II*.

Using Search Rules: Once you have set the search list, any COPY statement in a program can reference just the filename rather than the full pathname of the file. PRIMOS then searches the contents of the directories in the INCLUDE$ search list for the filename specified in the COPY statement. If PRIMOS finds the file, it stops searching and returns the full pathname of the file to the compiler. The compiler then uses this pathname to locate the file and inserts its contents into the source program.

Appendices

A

COBOL85 Formats

This appendix contains formats for all IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, and DATA DIVISION entries. It also contains formats for all PROCEDURE DIVISION verbs. Within each division the formats are listed in alphabetical order.

IDENTIFICATION DIVISION

```
{IDENTIFICATION DIVISION.}
{ID DIVISION.            }

PROGRAM-ID. program-name.

[AUTHOR. [comment-entry] . . .]

[INSTALLATION. [comment-entry] . . .]

[DATE-WRITTEN. [comment-entry] . . .]

[DATE-COMPILED. [comment-entry] . . .]

[SECURITY. [comment-entry] . . .]

[REMARKS. [comment-entry] . . .]
```

ENVIRONMENT DIVISION

```
ENVIRONMENT DIVISION.

[ CONFIGURATION SECTION.
  [SOURCE-COMPUTER. [computer-name.]]
  [ OBJECT-COMPUTER. [computer-name]
    [object-computer-entry].
  [ SPECIAL-NAMES.
    [special-names-entry] ... ]
]

[ INPUT-OUTPUT SECTION.
  [ FILE-CONTROL.
    {file-control-entry} ... ]
  [ I-O-CONTROL.
    [I-O-control-entry] ]
]
```

I-O-CONTROL

```
I-O-CONTROL.

[ SAME [ RECORD
         SORT
         SORT-MERGE ] AREA FOR file-name-1, {file-name-2} ... ] ...

[ RERUN [ON file-name-1]
  EVERY { { END OF { REEL } } OF file-name-3
          { UNIT   }
          { integer-4 RECORDS }
          { integer-5 CLOCK-UNITS }
          { condition-name } } ] ...

[ MULTIPLE FILE TAPE CONTAINS file-name-4
      [POSITION integer-6]
      [ file-name-5 [POSITION integer-7]] ... ] ...
```

OBJECT-COMPUTER

```
OBJECT-COMPUTER. [computer-name]

[ , MEMORY SIZE integer { WORDS      } ]
                        { CHARACTERS }
                        { MODULES    }

[ , PROGRAM COLLATING SEQUENCE IS alphabet-name-1]

[ , SEGMENT-LIMIT IS segment-number].
```

SELECT

Format 1

```
SELECT [OPTIONAL] file-name-1
  ASSIGN TO { device-name }
            { literal-1   }

[ RESERVE integer-1 [ AREA  ] ]
                    [ AREAS ]

[ [ORGANIZATION IS] SEQUENTIAL]

[ ACCESS MODE IS SEQUENTIAL]

[ FILE STATUS IS data-name-1].
```

Format 2

```
SELECT [OPTIONAL] file-name-1
  ASSIGN TO { device-name }
            { literal-1   }

[ RESERVE integer-1 [ AREA  ] ]
                    [ AREAS ]

[ORGANIZATION IS] RELATIVE

[ ACCESS MODE IS { SEQUENTIAL [, RELATIVE KEY IS data-name-1] } ]
                 { { RANDOM  } , RELATIVE KEY IS data-name-1   }
                 { { DYNAMIC }                                 }

[ FILE STATUS IS data-name-2].
```

Format 3

```
SELECT [OPTIONAL] file-name-1
  ASSIGN TO { device-name }
            { literal-1   }

[ RESERVE integer-1 [ AREA  ] ]
                    [ AREAS ]

[ORGANIZATION IS] INDEXED

[ ACCESS MODE IS { SEQUENTIAL } ]
                 { RANDOM     }
                 { DYNAMIC    }

RECORD KEY IS data-name-1

[ ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLICATES]] . . .

[ FILE STATUS IS data-name-3].
```

SPECIAL-NAMES

```
SPECIAL-NAMES.
[
[CONSOLE IS mnemonic-name-1] . . .

[ switch-name [IS mnemonic-name-2]
  [ { ON STATUS IS condition-name-1 [OFF STATUS IS condition-name-2] } ] ] . . .
  [ { OFF STATUS IS condition-name-2 [ON STATUS IS condition-name-1] } ]

[ ALPHABET alphabet-name-1 IS { STANDARD-1 } ] . . .
                              { STANDARD-2 }
                              { NATIVE     }
                              { EBCDIC     }

[ CLASS class-name-1 IS { literal-1 [ { THROUGH } literal-2 ] } . . . ] . . .
                        {            { THRU    }           }

[CURRENCY SIGN IS literal-3]

[DECIMAL-POINT IS COMMA].
]
```

SOURCE-COMPUTER

SOURCE-COMPUTER. [*computer-name* [WITH DEBUGGING MODE].]

DATA DIVISION

DATA DIVISION.

[FILE SECTION.

[*file-description-entry.*
[*record-description-entry*] . . .] . . .

[*sort-merge-file-description-entry.*
{*record-description-entry*} . . .] . . .]

[WORKING-STORAGE SECTION.

[*level-77-data-description-entry*
data-description-entry] . . .]

[LINKAGE SECTION.

[*level-77-data-description-entry*
data-description-entry] . . .]

BLANK

BLANK WHEN ZERO

BLOCK

BLOCK CONTAINS [*integer-1* TO] *integer-2* {RECORDS / CHARACTERS}

CODE-SET

CODE-SET IS *alphabet-name*

COMPRESSED/UNCOMPRESSED

```
FD file-name [ { COMPRESSED   } ]
             [ { UNCOMPRESSED } ]
```

data-name or FILLER

```
[ data-name ]
[ FILLER    ]
```

DATA RECORDS

```
DATA { RECORD IS  } data-name-1 [, data-name-2] . . .
     { RECORDS ARE }
```

EXTERNAL

```
IS EXTERNAL
```

file-description-entry

```
FD file-name [ { COMPRESSED    } ]
             [ { UNCOMPRESSED  } ]

[ ; IS EXTERNAL]

[ ; BLOCK CONTAINS [integer-1 TO] integer-2 { RECORDS    } ]
                                            { CHARACTERS } ]

[ ; CODE-SET IS alphabet-name]

[ ; DATA { RECORD IS  } data-name-1 [, data-name-2] . . . ]
         { RECORDS ARE }

[ ; LABEL { RECORD IS  } { STANDARD } ]
          { RECORDS ARE } { OMITTED  }

[ ; RECORD { CONTAINS integer-3 CHARACTERS                                    } ]
           { IS VARYING IN SIZE [[FROM integer-4] [TO integer-5] CHARACTERS] }
           { CONTAINS integer-6 TO integer-7 CHARACTERS                      }
           { IS NOT VARYING IN SIZE                                          }

[ ; RECORDING MODE IS {F, U, S, V}]

[ ; VALUE OF FILE-ID IS { data-name-3 } . . . ]
                        { literal-1   }

[record-description-entry] . . .
```

FILE SECTION

```
FILE SECTION.

[ file-description-entry. {record-description-entry} . . .                  ] . . .
[ sort-merge-file-description-entry. {record-description-entry} . . .       ]
```

LABEL RECORDS

```
LABEL { RECORD IS  } { STANDARD }
      { RECORDS ARE} { OMITTED  }
```

level-number

level-number

LINKAGE SECTION

```
LINKAGE SECTION.

[ level-77-description-entry ]
[ record-description-entry   ] ...
```

OCCURS

Format 1

```
OCCURS integer-2 TIMES

[ { ASCENDING  } KEY IS data-name-2 [, data-name-3] ... ] ...
  { DESCENDING }

[INDEXED BY index-name-1 [, index-name-2] ... ]
```

Format 2

```
OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name-1

[ { ASCENDING  } KEY IS data-name-2 [, data-name-3] ... ] ...
  { DESCENDING }

[INDEXED BY index-name-1 [, index-name-2] ... ]
```

PICTURE

```
{ PICTURE } IS character-string
{ PIC     }
```

record-description-entry

Format 1

level-number [*data-name-1* | FILLER]

[; BLANK WHEN ZERO]

[; IS EXTERNAL]

[; { JUSTIFIED | JUST } RIGHT]

[; OCCURS { *integer-1* TO *integer-2* TIMES DEPENDING ON *data-name-2* | *integer-2* TIMES }
[{ ASCENDING | DESCENDING } KEY IS *data-name-3* [, *data-name-4*] . . .] . . .
[INDEXED BY *index-name-1* [, *index-name-2*] . . .]]

[; { PICTURE | PIC } IS *character-string*]

[; REDEFINES *data-name-5*]

[; [SIGN IS] { LEADING | TRAILING } [SEPARATE CHARACTER]]

[; { SYNCHRONIZED | SYNC } [LEFT | RIGHT]]

[; [USAGE IS] { BINARY | COMPUTATIONAL | COMP | COMPUTATIONAL-1 | COMP-1 | COMPUTATIONAL-2 | COMP-2 | COMPUTATIONAL-3 | COMP-3 | DISPLAY | INDEX | PACKED-DECIMAL }]

[; VALUE IS *literal*]

Format 2

66 *data-name-1* ; RENAMES *data-name-2* [{ THROUGH / THRU } *data-name-3*]

Format 3

88 *condition-name*; { VALUE IS / VALUES ARE } *literal-1* [{ THROUGH / THRU } *literal-2*]

[, *literal-3* [{ THROUGH / THRU } *literal-4*]] . . .

RECORD

Format 1

RECORD CONTAINS *integer-1* CHARACTERS

Format 2

RECORD IS VARYING IN SIZE [[FROM *integer-2*] [TO *integer-3*] CHARACTERS]

Format 3

RECORD CONTAINS *integer-4* TO *integer-5* CHARACTERS

Format 4

RECORD IS NOT VARYING IN SIZE

RECORDING MODE

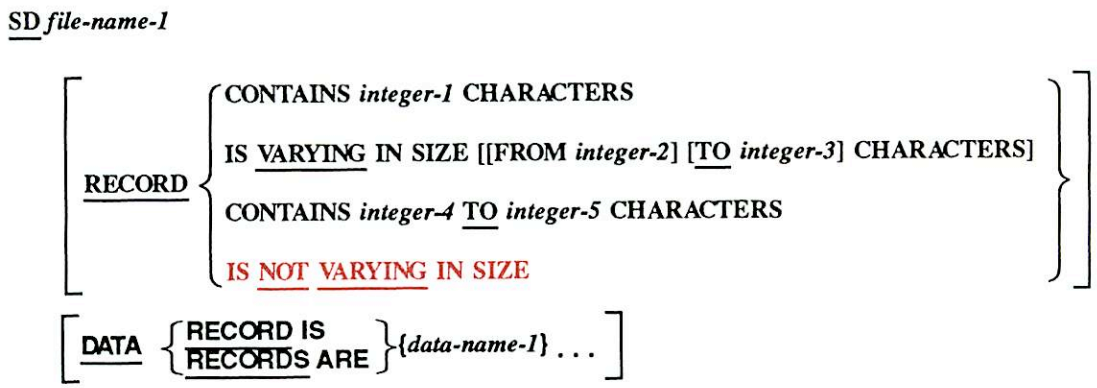
RECORDING MODE IS {F, U, S, V}

REDEFINES

level-number [*data-name-1* / FILLER] [; REDEFINES *data-name-2*]

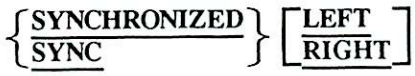
RENAMES

sort-merge-file-description-entry



SIGN

SYNCHRONIZED



USAGE

```
[USAGE IS] { BINARY
             COMPUTATIONAL
             COMP
             COMPUTATIONAL-1
             COMP-1
             COMPUTATIONAL-2
             COMP-2
             COMPUTATIONAL-3
             COMP-3
             DISPLAY
             INDEX
             PACKED-DECIMAL }
```

VALUE

Format 1

```
VALUE IS literal
```

Format 2

```
{VALUE IS  } literal-1 [ {THROUGH} literal-2 ]
{VALUES ARE}           [ {THRU   }           ]

    [ , literal-3 [ {THROUGH} literal-4 ] ] ...
                  [ {THRU   }           ]
```

VALUE OF FILE-ID

```
VALUE OF FILE-ID IS { data-name-3 }
                    { literal-2   }
```

WORKING-STORAGE SECTION

```
WORKING-STORAGE SECTION.

[ level-77-description-entry ] ...
[ record-description-entry   ]
```

PROCEDURE DIVISION

```
PROCEDURE DIVISION [USING data-name-1 [, data-name-2] . . . [data-name-64]] .

[DECLARATIVES.
 {section-name SECTION [segment-number]. USE-sentence.
 {[paragraph-name. [sentence] . . . ] . . .} . . .

END DECLARATIVES.]

{  [section-name SECTION [segment-number].]
 {[paragraph-name. [sentence] . . .] . . .}
 {[sentence] . . .                        } . . .
```

ACCEPT

Format 1

```
ACCEPT data-name [FROM mnemonic-name]
```

Format 2

```
ACCEPT data-name FROM {DATE}
                      {DAY }
                      {TIME}
```

ADD

Format 1

```
ADD {data-name-1 } [, data-name-2 ]
    {literal-1   }  [, literal-2   ] . . .
    {arith-expr-1}  [, arith-expr-2]

  TO data-name-3 [ROUNDED] [, data-name-n [ROUNDED]] . . .

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-ADD]
```

Format 2

```
ADD { data-name-1 } [ , data-name-2 ]  . . .  TO { data-name-3 }
    { literal-1   }  [ , literal-2   ]           { literal-3   }
    { arith-expr-1}  [ , arith-expr-2]           { arith-expr-3}

  GIVING {data-name-4 [ROUNDED]} [, data-name-n [ROUNDED]] . . .

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-ADD]
```

Format 3

```
ADD { CORRESPONDING } data-name-1 TO data-name-2 [ROUNDED]
    { CORR          }

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-ADD]
```

ALTER

```
ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2

  [, procedure-name-3 TO [PROCEED TO] procedure-name-4] . . .
```

CALL

```
CALL { data-name-1 } [USING data-name-2 [, data-name-3] . . .]
     { literal-1   }

  [ON OVERFLOW imperative-statement-1]

  [END-CALL]
```

CANCEL

```
CANCEL { identifier-1 } [ , identifier-2 ]  . . .
       { literal-1    }  [ , literal-2    ]
```

CLOSE

Format 1

```
CLOSE file-name-1 [,file-name-2] . . .
```

Format 2

```
CLOSE { file-name-1 [ REEL                ] } . . .
      {             [ UNIT                ] }
      {             [ WITH NO REWIND      ] }
```

COMPUTE

Format 1

```
COMPUTE data-name-1 [ROUNDED] [, data-name-2 [ROUNDED]] . . . = arith-expr

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-COMPUTE]
```

Format 2

```
COMPUTE { CORRESPONDING } data-name-1 [ROUNDED] = data-name-2
        { CORR          }

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-COMPUTE]
```

CONTINUE

```
CONTINUE
```

COPY

```
COPY { file-name   } [ { OF } { directory-name } ]
     { literal-1   }   { IN } { literal-2      }

[ REPLACING { = = pseudo-text-1 = = } BY { = = pseudo-text-2 = = } ] ...
            { data-name-1           }    { data-name-2           }
            { literal-3             }    { literal-4             }
            { reserved-word-1       }    { reserved-word-2       }
```

DECLARATIVES

```
DECLARATIVES.
{ section-name SECTION [segment-number]. USE-sentence.  } ...
{ [paragraph-name. [sentence] ... ] ...                 }

END DECLARATIVES.
```

DELETE

```
DELETE file-name RECORD

[INVALID KEY imperative-statement-1]

[NOT INVALID KEY imperative-statement-2]

[END-DELETE]
```

DISPLAY

```
DISPLAY { data-name-1 } [ , data-name-2 ] ... [UPON mnemonic-name]
        { literal-1   }   { , literal-2 }

[[WITH] NO ADVANCING]
```

DIVIDE

Format 1

```
DIVIDE { data-name-1 } INTO data-name-2 [ROUNDED] [, data-name-3 [ROUNDED]] ...
       { literal-1   }
       { arith-expr-1}

[ON SIZE ERROR imperative-statement-1]

[NOT ON SIZE ERROR imperative-statement-2]

[END-DIVIDE]
```

Format 2

```
DIVIDE { data-name-1 } { INTO } { data-name-2 }
       { literal-1   } { BY   } { literal-2   }
       { arith-expr-1}          { arith-expr-2}

GIVING data-name-3 [ROUNDED] [, data-name-4 [ROUNDED]] ...

[ON SIZE ERROR imperative-statement-1]

[NOT ON SIZE ERROR imperative-statement-2]

[END-DIVIDE]
```

Format 3

```
DIVIDE { data-name-1 } { INTO } { data-name-2 }
       { literal-1   } { BY   } { literal-2   }
       { arith-expr-1}          { arith-expr-2}

GIVING data-name-3 [ROUNDED]

REMAINDER data-name-4

[ON SIZE ERROR imperative-statement-1]

[NOT ON SIZE ERROR imperative-statement-2]

[END-DIVIDE]
```

Format 4

DIVIDE { CORRESPONDING / CORR } *data-name-1* { INTO / BY } *data-name-2* [ROUNDED]

[ON SIZE ERROR *imperative-statement-1*]

[NOT ON SIZE ERROR *imperative-statement-2*]

[END-DIVIDE]

EJECT

EJECT

ENTER

ENTER *language-name* [*routine-name*].

EXHIBIT

EXHIBIT { *literal* / [NAMED] *data-name* } ...

EXIT

EXIT.

EXIT PROGRAM

EXIT PROGRAM.

GOBACK

GOBACK

GO TO

Format 1

GO TO [*procedure-name*]

Format 2

GO TO *procedure-name-1* [, *procedure-name-2*] · · · [, *procedure-name-n*]

DEPENDING ON { *data-name* / *arith-expr* }

IF

IF CORRESPONDING *condition-1* [THEN] { *statement-1* / NEXT SENTENCE }

[{ ELSE {*statement-2*} · · · [END-IF] / OTHERWISE {*statement-2*} · · · [END-IF] / ELSE NEXT SENTENCE / OTHERWISE NEXT SENTENCE / END-IF }]

INSPECT

Format 1

INSPECT *data-name-1* TALLYING

{ *data-name-2* FOR { { { ALL / LEADING } { *data-name-3* / *literal-1* } / CHARACTERS } [{ BEFORE / AFTER } INITIAL { *data-name-4* / *literal-2* }] } · · · } · · ·

Format 2

INSPECT *data-name-1* REPLACING

{ CHARACTERS BY { *data-name-6* / *literal-4* } [{ BEFORE / AFTER } INITIAL { *data-name-7* / *literal-5* }]

{ { { ALL / LEADING / FIRST } { { *data-name-5* / *literal-3* } BY { *data-name-6* / *literal-4* } [{ BEFORE / AFTER } INITIAL { *data-name-7* / *literal-5* }] } · · · } · · · }

Format 3

```
INSPECT data-name-1 TALLYING

{ data-name-2 FOR { { {ALL|LEADING} {data-name-3|literal-1} }|CHARACTERS [ {BEFORE|AFTER} INITIAL {data-name-4|literal-2} ] } ... } ...

REPLACING

{ CHARACTERS BY {data-name-6|literal-4} [ {BEFORE|AFTER} INITIAL {data-name-7|literal-5} ]
{ { {ALL|LEADING|FIRST} } { {data-name-5|literal-3} } BY {data-name-6|literal-4} [ {BEFORE|AFTER} INITIAL {data-name-7|literal-5} ] } ... } ... }
```

MERGE

```
MERGE file-name-1 ON {ASCENDING|DESCENDING} KEY data-name-1 [, data-name-2] ...

[ ON {ASCENDING|DESCENDING} KEY data-name-3 [, data-name-4] ... ] ...

[COLLATING SEQUENCE IS alphabet-name]

USING file-name-2, file-name-3 [, file-name-4] ...

{ OUTPUT PROCEDURE IS procedure-name-1 [ {THROUGH|THRU} procedure-name-2 ]
{ GIVING file-name-5 [, file-name-6] ... }
```

MOVE

Format 1

```
MOVE {data-name-1|literal|arith-expr} TO data-name-2 [ROUNDED] [, data-name-3 [ROUNDED]] ...
```

Format 2

```
MOVE { CORRESPONDING } data-name-1 TO data-name-2 [ROUNDED]
     { CORR          }
```

MULTIPLY

Format 1

```
MULTIPLY { data-name-1   } BY data-name-2 [ROUNDED] [, data-name-3 [ROUNDED]] ...
         { literal-1     }
         { arith-expr-1  }

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-MULTIPLY]
```

Format 2

```
MULTIPLY { data-name-1   } BY { data-name-2   }
         { literal-1     }    { literal-2     }
         { arith-expr-1  }    { arith-expr-2  }

  GIVING data-name-3 [ROUNDED] [, data-name-4 [ROUNDED] ...

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-MULTIPLY]
```

Format 3

```
MULTIPLY { CORRESPONDING } data-name-1 BY data-name-2 [ROUNDED]
         { CORR          }

  [ON SIZE ERROR imperative-statement-1]

  [NOT ON SIZE ERROR imperative-statement-2]

  [END-MULTIPLY]
```

NOTE

NOTE *comment-entry.*

OPEN

Format 1

```
OPEN { INPUT    file-name-1 [, file-name-2] ... }
     { OUTPUT   file-name-3 [, file-name-4] ... }
     { I-O      file-name-5 [, file-name-6] ... }  ...
     { EXTEND   file-name-7 [, file-name-8] ... }
```

Format 2

```
OPEN { INPUT    file-name-1 [, file-name-2] ... }
     { OUTPUT   file-name-3 [, file-name-4] ... }  ...
     { I-O      file-name-5 [, file-name-6] ... }
```

Format 3

```
OPEN { INPUT   {file-name-1 [WITH NO REWIND]} ... }
     { OUTPUT  {file-name-2 [WITH NO REWIND]} ... }  ...
```

PERFORM

Format 1

```
PERFORM [ procedure-name-1 [ { THROUGH } procedure-name-2 ] ]
                             { THRU    }
```

[*imperative-statement-1* END-PERFORM]

Format 2

```
PERFORM [ procedure-name-1 [ { THROUGH } procedure-name-2 ] ]
                             { THRU    }

{ integer      }
{ data-name-1  } TIMES
{ arith-expr-1 }
```

[*imperative-statement-1* END-PERFORM]

Format 3

```
PERFORM [ procedure-name-1 [ { THROUGH } procedure-name-2 ] ]
        [ { THRU    }                                      ]

  UNTIL condition-1

[imperative-statement-1 END-PERFORM]
```

Format 4

```
PERFORM [ procedure-name-1 [ { THROUGH } procedure-name-2 ] ]
        [ { THRU    }                                      ]

VARYING { data-name-2  } FROM { data-name-3  }
        { index-name-1 }      { index-name-2 }
                              { literal-1    }
                              { arith-expr-1 }

   BY { data-name-4  } UNTIL condition-1
      { literal-2    }
      { arith-expr-2 }

[  AFTER { data-name-5  } FROM { data-name-6  }
         { index-name-3 }      { index-name-4 }
                               { literal-3    }
                               { arith-expr-2 }

   BY { data-name-7  } UNTIL condition-2 ] ...
      { literal-4    }
      { arith-expr-4 }

[imperative-statement-1 END-PEFORM]
```

READ

Format 1

READ *file-name* RECORD [INTO *data-name-1*]

[AT END *imperative-statement-1*]

[NOT AT END *imperative-statement-2*]

[END-READ]

Format 2

READ *file-name* [NEXT] RECORD [INTO *data-name-1*]

[AT END *imperative-statement-1*]

[NOT AT END *imperative-statement-2*]

[END-READ]

Format 3

READ *file-name* RECORD [INTO *data-name-1*]

[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-READ]

Format 4

READ *file-name* RECORD [INTO *data-name-1*]

[KEY IS *data-name-2*]

[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-READ]

READY TRACE

READY TRACE.

RELEASE

RELEASE *record-name* [FROM *data-name*]

RESET TRACE

RESET TRACE.

RETURN

RETURN *file-name* RECORD [INTO *data-name-1*]

AT END *imperative-statement-1*

[NOT AT END *imperative-statement-2*]

[END-RETURN]

REWRITE

Format 1

REWRITE *record-name* [FROM *data-name*]

[END-REWRITE]

Format 2

REWRITE *record-name* [FROM *data-name*]

[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-REWRITE]

SEARCH

Format 1

```
SEARCH data-name-1 [ VARYING { data-name-2  } ]
                             { index-name-1 }

[AT END imperative-statement-1]

{ WHEN condition-1 { imperative-statement-2 } } ...
                   { NEXT SENTENCE          }

[END-SEARCH]
```

Format 2

```
SEARCH ALL data-name-1 [AT END imperative-statement-1]

WHEN { data-name-2 { IS EQUAL TO } { data-name-3 }
                   { IS =        } { literal-1   }
                                   { arith-expr-1 }
     { condition-name-1 }

[ AND { data-name-4 { IS EQUAL TO } { data-name-5 } } ] ...
      {             { IS =        } { literal-2   } }
      { condition-name-2          } { arith-expr-2 }

{ imperative-statement-2 }
{ NEXT SENTENCE          }

[END-SEARCH]
```

SEEK

```
SEEK file-name RECORD
```

SET

Format 1

```
SET { index-name-1 [, index-name-2] ... } TO { index-name-3 }
    { data-name-1 [, data-name-2] ...  }     { data-name-3 }
                                             { integer-1   }
                                             { arith-expr-1 }
```

Format 2

```
SET index-name-4 [, index-name-5] ... { UP BY   } { data-name-4 }
                                      { DOWN BY } { integer-2   }
                                                  { arith-expr-2 }
```

Format 3

```
SET { {mnemonic-name-1} ... TO { ON  } } ...
                               { OFF }
```

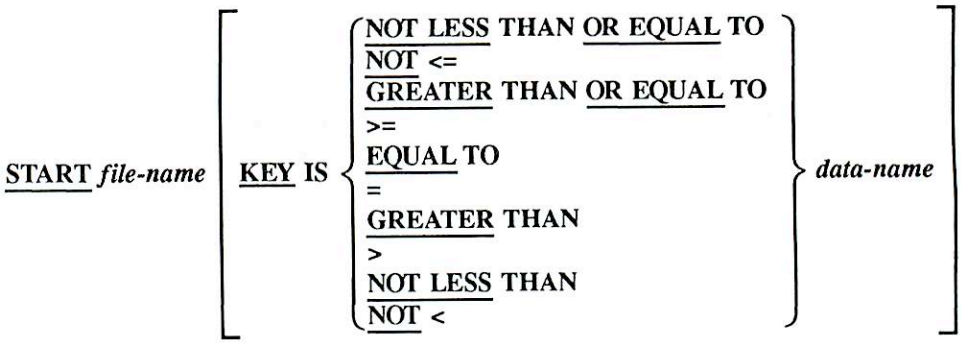
SKIP

```
SKIPn
```

SORT

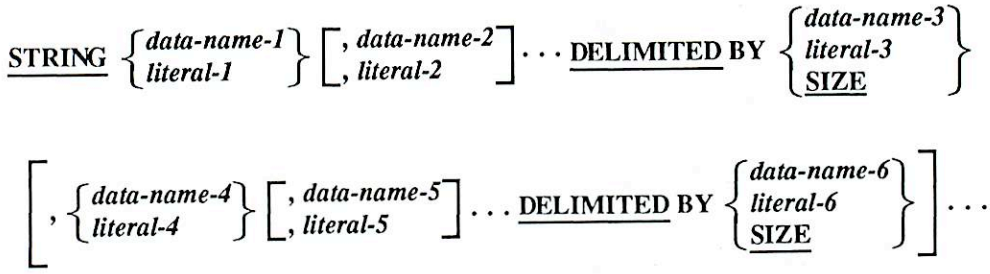
```
SORT file-name-1 { ON { ASCENDING  } KEY {data-name-1} ... } ...
                     { DESCENDING }

 [WITH DUPLICATES IN ORDER]

 [COLLATING SEQUENCE IS alphabet-name-1]

{ INPUT PROCEDURE IS procedure-name-1 [ { THROUGH } procedure-name-2 ] }
                                       { THRU    }
{ USING {file-name-2} ...                                              }

{ OUTPUT PROCEDURE IS procedure-name-3 [ { THROUGH } procedure-name-4 ] }
                                        { THRU    }
{ GIVING {file-name-3} ...                                              }
```

START



STOP

STRING



SUBTRACT

Format 1

```
SUBTRACT { data-name-1   } [ , data-name-2   ] ...
         { literal-1     }   , literal-2
         { arith-expr-1  }   , arith-expr-2
```

FROM *data-name-3* [ROUNDED] [, *data-name-n* [ROUNDED]] . . .

[ON SIZE ERROR *imperative-statement-1*]

[NOT ON SIZE ERROR *imperative-statement-2*]

[END-SUBTRACT]

Format 2

```
SUBTRACT { data-name-1   } [ , data-name-2   ] ...
         { literal-1     }   , literal-2
         { arith-expr-1  }   , arith-expr-2

FROM { data-name-3   } GIVING data-name-6 [ROUNDED] [, data-name-7 [ROUNDED]] . . .
     { literal-3     }
     { arith-expr-3  }
```

[ON SIZE ERROR *imperative-statement-1*]

[NOT ON SIZE ERROR *imperative-statement-2*]

[END-SUBTRACT]

Format 3

```
SUBTRACT { CORRESPONDING } data-name-1
         { CORR          }
```

FROM *data-name-2* [ROUNDED]

[ON SIZE ERROR *imperative-statement-1*]

[NOT ON SIZE ERROR *imperative-statement-2*]

[END-SUBTRACT]

UNSTRING

```
UNSTRING data-name-1

[ DELIMITED BY [ALL] { data-name-2 } [ , OR [ALL] { data-name-3 } ] ... ]
                     { literal-1   }               { literal-2   }

INTO data-name-4 [, DELIMITER IN data-name-5] [, COUNT IN data-name-6]

[, data-name-7 [, DELIMITER IN data-name-8] [, COUNT IN data-name-9]] ...

[WITH POINTER data-name-10] [TALLYING IN data-name-11]

[ON OVERFLOW imperative-statement-1]

[NOT ON OVERFLOW imperative-statement-2]

[END-STRING]
```

USE

```
USE AFTER STANDARD { EXCEPTION } PROCEDURE ON { file-name-1 [, file-name-2] ... }
                   { ERROR     }              { INPUT                          }
                                              { OUTPUT                         }
                                              { I-O                            }
                                              { EXTEND                         }.
```

WRITE

Format 1

```
WRITE record-name [FROM data-name-1]

[ { AFTER  } ADVANCING { { { data-name-2 } [LINE ] } } ]
  { BEFORE }           { { { integer     } [LINES] } }
                       { PAGE                       }

[END-WRITE]
```

Format 2

WRITE *record-name* [FROM *data-name*]

[INVALID KEY *imperative-statement-1*]

[NOT INVALID KEY *imperative-statement-2*]

[END-WRITE]

Format 3

WRITE *record-name* [FROM *data-name-1*]

[END-WRITE]

B

Reference Tables

The following tables are included in Appendix B:

- COBOL85 Symbols (Table B-1)
- COBOL85 Reserved Words (Table B-2)
- Prime Extended Character Set (Table B-3)
- Standard-1 ASCII Character Set (Table B-4)
- Standard-2 ASCII Character Set (Table B-5)
- EBCDIC Character Set and Collating Sequence (Table B-6)
- Availability of a File (Table B-7)
- Permissible Input-Output Statements After OPEN Options and Access Modes (Table B-8)
- Hexadecimal and Decimal Conversion Table (Table B-9)
- Octal and Decimal Conversion Table (Table B-10)
- Hexadecimal Addition Table (Table B-11)
- Decimal Data Type (Overpunch Symbols) (Table B-12)

TABLE B-1
COBOL85 Symbols

Symbol		*Functions*
Punctuation Symbols — Used to punctuate program entries		
.	Period	1. Terminates entries. Usually required. 2. Signifies the decimal point in numeric literals, or the comma in European notation.
,	Comma	1. Separates operands or clauses in a series. Optional. 2. European notation for the decimal point in numeric literals.
;	Semicolon	Separates operands or clauses in a series. Optional.
"	Quotation marks	Encloses nonnumeric literals.
'	Apostrophe or single quote mark	**Prime Extension**: Encloses nonnumeric literals.
==	Pseudo delimiter	Pseudo text delimiter used as a separator in COPY REPLACING statements.
Coding Symbols — Directives to the compiler		
*	Asterisk	Denotes an explanatory comment line when inserted in column 7 of a source program line.
/	Slash	Denotes a skip to the top of a new page during a source listing, when coded in column 7 of a source program line.
\	Backslash	Denotes the beginning of a nonnumeric literal mnemonic that corresponds to a character value in the Prime ECS chararacter set.
-	Hyphen	Denotes a continuation line when coded in column 7 of a source program line.
d	or D	When coded in column 7 of a source program line, denotes a line that COBOL85 treats as a comment line when you compile the program without the –DEBUG option, or when you omit the WITH DEBUGGING MODE clause in the SOURCE-COMPUTER paragraph. When you compile the program with the –DEBUG option, or when you specify the WITH DEBUGGING MODE clause in the SOURCE-COMPUTER paragraph, COBOL85 treats the line as a normal source statement.

TABLE B-1
COBOL85 Symbols - Continued

Symbol		Functions
Sign Symbols and Unary Operators — Used in numeric literals and arithmetic expressions		
+	Plus	1. Sign character in the high-order (leftmost) position of a numeric literal. 2. Unary operator for multiplication by numeric literal +1.
–	Minus	1. Sign character in the high-order (leftmost) position of a numeric literal. 2. Unary operator for multiplication by numeric literal –1.
Arithmetic Symbols — Used in arithmetic expressions		
+	Plus	Addition.
–	Minus	Subtraction.
*	Asterisk	Multiplication.
/	Slash	Division.
**	Double asterisk	Exponentiation.
=	Equal	Assignment.
( )	Parentheses	Enclose expressions to control the sequence in which they are evaluated.
Condition Symbols — Used in conditional test expressions		
=	Equal	Denotes *is equal to*.
>	Greater than	Denotes *is greater than*.
>=	Greater than or equal to	Denotes *is greater than or equal to*.
<	Less than	Denotes *is less than*.
<=	Less than or equal to	Denotes *is less than or equal to*.
( )	Parentheses	Enclose expressions to control the sequence in which conditions are evaluated.

TABLE B-1
COBOL85 Symbols - Continued

Symbol		Functions
Edit Symbols — Used in picture clauses		
.	Decimal point (insertion character)	Inserts a decimal point in the indicated position of an edited item.
,	Comma (insertion character)	Inserts a comma in the indicated position of an edited item. (May be used in conjunction with floating characters.)
$	Dollar sign (floating character)	Floats a dollar sign in an edited item so that exactly one dollar sign is inserted immediately to the left of the most significant nonzero digit in any position where the symbol is used.
/	Slash (insertion character)	Inserts a slash in the indicated position of an edited item.
*	Asterisk (replacement character)	Replaces leading zeros with an asterisk. Each asterisk represents a digit position in an edited item.
+ or –	Plus or minus (fixed sign control or floating characters)	1. Fixed sign control character in the low-order (rightmost) position of an edited item picture. The symbol does not replace a digit position. 2. Floats a plus or minus character (from left to right) in an edited item, so that exactly one plus or minus sign is inserted immediately to the left of the most significant nonzero digit in any position where the symbol is used.
B	(Insertion character)	Inserts blanks in the indicated positions of an edited item.
0	Zero (insertion character)	Inserts zeros in the indicated positions of an edited item.
Z	(Replacement character)	Replaces leading zeros with blanks in the indicated positions of an edited item.
CR	Credit (order fixed sign-control character)	Fixed sign control character in the low-order (rightmost) position of an edited item picture. It occupies two character positions in the edited result.
DB	Debit (fixed sign-control character)	Fixed sign-control character in the low-order (rightmost) position of an edited item picture. It occupies two character positions in the edited result.
P	(Decimal scaling character)	Specifies the location of an assumed decimal point when the point is not within the number that appears in the associated data item.
V	(Assumed decimal point)	Positions an assumed decimal point in a field.

TABLE B-2
COBOL85 Reserved Words

ACCEPT	COMP	DIVISION
ACCESS	COMP-1#	DOWN
ADD	COMP-2#	DUPLICATES
ADVANCING	COMP-3#	DYNAMIC
AFTER	COMPRESSED#	EGI*
ALL	COMPUTATIONAL	EJECT#
ALPHABET*	COMPUTATIONAL-1#	ELSE
ALPHABETIC	COMPUTATIONAL-2#	EMI*
ALPHABETIC-LOWER	COMPUTATIONAL-3#	ENABLE*
ALPHABETIC-UPPER	COMPUTE	END
ALPHANUMERIC*	CONFIGURATION	END-ADD
ALPHANUMERIC-EDITED*	CONTAINS	END-CALL
ALSO*	CONTENT*	END-COMPUTE
ALTER	CONTINUE	END-DELETE
ALTERNATE	CONTROL	END-DIVIDE
AND	CONTROLS	END-EVALUATE*
ANY*	CONVERTING	END-IF
ARE	COPY	END-MULTIPLY
AREA	CORR	END-OF-PAGE*
AREAS	CORRESPONDING	END-PERFORM
ASCENDING	COUNT	END-READ
ASSIGN	CURRENCY	END-RECEIVE*
AT	DATA	END-RETURN
AUTHOR	DATE	END-REWRITE
BEFORE	DATE-COMPILED	END-SEARCH
BINARY	DATE-WRITTEN	END-START
BLANK	DAY	END-STRING
BLOCK	DAY-OF-WEEK	END-SUBTRACT
BOTTOM*	DE*	END-UNSTRING
BY	DEBUG-CONTENTS*	END-WRITE
CALL	DEBUG-ITEM*	ENTER
CANCEL*	DEBUG-LINE*	ENVIRONMENT
CBL-SUBSCHEMA#	DEBUG-NAME*	EOP*
CD*	DEBUG-SUB-1*	EQUAL
CF*	DEBUG-SUB-2*	ERROR
CH*	DEBUG-SUB-3*	ESI*
CHARACTER	DEBUGGING	EVALUATE*
CHARACTERS	DECIMAL-POINT	EVERY*
CLASS	DECLARATIVES	EXCEPTION
CLOCK-UNITS*	DELETE	EXHIBIT#
CLOSE	DELIMITED	EXIT
COBOL	DELIMITER	EXTEND
CODE*	DEPENDING	EXTERNAL
CODE-SET	DESCENDING	FALSE*
COLLATING	DESTINATION*	FD
COLUMN	DETAIL*	FILE
COMMA	DISABLE*	FILE-CONTROL
COMMON*	DISPLAY	FILLER
COMMUNICATION*	DIVIDE	FINAL*

Prime reserved word.
* Not implemented.

TABLE B-2
COBOL85 Reserved Words - Continued

FIRST	LOCK*	POSITIVE
FOOTING*	LOW-VALUE	PRINTING*
FOR	LOW-VALUES	PROCEDURE
FROM	MEMORY	PROCEDURES*
GENERATE*	MERGE	PROCEED
GIVING	MESSAGE*	PROGRAM
GLOBAL*	MODE	PROGRAM-ID
GO	MODULES*	PURGE*
GOBACK#	MOVE	QUEUE*
GREATER	MULTIPLE*	QUOTE
GROUP*	MULTIPLY	QUOTES
HEADING*	NAMED#	RANDOM
HIGH-VALUE	NATIVE	RD*
HIGH-VALUES	NEGATIVE	READ
I-O	NEXT	READY#
I-O-CONTROL	NO	RECEIVE*
ID#	NOT	RECORD
IDENTIFICATION	NOTE#	RECORDS
IF	NUMBER*	REDEFINES
IN	NUMERIC	REEL
INDEX	NUMERIC-EDITED*	REFERENCE*
INDEXED	OBJECT-COMPUTER	REFERENCES*
INDICATE*	OCCURS	RELATIVE
INITIAL	OF	RELEASE
INITIALIZE*	OFF	REMAINDER
INITIATE*	OMITTED	REMARKS#
INPUT	ON	REMOVAL*
INPUT-OUTPUT	OPEN	RENAMES
INSPECT	OPTIONAL	REPLACE*
INSTALLATION	OR	REPLACING
INTO	ORDER*	REPORT*
INVALID	ORGANIZATION	REPORTING*
IS	OTHER*	REPORTS*
JUST	OTHERWISE#	RERUN*
JUSTIFIED	OUTPUT	RESERVE
KEY	OVERFLOW	RESET
LABEL	OWNER#	RETURN
LAST*	PACKED_DECIMAL	REVERSED*
LEADING	PADDING*	REWIND
LEFT	PAGE	REWRITE
LESS	PAGE-COUNTER*	RF*
LIMIT*	PERFORM	RH*
LIMITS*	PF*	RIGHT
LINAGE*	PH*	ROUNDED
LINAGE-COUNTER*	PIC	RUN
LINE	PICTURE	SAME
LINES	PLUS*	SD
LINE_COUNTER*	POINTER	SEARCH
LINKAGE	POSITION	SECTION

Prime reserved word.
* Not implemented.

TABLE B-2
COBOL85 Reserved Words - Continued

SECURITY	SUB-QUEUE-3*	USE
SEEK#	SUBTRACT	USING
SEGMENT*	SUM*	VALUE
SEGMENT-LIMIT	SUPPRESS*	VALUES
SELECT	SYMBOLIC*	VARYING
SEND*	SYNC	WHEN
SENTENCE	SYNCHRONIZED	WITH
SEPARATE	TABLE*	WORDS*
SEQUENCE	TALLYING	WORKING-STORAGE
SEQUENTIAL	TAPE	WRITE
SET	TERMINAL	ZERO
SIGN	TERMINATE*	ZEROES
SIZE	TEST*	ZEROS
SKIP1#	THAN	*
SKIP2#	THEN	**
SKIP3#	THROUGH	+
SORT	THRU	-
SORT-MERGE	TIME	/
SOURCE*	TIMES	<
SOURCE-COMPUTER	TO	<=
SPACE	TOP*	=
SPACES	TRACE#	>
SPECIAL-NAMES	TRAILING	>=
STANDARD	TRUE*	
STANDARD-1	TYPE*	
STANDARD-2	UNCOMPRESSED#	
START	UNIT*	
STATUS	UNSTRING	
STOP	UNTIL	
STRING	UP	
SUB-QUEUE-1*	UPON	
SUB-QUEUE-2*	USAGE	

Prime reserved word.
* Not implemented.

Prime Extended Character Set Table

Table B-3 contains all of the Prime ECS characters, arranged in ascending order. This order provides the collating sequence for sorting, merging, and comparing character strings. For each character, the table includes the graphic, the mnemonic, the description, and the binary, decimal, hexadecimal, and octal values. A blank entry indicates that the particular item does not apply to this character. The graphics for control characters are specified as ^*character*; for example, ^P represents the character produced when you type P while holding the control key down.

Characters with decimal values from 000 to 031 and from 128 to 159 are control characters.

Characters with decimal values from 032 to 127 and from 160 to 255 are graphic characters.

For additional information on the specification of Prime ECS mnemonics, see the section Nonnumeric Literals, in Chapter 4.

TABLE B-3
Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	RES1	Reserved for future standardization	0000 0000	000	00	000
	RES2	Reserved for future standardization	0000 0001	001	01	001
	RES3	Reserved for future standardization	0000 0010	002	02	002
	RES4	Reserved for future standardization	0000 0011	003	03	003
	IND	Index	0000 0100	004	04	004
	NEL	Next line	0000 0101	005	05	005
	SSA	Start of selected area	0000 0110	006	06	006
	ESA	End of selected area	0000 0111	007	07	007
	HTS	Horizontal tabulation set	0000 1000	008	08	010
	HTJ	Horizontal tab with justify	0000 1001	009	09	011
	VTS	Vertical tabulation set	0000 1010	010	0A	012
	PLD	Partial line down	0000 1011	011	0B	013
	PLU	Partial line up	0000 1100	012	0C	014
	RI	Reverse index	0000 1101	013	0D	015
	SS2	Single shift 2	0000 1110	014	0E	016
	SS3	Single shift 3	0000 1111	015	0F	017
	DCS	Device control string	0001 0000	016	10	020
	PU1	Private use 1	0001 0001	017	11	021
	PU2	Private use 2	0001 0010	018	12	022
	STS	Set transmission state	0001 0011	019	13	023
	CCH	Cancel character	0001 0100	020	14	024
	MW	Message waiting	0001 0101	021	15	025
	SPA	Start of protected area	0001 0110	022	16	026
	EPA	End of protected area	0001 0111	023	17	027
	RES5	Reserved for future standardization	0001 1000	024	18	030
	RES6	Reserved for future standardization	0001 1001	025	19	031
	RES7	Reserved for future standardization	0001 1010	026	1A	032
	CSI	Control sequence introducer	0001 1011	027	1B	033
	ST	String terminator	0001 1100	028	1C	034
	OSC	Operating system command	0001 1101	029	1D	035
	PM	Privacy message	0001 1110	030	1E	036

TABLE B-3
Prime Extended Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	APC	Application program command	0001 1111	031	1F	037
	NBSP	No-break space	0010 0000	032	20	040
¡	INVE	Inverted exclamation mark	0010 0001	033	21	041
¢	CENT	Cent sign	0010 0010	034	22	042
£	PND	Pound sign	0010 0011	035	23	043
¤	CURR	Currency sign	0010 0100	036	24	044
¥	YEN	Yen sign	0010 0101	037	25	045
¦	BBAR	Broken bar	0010 0110	038	26	046
§	SECT	Section sign	0010 0111	039	27	047
¨	DIA	Diaeresis, umlaut	0010 1000	040	28	050
©	COPY	Copyright sign	0010 1001	041	29	051
ª	FOI	Feminine ordinal indicator	0010 1010	042	2A	052
«	LAQM	Left angle quotation mark	0010 1011	043	2B	053
¬	NOT	Not sign	0010 1100	044	2C	054
	SHY	Soft hyphen	0010 1101	045	2D	055
®	TM	Registered trademark sign	0010 1110	046	2E	056
¯	MACN	Macron	0010 1111	047	2F	057
°	DEGR	Degree sign	0011 0000	048	30	060
±	PLMI	Plus/minus sign	0011 0001	049	31	061
²	SPS2	Superscript two	0011 0010	050	32	062
³	SPS3	Superscript three	0011 0011	051	33	063
´	AAC	Acute accent	0011 0100	052	34	064
µ	LCMU	Lowercase Greek letter µ, micro sign	0011 0101	053	35	065
¶	PARA	Paragraph sign, Pilgrow sign	0011 0110	054	36	066
·	MIDD	Middle dot	0011 0111	055	37	067
¸	CED	Cedilla	0011 1000	056	38	070
¹	SPS1	Superscript one	0011 1001	057	39	071
º	MOI	Masculine ordinal indicator	0011 1010	058	3A	072
»	RAQM	Right angle quotation mark	0011 1011	059	3B	073
¼	FR14	Common fraction one-quarter	0011 1100	060	3C	074

TABLE B-3
Prime Extended Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
1/2	FR12	Common fraction one-half	0011 1101	061	3D	075
3/4	FR34	Common fraction three-quarters	0011 1110	062	3E	076
¿	INVQ	Inverted question mark	0011 1111	063	3F	077
À	UCAG	Uppercase A with grave accent	0100 0000	064	40	100
Á	UCAA	Uppercase A with acute accent	0100 0001	065	41	101
Â	UCAC	Uppercase A with circumflex	0100 0010	066	42	102
Ã	UCAT	Uppercase A with tilde	0100 0011	067	43	103
Ä	UCAD	Uppercase A with diaeresis	0100 0100	068	44	104
Å	UCAR	Uppercase A with ring above	0100 0101	069	45	105
Æ	UCAE	Uppercase diphthong Æ	0100 0110	070	46	106
Ç	UCCC	Uppercase C with cedilla	0100 0111	071	47	107
È	UCEG	Uppercase E with grave accent	0100 1000	072	48	110
É	UCEA	Uppercase E with acute accent	0100 1001	073	49	111
Ê	UCEC	Uppercase E with circumflex	0100 1010	074	4A	112
Ë	UCED	Uppercase E with diaeresis	0100 1011	075	4B	113
Ì	UCIG	Uppercase I with grave accent	0100 1100	076	4C	114
Í	UCIA	Uppercase I with acute accent	0100 1101	077	4D	115
Î	UCIC	Uppercase I with circumflex	0100 1110	078	4E	116
Ï	UCID	Uppercase I with diaeresis	0100 1111	079	4F	117
Ð	UETH	Uppercase Icelandic letter Eth	0101 0000	080	50	120
Ñ	UCNT	Uppercase N with tilde	0101 0001	081	51	121
Ò	UCOG	Uppercase O with grave accent	0101 0010	082	52	122
Ó	UCOA	Uppercase O with acute accent	0101 0011	083	53	123

TABLE B-3
Prime Extended Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
Ô	UCOC	Uppercase O with circumflex	0101 0100	084	54	124
Õ	UCOT	Uppercase O with tilde	0101 0101	085	55	125
Ö	UCOD	Uppercase O with diaeresis	0101 0110	086	56	126
×	MULT	Multiplication sign used in mathematics	0101 0111	087	57	127
Ø	UCOO	Uppercase O with oblique line	0101 1000	088	58	130
Ù	UCUG	Uppercase U with grave accent	0101 1001	089	59	131
Ú	UCUA	Uppercase U with acute accent	0101 1010	090	5A	132
Û	UCUC	Uppercase U with circumflex	0101 1011	091	5B	133
Ü	UCUD	Uppercase U with diaeresis	0101 1100	092	5C	134
Ý	UCYA	Uppercase Y with acute accent	0101 1101	093	5D	135
Þ	UTHN	Uppercase Icelandic letter Thorn	0101 1110	094	5E	136
ß	LGSS	Lowercase German letter double s	0101 1111	095	5F	137
à	LCAG	Lowercase a with grave accent	0110 0000	096	60	140
á	LCAA	Lowercase a with acute accent	0110 0001	097	61	141
â	LCAC	Lowercase a with circumflex	0110 0010	098	62	142
ã	LCAT	Lowercase a with tilde	0110 0011	099	63	143
ä	LCAD	Lowercase a with diaeresis	0110 0100	100	64	144
å	LCAR	Lowercase a with ring above	0110 0101	101	65	145
æ	LCAE	Lowercase diphthong ae	0110 0110	102	66	146
ç	LCCC	Lowercase c with cedilla	0110 0111	103	67	147
è	LCEG	Lowercase e with grave accent	0110 1000	104	68	150
é	LCEA	Lowercase e with acute accent	0110 1001	105	69	151
ê	LCEC	Lowercase e with circumflex	0110 1010	106	6A	152

TABLE B-3
Prime Extended Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
ë	LCED	Lowercase e with diaeresis	0110 1011	107	6B	153
ì	LCIG	Lowercase i with grave accent	0110 1100	108	6C	154
í	LCIA	Lowercase i with acute accent	0110 1101	109	6D	155
î	LCiC	Lowercase i with circumflex	0110 1110	110	6E	156
ï	LCID	Lowercase i with diaeresis	0110 1111	111	6F	157
ð	LETH	Lowercase Icelandic letter Eth	0111 0000	112	70	160
ñ	LCNT	Lowercase n with tilde	0111 0001	113	71	161
ò	LCOG	Lowercase o with grave accent	0111 0010	114	72	162
ó	LCOA	Lowercase o with acute accent	0111 0011	115	73	163
ô	LCOC	Lowercase o with circumflex	0111 0100	116	74	164
õ	LCOT	Lowercase o with tilde	0111 0101	117	75	165
ö	LCOD	Lowercase o with diaeresis	0111 0110	118	76	166
÷	DIV	Division sign used in mathematics	0111 0111	119	77	167
ø	LCOO	Lowercase o with oblique line	0111 1000	120	78	170
ù	LCUG	Lowercase u with grave accent	0111 1001	121	79	171
ú	LCUA	Lowercase u with acute accent	0111 1010	122	7A	172
û	LCUC	Lowercase u with circumflex	0111 1011	123	7B	173
ü	LCUD	Lowercase u with diaeresis	0111 1100	124	7C	174
ý	LCYA	Lowercase y with acute accent	0111 1101	125	7D	175
þ	LTHN	Lowercase Icelandic letter Thorn	0111 1110	126	7E	176
ÿ	LCYD	Lowercase y with diaeresis	0111 1111	127	7F	177

TABLE B-3
Prime Extended Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	NUL	Null	1000 0000	128	80	200
^A	SOH/TC1	Start of heading	1000 0001	129	81	201
^B	STX/TC2	Start of text	1000 0010	130	82	202
^C	ETX/TC3	End of text	1000 0011	131	83	203
^D	EOT/TC4	End of transmission	1000 0100	132	84	204
^E	ENQ/TC5	Enquiry	1000 0101	133	85	205
^F	ACK/TC6	Acknowledge	1000 0110	134	86	206
^G	BEL	Bell	1000 0111	135	87	207
^H	BS/FE0	Backspace	1000 1000	136	88	210
^I	HT/FE1	Horizontal tab	1000 1001	137	89	211
^J	LF/NL/FE2	Line feed	1000 1010	138	8A	212
^K	VT/FE3	Vertical tab	1000 1011	139	8B	213
^L	FF/FE4	Form feed	1000 1100	140	8C	214
^M	CR/FE5	Carriage return	1000 1101	141	8D	215
^N	SO/LS1	Shift out	1000 1110	142	8E	216
^O	SI/LS0	Shift in	1000 1111	143	8F	217
^P	DLE/TC7	Data link escape	1001 0000	144	90	220
^Q	DC1/XON	Device control 1	1001 0001	145	91	221
^R	DC2	Device control 2	1001 0010	146	92	222
^S	DC3/XOFF	Device control 3	1001 0011	147	93	223
^T	DC4	Device control 4	1001 0100	148	94	224
^U	NAK/TC8	Negative acknowledge	1001 0101	149	95	225
^V	SYN/TC9	Synchronous idle	1001 0110	150	96	226
^W	ETB/TC10	End of transmission block	1001 0111	151	97	227
^X	CAN	Cancel	1001 1000	152	98	230
^Y	EM	End of medium	1001 1001	153	99	231
^Z	SUB	Substitute	1001 1010	154	9A	232
^[	ESC	Escape	1001 1011	155	9B	233
^\	FS/IS4	File separator	1001 1100	156	9C	234
^]	GS/IS3	Group separator	1001 1101	157	9D	235
^^	RS/IS2	Record separator	1001 1110	158	9E	236
^_	US/IS1	Unit separator	1001 1111	159	9F	237
	SP	Space	1010 0000	160	A0	240
!		Exclamation mark	1010 0001	161	A1	241
''		Quotation mark	1010 0010	162	A2	242
#	NUMB	Number sign	1010 0011	163	A3	243
$	DOLR	Dollar sign	1010 0100	164	A4	244
%		Percent sign	1010 0101	165	A5	245
&		Ampersand	1010 0110	166	A6	246

TABLE B-3
Prime Extended Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
'		Apostrophe	1010 0111	167	A7	247
(		Left parenthesis	1010 1000	168	A8	250
)		Right parenthesis	1010 1001	169	A9	251
*		Asterisk	1010 1010	170	AA	252
+		Plus sign	1010 1011	171	AB	253
,		Comma	1010 1100	172	AC	254
–		Minus sign	1010 1101	173	AD	255
.		Period	1010 1110	174	AE	256
/		Slash	1010 1111	175	AF	257
0		Zero	1011 0000	176	B0	260
1		One	1011 0001	177	B1	261
2		Two	1011 0010	178	B2	262
3		Three	1011 0011	179	B3	263
4		Four	1011 0100	180	B4	264
5		Five	1011 0101	181	B5	265
6		Six	1011 0110	182	B6	266
7		Seven	1011 0111	183	B7	267
8		Eight	1011 1000	184	B8	270
9		Nine	1011 1001	185	B9	271
:		Colon	1011 1010	186	BA	272
;		Semicolon	1011 1011	187	BB	273
<		Less than sign	1011 1100	188	BC	274
=		Equal sign	1011 1101	189	BD	275
>		Greater than sign	1011 1110	190	BE	276
?		Question mark	1011 1111	191	BF	277
@	AT	Commercial at sign	1100 0000	192	C0	300
A		Uppercase A	1100 0001	193	C1	301
B		Uppercase B	1100 0010	194	C2	302
C		Uppercase C	1100 0011	195	C3	303
D		Uppercase D	1100 0100	196	C4	304
E		Uppercase E	1100 0101	197	C5	305
F		Uppercase F	1100 0110	198	C6	306
G		Uppercase G	1100 0111	199	C7	307
H		Uppercase H	1100 1000	200	C8	310
I		Uppercase I	1100 1001	201	C9	311
J		Uppercase J	1100 1010	202	CA	312
K		Uppercase K	1100 1011	203	CB	313
L		Uppercase L	1100 1100	204	CC	314
M		Uppercase M	1100 1101	205	CD	315
N		Uppercase N	1100 1110	206	CE	316

TABLE B-3
Prime Extended Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
O		Uppercase O	1100 1111	207	CF	317
P		Uppercase P	1101 0000	208	D0	320
Q		Uppercase Q	1101 0001	209	D1	321
R		Uppercase R	1101 0010	210	D2	322
S		Uppercase S	1101 0011	211	D3	323
T		Uppercase T	1101 0100	212	D4	324
U		Uppercase U	1101 0101	213	D5	325
V		Uppercase V	1101 0110	214	D6	326
W		Uppercase W	1101 0111	215	D7	327
X		Uppercase X	1101 1000	216	D8	330
Y		Uppercase Y	1101 1001	217	D9	331
Z		Uppercase Z	1101 1010	218	DA	332
[	LBKT	Left bracket	1101 1011	219	DB	333
\	REVS	Reverse slash, backslash	1101 1100	220	DC	334
]	RBKT	Right bracket	1101 1101	221	DD	335
^	CFLX	Circumflex	1101 1110	222	DE	336
_		Underline, underscore	1101 1111	223	DF	337
`	GRAV	Left single quote, grave accent	1110 0000	224	E0	340
a		Lowercase a	1110 0001	225	E1	341
b		Lowercase b	1110 0010	226	E2	342
c		Lowercase c	1110 0011	227	E3	343
d		Lowercase d	1110 0100	228	E4	344
e		Lowercase e	1110 0101	229	E5	345
f		Lowercase f	1110 0110	230	E6	346
g		Lowercase g	1110 0111	231	E7	347
h		Lowercase h	1110 1000	232	E8	350
i		Lowercase i	1110 1001	233	E9	351
j		Lowercase j	1110 1010	234	EA	352
k		Lowercase k	1110 1011	235	EB	353
l		Lowercase l	1110 1100	236	EC	354
m		Lowercase m	1110 1101	237	ED	355
n		Lowercase n	1110 1110	238	EE	356
o		Lowercase o	1110 1111	239	EF	357
p		Lowercase p	1111 0000	240	F0	360
q		Lowercase q	1111 0001	241	F1	361
r		Lowercase r	1111 0010	242	F2	362
s		Lowercase s	1111 0011	243	F3	363
t		Lowercase t	1111 0100	244	F4	364

TABLE B-3
Prime Extended Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
u		Lowercase u	1111 0101	245	F5	365
v		Lowercase v	1111 0110	246	F6	366
w		Lowercase w	1111 0111	247	F7	367
x		Lowercase x	1111 1000	248	F8	370
y		Lowercase y	1111 1001	249	F9	371
z		Lowercase z	1111 1010	250	FA	372
{	LBCE	Left brace	1111 1011	251	FB	373
|	VERT	Vertical line	1111 1100	252	FC	374
}	RBCE	Right brace	1111 1101	253	FD	375
~	TIL	Tilde	1111 1110	254	FE	376
	DEL	Delete	1111 1111	255	FF	377

Standard-1 ASCII Character Set Table

Table B-4 contains the standard ASCII character set as defined by ANSI X3.4-1977, arranged in ascending order. This order provides the collating sequence for nonnumeric comparisons in conditional expressions when the *alphabet-name* of the PROGRAM COLLATING SEQUENCE clause is an *alphabet-name* specified as STANDARD-1.

For each character, the table includes the graphic, the mnemonic, the description, and the binary, decimal, hexadecimal, and octal value. A blank entry indicates that the particular item does not apply to this character. The graphics for control characters are specified as ^*character*; for example, ^P represents the character produced when you type P while holding the control key down.

TABLE B-4
Standard-1 ASCII Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	NUL	Null	0000 0000	000	00	000
^A	SOH/TC1	Start of heading	0000 0001	001	01	001
^B	STX/TC2	Start of text	0000 0010	002	02	002
^C	ETX/TC3	End of text	0000 0011	003	03	003
^D	EOT/TC4	End of transmission	0000 0100	004	04	004
^E	ENQ/TC5	Enquiry	0000 0101	005	05	005
^F	ACK/TC6	Acknowledge	0000 0110	006	06	006
^G	BEL	Bell	0000 0111	007	07	007
^H	BS/FE0	Backspace	0000 1000	008	08	010
^I	HT/FE1	Horizontal tab	0000 1001	009	09	011
^J	LF/NL/FE2	Line feed	0000 1010	010	0A	012
^K	VT/FE3	Vertical tab	0000 1011	011	0B	013
^L	FF/FE4	Form feed	0000 1100	012	0C	014
^M	CR/FE5	Carriage return	0000 1101	013	0D	015
^N	SO/LS1	Shift out	0000 1110	014	0E	016
^O	SI/LS0	Shift in	0000 1111	015	0F	017
^P	DLE/TC7	Data link escape	0001 0000	016	10	020
^Q	DC1/XON	Device control 1	0001 0001	017	11	021
^R	DC2	Device control 2	0001 0010	018	12	022
^S	DC3/XOFF	Device control 3	0001 0011	019	13	023
^T	DC4	Device control 4	0001 0100	020	14	024
^U	NAK/TC8	Negative acknowledge	0001 0101	021	15	025
^V	SYN/TC9	Synchronous idle	0001 0110	022	16	026
^W	ETB/TC10	End of transmission block	0001 0111	023	17	027
^X	CAN	Cancel	0001 1000	024	18	030
^Y	EM	End of medium	0001 1001	025	19	031
^Z	SUB	Substitute	0001 1010	026	1A	032
^[	ESC	Escape	0001 1011	027	1B	033
^\	FS/IS4	File separator	0001 1100	028	1C	034
^]	GS/IS3	Group separator	0001 1101	029	1D	035
^^	RS/IS2	Record separator	0001 1110	030	1E	036
^_	US/IS1	Unit separator	0001 1111	031	1F	037
	SP	Space	0010 0000	032	20	040
!		Exclamation mark	0010 0001	033	21	041
"		Quotation mark	0010 0010	034	22	042
#	NUMB	Number sign	0010 0011	035	23	043
$	DOLR	Dollar sign	0010 0100	036	24	044
%		Percent sign	0010 0101	037	25	045
&		Ampersand	0010 0110	038	26	046

TABLE B-4
Standard-1 ASCII Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
'		Apostrophe	0010 0111	039	27	047
(		Left parenthesis	0010 1000	040	28	050
)		Right parenthesis	0010 1001	041	29	051
*		Asterisk	0010 1010	042	2A	052
+		Plus sign	0010 1011	043	2B	053
,		Comma	0010 1100	044	2C	054
–		Minus sign	0010 1101	045	2D	055
.		Period	0010 1110	046	2E	056
/		Slash	0010 1111	047	2F	057
0		Zero	0011 0000	048	30	060
1		One	0011 0001	049	31	061
2		Two	0011 0010	050	32	062
3		Three	0011 0011	051	33	063
4		Four	0011 0100	052	34	064
5		Five	0011 0101	053	35	065
6		Six	0011 0110	054	36	066
7		Seven	0011 0111	055	37	067
8		Eight	0011 1000	056	38	070
9		Nine	0011 1001	057	39	071
:		Colon	0011 1010	058	3A	072
;		Semicolon	0011 1011	059	3B	073
<		Less than sign	0011 1100	060	3C	074
=		Equal sign	0011 1101	061	3D	075
>		Greater than sign	0011 1110	062	3E	076
?		Question mark	0011 1111	063	3F	077
@	AT	Commercial at sign	0100 0000	064	40	100
A		Uppercase A	0100 0001	065	41	101
B		Uppercase B	0100 0010	066	42	102
C		Uppercase C	0100 0011	067	43	103
D		Uppercase D	0100 0100	068	44	104
E		Uppercase E	0100 0101	069	45	105
F		Uppercase F	0100 0110	070	46	106
G		Uppercase G	0100 0111	071	47	107
H		Uppercase H	0100 1000	072	48	110
I		Uppercase I	0100 1001	073	49	111
J		Uppercase J	0100 1010	074	4A	112
K		Uppercase K	0100 1011	075	4B	113
L		Uppercase L	0100 1100	076	4C	114
M		Uppercase M	0100 1101	077	4D	115
N		Uppercase N	0100 1110	078	4E	116

TABLE B-4
Standard-1 ASCII Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
O		Uppercase O	0100 1111	079	4F	117
P		Uppercase P	0101 0000	080	50	120
Q		Uppercase Q	0101 0001	081	51	121
R		Uppercase R	0101 0010	082	52	122
S		Uppercase S	0101 0011	083	53	123
T		Uppercase T	0101 0100	084	54	124
U		Uppercase U	0101 0101	085	55	125
V		Uppercase V	0101 0110	086	56	126
W		Uppercase W	0101 0111	087	57	127
X		Uppercase X	0101 1000	088	58	130
Y		Uppercase Y	0101 1001	089	59	131
Z		Uppercase Z	0101 1010	090	5A	132
[	LBKT	Left bracket	0101 1011	091	5B	133
\	REVS	Reverse slash, backslash	0101 1100	092	5C	134
]	RBKT	Right bracket	0101 1101	093	5D	135
^	CFLX	Circumflex, caret	0101 1110	094	5E	136
_		Underline, underscore	0101 1111	095	5F	137
`	GRAV	Left single quote, grave accent	0110 0000	096	60	140
a		Lowercase a	0110 0001	097	61	141
b		Lowercase b	0110 0010	098	62	142
c		Lowercase c	0110 0011	099	63	143
d		Lowercase d	0110 0100	100	64	144
e		Lowercase e	0110 0101	101	65	145
f		Lowercase f	0110 0110	102	66	146
g		Lowercase g	0110 0111	103	67	147
h		Lowercase h	0110 1000	104	68	150
i		Lowercase i	0110 1001	105	69	151
j		Lowercase j	0110 1010	106	6A	152
k		Lowercase k	0110 1011	107	6B	153
l		Lowercase l	0110 1100	108	6C	154
m		Lowercase m	0110 1101	109	6D	155
n		Lowercase n	0110 1110	110	6E	156
o		Lowercase o	0110 1111	111	6F	157
p		Lowercase p	0111 0000	112	70	160
q		Lowercase q	0111 0001	113	71	161
r		Lowercase r	0111 0010	114	72	162
s		Lowercase s	0111 0011	115	73	163
t		Lowercase t	0111 0100	116	74	164

TABLE B-4
Standard-1 ASCII Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
u		Lowercase u	0111 0101	117	75	165
v		Lowercase v	0111 0110	118	76	166
w		Lowercase w	0111 0111	119	77	167
x		Lowercase x	0111 1000	120	78	170
y		Lowercase y	0111 1001	121	79	171
z		Lowercase z	0111 1010	122	7A	172
{	LBCE	Left brace	0111 1011	123	7B	173
|	VERT	Vertical line	0111 1100	124	7C	174
}	RBCE	Right brace	0111 1101	125	7D	175
~	TIL	Tilde	0111 1110	126	7E	176
	DEL	Delete	0111 1111	127	7F	177
	RES1	Reserved for future standardization	1000 0000	128	80	200
	RES2	Reserved for future standardization	1000 0001	129	81	201
	RES3	Reserved for future standardization	1000 0010	130	82	202
	RES4	Reserved for future standardization	1000 0011	131	83	203
	IND	Index	1000 0100	132	84	204
	NEL	Next line	1000 0101	133	85	205
	SSA	Start of selected area	1000 0110	134	86	206
	ESA	End of selected area	1000 0111	135	87	207
	HTS	Horizontal tabulation set	1000 1000	136	88	210
	HTJ	Horizontal tab with justify	1000 1001	137	89	211
	VTS	Vertical tabulation set	1000 1010	138	8A	212
	PLD	Partial line down	1000 1011	139	8B	213
	PLU	Partial line up	1000 1100	140	8C	214
	RI	Reverse index	1000 1101	141	8D	215
	SS2	Single shift 2	1000 1110	142	8E	216
	SS3	Single shift 3	1000 1111	143	8F	217
	DCS	Device control string	1001 0000	144	90	220
	PU1	Private use 1	1001 0001	145	91	221
	PU2	Private use 2	1001 0010	146	92	222
	STS	Set transmission state	1001 0011	147	93	223
	CCH	Cancel character	1001 0100	148	94	224
	MW	Message waiting	1001 0101	149	95	225
	SPA	Start of protected area	1001 0110	150	96	226
	EPA	End of protected area	1001 0111	151	97	227

TABLE B-4
Standard-1 ASCII Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	RES5	Reserved for future standardization	1001 1000	152	98	230
	RES6	Reserved for future standardization	1001 1001	153	99	231
	RES7	Reserved for future standardization	1001 1010	154	9A	232
	CSI	Control sequence introducer	1001 1011	155	9B	233
	ST	String terminator	1001 1100	156	9C	234
	OSC	Operating system command	1001 1101	157	9D	235
	PM	Privacy message	1001 1110	158	9E	236
	APC	Application program command	1001 1111	159	9F	237
	NBSP	No-break space	1010 0000	160	A0	240
¡	INVE	Inverted exclamation mark	1010 0001	161	A1	241
¢	CENT	Cent sign	1010 0010	162	A2	242
£	PND	Pound sign	1010 0011	163	A3	243
¤	CURR	Currency sign	1010 0100	164	A4	244
¥	YEN	Yen sign	1010 0101	165	A5	245
¦	BBAR	Broken bar	1010 0110	166	A6	246
§	SECT	Section sign	1010 0111	167	A7	247
¨	DIA	Diaeresis, umlaut	1010 1000	168	A8	250
©	COPY	Copyright sign	1010 1001	169	A9	251
ª	FOI	Feminine ordinal indicator	1010 1010	170	AA	252
«	LAQM	Left angle quotation mark	1010 1011	171	AB	253
¬	NOT	Not sign	1010 1100	172	AC	254
	SHY	Soft hyphen	1010 1101	173	AD	255
®	TM	Registered trademark sign	1010 1110	174	AE	256
¯	MACN	Macron	1010 1111	175	AF	257
°	DEGR	Degree sign	1011 0000	176	B0	260
±	PLMI	Plus/minus sign	1011 0001	177	B1	261
²	SPS2	Superscript two	1011 0010	178	B2	262
³	SPS3	Superscript three	1011 0011	179	B3	263
´	AAC	Acute accent	1011 0100	180	B4	264
µ	LCMU	Lowercase Greek letter µ, micro sign	1011 0101	181	B5	265

TABLE B-4
Standard-1 ASCII Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
¶	PARA	Paragraph sign, Pilgrow sign	1011 0110	182	B6	266
·	MIDD	Middle dot	1011 0111	183	B7	267
¸	CED	Cedilla	1011 1000	184	B8	270
¹	SPS1	Superscript one	1011 1001	185	B9	271
º	MOI	Masculine ordinal indicator	1011 1010	186	BA	272
»	RAQM	Right angle quotation mark	1011 1011	187	BB	273
1/4	FR14	Common fraction one-quarter	1011 1100	188	BC	274
1/2	FR12	Common fraction one-half	1011 1101	189	BD	275
3/4	FR34	Common fraction three-quarters	1011 1110	190	BE	276
¿	INVQ	Inverted question mark	1011 1111	191	BF	277
À	UCAG	Uppercase A with grave accent	1100 0000	192	C0	300
Á	UCAA	Uppercase A with acute accent	1100 0001	193	C1	301
Â	UCAC	Uppercase A with circumflex	1100 0010	194	C2	302
Ã	UCAT	Uppercase A with tilde	1100 0011	195	C3	303
Ä	UCAD	Uppercase A with diaeresis	1100 0100	196	C4	304
Å	UCAR	Uppercase A with ring above	1100 0101	197	C5	305
Æ	UCAE	Uppercase diphthong Æ	1100 0110	198	C6	306
Ç	UCCC	Uppercase C with cedilla	1100 0111	199	C7	307
È	UCEG	Uppercase E with grave accent	1100 1000	200	C8	310
É	UCEA	Uppercase E with acute accent	1100 1001	201	C9	311
Ê	UCEC	Uppercase E with circumflex	1100 1010	202	CA	312
Ë	UCED	Uppercase E with diaeresis	1100 1011	203	CB	313
Ì	UCIG	Uppercase I with grave accent	1100 1100	204	CC	314

TABLE B-4
Standard-1 ASCII Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
Í	UCIA	Uppercase I with acute accent	1100 1101	205	CD	315
Î	UCIC	Uppercase I with circumflex	1100 1110	206	CE	316
Ï	UCID	Uppercase I with diaeresis	1100 1111	207	CF	317
Ð	UETH	Uppercase Icelandic letter Eth	1101 0000	208	D0	320
Ñ	UCNT	Uppercase N with tilde	1101 0001	209	D1	321
Ò	UCOG	Uppercase O with grave accent	1101 0010	210	D2	322
Ó	UCOA	Uppercase O with acute accent	1101 0011	211	D3	323
Ô	UCOC	Uppercase O with circumflex	1101 0100	212	D4	324
Õ	UCOT	Uppercase O with tilde	1101 0101	213	D5	325
Ö	UCOD	Uppercase O with diaeresis	1101 0110	214	D6	326
×	MULT	Multiplication sign used in mathematics	1101 0111	215	D7	327
Ø	UCOO	Uppercase O with oblique line	1101 1000	216	D8	330
Ù	UCUG	Uppercase U with grave accent	1101 1001	217	D9	331
Ú	UCUA	Uppercase U with acute accent	1101 1010	218	DA	332
Û	UCUC	Uppercase U with circumflex	1101 1011	219	DB	333
Ü	UCUD	Uppercase U with diaeresis	1101 1100	220	DC	334
Ý	UCYA	Uppercase Y with acute accent	1101 1101	221	DD	335
Þ	UTHN	Uppercase Icelandic letter Thorn	1101 1110	222	DE	336
ß	LGSS	Lowercase German letter double s	1101 1111	223	DF	337
à	LCAG	Lowercase a with grave accent	1110 0000	224	E0	340
á	LCAA	Lowercase a with acute accent	1110 0001	225	E1	341
â	LCAC	Lowercase a with circumflex	1110 0010	226	E2	342
ã	LCAT	Lowercase a with tilde	1110 0011	227	E3	343

TABLE B-4
Standard-1 ASCII Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
ä	LCAD	Lowercase a with diaeresis	1110 0100	228	E4	344
å	LCAR	Lowercase a with ring above	1110 0101	229	E5	345
æ	LCAE	Lowercase diphthong ae	1110 0110	230	E6	346
ç	LCCC	Lowercase c with cedilla	1110 0111	231	E7	347
è	LCEG	Lowercase e with grave accent	1110 1000	232	E8	350
é	LCEA	Lowercase e with acute accent	1110 1001	233	E9	351
ê	LCEC	Lowercase e with circumflex	1110 1010	234	EA	352
ë	LCED	Lowercase e with diaeresis	1110 1011	235	EB	353
ì	LCIG	Lowercase i with grave accent	1110 1100	236	EC	354
í	LCIA	Lowercase i with acute accent	1110 1101	237	ED	355
î	LCIC	Lowercase i with circumflex	1110 1110	238	EE	356
ï	LCID	Lowercase i with diaeresis	1110 1111	239	EF	357
ð	LETH	Lowercase Icelandic letter Eth	1111 0000	240	F0	360
ñ	LCNT	Lowercase n with tilde	1111 0001	241	F1	361
ò	LCOG	Lowercase o with grave accent	1111 0010	242	F2	362
ó	LCOA	Lowercase o with acute accent	1111 0011	243	F3	363
ô	LCOC	Lowercase o with circumflex	1111 0100	244	F4	364
õ	LCOT	Lowercase o with tilde	1111 0101	245	F5	365
ö	LCOD	Lowercase o with diaeresis	1111 0110	246	F6	366
÷	DIV	Division sign used in mathematics	1111 0111	247	F7	367
ø	LCOO	Lowercase o with oblique line	1111 1000	248	F8	370
ù	LCUG	Lowercase u with grave accent	1111 1001	249	F9	371
ú	LCUA	Lowercase u with acute accent	1111 1010	250	FA	372

TABLE B-4
Standard-1 ASCII Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
û	LCUC	Lowercase u with circumflex	1111 1011	251	FB	373
ü	LCUD	Lowercase u with diaeresis	1111 1100	252	FC	374
ý	LCYA	Lowercase y with acute accent	1111 1101	253	FD	375
þ	LTHN	Lowercase Icelandic letter Thorn	1111 1110	254	FE	376
ÿ	LCYD	Lowercase y with diaeresis	1111 1111	255	FF	377

Standard-2 ASCII Character Set Table

Table B-5 contains the International Reference Version of ISO 646 7-bit Coded Character Set, arranged in ascending order. This order provides the collating sequence for nonnumeric comparisons in conditional expressions when the *alphabet-name* of the PROGRAM COLLATING SEQUENCE clause is an *alphabet-name* specified as STANDARD-2.

For each character, the table includes the graphic, the mnemonic, the description, and the binary, decimal, hexadecimal, and octal value. A blank entry indicates that the particular item does not apply to this character. The graphics for control characters are specified as ^*character*; for example, ^P represents the character produced when you type P while holding the control key down.

TABLE B-5
Standard-2 ASCII Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	NUL	Null	0000 0000	000	00	000
^A	SOH/TC1	Start of heading	0000 0001	001	01	001
^B	STX/TC2	Start of text	0000 0010	002	02	002
^C	ETX/TC3	End of text	0000 0011	003	03	003
^D	EOT/TC4	End of transmission	0000 0100	004	04	004
^E	ENQ/TC5	Enquiry	0000 0101	005	05	005
^F	ACK/TC6	Acknowledge	0000 0110	006	06	006
^G	BEL	Bell	0000 0111	007	07	007
^H	BS/FE0	Backspace	0000 1000	008	08	010
^I	HT/FE1	Horizontal tab	0000 1001	009	09	011
^J	LF/NL/FE2	Line feed	0000 1010	010	0A	012
^K	VT/FE3	Vertical tab	0000 1011	011	0B	013
^L	FF/FE4	Form feed	0000 1100	012	0C	014
^M	CR/FE5	Carriage return	0000 1101	013	0D	015
^N	SO/LS1	Shift out	0000 1110	014	0E	016
^O	SI/LS0	Shift in	0000 1111	015	0F	017
^P	DLE/TC7	Data link escape	0001 0000	016	10	020
^Q	DC1/XON	Device control 1	0001 0001	017	11	021
^R	DC2	Device control 2	0001 0010	018	12	022
^S	DC3/XOFF	Device control 3	0001 0011	019	13	023
^T	DC4	Device control 4	0001 0100	020	14	024
^U	NAK/TC8	Negative acknowledge	0001 0101	021	15	025
^V	SYN/TC9	Synchronous idle	0001 0110	022	16	026
^W	ETB/TC10	End of transmission block	0001 0111	023	17	027
^X	CAN	Cancel	0001 1000	024	18	030
^Y	EM	End of medium	0001 1001	025	19	031
^Z	SUB	Substitute	0001 1010	026	1A	032
^[	ESC	Escape	0001 1011	027	1B	033
^\	FS/IS4	File separator	0001 1100	028	1C	034
^]	GS/IS3	Group separator	0001 1101	029	1D	035
^^	RS/IS2	Record separator	0001 1110	030	1E	036
^_	US/IS1	Unit separator	0001 1111	031	1F	037
	SP	Space	0010 0000	032	20	040
!		Exclamation mark	0010 0001	033	21	041
"		Quotation mark	0010 0010	034	22	042
#	NUMB	Number sign	0010 0011	035	23	043
⌑		Open Lozenge	0010 0100	036	24	044
%		Percent sign	0010 0101	037	25	045
&		Ampersand	0010 0110	038	26	046

TABLE B-5
Standard-2 ASCII Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
'		Apostrophe	0010 0111	039	27	047
(		Left parenthesis	0010 1000	040	28	050
)		Right parenthesis	0010 1001	041	29	051
*		Asterisk	0010 1010	042	2A	052
+		Plus sign	0010 1011	043	2B	053
,		Comma	0010 1100	044	2C	054
–		Minus sign	0010 1101	045	2D	055
.		Period	0010 1110	046	2E	056
/		Slash	0010 1111	047	2F	057
0		Zero	0011 0000	048	30	060
1		One	0011 0001	049	31	061
2		Two	0011 0010	050	32	062
3		Three	0011 0011	051	33	063
4		Four	0011 0100	052	34	064
5		Five	0011 0101	053	35	065
6		Six	0011 0110	054	36	066
7		Seven	0011 0111	055	37	067
8		Eight	0011 1000	056	38	070
9		Nine	0011 1001	057	39	071
:		Colon	0011 1010	058	3A	072
;		Semicolon	0011 1011	059	3B	073
<		Less than sign	0011 1100	060	3C	074
=		Equal sign	0011 1101	061	3D	075
>		Greater than sign	0011 1110	062	3E	076
?		Question mark	0011 1111	063	3F	077
@	AT	Commercial at sign	0100 0000	064	40	100
A		Uppercase A	0100 0001	065	41	101
B		Uppercase B	0100 0010	066	42	102
C		Uppercase C	0100 0011	067	43	103
D		Uppercase D	0100 0100	068	44	104
E		Uppercase E	0100 0101	069	45	105
F		Uppercase F	0100 0110	070	46	106
G		Uppercase G	0100 0111	071	47	107
H		Uppercase H	0100 1000	072	48	110
I		Uppercase I	0100 1001	073	49	111
J		Uppercase J	0100 1010	074	4A	112
K		Uppercase K	0100 1011	075	4B	113
L		Uppercase L	0100 1100	076	4C	114
M		Uppercase M	0100 1101	077	4D	115
N		Uppercase N	0100 1110	078	4E	116

TABLE B-5
Standard-2 ASCII Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
O		Uppercase O	0100 1111	079	4F	117
P		Uppercase P	0101 0000	080	50	120
Q		Uppercase Q	0101 0001	081	51	121
R		Uppercase R	0101 0010	082	52	122
S		Uppercase S	0101 0011	083	53	123
T		Uppercase T	0101 0100	084	54	124
U		Uppercase U	0101 0101	085	55	125
V		Uppercase V	0101 0110	086	56	126
W		Uppercase W	0101 0111	087	57	127
X		Uppercase X	0101 1000	088	58	130
Y		Uppercase Y	0101 1001	089	59	131
Z		Uppercase Z	0101 1010	090	5A	132
[	LBKT	Left bracket	0101 1011	091	5B	133
\	REVS	Reverse slash, backslash	0101 1100	092	5C	134
]	RBKT	Right bracket	0101 1101	093	5D	135
^	CFLX	Circumflex, caret	0101 1110	094	5E	136
_		Underline, underscore	0101 1111	095	5F	137
`	GRAV	Left single quote, grave accent	0110 0000	096	60	140
a		Lowercase a	0110 0001	097	61	141
b		Lowercase b	0110 0010	098	62	142
c		Lowercase c	0110 0011	099	63	143
d		Lowercase d	0110 0100	100	64	144
e		Lowercase e	0110 0101	101	65	145
f		Lowercase f	0110 0110	102	66	146
g		Lowercase g	0110 0111	103	67	147
h		Lowercase h	0110 1000	104	68	150
i		Lowercase i	0110 1001	105	69	151
j		Lowercase j	0110 1010	106	6A	152
k		Lowercase k	0110 1011	107	6B	153
l		Lowercase l	0110 1100	108	6C	154
m		Lowercase m	0110 1101	109	6D	155
n		Lowercase n	0110 1110	110	6E	156
o		Lowercase o	0110 1111	111	6F	157
p		Lowercase p	0111 0000	112	70	160
q		Lowercase q	0111 0001	113	71	161
r		Lowercase r	0111 0010	114	72	162
s		Lowercase s	0111 0011	115	73	163
t		Lowercase t	0111 0100	116	74	164

TABLE B-5
Standard-2 ASCII Character Set - Continued

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
u		Lowercase u	0111 0101	117	75	165
v		Lowercase v	0111 0110	118	76	166
w		Lowercase w	0111 0111	119	77	167
x		Lowercase x	0111 1000	120	78	170
y		Lowercase y	0111 1001	121	79	171
z		Lowercase z	0111 1010	122	7A	172
{	LBCE	Left brace	0111 1011	123	7B	173
|	VERT	Vertical line	0111 1100	124	7C	174
}	RBCE	Right brace	0111 1101	125	7D	175
~	TIL	Tilde	0111 1110	126	7E	176
	DEL	Delete	0111 1111	127	7F	177

TABLE B-6
EBCDIC Character Set and Collating Sequence

	EBCDIC Codes		
COBOL85 Character Set	*Hex*	*Decimal*	*Octal*
. (period)	4B	075	113
<	4C	076	114
(	4D	077	115
+	4E	078	116
$	5B	091	133
*	5C	092	134
)	5D	093	135
;	5E	094	136
- (minus or hyphen)	60	096	140
/	61	097	141
, (comma)	6B	107	153
	6C	108	154
>	6E	110	156
	6F	111	157
' (apostrophe)	7D	125	175
=	7E	126	176
" (quotation marks)	7F	127	177
a	81	129	201
b	82	130	202
c	83	131	203
d	84	132	204
e	85	133	205
f	86	134	206
g	87	135	207
h	88	136	210
i	89	137	211
j	91	145	221
k	92	146	222
l	93	147	223
m	94	148	224
n	95	149	225
o	96	150	226
p	97	151	227
q	98	152	230
r	99	153	231
s	A2	162	242
t	A3	163	243
u	A4	164	244
v	A5	165	245
w	A6	166	246
x	A7	167	247
y	A8	168	250
z	A9	169	251
A	C1	193	301
B	C2	194	302
C	C3	195	303
D	C4	196	304

TABLE B-6
EBCDIC Character Set and Collating Sequence - Continued

	EBCDIC Codes		
COBOL85 Character Set	*Hex*	*Decimal*	*Octal*
E	C5	197	305
F	C6	198	306
G	C7	199	307
H	C8	200	310
I	C9	201	311
J	D1	209	321
K	D2	210	322
L	D3	211	323
M	D4	212	324
N	D5	213	325
O	D6	214	326
P	D7	215	327
Q	D8	216	330
R	D9	217	331
S	E2	226	342
T	E3	227	343
U	E4	228	344
V	E5	229	345
W	E6	230	346
X	E7	231	347
Y	E8	232	350
Z	E9	233	351
0	F0	240	360
1	F1	241	361
2	F2	242	362
3	F3	243	363
4	F4	244	364
5	F5	245	365
6	F6	246	366
7	F7	247	367
8	F8	248	370
9	F9	249	371

TABLE B-7
Availability of a File

OPEN Mode	*File Present*	*File Not Present*
INPUT	Normal OPEN. Status code 00.	OPEN is unsuccessful. Status code 35.
INPUT (optional file)	Normal OPEN. Status code 00.	Normal OPEN. Status code 05. The first READ causes the AT END or INVALID KEY condition.
I-O	Normal OPEN. Status code 00.	OPEN is unsuccessful. Status code 35.
I-O (optional file)	Normal OPEN. Status code 00.	OPEN causes the file to be created. Status code 05.[1,2]
OUTPUT	Normal OPEN. Status code 00. The file contains no records.[3]	OPEN causes the file to be created. Status code 00.[1,2]
EXTEND	Normal OPEN. Status code 00.	OPEN is unsuccessful. Status code 35.
EXTEND (optional file)	Normal OPEN. Status code 00.	OPEN causes the file to be created. Status code 05.[1,2]

1 PRISAM files must have an existing template created by FAU. Otherwise, the OPEN is unsuccessful, and status code 37 is returned.

2 MIDASPLUS files must have alphanumeric keys and must be fixed-length files. Otherwise, the OPEN is unsuccessful, and status code 37 is returned.

3 Sequential disk and tape files (assigned to PRIMOS or MT9) are truncated if the file contains records. MIDASPLUS and PRISAM files are not truncated. If the file contains records, the OPEN is unsuccessful, and status code 37 is returned.

TABLE B-8
Permissible Input-Output Statements After OPEN Options and Access Modes

File Organization	*File Access Mode*	*Procedure Statement*	*OPEN Option in Effect*			
			INPUT	*OUTPUT*	*I-O*	*EXTEND*
SEQUENTIAL	SEQUENTIAL	READ	X		X	
		WRITE		X		X
		REWRITE			X	
INDEXED	SEQUENTIAL	READ	X		X	
		WRITE		X		
		REWRITE			X	
		START	X		X	
		DELETE			X	
	RANDOM	READ	X		X	
		WRITE			X	
		REWRITE		X	X	
		START				
		DELETE			X	
	DYNAMIC	READ	X		X	
		WRITE		X	X	
		REWRITE			X	
		START	X		X	
		DELETE			X	
RELATIVE	SEQUENTIAL	READ	X		X	
		WRITE		X		
		REWRITE			X	
		START	X		X	
		DELETE			X	
	RANDOM	READ	X		X	
		WRITE		X	X	
		REWRITE			X	
		START				
		DELETE			X	
	DYNAMIC	READ	X		X	
		WRITE		X	X	
		REWRITE			X	
		START	X		X	
		DELETE			X	

X = Permitted.

Hexadecimal, Octal, and Decimal Conversion

To convert a hexadecimal or octal number to decimal with one of the following tables, locate each digit in the correct column position and add the decimal equivalents of all digits. For example, 6C5 in hex equals 1,536 + 192 + 5 in decimal, or 1,733.

To convert from decimal to hexadecimal or octal, locate the largest decimal value in the table that is still smaller than the number to be converted. Note the corresponding hexadecimal or octal value on the left. Then subtract that value from your number, and repeat. Each time, write down the new digit to the right of the last one. For example, 95 is 80 + 15, or 5 from the second hexadecimal column and F from the third column; 95 decimal equals 5F in hexadecimal.

TABLE B-9
Hexadecimal and Decimal Conversion

HEX	*DEC*	*HEX*	*DEC*	*HEX*	*DEC*	*HEX*	*DEC*
0	0	0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	2	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10
B	45056	B	2816	B	176	B	11
C	49152	C	3072	C	192	C	12
D	53248	D	3328	D	208	D	13
E	57344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15
16**3		16**2		16**1		16**0	

TABLE B-10
Octal and Decimal Conversion

OCT	*DEC*	*OCT*	*DEC*	*OCT*	*DEC*	*OCT*	*DEC*	*OCT*	*DEC*
0	0	0	0	0	0	0	0	0	0
1	4096	1	512	1	64	1	8	1	1
2	8192	2	1024	2	128	2	16	2	2
3	12288	3	1536	3	192	3	24	3	3
4	16384	4	2048	4	256	4	32	4	4
5	20480	5	2560	5	320	5	40	5	5
6	24576	6	3072	6	384	6	48	6	6
7	28672	7	3584	7	448	7	56	7	7
8**4		8**3		8**2		8**1		8**0	

TABLE B-11
Hexadecimal Addition Table

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Decimal Data Types

COBOL85 operates on five types of decimal data. Table B-12 summarizes the characteristics of each type.

TABLE B-12
Decimal Data Types (Overpunch Symbols)

Type	*Code*	*Size of Decimal Digit*	*Comments*
Leading Separate Sign	0	8	A plus sign (+) or a space represents a positive number. Operations generate +. A minus sign (–) represents a negative number.
Trailing Separate Sign	1	8	Same as leading separate sign.
Packed Decimal	3	4	Each four bits represent a digit in binary-coded decimal form. The right-most four bits represent the sign of the entire decimal field: bit values 1100 = plus, bit values 1101 = minus.
Leading Embedded Sign	4		A single character represents a digit and the sign of the field. When more than one character is listed, all are recognized but only the first is given in the result field.
Trailing Embedded Sign	5	8	Embedded sign characters have the following meanings:

Digit	*Positive*	*Negative*
0	0,+{	–,}
1	1A	J
2	2B	K
3	3C	L
4	4D	M
5	5E	N
6	6F	O
7	7G	P
8	8H	Q
9	9I	R

Error Messages

The following categories of error messages can be generated when you compile and run COBOL85 programs. These messages appear on the screen if you interactively compile and run the program:

- COBOL85 compile time error messages
- COBOL85 runtime error messages
- PRIMOS error messages

Compile Time Error Messages

These messages are displayed on the screen. They are also stored in the file *program*.ERROR and inserted after the source listing, if you specify either the –LISTING or the –ERRORFILE option at compile time. The format of compile time error messages is explained in the section Compiler Error Messages, in Chapter 2. For example,

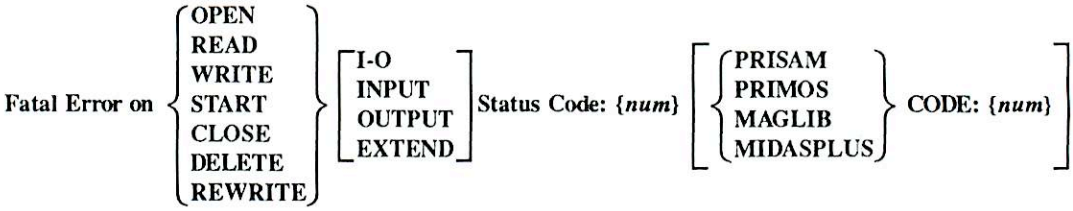
```
ERROR 407  SEVERITY 2  LINE 15  COLUMN 19  [WARNING, SEMANTICS]
The initial value for "A9" exceeds the range of values allowed by
the PICTURE or by the default implementation size.  The initial value
may be truncated or unpredictable.

[1 WARNING IN PROGRAM: TRUNCATE.COBOL85]

ERROR 300  SEVERITY 3  LINE 25  COLUMN 40  [FATAL, SYNTAX]
"PICTURE" found when expecting one of {"RIGHT"}.
  [syntax checking suspended at THE RESERVED WORD "PICTURE"]

ERROR 64  SEVERITY 3  LINE 25  COLUMN 12  [FATAL, SEMANTICS]
This clause has been specified more than once in this entry.

ERROR 329  SEVERITY 3  LINE 77  COLUMN 16  [FATAL, SEMANTICS]
"ACCT-IN" is an undefined data reference.
```

```
ERROR 347  SEVERITY 1  LINE 80  COLUMN 26  [OBSERVATION, SEMANTICS]
The section that immediately contains the RELEASE statement was not
named as an input procedure associated with a SORT or MERGE statement.
Check that the perform range of the applicable procedure contains
this section.

[3 FATALS 1 OBSERVATION IN PROGRAM: <MYMFD>MYDIR>FATAL.ERROR.COBOL85]
```

The compile time error messages are self-explanatory. If you encounter a SEVERITY 4 message, recompile the program after eliminating all other error messages. Recompilation usually eliminates the SEVERITY message also. If a SEVERITY 4 message persists, consult a Prime System Analyst.

COBOL85 Runtime Error Messages

These messages are displayed when, for example, subroutines called by COBOL85 cannot perform operations such as file I-O. The message, which is self-explanatory, describes the error and file involved, and gives the name of the subroutine that raised the error condition. If applicable, the message also includes the PRISAM, PRIMOS, MIDASPLUS, or MAGLIB error code. The format of the runtime error message is



[ERRORTEXT]
[E$ ***message from*** **PRIMOS]** ***pathname (caller)***

Examples of runtime error messages follow:

```
Fatal Error on OPEN INPUT. Status Code: 39.

The file ORGANIZATION specified (INDEXED) does not match the actual file
ORGANIZATION found for this file (RELATIVE).
MYRELFILE (CB$OPEN)
ER!

Fatal Error on READ. Status Code: 47.

Attempted READ from file not open in INPUT or I-O mode.
MYFILE (CB$RS)
ER!

Fatal Error on WRITE. Status Code: 99.  PRISAM Code: 212.
MYFILE.PRISAM (CB$WS)
ER!
```

```
Fatal Error on OPEN INPUT. Status Code: 30.  PRIMOS Code: 10.
Insufficient access rights.  *>NO_RIGHTS>YOURFILE (CB$OPEN)
ER!

Fatal Error on OPEN INPUT. Status Code: 35.
Not found.  MYFILE (CB$OPEN)
ER!

Fatal Error on OPEN INPUT. Status Code: 39.

Midasplus has no information for minimum and maximum variable length record
sizes for this file.  Use CREATK Initialize Function if file is empty; or Get
Function for non-empty files.
MYFILE.MIDAS (CB$OPEN)
ER!

Fatal Error on OPEN INPUT. Status Code: 39.

The maximum or minimum record sizes specified in program exceed the actual
minimum or maximum record sizes found for this file:
   Program minimum: 88wds    Actual minimum: 50wds
   Program maximum: 175wds   Actual maximum: 175wds
MYFILE.MIDAS (CB$OPEN)
ER!
```

MIDASPLUS error codes and messages are listed numerically in the *MIDASPLUS User's Guide*. For more information on PRISAM error codes, see Appendix D and the *PRISAM User's Guide*. For a complete discussion of COBOL85 file status codes, see Chapter 4.

PRIMOS Error Messages

These messages are generated by PRIMOS, the Prime operating system. They are explained in the *PRIMOS User's Guide*. An example is

```
ERROR:  condition POINTER_FAULT$ raised at 4011(3)1011
```

In this case, a sort program was run without loading VSRTLI.

Common System Runtime Messages

The following system error conditions can occur during execution of a COBOL85 program.

ACCESS_VIOLATIONS$ The run unit or runfile attempted to violate the CPU access rules. This condition aborts the run unit and can be caused by a variety of factors. One common cause is reference to a table with an out-of-range subscript. Use the –RANGE option to locate the statement that caused the subscript to go out of range.

ARITH$ An arithmetic exception involved data overflow of fixed or floating point operands. This condition can occur only if the –SIGNALERRORS option was specified, or for an exponential operation. The run unit aborts. If –SIGNALERRORS was not specified, execution of the run unit continues with truncation of the value that would have caused the exception. A division by zero causes this condition to occur when –SIGNALERRORS is specified.

ERROR$	A conversion involving illegal characters was attempted. Implicit conversions can occur in COBOL85 programs when fields of different types are moved. The run unit aborts. In general, this condition does not occur unless the –SIGNALERRORS compile-time option was specified. Recompiling the program with the READY TRACE statement, or recompiling with the –DEBUG option and then executing the run unit under Debugger control, helps to locate the illegal conversion.
LINKAGE_FAULT$	An **unsnapped** link (unresolved call) was encountered but the reference could not be found in the system entry point table. The run unit aborts. Either not enough libraries were loaded or a program referenced in a COBOL85 CALL statement was not loaded when this run unit was linked with the BIND or SEG utility. If a map was created when the run unit was linked, check it for unresolved entries. Otherwise, invoke BIND or SEG again and, after LI, enter MAP 3.
OUT_OF_BOUNDS$	*See* ACCESS_VIOLATION$.
POINTER_FAULT$	A reference was made to an Indirect Pointer (IP) but the pointer does not appear to be valid. The run unit aborts. The most likely cause is a link base that was destroyed by a MOVE statement with out-of-range subscripts. Recompile with the –RANGE option.
ERROR	This condition can be caused by a variety of errors, but if the program is compiled with –RANGE and an out of bounds reference is found, this condition is preceded by a message containing the line number and subscript value where the invalid array reference occurred.

D

PRISAM to COBOL85 Status Code Mapping

This appendix includes three tables listing PRISAM status codes and their corresponding COBOL85 status codes. The tables are sorted as follows:

- Table D-1 numerically sorts the COBOL85 status codes.
- Table D-2 numerically sorts the PRISAM status codes.
- Table D-3 alphabetically lists the PRISAM status names that translate to COBOL85 status codes.

Descriptions of the COBOL85 status codes can be found in Chapter 4. For more information on PRISAM status codes, see the *PRISAM User's Guide.*

TABLE D-1
PRISAM to COBOL85 Status Code Mapping Sorted Numerically by COBOL85 Status Code

COBOL85 Status Code	*PRISAM Status Number*	*PRISAM Status Name*	*Description*
00	0	OKOK	PRISAM function successfully completed
10	24	ER$HOF	High end of file
10	25	ER$LOF	Low end of file
22	141	ER$KNM	Key cannot be modified during update
22	137	ER$NDA	No duplicate keys allowed
23	23	ER$NFD	Record not found
24	153	ER$IRN	Invalid relative record number
30	9	ER$DKF	Disk is full or maximum quota exceeded (indexed or relative files)

TABLE D-1
PRISAM to COBOL85 Status Code Mapping Sorted Numerically by COBOL85 Status Code - Continued

COBOL85 Status Code	PRISAM Status Number	PRISAM Status Name	Description
34	9	ER$DKF	Disk is full or maximum quota exceeded (sequential files)
43	148	ER$RNL	Record not active
44	129	ER$RBL	Invalid record buffer length
46	130	ER$NCR	No current record position
97	184	ER$ABT	Transaction was aborted
97	185	ER$TAB	Transaction timeout abort
97	183	ER$TIM	Timeout occurred
99	170	ER$AIF	Active AI file is full
99	187	ER$APP	PRISAM process was aborted
99	188	ER$APS	PRISAM system was aborted
99	171	ER$BIO	BI file overflow
99	207	ER$BKI	Bad key information
99	193	ER$CLT	File close invalid during transaction
99	234	ER$DDM	Unspecified DDM error
99	161	ER$DEX	Maximum number of duplicates exceeded
99	182	ER$DLK	Record deadlock detected
99	202	ER$FAB	File access aborted
99	215	ER$FBS	Invalid found key buffer length
99	216	ER$FBT	Found key value truncated
99	20	ER$FCD	Invalid function code
99	134	ER$FID	Invalid file indentifier
99	194	ER$IDB	Invalid decimal digit in key

TABLE D-1
PRISAM to COBOL85 Status Code Mapping Sorted Numerically by COBOL85 Status Code - Continued

COBOL85 Status Code	*PRISAM Status Number*	*PRISAM Status Name*	*Description*
99	196	ER$IDD	Invalid packed decimal digit in key
99	195	ER$IDS	Invalid decimal sign in key
99	232	ER$IDT	Invalid distributed transaction
99	168	ER$IFF	Invalid file organization
99	213	ER$IFU	Insufficient PRIMOS file-units
99	205	ER$INF	Invalid key information length
99	191	ER$IOM	File in extend only mode cannot be read
99	142	ER$IPK	Invalid partial key value
99	21	ER$KBS	Invalid key buffer length
99	204	ER$KDB	Key descriptor block too small
99	22	ER$KRB	Invalid key of reference number
99	206	ER$NKF	No such key defined in the file
99	181	ER$NOI	File not opened for inserting records
99	180	ER$NOU	File not opened for updating records
99	233	ER$NRR	No remote receive
99	135	ER$NUK	No unique matching key definition
99	189	ER$PNA	PRISAM system not available
99	190	ER$PNP	PRISAM system not available to process
99	12	ER$PRI	Nonspecific PRIMOS error
99	222	ER$ROM	Nonspecific ROAM error
99	231	ER$RSU	Remote system unavailable
99	217	ER$RTV	Update invalid in retrieval transaction

TABLE D-2
PRISAM to COBOL85 Status Code Mapping Sorted Numerically by PRISAM Status Number

COBOL85 Status Code	PRISAM Status Number	PRISAM Status Name	Description
00	0	OKOK	PRISAM function successfully completed
30	9	ER$DKF	Disk is full or maximum quota exceeded (indexed or relative files)
34	9	ER$DKF	Disk is full or maximum quota exceeded (sequential files)
99	12	ER$PRI	Nonspecific PRIMOS error
99	20	ER$FCD	Invalid function code
99	21	ER$KBS	Invalid key buffer length
99	22	ER$KRB	Invalid key of reference number
23	23	ER$NFD	Record not found
10	24	ER$HOF	High end of file
10	25	ER$LOF	Low end of file
44	129	ER$RBL	Invalid record buffer length
46	130	ER$NCR	No current record position
99	134	ER$FID	Invalid file identifier
99	135	ER$NUK	No unique matching key definition
22	137	ER$NDA	No duplicate keys allowed
22	141	ER$KNM	Key cannot be modified during update
99	142	ER$IPK	Invalid partial key value
43	148	ER$RNL	Record not active
24	153	ER$IRN	Invalid relative record number
99	161	ER$DEX	Maximum number of duplicates exceeded
99	168	ER$IFF	Invalid file organization
99	170	ER$AIF	Active AI file is full

TABLE D-2
PRISAM to COBOL85 Status Code Mapping Sorted Numerically by PRISAM Status Number - Continued

COBOL85 Status Code	*PRISAM Status Number*	*PRISAM Status Name*	*Description*
99	171	ER$BIO	BI file overflow
99	180	ER$NOU	File not opened for updating records
99	181	ER$NOI	File not opened for inserting records
99	182	ER$DLK	Record deadlock detected
97	183	ER$TIM	Timeout occurred
97	184	ER$ABT	Transaction was aborted
97	185	ER$TAB	Transaction timeout abort
99	187	ER$APP	PRISAM process was aborted
99	188	ER$APS	PRISAM system was aborted
99	189	ER$PNA	PRISAM system not available
99	190	ER$PNP	PRISAM system not available to process
99	191	ER$IOM	File in extend only mode cannot be read
99	193	ER$CLT	File close invalid during transaction
99	194	ER$IDB	Invalid decimal digit in key
99	195	ER$IDS	Invalid decimal sign in key
99	196	ER$IDD	Invalid packed decimal digit in key
99	202	ER$FAB	File access aborted
99	204	ER$KDB	Key descriptor block too small
99	205	ER$INF	Invalid key information length
99	206	ER$NKF	No such key defined in the file
99	207	ER$BKI	Bad key information
99	213	ER$IFU	Insufficient PRIMOS file-units
99	215	ER$FBS	Invalid found key buffer length

TABLE D-2
PRISAM to COBOL85 Status Code Mapping Sorted Numerically by PRISAM Status Number - Continued

COBOL85 Status Code	*PRISAM Status Number*	*PRISAM Status Name*	*Description*
99	216	ER$FBT	Found key value truncated
99	217	ER$RTV	Update invalid in retrieval transaction
99	222	ER$ROM	Nonspecific ROAM error
99	231	ER$RSU	Remote system unavailable
99	232	ER$IDT	Invalid distributed transaction
99	233	ER$NRR	No remote receive
99	234	ER$DDM	Unspecified DDM error

TABLE D-3
PRISAM to COBOL85 Status Code Mapping (Sorted Alphabetically by PRISAM Status Name)

COBOL85 Status Code	*PRISAM Status Number*	*PRISAM Status Name*	*Description*
97	184	ER$ABT	Transaction was aborted
99	170	ER$AIF	Active AI file is full
99	187	ER$APP	PRISAM process was aborted
99	188	ER$APS	PRISAM system was aborted
99	171	ER$BIO	BI file overflow
99	207	ER$BKI	Bad key information
99	193	ER$CLT	File close invalid during transaction
99	161	ER$DEX	Maximum number of duplicates exceeded
99	234	ER$DDM	Unspecified DDM error
30	9	ER$DKF	Disk is full or maximum quota exceeded (indexed or relative files)

TABLE D-3
PRISAM to COBOL85 Status Code Mapping (Sorted Alphabetically by PRISAM Status Name) - Continued

COBOL85 Status Code	*PRISAM Status Number*	*PRISAM Status Name*	*Description*
34	9	ER$DKF	Disk is full or maximum quota exceeded (sequential files)
99	182	ER$DLK	Record deadlock detected
99	202	ER$FAB	File access aborted
99	215	ER$FBS	Invalid found key buffer length
99	216	ER$FBT	Found key value truncated
99	20	ER$FCD	Invalid function code
99	134	ER$FID	Invalid file indentifier
10	24	ER$HOF	High end of file
99	194	ER$IDB	Invalid decimal digit in key
99	196	ER$IDD	Invalid packed decimal digit in key
99	195	ER$IDS	Invalid decimal sign in key
99	232	ER$IDT	Invalid distributed transaction
99	168	ER$IFF	Invalid file organization
99	213	ER$IFU	Insufficient PRIMOS file units
99	205	ER$INF	Invalid key information length
99	191	ER$IOM	File in extend only mode cannot be read
99	142	ER$IPK	Invalid partial key value
24	153	ER$IRN	Invalid relative record number
99	21	ER$KBS	Invalid key buffer length
99	204	ER$KDB	Key descriptor block too small
22	141	ER$KNM	Key cannot be modified during update
99	22	ER$KRB	Invalid key of reference number

TABLE D-3
PRISAM to COBOL85 Status Code Mapping (Sorted Alphabetically by PRISAM Status Name) - Continued

COBOL85 Status Code	*PRISAM Status Number*	*PRISAM Status Name*	*Description*
10	25	ER$LOF	Low end of file
46	130	ER$NCR	No current record position
22	137	ER$NDA	No duplicate keys allowed
23	23	ER$NFD	Record not found
99	206	ER$NKF	No such key defined in the file
99	181	ER$NOI	File not opened for inserting records
99	180	ER$NOU	File not opened for updating records
99	233	ER$NRR	No remote receive
99	135	ER$NUK	No unique matching key definition
99	189	ER$PNA	PRISAM system not available
99	190	ER$PNP	PRISAM system not available to process
99	12	ER$PRI	Nonspecific PRIMOS error
44	129	ER$RBL	Invalid record buffer length
43	148	ER$RNL	Record not active
99	222	ER$ROM	Nonspecific ROAM error
99	231	ER$RSU	Remote system unavailable
99	217	ER$RTV	Update invalid in retrieval transaction
97	185	ER$TAB	Transaction timeout abort
97	183	ER$TIM	Timeout occurred
00	0	OKOK	PRISAM function successfully completed

The Debugger Interface

The *Source Level Debugger User's Guide* documents the symbolic debugger. This appendix describes the COBOL85 interface to the Debugger, and restrictions to debugging in COBOL85.

Overview

To use the Debugger with COBOL85, follow these steps:

1. Compile the program using the –DEBUG option. For example, if the program to be debugged is RANDOM.COBOL85, enter the following:

```
OK, COBOL85 RANDOM -DEBUG
```

2. Link the program in the usual way:

```
OK, BIND
[BIND Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
: LO RANDOM
: LI COBOL85LIB
: LI
BIND COMPLETE
: FILE
```

3. The format to invoke the Debugger is

   DBG RANDOM [*option-1* [*option-2*]...]

   If the program was compiled with the default loading steps described in Chapter 3, the source filename may be used after DBG. Otherwise, use the runfile name.

4. Follow instructions in the the *Source Level Debugger User's Guide*. Note the following special definitions:

   - *procedure-name* and *program-block-name* mean the entire COBOL85 program named by *program-id*. Separately called programs define other *procedure-names* or *program-block-names*.

- *labels* mean *paragraph-names* and *section-names*. For the Debugger, names that begin with a number must be preceded by a dollar sign ($). Thus, 040-EDIT would be entered as $040-EDIT.
- Array elements are referred to by the *array-name* followed by the element number within parentheses. Multiple subscripts are separated by commas. Thus, element 3 of the array TABLE1 is referred to as TABLE1(3). Element 3 of the second dimension of TABLE2 is TABLE2(2,3).
- Elements with the same names must be referred to as

  ***element-name* OF *group-name*.**

- There is no way to set a breakpoint on a *paragraph-name* that is not unique. *paragraph-names*, therefore, may not be qualified by *section-names* in breakpoint identifiers. Instead, set the breakpoint on the first statement in the paragraph.
- The Debugger does not accept numeric literals longer than 14 digits.
- LET cannot be used with edited data types.
- LET cannot be used with an index name or an index data item.
- Evaluation of abbreviated conditional expressions is not supported.
- Results of arithmetic operations on scaled binary values are incorrect if decimal points are not aligned.
- Switches cannot be displayed.
- Assignments are not right-justified to justified data items.
- Any source line with D in column 7 is executed.

5. To leave the Debugger, enter Q.

Examples

The first example uses the Debugger to examine data elements of the program RANDOM.COBOL85 at the end of Chapter 10. The program uses three Debugger commands: BRK to set a breakpoint, the colon (:) to display data values, and LET to modify data values. Many other Debugger commands are discussed in the *Source Level Debugger User's Guide*.

The Debugger allows the program to present the usual requests for file assignments. If the program opens files, only C (continue) should be used after a breakpoint. RESTART may cause the program to attempt to open files that are not closed.

```
OK, DBG RANDOM

**Dbg**   revision 22   (25-March (c) Prime Computer 1988)

> BRK $200-CREATE-ERROR-FILE
> RESTART
trace:  010-PRINT-HEADINGS
trace:  020-PROCESS-TRANS
trace:  100-UPDATE
trace:  020-PROCESS-TRANS
trace:  110-ADD
```

```
**** breakpointed at RANDOM1\151 ($200-CREATE-ERROR-FILE)
> : PRINT-COUNT
PRINT-COUNT = 0
> : ENTRY-CODE
ENTRY-CODE = 'D'
> : NEW-CODE
NEW-CODE = ''
> : TRANS-ENTRY
TRANS-ENTRY.ACCT-ENTRY = '414'
TRANS-ENTRY.FILLER = '061185   PRIME COMPUTER      4360000123'
> LET PRINT-COUNT = 1
> C
trace:  200-CREATE-ERROR-FILE
trace:  020-PROCESS-TRANS
trace:  120-DELETE

**** breakpointed at RANDOM1\151 ($200-CREATE-ERROR-FILE)
> : PRINT-COUNT
PRINT-COUNT = 2
> Q
```

The next example illustrates the use of ED commands within the Debugger environment. The command SRC NAME elicits the *program-name* of the program being debugged. The command SRC L PROC causes ED to locate the first instance of PROCEDURE. The command SRC P15 causes ED to display 15 lines of source code.

```
OK, DBG RANDOM

**Dbg**   revision 22   (25-March (c) Prime Computer 1988)

> SRC NAME
Source file is "<PUBS>EVELYN>RANDOM.COBOL85", based on evaluation environment.
> SRC L PROC
   95:         PROCEDURE DIVISION.
> SRC P15
   95:         PROCEDURE DIVISION.
   96:        *
   97:        *DECLARATIVES.
   98:        *   THIS SECTION SHOULD DISPLAY FILE-STATUS FOR ANY ERRORS
   99:        *   NOT CAUGHT BY INVALID KEY OR AT END CLAUSES.
  100:        *END DECLARATIVES.
  101:        *
  102:         000-MAINLINE.
  103:             READY TRACE.
  104:             OPEN INPUT TRANS-FILE,
  105:                  I-O MASTER-FILE,
  106:                  I-O NEW-FILE,
  107:                  OUTPUT PRINT-FILE.
  108:             PERFORM 010-PRINT-HEADINGS.
  109:             READ TRANS-FILE AT END
> Q
OK,
```

F

Prime Support of the ANSI Standard

COBOL85 implements the intermediate level (level 1) of American National Standards Institute (ANSI) COBOL, as defined in the document *American National Standard Programming Language COBOL* X3.23-1985, published by the American National Standards Institute, New York, 1985. COBOL85 also implements a number of high-level (level 2) features of Standard COBOL. This appendix lists all of the Standard COBOL features available in COBOL85.

Prime has added a number of extensions to the ANSI standard. Throughout this book, Prime extensions are printed in red to identify them as such. This appendix also lists all of these extensions.

Standard COBOL Features in COBOL85

ANSI Module	*Features Available in COBOL85*
Nucleus	Full level 2, with these exceptions: Reference modification SYMBOLIC CHARACTER ALPHABET-NAME IS literal END-PROGRAM ACCEPT multiple transfers ACCEPT DAY-OF-WEEK EVALUATE INITIALIZE INSPECT with CONVERTING phrase PERFORM with TEST phrase PERFORM with unlimited AFTER SET condition-name TO TRUE MOVE with de-editing

ANSI Module	*Features Available in COBOL85*
Sequential I-O	Full level 2, with these exceptions: LINAGE PADDING CHARACTER RECORD DELIMITER RECORD IS VARYING DEPENDING ON CLOSE FOR REMOVAL/LOCK
Relative I-O	Full level 2, with these exceptions: OPTIONAL (I-O and EXTEND modes) EXTEND mode RECORD IS VARYING DEPENDING ON CLOSE with LOCK phrase REWRITE records of unequal lengths
Indexed I-O	Full level 2, with these exceptions: OPTIONAL (I-O and EXTEND modes) EXTEND mode RECORD IS VARYING DEPENDING ON CLOSE with LOCK phrase REWRITE records of unequal lengths READ not required for REWRITE with secondary keys (dynamic/random)
Interprogram Communication	Full level 2, with these exceptions: Nested source programs COMMON INITIAL GLOBAL CALL BY REFERENCE/CONTENT CALL ON OVERFLOW/EXCEPTION CANCEL
Sort-merge	Full level 2, with this exception: RECORD IS VARYING DEPENDING ON
Source Text Manipulation	Full level 2, with this exception: REPLACE
Report-Writer	Not included
Debug	Not included
Communication	Not included

Prime Extensions to the ANSI Standard

The –STANDARD compiler option causes the compiler to issue observations when the COBOL85 source code contains Prime extensions to the ANSI standard.

The basic format for observations generated when you specify –STANDARD is

```
ERROR 5xx SEVERITY 1 LINE...

Prime extension: <explanatory message>
```

The following sections list Prime extensions to ANSI COBOL 85.

All Program Divisions

- Missing periods on defined divisions, sections, and paragraph headers are allowed.
- Support of EJECT as a compiler-directing statement is provided.
- Support of SKIP1, SKIP2, and SKIP3 as compiler-directing statements is provided.
- Support of the single quote character in place of the double quote character is provided.

IDENTIFICATION DIVISION

- Optional ID DIVISION entries are allowed to be out of order.
- Support of the REMARKS paragraph in the ID DIVISION is provided.
- ID is allowed as an abbreviation for IDENTIFICATION.

ENVIRONMENT DIVISION

- ENVIRONMENT DIVISION entries are allowed to be out of order.
- Empty FILE-CONTROL paragraph is allowed.
- The FILE-STATUS entry is allowed to be numeric.

DATA DIVISION

- Missing terminating periods on *data-descriptions* not ending with a VALUE clause are allowed.
- Specification of the FILLER clause on a level 01, group item, or redefined item is allowed.
- Specification of the OCCURS clause on an 01 level is allowed.
- Specification of the COMPRESSED and UNCOMPRESSED phrases in the FD entry is allowed.
- Specification of COMP-1 and COMP-2 as floating point descriptions in the USAGE clause is allowed.
- Support of an eighth level of subscripting is provided.

- Support of subscripted variables as subscripts is provided.
- Support of qualified *index-names* as subscripts is provided.
- Specification of non-01 level numbers in Area A is allowed.
- Support of the underscore character in user-defined words is provided.
- Support of user-defined words with underscores and no alphabetic characters is provided.
- Support of COMP-3 as a synonym for PACKED-DECIMAL is provided.
- Support of hyphens at the end of user-defined words is provided.
- RECORDING MODE IS clause is supported.

PROCEDURE DIVISION

- Nonnumeric literals may be specified using mnemonic-codes through the support of backslash (\) in the character set.
- Support of EXHIBIT NAMED as a PROCEDURE DIVISION verb is provided.
- Support of the GOBACK statement as a synonym for EXIT PROGRAM is provided.
- Specification of the ROUNDED clause on numeric MOVEs is allowed.
- Support of the NOTE statement as a comment entry in the PROCEDURE DIVISION is provided.
- Specification of the OTHERWISE phrase in the IF statement is allowed.
- Support of the READY TRACE and RESET TRACE statements as debugging tools in the PROCEDURE DIVISION is provided.
- Support of arithmetic expressions in place of *data-names* for ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPUTE, GO TO...DEPENDING ON, MOVE, PERFORM...VARYING, SET, and IF statements as well as indexing and subscripting is provided.
- Specification of the CORRESPONDING clause on the IF, MULTIPLY, DIVIDE, and COMPUTE statements is allowed.
- Support of literals as operands of some clauses of the INSPECT statement is provided.
- Support of passing subgroup data items through the CALL statement is provided.
- The PROGRAM-ID and DATE-COMPILED entries are allowed to be referenced in the PROCEDURE DIVISION.
- The reserved word ERROR is always optional in the ON SIZE ERROR clause.
- Support of Area A for PROCEDURE DIVISION statements that contain only one reserved word in Area A is provided.
- The reserved word SENTENCE in the NEXT SENTENCE phrase is optional.
- *data-name* operands that contain a REDEFINES clause can be specified in the USING clause of the PROCEDURE DIVISION header.
- Optional spaces around arithmetic operators are allowed.

G

Obsolete Language Elements

The –ANSI_OBSOLETE compiler option causes the compiler to generate observations for all language elements that ANSI X3.23-1985 includes on the obsolete language element list. The presence of an element on this list means that the element will not appear in the next ANSI COBOL standard. This does not imply that the element will be removed from future revisions of Prime COBOL85.

The basic format for observations generated when you specify –ANSI_OBSOLETE is

```
ERROR 550 SEVERITY 1 LINE...
Use of the <statement, clause, literal, or paragraph> has been placed
in the obsolete element category of ANSI X3.23-1985 and will become obsolete at
the next revision of the ANSI standard.
```

The following COBOL85 language elements are on the ANSI obsolete element list:

- ALL literal with numeric or numeric edited items

  When the literal ALL is used with literals longer than one character that are being sent to numeric or numeric edited data items, the compiler issues an observation.
- AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED and SECURITY paragraphs

  When any of these paragraphs is specified in the ID DIVISION, the compiler issues an observation.
- MEMORY SIZE clause

  The MEMORY SIZE clause currently only serves as documentation to the OBJECT-COMPUTER paragraph and is checked for proper syntax. When this clause is specified in the OBJECT-COMPUTER paragraph, the compiler issues an observation.
- RERUN clause

  The RERUN clause is currently checked only for syntax in the I-O-CONTROL paragraph. When this clause is specified in the I-O-CONTROL paragraph, the compiler issues an observation.
- MULTIPLE FILE TAPE clause

  When this clause is specified in the I-O-CONTROL paragraph, the compiler issues an observation.

- LABEL RECORDS clause

  When this clause is specified in an FD entry, the compiler issues an observation.

- VALUE OF clause

  When this clause is specified in an FD entry, the compiler issues an observation.

- DATA RECORDS clause

  When the DATA RECORDS clause is specified in an FD entry, the compiler issues an observation.

- ALTER statement

  When the ALTER statement is used, the compiler issues an observation.

- ENTER statement

  The compiler generates a warning stating that the ENTER statement is not supported in this implementation. When the ENTER statement is used, the compiler also issues an observation.

- Optional *paragraph-name* in GO TO statement

  A GO TO statement without a *paragraph-name* must be acted upon by an ALTER statement prior to the execution of the GO TO statement. Therefore, when a GO TO statement is used without the corresponding *paragraph-name*, the compiler issues an observation.

- REVERSED phrase of the OPEN statement

  The REVERSED phrase of the OPEN statement generates a fatal error stating that it is not implemented. In addition to the fatal error, the compiler issues an observation.

- STOP literal statement

  When the STOP literal statement is used, the compiler issues an observation.

- Segmentation module

  When segment numbers are specified after a *section-name* and the reserved word SECTION, the compiler issues an observation.

H

Conversion From CBL to COBOL85

If you wish to take advantage of new COBOL85 functionality in an existing CBL program, you must first convert the CBL program to COBOL85. Then you must recompile and relink the program before you execute it. If the resulting COBOL85 program is larger than one segment, you *must* use BIND to link and execute it. If your program is less than one segment, you may use either BIND or SEG.

COBOL85 contains 112 new reserved words, 21 new I-O status codes, and many new error conditions that are handled differently from the way CBL handles them. You must be aware of these differences when you convert a CBL program to COBOL85.

COBOL85 also contains other features not available in CBL that you may wish to use in your converted CBL program.

This appendix documents the differences between CBL and COBOL85 that may require attention during program conversion. For additional information see the *CBL to COBOL85 Conversion Program Guide.*

New Reserved Words

COBOL85 has 112 new reserved words. If your CBL program uses any of these reserved words as identifiers, COBOL85 flags each such occurrence as a fatal error. Replace all such occurrences of COBOL85 reserved words with new identifier names. The following list contains the reserved words that are new in COBOL85:

ALPHABETIC-LOWER	ALPHABETIC-UPPER	ALPHANUMERIC
ALPHANUMERIC-EDITED	ANY	BINARY
CD	CF	CH
CLASS	COBOL	CODE
COLUMN	COMMON	COMMUNICATION
CONTENT	CONTINUE	CONTROL
CONTROLS	CONVERTING	DAY-OF-WEEK
DE	DEBUG-CONTENTS	DEBUG-ITEM
DEBUG-LINE	DEBUG-NAME	DEBUG-SUB-1
DEBUG-SUB-2	DEBUG-SUB-3	DESTINATION
DETAIL	DISABLE	EGI

EMI	ENABLE	END-ADD
END-CALL	END-COMPUTE	END-DELETE
END-DIVIDE	END-EVALUATE	END-IF
END-MULTIPLY	END-PERFORM	END-READ
END-RECEIVE	END-RETURN	END-REWRITE
END-SEARCH	END-START	END-STRING
END-SUBTRACT	END-UNSTRING	END-WRITE
ESI	EVALUATE	FALSE
FINAL	GENERATE	GLOBAL
GROUP	HEADING	INDICATE
INITIALIZE	INITIATE	LAST
LENGTH	LIMIT	LIMITS
LINAGE-COUNTER	LINE-COUNTER	MESSAGE
NUMBER	NUMERIC-EDITED	OTHER
PACKED-DECIMAL	PADDING	PAGE-COUNTER
PF	PH	PLUS
PRINTING	PURGE	QUEUE
RD	RECEIVE	REFERENCE
REPLACE	REPORT	REPORTING
REPORTS	RF	RH
SEGMENT	SEND	SOURCE
STANDARD-2	SUB-QUEUE-1	SUB-QUEUE-2
SUB-QUEUE-3	SUM	SUPPRESS
SYMBOLIC	TABLE	TERMINAL
TERMINATE	TEST	TEXT
TRUE	TYPE	<=
>=		

See Table B-2 in Appendix B for a complete list of COBOL85 reserved words.

New I-O Status Codes and Error Handling

COBOL85 contains 21 new I-O status codes and has slightly changed meanings for several other existing status codes. COBOL85 recognizes many new error conditions that are handled differently from the way CBL handles them. This may cause conversion problems depending on the application.

For example, a CBL program may check for specific status codes that are no longer meaningful or whose meaning has changed. Consequently, the program may invoke a USE procedure to handle errors that the USE procedure was not designed to handle. This discrepancy, in turn, may affect program behavior depending on the application.

This section lists all COBOL85 I-O status codes, and discusses differences between the meanings of each code in CBL and COBOL85 that may require attention during program conversion.

Note

For complete definitions of COBOL85 I-O status codes, and a complete discussion of error recovery, see Chapter 4.

Status Code 00

No conversion concerns. The meaning of this status code in COBOL85 is the same as its meaning in CBL.

Status Code 02 (Indexed)

CBL allows duplicate keys on a READ, WRITE, or REWRITE, and returns status code 00. COBOL85 returns status code 02 when it detects duplicate keys. If a program checks for 00 following any of these statements, and duplicate keys are a possibility in the application, modify the program to check for status code 02.

Status Code 04 (Sequential, Relative, Indexed)

CBL programs that check the file status for 00 to indicate a successful READ for variable-length records must be modified. However, modification is required only for programs that process filetypes for which size conflicts are permissible in CBL. Some size conflicts are ignored in CBL; some generate an invalid key condition; some generate a runtime abort. See Tables H-1 through H-4 for a complete list of record size conflict differences between CBL and COBOL85.

Status Code 05 (Sequential, Indexed, Relative)

CBL programs that open, in I-O mode or EXTEND mode, sequential disk files that do not exist prior to program execution must be modified to describe such files as OPTIONAL in the SELECT clause in order for COBOL85 to create them. Any associated checking for a file status of 00 that is done after the OPEN must be modified.

Status Code 07 (Sequential)

No conversion concerns. CBL issues a fatal diagnostic at compile time for optional phrases on an OPEN or CLOSE statement.

Status Code 10 (Sequential, Relative, Indexed)

No conversion concerns. Although the second meaning for status code 10 is new in COBOL85, CBL does not support OPTIONAL, so such a program fails at OPEN time if an input file does not exist.

Status Code 14 (Relative)

If a relative key data item is not large enough to contain the key value of the record read, in CBL the record is returned, the relative key data item overflows, and the file status item is set to 00. The value returned to the relative key data item is undefined. In COBOL85 the record is not returned; the file status item is set to 14; the imperative statement following AT END is executed (or a USE procedure is invoked if no AT END clause exists); and the value of the relative key data item is undefined.

Status Code 21 (Indexed)

When a READ statement is required prior to a REWRITE, and the primary key value is changed by the program between the successful execution of a READ and the execution of the next REWRITE statement for the file, CBL returns status code 22. COBOL85 returns 21 to signal this condition.

CBL does not check that the successive primary record key values are in ascending order during a WRITE operation for a sequentially accessed file; CBL writes the record to the file. COBOL85 does such a check, and returns status code 21 if successive primary record key values are not in ascending order.

Status Code 22 (Relative, Indexed)

Creating invalid duplicate alternate keys returns a 92 in CBL for MIDASPLUS files.

Also, using the DUPLICATES phrase specified in the program is new in COBOL85. If a CREATK specification allows DUPLICATES, but the program does not specify DUPLICATES, the record is not written in COBOL85, but the record is written in CBL.

Status Code 23 (Relative, Indexed)

No conversion concerns. Although the second meaning for status code 23 is new in COBOL85, CBL does not support OPTIONAL, so such a program fails at OPEN time if an input file does not exist.

Status Code 24 (Relative)

If a program attempts to write beyond the externally defined boundaries of a relative file, CBL returns status code 96. COBOL85 returns 24 to signal this condition.

When a sequential WRITE statement is attempted for a relative file and the number of significant digits in the relative record number is larger than the size of the relative key data item described for the file, CBL gives varying results depending on the sizes being used. In most cases, CBL does not write the record, and returns status code 22. COBOL85 does not write the record, but returns status code 24; the imperative statement following INVALID KEY is executed (or a USE procedure is invoked if no INVALID KEY clause exists); and the value of the relative key data item is undefined.

Also, CBL writes the record based on the size of the relative key data item described in the program and in conjunction with the size of the CREATK primary key definition. Overflow is controlled by these two factors. It is not checked explicitly. COBOL85 writes the record based on the size of the relative key data item, and explicitly checks for overflow.

Status Code 30 (Sequential, Relative, Indexed)

When CBL returns status code 30, the INVALID KEY path is taken for indexed and relative files; declaratives, if present, are invoked for sequential files; and control returns to the next executable statement. If declaratives are not present and are required, the program terminates. In COBOL85, after executing declaratives, if present, the program terminates.

Also, in CBL a badspot error for magnetic tape files generates a status code 30. In COBOL85 a badspot error generates a status code 98.

Status Code 34 (Sequential)

When an attempt is made to write beyond the externally defined boundaries of a sequential file, CBL returns status code 30. COBOL85 returns 34 to signal this condition.

Also, in CBL, declaratives, if present, are invoked, and control returns to the next executable statement. In COBOL85, after executing declaratives, if present, the program terminates.

Status Code 35 (Sequential, Indexed, Relative)

In CBL implied OPTIONAL support exists for PRIMOS sequential files only. (That is, a file is created if opened in I-O or EXTEND mode for PRIMOS sequential files). In COBOL85 the OPTIONAL phrase must be specified in the SELECT clause to achieve this functionality; otherwise, a status code 35 is generated for a sequential file that does not exist when opened in I-O or EXTEND mode.

Also, in CBL an OPEN of a magnetic tape file that does not locate the correct *file-ID* on the reel, aborts in such a way that the user is allowed the opportunity to mount another volume and retry the operation. In COBOL85, after invoking any declaratives for status code 35, the program terminates.

Status Code 37 (Sequential, Relative, Indexed)

In CBL a file assigned to PFMS in a program that has WRITE WITH ADVANCING statements can be opened in EXTEND mode. COBOL85 does not support this functionality.

In CBL an indexed or relative file can be opened in OUTPUT mode and contain existing records. WRITE statements add records to the file and may be rejected if an attempt is made to add duplicate primary keys. The integrity of the file cannot be assured. In COBOL85 this error condition is recognized, and the I-O system ensures that the file does not contain existing records.

Status Code 39 (Sequential, Relative, Indexed)

In CBL there is virtually no checking for file attribute conflicts. In some cases, it is possible that executing a program with an undetected file attribute conflict could be a meaningful application, depending on the particular conflict. In COBOL85 checking of attributes prohibits such programs from running, which ensures consistent and expected runtime behavior.

For example, in CBL, key types specified in CREATK are not checked against the key type specified in the source program. Depending on the key types and lengths involved, it is possible that the program could access a file properly. However, it is also possible that a corrupted file may result. COBOL85 does not allow the file to be opened.

See Tables H-1 through H-4 for a complete list of record size conflict differences between CBL and COBOL85. See also the discussion of status code 39 in Chapter 4 for a full list of the file attributes that COBOL85 checks.

Note

In order for COBOL85 to open a MIDASPLUS variable-length record file, you must first establish the minimum and maximum size for the file by using either the CREATK GET function for files that already contain records, or the CREATK INIT function for empty files. See the *MIDASPLUS User's Guide* for details.

Status Code 41 (Sequential, Relative, Indexed)

No conversion concerns. However, see Status Code 93, below.

Status Code 42 (Sequential, Relative, Indexed)

In CBL a CLOSE statement is ignored when the file is not open. COBOL85 returns status code 42 to signal this condition.

Status Code 43 (Sequential, Relative, Indexed)

CBL returns status code 91 for an indexed REWRITE attempt without a successfully executed READ. The ability to REWRITE without a prior READ for MIDASPLUS files that do not have secondary keys is new in COBOL85.

For PRIMOS sequential files, CBL permits multiple REWRITEs for the same record without reestablishing the file position. COBOL85 disallows this practice by requiring a READ before a REWRITE.

Status Code 44 (Sequential, Relative, Indexed)

See Tables H-1 through H-4 for a complete list of record size conflict differences between CBL and COBOL85.

Status Code 46 (Sequential, Relative, Indexed)

In CBL a READ NEXT statement following an unsuccessful START or READ statement causes unpredictable results. A status code of 10 may be returned in some cases. COBOL85 returns status code 46 to signal this condition.

Status Code 47 (Sequential, Relative, Indexed)

No conversion concerns.

Status Code 48 (Sequential, Relative, Indexed)

In CBL a WRITE statement is allowed for a sequential file opened in I-O mode, or for a sequential access indexed file opened in I-O mode. A WRITE statement is allowed for an indexed or relative file in EXTEND mode (although unsupported at runtime). COBOL85 returns status code 48 to signal these conditions.

Status Code 49 (Sequential, Relative, Indexed)

No conversion concerns.

Status Code 82

No conversion concerns.

Status Code 90

No conversion concerns. The meaning of this status code in COBOL85 is the same as its meaning in CBL.

Status Code 91

Obsolete. Replaced by status code 43.

Status Code 92

Obsolete. Replaced by status code 22.

Status Code 93

CBL returns status code 41 for a FORMS validation error on a READ statement. CBL does not support status code 93.

Status Code 94

No conversion concerns. The meaning of this status code in COBOL85 is the same as its meaning in CBL.

Status Code 95

Obsolete. See Status Code 04, 39, 44 and Tables H-1 through H-4.

Status Code 96

Obsolete. Replaced by status code 24.

Status Code 97

No conversion concerns. The meaning of this status code in COBOL85 is the same as its meaning in CBL.

Status Code 98

CBL returns status code 30 when an input-output operation is unsuccessful due to a recoverable error associated with a tape file, such as a badspot on the tape. COBOL85 returns status code 98 to signal this condition.

Note

In CBL status code 98 is documented but not supported. The error condition that 98 is meant to signal in CBL returns status code 39 in COBOL85.

Status Code 99

No conversion concerns. The meaning of this status code in COBOL85 is the same as its meaning in CBL.

Other CBL/COBOL85 Differences Requiring Conversion

The following sections document other differences between CBL and COBOL85 that may require attention during program conversion. The differences are grouped as follows:

- Compiler options
- ENVIRONMENT DIVISION
- DATA DIVISION
- PROCEDURE DIVISION

Compiler Options

This section discusses differences between CBL and COBOL85 compiler options. For a discussion of all COBOL85 compiler options, see Chapter 2.

Removal of FIPS Flagging

COBOL85 does not support FIPS flagging.

Note

The –STANDARD compiler option generates observations for all Prime extensions to the ANSI standard. See Chapter 2 for details.

Removal of –UPCASE

COBOL85 does not support the –UPCASE compiler option. Except for quoted literals, COBOL85 and CBL treat all source code as uppercase.

Removal of –OLDIO

COBOL85 does not support the –OLDIO compiler option. Therefore, the following CBL functionality provided by –OLDIO to support Prime's older COBOL compiler is *not* available in COBOL85:

- COMPRESSED default file attribute

  In COBOL85 UNCOMPRESSED is the default file attribute. For more information on COMPRESSED/UNCOMPRESSED, see Chapter 7.

WARNING

If your program reads a compressed file as uncompressed, a premature end of file results, and data is transferred to seemingly inappropriate fields.

- INVALID KEY on sequential READ NEXT

  In COBOL85 the INVALID KEY phrase is not allowed in a READ NEXT statement. Remove any such occurrences.

- Optional AT END for READ of TERMINAL file

  In COBOL85 the AT END phrase is optional; however, if you do not specify AT END, then you must specify a USE procedure for the file.

- PRINTER filenames default to first four characters of PROGRAM-ID followed by a sequence number

  In COBOL85, if you do not specify the VALUE OF FILE-ID phrase for a file, the compiler uses the internal filename defined in the SELECT clause as the default.

- Non-PRINTER filenames default to F1, F2 ... F9

  In COBOL85, if you do not specify the VALUE OF FILE-ID phrase for a file, the compiler uses the internal filename defined in the SELECT clause as the default.

- VALUE OF FILE-ID literal limited to 8 characters

  In COBOL85 the VALUE OF FILE-ID literal is not truncated to 8 characters.

The ability to reassign files at runtime is still available in COBOL85, but is controlled by the –FILE_ASSIGN compiler option instead of –OLDIO, as in CBL. For information on –FILE_ASSIGN see Appendix N.

–VARYING Default

In COBOL85 –VARYING is the default compiler option for formatting variable-length records.

In CBL records that are defined with multiple 01s of different sizes or that contain tables with the OCCURS DEPENDING ON clause are not written out in variable-length format unless –VARYING is specified. In order to process such files as fixed-length files in COBOL85, –NO_VARYING must be specified, or the RECORD IS NOT VARYING clause must be used in the file description.

ENVIRONMENT DIVISION

This section discusses COBOL85 ENVIRONMENT DIVISION features that differ from the corresponding CBL features.

alphabet-name Clause of SPECIAL-NAMES Paragraph

In COBOL85 the *alphabet-name* clause of the SPECIAL-NAMES paragraph includes the new reserved word ALPHABET. CBL programs that contain this clause must be modified to include the reserved word ALPHABET.

Obsolete Device Types

Device types of MT7, READER, and PUNCH are invalid in COBOL85. No runtime support for these devices exists in CBL. Table 6-1 in Chapter 6 lists allowable COBOL85 device types.

DATA DIVISION

This section discusses COBOL85 DATA DIVISION features that differ from the corresponding CBL features.

Multiple Sign Clause

In CBL the specification of a SIGN clause on a group item does not extend to that group's subordinate data items. In COBOL85 the specification of a SIGN clause on a group item does extend to that group's subordinate data items. In COBOL85 the specification of a SIGN clause on a subordinate data item takes precedence over a SIGN clause specified at the group level.

Propagation of USAGE Clause

In CBL the specification of a USAGE clause on a group item does not extend to that group's subordinate data items when subgroups intervene. In COBOL85 the specification of a USAGE clause on a group item does extend to that group's subordinate data items regardless of its structure. If such structures occur in record descriptions, this may affect the size of the record if the type of USAGE being propagated causes alternate size calculations to be made.

Redefined Items of Unequal Length

In CBL the Prime extension for allowing intervening data descriptions between the redefining item and redefined item allows the redefining item to be greater than the redefined item. In COBOL85 such redefinition causes a fatal error.

VALUE OF FILE-ID IS data-name

If the VALUE OF FILE-ID IS *data-name* clause is used, a search is made during the OPEN operation to determine whether a file reassignment was made using the –FILE_ASSIGN option. In CBL the VALUE OF FILE-ID IS *data-name* clause and file reassignments are mutually exclusive. For example,

```
VALUE OF FILE-ID IS DATANAME.
      .
      .
01  DATANAME PIC X(32) VALUE 'MYFILE'.
      .
      .
ENTER FILE ASSIGNMENTS:
> MYFILE = YOURFILE
> /
```

In COBOL85 YOURFILE is opened. In CBL MYFILE is opened.

PROCEDURE DIVISION

This section discusses COBOL85 PROCEDURE DIVISION features that differ from the corresponding CBL features.

MOVE ON SIZE ERROR

The CBL Prime extension of specifying the ON SIZE ERROR clause for the MOVE verb is not supported in COBOL85 because of conflicts with the ANSI X3.23-1985 standard syntax. The same functionality is available in COBOL85 by using the COMPUTE verb as follows:

The CBL code

```
MOVE A TO B
     ON SIZE ERROR DISPLAY 'SIZE ERROR'.
```

can be recoded as

```
COMPUTE B = A
     ON SIZE ERROR DISPLAY 'SIZE ERROR'.
```

ALPHABETIC Class Condition

In COBOL85 the ALPHABETIC class test returns a result of true if the content of the identifier consists of lowercase, uppercase, or space characters. In CBL only uppercase characters are classified as ALPHABETIC.

Note

Using the new ALPHABETIC-UPPER class test can aid conversions of applications that require uppercase testing exclusively.

OPEN OUTPUT

In COBOL85 if an indexed or relative file is opened in OUTPUT mode and the file exists, it must not contain data, or the OPEN statement is in error. In CBL such a file is opened, and any WRITE statements cause new records to be added to the existing file, providing duplicate primary keys are not created.

Duplicate Alternate Keys

In CBL, if an alternate key is the key of reference during sequential access and duplicates exist in the file, the records with duplicate alternate keys are read, even if the program specifies that duplicates are not allowed. In COBOL85, however, the duplicate keys are bypassed, and the next record returned is the first record in the file whose key value is greater than the file position indicator.

Relative Record Numbers

In CBL the lowest relative record number that you can specify for a record in a relative file is 0. In COBOL85 the lowest relative record number that you can specify is 1. COBOL85 returns status code 24 if you attempt to WRITE a record with a relative record number less than 1.

Opening an Indexed or Relative File in EXTEND Mode

Opening an indexed or relative file in EXTEND mode is explicitly disallowed in COBOL85. A diagnostic is issued at compile time. In CBL the open mode is ignored.

NEXT SENTENCE

In COBOL85 the NEXT SENTENCE phrase is not allowed in formats that require an imperative statement. It is only permitted in IF and SEARCH statements. In CBL the NEXT SENTENCE phrase is allowed anywhere an imperative statement is allowed. You can use the CONTINUE statement in those places where COBOL85 disallows the NEXT SENTENCE phrase.

Conditional and Imperative Statements

It is a Prime extension in CBL that conditional statements may appear where imperative statements are required. In COBOL85 this substitution is not allowed. You can convert an illegal conditional statement to an imperative statement by adding the appropriate explicit scope delimiter to the conditional statement.

For example, the following valid CBL code

```
READ filename
     INVALID KEY
         IF file-status = '23'
            PERFORM get-name
         ELSE
            PERFORM error-routine.
```

can be converted to valid COBOL85 code by adding the END-IF scope delimiter as shown below:

```
READ filename
     INVALID KEY
         IF file-status = '23'
            PERFORM get-name
         ELSE
            PERFORM error-routine
         END-IF.
```

Record Size Conflict Tables

The following tables list record size conflict differences between CBL and COBOL85. The tables are organized as follows:

- PRIMOS Sequential Files (Table H-1)
- Magtape Sequential Files (Table H-2)
- MIDASPLUS Indexed/Relative Files (Table H-3)
- PRISAM Indexed/Relative/Sequential Files (Table H-4)

Notes pertaining to all of the tables follow Table H-4.

TABLE H-1
PRIMOS Sequential Files[1] Record Size Conflicts

Filetype	*Program Size*	*Actual Size*	*Verb*	*CBL*	*COBOL85*
Uncompressed Fixed-length	50	50	ALL	00	00
	100	50	OPEN	00	39
	100	50	READ	00[3]	—
	100	50	WRITE	00[3]	—
	50	100	OPEN	00	39
	50	100	READ	00[3]	—
	50	100	WRITE	00[3]	—
	102	50	OPEN	00	00[4]
	102	50	READ	00[3]	00[3]
	102	50	WRITE	00[3]	00[3]
	99	100	OPEN	00	00[1]
	99	100	READ	00	00
	99	100	WRITE	00	00
Variable-length[5]	50...100	50	READ	00	00
	50...100	10	READ	00	04
	50...100	120	READ	95	04[2]
	50...100	50	WRITE	00	00
	50...100	10	WRITE	00	44[5]
	50...100	120	WRITE	95	44[5]

TABLE H-2
Magtape Sequential Files[1] Record Size Conflicts

Filetype	*Program Size*	*Actual Size*	*Verb*	*CBL*	*COBOL85*
Labeled fixed-length input files	50	50	OPEN	00	00
	50	50	READ	00	00
	50	100	OPEN	00	39
	50	100	READ	00[3]	—
	100	50	OPEN	00	39
	100	50	READ	00[3]	—
Unlabeled fixed-length		--- no checking for unlabeled tapes ---			
Labeled variable-length	50...100	50...100	OPEN	00	00
	50...100	50...100	READ	00	00
	50...120	50...100	OPEN	00	39
	50...120	50...100	READ	00[3]	—
	30...100	50...100	OPEN	00	00
	30...100	50...100	READ	00	00
	60...100	50...100	OPEN	00	00
	60...100	50...100	READ < 60	00	04
	50...80	50...100	OPEN	00	00[7]
	50...80	50...100	READ > 80	00	04[2]
Unlabeled variable-length	60...100	50...100	OPEN	00	00
	60...100	50...100	READ < 60	00	04
	50...80	50...100	OPEN	00	00
	50...80	50...100	READ > 80	00	04[2]

TABLE H-3
MIDASPLUS Indexed/Relative Files[1] Record Size Conflicts

Filetype	*Program Size*	*CREATK Size*	*Verb*	*CBL*	*COBOL85*
Fixed-length	50	50	ALL	00	00
	100	50	OPEN	00	39
	100	50	READ	00	—
	100	50	WRITE	00[2]	—
	50	100	OPEN	abort	39
Filetype	*Program Size*	*MIDAS+ Stored Actual Min/Max*	*Verb*	*CBL*	*COBOL85*
Variable-length (indexed only)	50...100	0	OPEN	00	39
			READ > 100	abort	—
			WRITE	00	—
	50...100	50...100	ALL	00	00
	50...150	50...100	OPEN	00	39
	50...150	50...100	READ	00	—
	50...150	50...100	WRITE	00	—
	10...100	50...100	OPEN	00	39
	10...100	50...100	READ	00	—
	10...100	50...100	WRITE	00	—
	70...80	50...100	OPEN	00	00[7]
	70...80	50...100	READ > 80	abort	04[2]
	70...80	50...100	READ < 70	00	04
	70...80	50...100	WRITE 70...80	00	00
	70...80	50...100	WRITE > 80	00	44[6]

TABLE H-4
PRISAM Indexed/Relative/Sequential Files Record Size Conflicts

Filetype	*Program Size*	*DDL Size*	*Verb*	*CBL*	*COBOL85*
Fixed-length	50	50	ALL	00	00
	100	50	OPEN	00	39
	100	50	READ	00	—
	100	50	WRITE	95	—
	50	100	OPEN	00	39
	50	100	READ	95	—
	50	100	WRITE	00	—
Variable-length (DDL contains	50...100	50...100	ALL	00	00
OCCURS	50...150	50...100	OPEN	00	39
DEPENDING ON	50...150	50...100	READ	00	—
clause)	50...150	50...100	WRITE > 100	95	—
	10...100	50...100	OPEN	00	39
	10...100	50...100	READ	00	—
	10...100	50...100	WRITE	00	—
	70...80	50...100	OPEN	00	00[7]
	70...80	50...100	READ > 80	95	04[2]
	70...80	50...100	READ < 70	00	04
	70...80	50...100	WRITE 70...80	00	00
	70...80	50...100	WRITE > 80	00	44[6]
Pseudo-variable-length (DDL contains multiple 01 levels)	50...70	50, 70	ALL	00	00
		--- Program maximum larger than DDL maximum ---			
	40...80	40, 70	OPEN	00	39
	40...80	40, 70	READ 40	00	—
	40...80	40, 70	READ 70	00	—
	40...80	40, 70	WRITE ^=40,70	95	—
		--- Program minimum smaller than DDL minimum ---			
	20...70	40, 70	OPEN	00	39
	20...70	40, 70	READ 40	00	—
	20...70	40, 70	READ 70	00	—
	20...70	40, 70	WRITE ^=40,70	95	—

TABLE H-4
PRISAM Indexed/Relative/Sequential Files Record Size Conflicts - Continued

Filetype	*Program Size*	*DDL Size*	*Verb*	*CBL*	*COBOL85*
	--- Program maximum smaller than DDL maximum ---				
	40...60	40, 70	OPEN	00	00[7]
	40...60	40, 70	READ 40	00	00
	40...60	40, 70	READ 70	95	04[2]
	40...60	40, 70	WRITE ^=40,70	95	44
	--- Program minimum larger than DDL minimum ---				
	50...70	40, 70	OPEN	00	00[7]
	50...70	40, 70	READ 40	00	04
	50...70	40, 70	READ 70	00	00
	50...70	40, 70	WRITE ^=40,70	95	44

Notes

1. Size comparisons are based on word size.
2. Truncation occurs.
3. Undefined results.
4. There is no way to determine the actual size of a PRIMOS fixed-length record. The program record size is used to read a record. If the subsequent word is a newline character, the OPEN is successful. Therefore, when multiples of actual record lengths (plus newline words) equal the program record size, a file attribute conflict is not detected. Subsequent file operations are undefined.
5. There are no minimum or maximum size attributes associated with PRIMOS variable-length files to be checked during an OPEN. Sizes are checked during READ only. WRITE statements are subject only to the following note.
6. If a program's record description for a variable-length file contains a DEPENDING ON phrase, and the maximum is out of range so that the size of the record being written is larger than the valid maximum for the file, the program is invalid and status code 44 applies.
7. This OPEN statement is permitted only if the RECORD IS VARYING clause is specified with a minimum and a maximum that are equal to the actual file minimum and maximum.

Other Notes Related to Size Conflict CBL/COBOL85 Differences

1. The size of a rewritten variable-length record must equal the size of the record being replaced. This rule applies to all filetypes in both CBL and COBOL85. However, CBL returns status code 95; COBOL85 returns status code 44.

2. In CBL status code 95 is treated as an INVALID KEY error for keyed access. The imperative statement after INVALID KEY is executed if you specify one. Otherwise, any applicable USE procedure is executed. In either case, control returns to the statement following the I-O statement that detected status code 95. If neither an INVALID KEY clause nor declaratives are specified, the program terminates.
3. In COBOL85 status code 44 is a fatal logic error. After executing any applicable USE procedure, the program terminates.
4. In COBOL85 status code 04 is an informational error. The I-O operation is successful.

Implementation-dependent Features of COBOL85

Maximum Sizes

Unpacked decimal (DISPLAY) number	18 digits
Packed decimal number (COMP-3 and PACKED-DECIMAL)	18 digits
Binary number (COMP and BINARY)	18 digits, default S9999
Floating-point-1 (COMP-1) value	10E38, minimum 10E-38
mantissa	7 digits
Floating-point-2 (COMP-2) value	10E+9823, minimum 10E-9902
mantissa	14 digits
Index value (max occurrence number)	128K(131,071)
Index size	64 bits
WORKING-STORAGE size	100 128K-byte segments
Program size (PROCEDURE DIVISION)	100 128K-byte segments
Internal table (array) size	100 128K-byte segments
External table (array) size	Unlimited by COBOL85
Length of *data-names* and other programmer-defined words	32 characters
Filename as literal, including pathname	128 characters
program-id	8 characters through SEG; 32 characters through BIND
Size of an elementary item	32K bytes
Record sizes:	
In WORKING-STORAGE	100 128K-byte segments
PRIMOS sequential disk file	64K bytes
PRISAM file	32K bytes
MIDASPLUS file	32K bytes
Tape file (fixed-length)	12K bytes (blocking factor of 1)
Tape file (variable-length)	9995 bytes

Block sizes:	
Disk (all file types)	64K bytes
Tape	12K bytes

Maximum Numbers

Characters in ACCEPT or DISPLAY	256
Delimiters of UNSTRING	5
Number of subscripts (array dimensions)	8
Number of secondary keys	17
Number of files open at once	128 (0 and 127 are reserved for PRIMOS)
Number of files merged	2-11
Number of files sorted	1-20
Number of qualifiers:	
for *paragraph-names*	1
for *data-names* or *condition-names*	50
Operands of PROCEDURE DIVISION USING	64

Other Information

High values	Hex FF
Low values	Hex 00

J

COBOL85 Library Files

Note

COBOL85 programs do not have access to any part of the CBL library. Likewise, CBL programs do not have access to any part of the COBOL85 library. All CBL and COBOL85 entry points are unique.

To use COBOL85, you must have the following files available in the directories specified:

Directory	*Filename*	*Function*
CMDNC0	COBOL85.RUN	EPF COBOL85 compiler
SYSOVL	COBOL85DATA	Diagnostic file
LIB	COBOL85LIB	COBOL85 library
	NCOBOL85LIB	Nonshared COBOL85 library routines
LIBRARIES*	COBOL85_LIBRARY.RUN	EPF COBOL85 library routines

The COBOL85 library (COBOL85LIB or NCOBOL85LIB) contains the following subroutines:

Subroutine	*Function*
CB$ACLT	Alphabetic class tests
CB$ADAT	Returns current date in format *YYMMDD*
CB$ADAY	Returns Julian date in format *YYDDD*
CB$ADDFCB	Adds FCB to active list
CB$ALC	Merge internal routine
CB$ANY2/CB$ANY3	Runtime interface between object program and system conversion
CB$ART	Merge internal routine
CB$ATIM	Returns current time in format *HHMMSSFF*: *H* = Hour *M* = Minutes *S* = Seconds *F* = Hundredths of seconds
CB$ATOA	Performs compile/object time data conversion
CB$BRD	Merge internal routine
CB$CA	Closes all open files
CB$CLL	Merge internal routine

Subroutine	*Function*
CB$CLOSE	Closes a file
CB$CLU	Merge internal routine
CB$CM	Closes magnetic tape file
CB$CMB	Merge internal routine
CB$CMP	Merge internal routine
CB$CPR	Merge internal routine
CB$CRLSE	Sort/merge release interlude
CB$CSMOP	Sort/merge open interlude
CB$CVRT	Miscellaneous conversion routine
CB$DI/CB$DR	Deletes record from an indexed/relative file
CB$DR	Deletes record from an indexed/relative file
CB$EM	Magnetic tape error processing
CB$EMT	Merge internal routine
CB$ERR	Error processing
CB$EXP	Merge internal routine
CB$EXP	Runtime decimal exponentiation
CB$EZCM	EBCDIC compare
CB$FLL	Merge internal routine
CB$FLU	Merge internal routine
CB$GETFID	Gets pathname for file
CB$IBL	Merge internal routine
CB$ICA	Merge internal routine
CB$IN/CB$IN1	File assignment initialization
CB$INS	Merge internal routine
CB$INSP	INSPECT statement processor
CB$KGT	Merge internal routine
CB$LNK	Merge internal routine
CB$LNL	Merge internal routine
CB$MER	Maps MIDASPLUS error codes to file status
CB$MG1	Initiates the merge process
CB$MG2	Merge internal routine
CB$MG3	Closes all units opened by other merge routines
CB$MLT	Merge internal routine
CB$MOV	Merge internal routine
CB$MSU	Initialize the merge process
CB$NCLT	Numeric class test
CB$NED	Miscellaneous numeric editing
CB$OBL	Merge internal routine
CB$OM	Opens magnetic tape file
CB$OPEN	Opens a file
CB$OUT	Merge internal routine
CB$PER	Maps PRISAM error codes to file status
CB$PRC	Merge internal routine
CB$PRTN	Used with an EXIT PROGRAM to return to caller
CB$RCK	Range checking
CB$RDR	Merge internal routine
CB$RI/CB$RR	Reads indexed/relative file
CB$RLS	Merge internal routine
CB$RM	Reads input from a magnetic tape file
CB$RMVFCB	Removes FCB from active list
CB$RS	Reads sequential file
CB$RTN	Returns the next merged record
CB$SHL	Merge internal routine

Subroutine	*Function*
CB$SI/CB$SR	Starts indexed/relative file
CB$SLR	Merge select read interlude
CB$SMUT	Sort/merge utility routines
CB$SPC	Merge internal routine
CB$SRT	Merge internal routine
CB$STR	String statement processor
CB$STR1/CB$STR2/CB$STR3	STRING statement
CB$SW/CB$SW0	Sense switches setting
CB$TIN	Reads input from terminal
CB$TOUT	Writes output to user terminal
CB$TRN	Merge internal routine
CB$UCLT	User-defined class test
CB$UN	Unlocks an indexed/relative entry
CB$UNS	Unstring statement processor
CB$UNS1/CB$UNS2	UNSTRING statement
CB$WI/CB$WR	Writes indexed/relative file
CB$WM	Writes output to a magnetic tape file
CB$WR	Writes indexed/relative file
CB$WS	Writes a sequential file
CB$WTR	Merge internal routine
CB$XBTD	Binary to decimal conversion routine
CB$XDTB	Decimal to binary conversion routine
CB$XI/CB$XR	Rewrites indexed/relative file
CB$XMV	Numeric move statement
CB$XR	Rewrites indexed/relative file
CB$XS	Rewrites a sequential file
CB$XTR	Merge internal routine
CB$ZCM	Multi-segment character comparison
CB$ZED	Alphanumeric edited move
CB$ZMVD	Multi-segment block move
N$ATOA	Interfaces to symbolic debugger

K

The MAP Option

The sample program listing in this appendix includes a map created with the –MAP compiler option. The listing includes the names of all data items, the program name, section and paragraph headings, with the following information:

LEVEL The level number specified by the user in the *data-description-entry*:

1-49, 66, 77, 88 represent *data-names* in the DATA DIVISION

FD represents *file-description-entries*

SIZE The size in 16-bit halfwords, unless followed by C, indicating characters.

LOC The memory address in octal notation, unless the –HEXADDRESS option is used. The address may be followed by a number and a letter. If there is no number, the data is allocated in the link frame. Otherwise, the data is in a common block of that number. The common block names are at the end of the map.

ATTRIBUTES The first line shows whether the *data-name* is COMP-1, COMP-2, COMP-3, PACKED-DECIMAL, INDEX, DISPLAY, FD-FILE, ALPHANUMERIC, ALPHANUMERIC GROUP ITEM, BINARY, or US-BINARY. The US means unsigned. The BINARY attribute corresponds to COBOL85 data types in the following way:

BINARY-1 16-bit COMP, PIC 9 through PIC 9(4)

BINARY-2 32-bit COMP, PIC 9(5) through PIC 9(9)

BINARY-4 64-bit COMP, PIC 9(10) through PIC 9(18)

For DISPLAY items, the listing shows whether they have a separate sign or an overpunch. The listing also indicates group items.

The second line shows the line where the item is declared. An asterisk indicates a line where the value of the item is changed.

For items from a copy file, the line number is shown in the format

n<m>

where *n* is the line of the COPY statement in the main program, and *m* is the line number in the copy file.

In addition, at the end of the listing, the amount of 16-bit halfwords of working storage is given. Working storage is divided into

- LINK BASE — the link frame space.
- The names of common blocks used. These names, created by the compiler, end with a number plus a dollar sign to avoid possible use in a program.

The program below uses two common blocks, QWTB1$ and QWTB2$, created by the compiler with a hashing formula based on the *program-id.*

Example

```
SOURCE FILE: <MYMFD>MYDIR>MAP.COBOL85
COMPILED ON: WED, MAR 16 1988  AT: 10:04  BY: COBOL85 REV. 1.0-22.0  02/01/88.14:18
Options selected:  map -map
Optimization note: Currently "-OPTimize" means "-OPTimize 2",
Options used (* follows those that are not default):
     64V No_Ansi_Obsolete Binary CALCindex No_COMP No_CORrMap No_DeBuG
     No_ERRorFile ERRTty No_EXPlist No_File_Assign Formatted_DISplay
     No_HEXaddress Listing* MAp* No_OFFset OPTimize(2) No_PRODuction No_RAnge
     No_SIGnalerrors SIlent(0) No_SLACKbytes TIME No_STANdard No_STATistics
     Store_Owner_Field SYNtaxmsg No_TRUNCdiags UPcase VARYing No_XRef

    1            IDENTIFICATION DIVISION.
    2            PROGRAM-ID. REL2HUGE.
    3            ENVIRONMENT DIVISION.
    4            CONFIGURATION SECTION.
    5            SOURCE-COMPUTER. PRIME.
    6            OBJECT-COMPUTER. PRIME.
    7            INPUT-OUTPUT SECTION.
    8            FILE-CONTROL.
    9                SELECT A-FILE ASSIGN TO MT9
   10                        ORGANIZATION IS SEQUENTIAL.
   11                 SELECT B-FILE ASSIGN TO PRIMOS
   12                ORGANIZATION IS SEQUENTIAL.
   13                SELECT C-FILE ASSIGN TO PRIMOS
   14                ORGANIZATION IS SEQUENTIAL.
   15                SELECT D-FILE ASSIGN TO PRIMOS
   16                ORGANIZATION IS SEQUENTIAL.
   17                SELECT T-FILE ASSIGN TO MIDASPLUS
   18                  ORGANIZATION IS INDEXED
   19                   ACCESS IS DYNAMIC
   20                           RECORD KEY IS T-KEY0
   21                           ALTERNATE RECORD KEY IS T-KEY1
   22                           ALTERNATE RECORD KEY IS T-KEY2
   23                           ALTERNATE RECORD KEY IS T-KEY3
   24                           FILE STATUS IS T-FILE-STATUS.
   25                       SELECT MIDAS-S-FILE ASSIGN TO MIDASPLUS
   26                          ORGANIZATION IS INDEXED
   27                          ACCESS IS DYNAMIC
   28                          RECORD KEY IS S-KEY0
   29                          ALTERNATE RECORD KEY IS S-KEY1 WITH DUPLICATES
```

```
30                      ALTERNATE RECORD KEY IS S-KEY2 WITH DUPLICATES
31                      FILE STATUS IS FILE-STATUS.
32                    SELECT REL-1 ASSIGN TO MIDASPLUS
33                      ORGANIZATION RELATIVE
34                      ACCESS RANDOM
35                      RELATIVE KEY REL-KEY
36                      FILE STATUS REL-1-STATUS.
37            DATA DIVISION.
38            FILE SECTION.
39            FD A-FILE
41                VALUE OF FILE-ID IS 'YIKES'.
42            01 A-REC.
43                03 ATAB OCCURS 1000.
44                  05 A-ENT        PIC X(32).
45            FD B-FILE
47                VALUE OF FILE-ID IS 'B-FILE'.
48            01 B-REC.
49                03 ATAB OCCURS 100.
50                  05 A-ENT        PIC X(314).
51            FD C-FILE
53                VALUE OF FILE-ID IS 'C-FILE'.
54            01 C-REC.
55                03 ATAB OCCURS 1000.
56                  05 A-ENT        PIC X(32).
57            FD D-FILE
59                VALUE OF FILE-ID IS 'D-FILE'.
60            01 D-REC.
61                03 ATAB OCCURS 1000.
62                  05 A-ENT        PIC X(32).
63            FD T-FILE
65                VALUE OF FILE-ID IS 'TF-FILE1'.
66            01 TREC.
67                03 T-KEY0         PIC 9(4).
68                03 FILLER         PIC X.
69                03 T-KEY1         PIC 9(4).
70                03 T-KEY2         PIC 9(6).
71                03 T-KEY3         PIC 9(2).
72                03 T-DATA         PIC X(33).
73            FD MIDAS-S-FILE
75                VALUE OF FILE-ID IS SHORTREC-TREE.
76            01 SHORT-REC.
77                03 S-KEY0         PIC 9(5).
78                03 S-KEY1.
79                   05 S-KEY-1-X   PIC X.
80                   05 S-KEY1-9    PIC 9(6).
81                03 S-KEY2.
82                   05 S-KEY2-X    PIC X.
83                   05 K-KEY2-9    PIC 9(4).
84                03 S-DATA         PIC X(33).
85            FD REL-1 UNCOMPRESSED
87                VALUE OF FILE-ID IS REL-1-TREE.
88            01 RECORD-1.
89                02 PRIM-KEY       PIC 9(16).
90                02 ALT-KEY1       PIC 9(16).
91                02 ALT-KEY2       PIC 9(16).
92                02 FILLER         PIC X(12).
93            WORKING-STORAGE SECTION.
94            01 SHORTREC-TREE      PIC X(40) VALUE 'SHORTREC'.
95            01 RTREE.
96                03 FILLER         PIC X.
97                03 REL-1-TREE     PIC X(40) VALUE 'REL-1'.
98            01 REL-STATUS-STUFF.
99               03 FILLER          PIC X.
100              03 REL-1-STATUS    PIC X(2).
101              03 T-FILE-STATUS   PIC X(2).
102              03 FILE-STATUS     PIC X(2).
103           01 REL-KEY-STUFF.
104              03 FILLER          PIC X.
```

```
105              03 REL-KEY        PIC 9(6).
106           01 CPU-START          PIC X(4).
107           01 CPU-FIN            PIC X(4).
108           01 DISK-START         PIC X(4).
109           01 DISK-FIN           PIC X(4).
110           01 LOOP-COUNT          PIC S999 COMP VALUE 100.
111           PROCEDURE DIVISION.
112           DECLARATIVES.
113           DECLARE-1-SECTION SECTION.
114               USE AFTER ERROR PROCEDURE ON REL-1.
115           DECLARE-1-P1.
116               DISPLAY 'IO ERROR ON REL-1'.
117               DISPLAY 'STATUS = '  REL-1-STATUS.
118               DISPLAY 'REL-KEY = ' REL-KEY.
119               CLOSE REL-1.
120               GO TO ALL-DONE.
121           END DECLARATIVES.
122           START-SECTION SECTION.
123           P1.
124               DISPLAY 'ENTER RELATIVE FILE PATHNAME'.
125               ACCEPT REL-1-TREE.
126               DISPLAY 'ENTER ISAM FILE NAME'.
127               ACCEPT SHORTREC-TREE.
128               OPEN INPUT REL-1.
129               MOVE 0 TO PRIM-KEY.
130               MOVE 0 TO REL-KEY.
131               MOVE 0 TO ALT-KEY1.
132               MOVE 0 TO ALT-KEY2.
133               DISPLAY 'READ RELATIVE FILE TEST'.
134               PERFORM READ-1 LOOP-COUNT TIMES.
135               GO TO CLOSE-FILES.
136           READ-1.
137               DISPLAY 'ENTER KEY AS 9(6) ITEM'.
138               ACCEPT REL-KEY.
139               READ REL-1 RECORD.
140               DISPLAY RECORD-1.
141           CLOSE-FILES.
142               CLOSE REL-1.
143           ALL-DONE.
144               EXIT.
145           A100-MAIN.
146               DISPLAY 'READ ISAM FILE TEST. (USING BYTE-ALIGNED KEY)'.
147               DISPLAY 'ENTER Q$ TO QUIT'.
148               OPEN I-O MIDAS-S-FILE.
149           A100-KEEP-GOING.
150               DISPLAY 'ENTER KEY AS X(7) ITEM'.
151               ACCEPT S-KEY1.
152               IF S-KEY1 = 'Q$' THEN GO TO A999-END.
153               READ MIDAS-S-FILE KEY IS S-KEY1
154                    INVALID KEY
155                    DISPLAY FILE-STATUS.
156               EXHIBIT SHORT-REC.
157               GO TO A100-KEEP-GOING.
158           A999-END.
159               CLOSE MIDAS-S-FILE.
160               DISPLAY FILE-STATUS.
161           A200-MAIN.
162               DISPLAY 'READ ISAM FILE TEST (WORD ALIGNED KEY)'.
163               DISPLAY 'ENTER Q$ TO QUIT'.
164               OPEN I-O MIDAS-S-FILE.
165           A200-KEEP-GOING.
166               DISPLAY 'ENTER KEY AS X(5) ITEM'.
167               ACCEPT S-KEY2.
168               IF S-KEY2 = 'Q$' THEN GO TO A9999-END.
169               READ MIDAS-S-FILE KEY IS S-KEY2
170                  INVALID KEY
171                  DISPLAY FILE-STATUS.
172               EXHIBIT SHORT-REC.
```

```
  173                GO TO A200-KEEP-GOING.
  174             A9999-END.
  175                CLOSE MIDAS-S-FILE.
  176                DISPLAY FILE-STATUS.
  177                STOP RUN.

DATA NAMES DECLARED IN REL2HUGE

REL2HUGE                                        ENTRY PT EXTERNAL
                                                DECLARED ON LINE 2

NAME            LEVEL SIZE    LOC (OCTAL)       ATTRIBUTES

A-FILE          FD    135     001405            FD-FILE
                                                DECLARED ON LINE 9
B-FILE          FD    161     100214            FD-FILE
                                                DECLARED ON LINE 11
C-FILE          FD    161     100455            FD-FILE
                                                DECLARED ON LINE 13
D-FILE          FD    161     100716            FD-FILE
                                                DECLARED ON LINE 15
T-FILE          FD    213     101157            FD-FILE
                                                DECLARED ON LINE 17
MIDAS-S-FILE    FD    191     101661            FD-FILE
                                                DECLARED ON LINE 25
REL-1           FD    152     102336            FD-FILE
                                                DECLARED ON LINE 32
A-REC           1     32000C  001614            ALPHANUMERIC GROUP ITEM
                                                REDEFINING ITEM
                                                DECLARED ON LINE 42
ATAB            3     32000C  001614            ALPHANUMERIC GROUP ITEM
                                                OCCURRING ITEM
                                                DECLARED ON LINE 43
A-ENT           5     32C     001614            ALPHANUMERIC
                                                DECLARED ON LINE 44
B-REC           1     31400C  000000            ALPHANUMERIC GROUP ITEM
                                                REDEFINING ITEM
                                                DECLARED ON LINE 48
ATAB            3     31400C  C1+000432         ALPHANUMERIC GROUP ITEM
                                                OCCURRING ITEM
                                                DECLARED ON LINE 49
A-ENT           5     314C    C1+000432         ALPHANUMERIC
                                                DECLARED ON LINE 50
C-REC           1     32000C  075250            ALPHANUMERIC GROUP ITEM
                                                REDEFINING ITEM
                                                DECLARED ON LINE 54
ATAB            3     32000C  C1+000432         ALPHANUMERIC GROUP ITEM
                                                OCCURRING ITEM
                                                DECLARED ON LINE 55
A-ENT           5     32C     C1+000432         ALPHANUMERIC
                                                DECLARED ON LINE 56
D-REC           1     32000C  000000            ALPHANUMERIC GROUP ITEM
                                                REDEFINING ITEM
                                                DECLARED ON LINE 60
ATAB            3     32000C  C2+000434         ALPHANUMERIC GROUP ITEM
                                                OCCURRING ITEM
                                                DECLARED ON LINE 61
A-ENT           5     32C     C2+000434         ALPHANUMERIC
                                                DECLARED ON LINE 62
TREC            1     50C     101504            ALPHANUMERIC GROUP ITEM
                                                REDEFINING ITEM
                                                DECLARED ON LINE 66
T-KEY0          3     4C      101504            US-DISPLAY(4,0)
                                                DECLARED ON LINE 67
FILLER          3     1C      101506            ALPHANUMERIC
                                                DECLARED ON LINE 68
T-KEY1          3     4C      101506+1C         US-DISPLAY(4,0)
                                                DECLARED ON LINE 69
```

```
T-KEY2          3     6C     101510+1C       US-DISPLAY(6,0)
                                             DECLARED ON LINE 70
T-KEY3          3     2C     101513+1C       US-DISPLAY(2,0)
                                             DECLARED ON LINE 71
T-DATA          3     33C    101514+1C       ALPHANUMERIC
                                             DECLARED ON LINE 72
SHORT-REC       1     50C    102160          ALPHANUMERIC GROUP ITEM
                                             REDEFINING ITEM
                                             DECLARED ON LINE 76
S-KEY0          3     5C     102160          US-DISPLAY(5,0)
                                             DECLARED ON LINE 77
S-KEY1          3     7C     102162+1C       ALPHANUMERIC GROUP ITEM
                                             DECLARED ON LINE 78
S-KEY-1-X       5     1C     102162+1C       ALPHANUMERIC
                                             DECLARED ON LINE 79
S-KEY1-9        5     6C     102163          US-DISPLAY(6,0)
                                             DECLARED ON LINE 80
S-KEY2          3     5C     102166          ALPHANUMERIC GROUP ITEM
                                             DECLARED ON LINE 81
S-KEY2-X        5     1C     102166          ALPHANUMERIC
                                             DECLARED ON LINE 82
K-KEY2-9        5     4C     102166+1C       US-DISPLAY(4,0)
                                             DECLARED ON LINE 83
S-DATA          3     33C    102170+1C       ALPHANUMERIC
                                             DECLARED ON LINE 84
RECORD-1        1     60C    102566          ALPHANUMERIC GROUP ITEM
                                             REDEFINING ITEM
                                             DECLARED ON LINE 88
PRIM-KEY        2     16C    102566          US-DISPLAY(16,0)
                                             DECLARED ON LINE 89
ALT-KEY1        2     16C    102576          US-DISPLAY(16,0)
                                             DECLARED ON LINE 90
ALT-KEY2        2     16C    102606          US-DISPLAY(16,0)
                                             DECLARED ON LINE 91
FILLER          2     12C    102616          ALPHANUMERIC
                                             DECLARED ON LINE 92
SHORTREC-TREE   1     40C    102771          ALPHANUMERIC
                                             DECLARED ON LINE 94
RTREE           1     41C    103015          ALPHANUMERIC GROUP ITEM
                                             DECLARED ON LINE 95
FILLER          3     1C     103015          ALPHANUMERIC
                                             DECLARED ON LINE 96
REL-1-TREE      3     40C    103015+1C       ALPHANUMERIC
                                             DECLARED ON LINE 97
REL-STATUS-STUFF
                1     7C     103042          ALPHANUMERIC GROUP ITEM
                                             DECLARED ON LINE 98
FILLER          3     1C     103042          ALPHANUMERIC
                                             DECLARED ON LINE 99
REL-1-STATUS    3     2C     103042+1C       ALPHANUMERIC
                                             DECLARED ON LINE 100
T-FILE-STATUS   3     2C     103043+1C       ALPHANUMERIC
                                             DECLARED ON LINE 101
FILE-STATUS     3     2C     103044+1C       ALPHANUMERIC
                                             DECLARED ON LINE 102
REL-KEY-STUFF   1     7C     103046          ALPHANUMERIC GROUP ITEM
                                             DECLARED ON LINE 103
FILLER          3     1C     103046          ALPHANUMERIC
                                             DECLARED ON LINE 104
REL-KEY         3     6C     103046+1C       US-DISPLAY(6,0)
                                             DECLARED ON LINE 105
CPU-START       1     4C     103052          ALPHANUMERIC
                                             DECLARED ON LINE 106
CPU-FIN         1     4C     103054          ALPHANUMERIC
                                             DECLARED ON LINE 107
DISK-START      1     4C     103056          ALPHANUMERIC
                                             DECLARED ON LINE 108
DISK-FIN        1     4C     103060          ALPHANUMERIC
                                             DECLARED ON LINE 109
LOOP-COUNT      1     1      103062          BINARY-1(10,0)
```

```
                                                    DECLARED ON LINE 110
PROCEDURE NAMES DEFINED IN REL2HUGE

NAME                                                ATTRIBUTES

DECLARE-1-SECTION                                   SECTION END OF PERFORM RANGE
                                                    DECLARED ON LINE 113
DECLARE-1-P1                                        PARAGRAPH
                                                    DECLARED ON LINE 115
START-SECTION                                       SECTION
                                                    DECLARED ON LINE 122
P1                                                  PARAGRAPH
                                                    DECLARED ON LINE 123
READ-1                                              PARAGRAPH END OF PERFORM RANGE
                                                    DECLARED ON LINE 136
CLOSE-FILES                                         PARAGRAPH
                                                    DECLARED ON LINE 141
ALL-DONE                                            PARAGRAPH
                                                    DECLARED ON LINE 143
A100-MAIN                                           PARAGRAPH
                                                    DECLARED ON LINE 145
A100-KEEP-GOING                                     PARAGRAPH
                                                    DECLARED ON LINE 149
A999-END                                            PARAGRAPH
                                                    DECLARED ON LINE 158
A200-MAIN                                           PARAGRAPH
                                                    DECLARED ON LINE 161
A200-KEEP-GOING                                     PARAGRAPH
                                                    DECLARED ON LINE 165
A9999-END                                           PARAGRAPH
                                                    DECLARED ON LINE 174

PROGRAMS CALLED FROM REL2HUGE

(NONE)


COMMON (EXTERNAL) AREAS

QWTB1$              63400  HALFWORDS IN AREA

QWTB2$              32000  HALFWORDS IN AREA
```

L

The XREF Option

This appendix contains a sample program followed by a cross-reference listing created with the –XREFSORT compiler option. The program includes two COPY statements, one in the WORKING-STORAGE SECTION, and one in the PROCEDURE DIVISION.

The cross-reference listing includes all features provided by the –MAP option. In addition, it provides a list of all lines on which each data item is referenced, including destructive references. –XREF lists the names in the map in source program order. –XREFSORT lists the names in alphabetic order. The cross-reference listing gives each *data-name*, and the following information:

LEVEL The *level-number* specified by the user in the *data-description-entry*.

SIZE The size in 16-bit halfwords, unless followed by C, indicating characters.

LOC The memory address in octal notation, unless the –HEXADDRESS option is used. This address may be followed by a two-character code. If there is no code, the data is allocated in the link frame. Any other code is the number of the common block at the end of the listing.

ATTRIBUTES The first line shows whether the *data-name* is COMP-1, COMP-2, COMP-3, PACKED-DECIMAL, INDEX, DISPLAY, FD-FILE, ALPHANUMERIC, ALPHANUMERIC GROUP ITEM, BINARY, or US-BINARY. The US means unsigned. The BINARY attribute corresponds to COBOL85 data types in the following way:

BINARY-1 16-bit COMP, PIC 9 through PIC 9(4)

BINARY-2 32-bit COMP, PIC 9(5) through PIC 9(9))

BINARY-4 64-bit COMP, PIC 9(10) through PIC 9(18)

For DISPLAY items, the listing shows whether they have a separate sign or an overpunch. The listing also indicates group items, and shows the precision of the data items.

The second line shows the line on which the item is declared, and all lines that contain references to the item. Line numbers prefixed with an asterisk indicate destructive references.

For items from a copy file, the line number is shown in the format

n<m>

where *n* is the line of the COPY statement in the main program, and *m* is the line number in the copy file.

Example

```
SOURCE FILE: <MYMFD>MYDIR>XREF.COBOL85
COMPILED ON: WED, MAR 16 1988  AT: 10:26  BY: COBOL85 REV. 1.0-22.0  02/01/88.14:18
Options selected:  xref -xrefsort
Optimization note: Currently "-OPTimize" means "-OPTimize 2",
Options used (* follows those that are not default):
     64V No_Ansi_Obsolete Binary CALCindex No_COMP No_CORrMap No_DeBuG
     No_ERRorFile ERRTty No_EXPlist No_File_Assign Formatted_DISplay
     No_HEXaddress Listing* MAp* MAPSort* No_OFFset OPTimize(2) No_PRODuction
     No_RAnge No_SIGnalerrors SIlent(0) No_SLACKbytes TIME No_STANdard
     No_STATistics Store_Owner_Field SYNtaxmsg No_TRUNCdiags UPcase VARYing
     XRef* XRefSort*

    1            IDENTIFICATION DIVISION.
    2            PROGRAM-ID. XREF3.
    3           ******************************************
    4            ENVIRONMENT DIVISION.
    5            CONFIGURATION SECTION.
    6            SOURCE-COMPUTER. PRIME.
    7            OBJECT-COMPUTER. PRIME.
    8           ******************************************
    9            DATA DIVISION.
   10                 COPY 'XREF3.LIB'.
<     1>          WORKING-STORAGE SECTION.
<     2>          01 LEADING-SEP      PIC S99 COMP VALUE 51.
<     3>          01 TRAILING-SEP     PIC S99 COMP VALUE 52.
<     4>          01 FIXED-DEC        PIC S99 COMP VALUE 4.
<     5>          01 LEADING-OVP      PIC S99 COMP VALUE 54.
<     6>          01 TRAILING-OVP     PIC S99 COMP VALUE 55.
<     7>          01 US-DISPLAY       PIC S99 COMP VALUE 63.
<     8>          01 INDEX-NAME       PIC S99 COMP VALUE 67.
<     9>          01 INDEX-ITEM       PIC S99 COMP VALUE 67.
<    10>          01 LS               PIC S9(7)V9(2) LEADING SEPARATE
<    11>                     VALUE -1234567.89.
<    12>          01 TS               PIC S9(7)V9(2) TRAILING SEPARATE
<    13>                     VALUE +9876543.21.
<    14>          01 C3               PIC S9(7)V9(2) COMP-3
<    15>                     VALUE +0.
<    16>         *
<    17>          01 LOP              PIC S9(7)V9(2) LEADING
<    18>                     VALUE -7654321.98.
<    19>          01 TROP             PIC S9(7)V9(2) TRAILING
<    20>                     VALUE +23.45.
   11            01 USD                    PIC 9(7)V9(2)
   12                                          VALUE 5544773.32.
   13            01 IND-DATA-ITEM USAGE IS INDEX.
   14            01 TAB.
   15               05 T-ELEMENT PIC 99 COMP OCCURS 10 INDEXED BY
   16                                    INDX.
   17            01 FB1          PIC S99 COMP VALUE 32765.
   18            01 FB2          PIC S99999 COMP VALUE 1234567.
   19            01 FB4        PIC S99999999999 COMP VALUE 12345678901.
   20            01 USFB1         PIC 99  COMP VALUE 3000.
   21            01 USFB2             PIC 999999 COMP VALUE 223344.
   22            01 USFB4        PIC 999999999999999 COMP
```

```
 23                                        VALUE 123451234512345.
 24           01 CMP1          COMP-1 VALUE 2345.67E+4.
 25           01 CMP           COMP-2 VALUE -764321.98E+14.
 26           01 SB            PIC S9(2)V9(2) COMP VALUE 76.54.
 27           01 SB2           PIC S9(3)V9(3) COMP VALUE 987.654.
 28           01 SB4  PIC S9(12)V9(4) COMP VALUE -555554444433.1234.
 29           01 P             PIC S99 COMP VALUE ZERO.
 30           01 Q             PIC S99 COMP VALUE ZERO.
 31           01 TY            PIC S99 COMP VALUE ZERO.
 32           01 PVALUE        PIC S9 COMP VALUE 9.
 33           01 QVALUE        PIC S9 COMP VALUE 2.
 34           01 CVAR.
 35               05 CSIZE     PIC S9 COMP VALUE 0.
 36               05 CSTRING   PIC X(30) VALUE SPACES.
 37           01 FAKE1         PIC S9(5) VALUE 000000015.
 38           01 FAKE2         PIC S9(4)V9(2) VALUE -0000001234.56.
 39           01 FAKE3         PIC S9(4)V9(2) VALUE +00000001234.56.
 40           01 LSA           PIC S9(7)V9(2) LEADING SEPARATE
 41                            VALUE -1234567.89.
 42           01 TSA           PIC S9(7)V9(2) TRAILING SEPARATE
 43                              VALUE +9876543.21.
 44           01 C3A                    PIC S9(7)V9(2) COMP-3
 45                 VALUE +0.
 46         *
 47           01 LOPA                   PIC S9(7)V9(2) LEADING
 48                           VALUE -7654321.98.
 49           01 TROPA                  PIC S9(7)V9(2) TRAILING
 50                            VALUE +23.45.
 51           01 USDA                   PIC 9(7)V9(2)
 52                            VALUE 5544773.32.
 53           01 FB1A          PIC S99 COMP VALUE 32765.
 54           01 FB2A          PIC S99999 COMP VALUE 123567.
 55           01 FB4A     PIC S99999999999 COMP VALUE 12345678901.
 56           01 USFB1A   PIC 99  COMP VALUE 3000.
 57           01 USFB2A   PIC 999999 VALUE 223344.
 58           01 USFB4A   PIC 999999999999999 VALUE 123451234512345.
 59           01 CMP1A       COMP-1 VALUE 2345.67E+4.
 60           01 CMP2A       COMP-2 VALUE -764321.98E+14.
 61           01 SB1A         PIC S9(2)V9(2) COMP VALUE 76.54.
 62           01 SB2A         PIC S9(3)V9(3) COMP VALUE 987.654.
 63           01 SB4A         PIC S9(12)V9(4) COMP VALUE -555554444433.1234.
 64          ********************************************************
 65           PROCEDURE DIVISION.
 66           S1 SECTION.
 67           P1.
 68               MOVE LS TO LSA.
 69               MOVE LS TO TSA.
 70               COPY 'XREF2.LIB'.
<    1>           MOVE TS TO C3A.
<    2>           MOVE TS TO LOPA.
<    3>           MOVE TS TO TROP.
<    4>           MOVE TS TO USDA.
<    5>           MOVE TS TO FB1A.
<    6>           MOVE TS TO FB2A.
<    7>           MOVE TS TO FB4A.
<    8>           MOVE TS TO USFB1A.
<    9>           MOVE TS TO USFB2A.
<   10>           MOVE TS TO USFB4A.
<   11>           MOVE TS TO CMP1A.
<   12>           MOVE TS TO SB1A.
<   13>           MOVE SB2 TO SB2A.
<   14>           MOVE SB4 TO SB4A.
```

```
DATA NAMES DECLARED IN XREF3

NAME            LEVEL SIZE   LOC (OCTAL)      ATTRIBUTES   ("*" = DESTRUCTIVE REF)

C3              1     5C     000426           COMP-3(9,2)
                                              DECLARED ON LINE 10<14>
C3A             1     5C     000604           COMP-3(9,2)
                                              DECLARED ON LINE 44
                                              REFERENCES:  *70<1>
CMP             1     4      000514           COMP-2(47,0)
                                              DECLARED ON LINE 25
CMP1            1     2      000512           COMP-1(23,0)
                                              DECLARED ON LINE 24
CMP1A           1     2      000654           COMP-1(23,0)
                                              DECLARED ON LINE 59
                                              REFERENCES:  *70<11>
CMP2A           1     4      000656           COMP-2(47,0)
                                              DECLARED ON LINE 60
CSIZE           5     1      000535           BINARY-1(4,0)
                                              DECLARED ON LINE 35
CSTRING         5     30C    000536           ALPHANUMERIC
                                              DECLARED ON LINE 36
CVAR            1     32C    000535           ALPHANUMERIC GROUP ITEM
                                              DECLARED ON LINE 34
FAKE1           1     5C     000556           TRAILING OVP(5,0)
                                              DECLARED ON LINE 37
FAKE2           1     6C     000562           TRAILING OVP(6,2)
                                              DECLARED ON LINE 38
FAKE3           1     6C     000566           TRAILING OVP(6,2)
                                              DECLARED ON LINE 39
FB1             1     1      000472           BINARY-1(7,0)
                                              DECLARED ON LINE 17
FB1A            1     1      000626           BINARY-1(7,0)
                                              DECLARED ON LINE 53
                                              REFERENCES:  *70<5>
FB2             1     2      000474           BINARY-2(17,0)
                                              DECLARED ON LINE 18
FB2A            1     2      000630           BINARY-2(17,0)
                                              DECLARED ON LINE 54
                                              REFERENCES:  *70<6>
FB4             1     4      000476           BINARY-4(37,0)
                                              DECLARED ON LINE 19
FB4A            1     4      000632           BINARY-4(37,0)
                                              DECLARED ON LINE 55
                                              REFERENCES:  *70<7>
FIXED-DEC       1     1      000406           BINARY-1(7,0)
                                              DECLARED ON LINE 10<4>
IND-DATA-ITEM   1     4      000450           INDEX-DATA-NAME
                                              DECLARED ON LINE 13
INDEX-ITEM      1     1      000413           BINARY-1(7,0)
                                              DECLARED ON LINE 10<9>
INDEX-NAME      1     1      000412           BINARY-1(7,0)
                                              DECLARED ON LINE 10<8>
INDX                  4      000466           INDEX-NAME
                                              DECLARED ON LINE 16
LEADING-OVP     1     1      000407           BINARY-1(7,0)
                                              DECLARED ON LINE 10<5>
LEADING-SEP     1     1      000404           BINARY-1(7,0)
                                              DECLARED ON LINE 10<2>
LOP             1     9C     000431           LEADING OVP(9,2)
                                              DECLARED ON LINE 10<17>
LOPA            1     9C     000607           LEADING OVP(9,2)
                                              DECLARED ON LINE 47
                                              REFERENCES:  *70<2>
LS              1     10C    000414           LEADING SEP(9,2)
                                              DECLARED ON LINE 10<10>
                                              REFERENCES:  68 69
LSA             1     10C    000571           LEADING SEP(9,2)
                                              DECLARED ON LINE 40
```

```
                                                REFERENCES:  *68
P               1       1       000530          BINARY-1(7,0)
                                                DECLARED ON LINE 29
PVALUE          1       1       000533          BINARY-1(4,0)
                                                DECLARED ON LINE 32
Q               1       1       000531          BINARY-1(7,0)
                                                DECLARED ON LINE 30
QVALUE          1       1       000534          BINARY-1(4,0)
                                                DECLARED ON LINE 33
SB              1       1       000520          BINARY-1(14,7)
                                                DECLARED ON LINE 26
SB1A            1       1       000662          BINARY-1(14,7)
                                                DECLARED ON LINE 61
                                                REFERENCES:  *70<12>
SB2             1       2       000522          BINARY-2(20,10)
                                                DECLARED ON LINE 27
                                                REFERENCES:  70<13>
SB2A            1       2       000664          BINARY-2(20,10)
                                                DECLARED ON LINE 62
                                                REFERENCES:  *70<13>
SB4             1       4       000524          BINARY-4(54,14)
                                                DECLARED ON LINE 28
                                                REFERENCES:  70<14>
SB4A            1       4       000666          BINARY-4(54,14)
                                                DECLARED ON LINE 63
                                                REFERENCES:  *70<14>
T-ELEMENT       5       10      000454          US-BINARY-1(7,0) OCCURRING ITEM
                                                DECLARED ON LINE 15
TAB             1       10      000454          ALPHANUMERIC GROUP ITEM
                                                DECLARED ON LINE 14
TRAILING-OVP    1       1       000410          BINARY-1(7,0)
                                                DECLARED ON LINE 10<6>
TRAILING-SEP    1       1       000405          BINARY-1(7,0)
                                                DECLARED ON LINE 10<3>
TROP            1       9C      000436          TRAILING OVP(9,2)
                                                DECLARED ON LINE 10<19>
                                                REFERENCES:  *70<3>
TROPA           1       9C      000614          TRAILING OVP(9,2)
                                                DECLARED ON LINE 49
TS              1       10C     000421          TRAILING SEP(9,2)
                                                DECLARED ON LINE 10<12>
                                                REFERENCES:  70<1> 70<2> 70<3>
                                                70<4> 70<5> 70<6> 70<7> 70<8>
                                                70<9> 70<10> 70<11> 70<12>
TSA             1       10C     000576          TRAILING SEP(9,2)
                                                DECLARED ON LINE 42
                                                REFERENCES:  *69
TY              1       1       000532          BINARY-1(7,0)
                                                DECLARED ON LINE 31
US-DISPLAY      1       1       000411          BINARY-1(7,0)
                                                DECLARED ON LINE 10<7>
USD             1       9C      000443          US-DISPLAY(9,2)
                                                DECLARED ON LINE 11
USDA            1       9C      000621          US-DISPLAY(9,2)
                                                DECLARED ON LINE 51
                                                REFERENCES:  *70<4>
USFB1           1       1       000502          US-BINARY-1(7,0)
                                                DECLARED ON LINE 20
USFB1A          1       1       000636          US-BINARY-1(7,0)
                                                DECLARED ON LINE 56
                                                REFERENCES:  *70<8>
USFB2           1       2       000504          US-BINARY-2(20,0)
                                                DECLARED ON LINE 21
USFB2A          1       6C      000640          US-DISPLAY(6,0)
                                                DECLARED ON LINE 57
                                                REFERENCES:  *70<9>
USFB4           1       4       000506          US-BINARY-4(50,0)
                                                DECLARED ON LINE 22
USFB4A          1       15C     000643          US-DISPLAY(15,0)
                                                DECLARED ON LINE 58
```

```
                                                REFERENCES:  *70<10>
XREF3                                           ENTRY PT EXTERNAL
                                                DECLARED ON LINE 2

PROCEDURE NAMES DEFINED IN XREF3

NAME                                            ATTRIBUTES   ("*" = DESTRUCTIVE REF)

P1                                              PARAGRAPH
                                                DECLARED ON LINE 67
S1                                              SECTION
                                                DECLARED ON LINE 66

PROGRAMS CALLED FROM XREF3

(NONE)

LINK BASE SIZE:   224   HALFWORDS
```

Loading and Executing With SEG

After a program is compiled as discussed in Chapter 2, it must be loaded into an executable file before being run or executed. The PRIMOS SEG utility loads and executes COBOL85 programs. The loading steps create a **runfile**, or executable file consisting of one or more object programs plus any necessary subroutines and libraries. This runfile, or **run unit**, can then be executed at will. This appendix describes normal loading and execution. Loading is described in more detail in the *PRIMOS User's Guide*. For extended loading features and a complete description of all SEG commands, including those for system-level programming, refer to the *SEG and LOAD Reference Guide*.

Note

If your COBOL85 program is larger than one segment, you *must* use BIND to link and execute it, as described in Chapter 3. If your program is less than one segment, you may use either BIND or SEG.

Loading Programs

Default Loading

The SEG utility can create a default runfile named *program*.SEG and load default object *filenames*. Use the –LOAD parameter after SEG, providing the object filename ends in .BIN.

Perform the following steps for default loading:

1. Give the command SEG –LOAD. The response is a dollar sign ($), indicating that the load subprocessor is ready.
2. Use the LOAD command with either the binary filename or the source filename. In the latter case, SEG looks for a binary file of the same name, followed by .BIN.
3. Use the LOAD command to load the object files of any separately compiled subroutines (preferably in order of frequency of use).
4. Use the LIBRARY command to load subroutines called from libraries in the following order:

   - The COBOL85 library (COBOL85LIB)

- The sort-merge library, if sort-merge files are loaded (VSRTLI, or NVSRTLI if a nonshared library is needed)
- Other Prime libraries, if required (filename)
- The PRIMOS system subroutine library — required (LI with no filename)

At this point, you should receive a LOAD COMPLETE message. If the message is absent, check whether any required libraries, programs to be called, or subroutines are missing. If necessary, enter MAP 3 (described in the *PRIMOS User's Guide* and the *SEG and LOAD Reference Guide*) to identify the unresolved references and load them. If the unresolved references are caused by missing subroutine names, enter QUIT and restart from Step 1. If some other SEG error message appears, refer to the *SEG and LOAD Reference Guide* for the probable cause and correction.

5. Enter QUIT to save the runfile and exit from the utility.

   SEG gives the runfile the default name *filename*.SEG, where *filename* is the name of the first object file loaded.

For example, suppose you have a main program called MYPROG.COBOL85 compiled to produce an object file called MYPROG.BIN. The runfile can be created as follows:

```
OK, SEG -LOAD
[SEG Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
$ LO MYPROG
$ LI COBOL85LIB
$ LI
LOAD COMPLETE
$ Q
OK,
```

The command LO MYPROG loads MYPROG.BIN. The resulting runfile is automatically named MYPROG.SEG.

The Older Loading Procedure

Loads can also be accomplished by the following procedure:

1. Invoke the SEG loader with the SEG command without options. A pound sign (#) is the response and prompt symbol.
2. Enter the SEG-level LOAD command to start the load subprocessor and to set up the runfile with a name selected by you (LO runfilename). A dollar sign appears as the next prompt symbol.
3. Use the LOAD command to load the object files in the following order:

   - The object file of the main program (B_filename)
   - The object files of any separately compiled programs or subroutines to be called (preferably in order of frequency of use)

4. Use the LIBRARY command to load subroutines called from libraries in the same order as in Step 4 in Default Loading above.
5. Enter QUIT to save the runfile and exit from the utility.

As an example of loading, assume that you compiled a main program, MAIN, and a subroutine in a separate source file, SUBR. Both were compiled using the default object filenames B_MAIN and B_SUBR. They can be loaded as follows:

```
OK, SEG              Brings SEG into memory
[SEG Rev. 22.0 Copyright (c) Prime Computer, Inc. 1988]
# LO MAIN.SEG        Invokes the loader and establishes a runfile
$ LO B_MAIN          Loads the main program
$ LO B_SUBR          Loads any separately compiled subroutine
$ LI COBOL85LIB      Loads the COBOL85 library
$ LI                 Loads the subroutine library
LOAD COMPLETE             Loader indicates all references are satisfied
$ Q                  Returns to PRIMOS level
OK,
```

Note

Any name may be supplied for the runfile. A name ending with .SEG is suggested to identify runfiles, and to allow the default execution method described below.

Load Error Messages

If the message `WARNING – LOAD NOT COMPLETE` is displayed, the cause may be

- Not loading necessary libraries such as VSRTLI or COBOL85LIB
- Not loading a program named in a CALL statement

To list all unresolved references, go through the loading routine and, after the final LI, enter MAP 3. The MAP command is discussed in the *SEG and LOAD Reference Guide.*

Executing Loaded Programs — Runtime

Any of the following methods start program execution.

Executing Default Runfiles

If the runfile name ends in .SEG, you can execute the runfile by using only the source program name, because, given *pathname*, SEG looks first for *pathname.SEG*, then for *pathname*. For the default runfile MYPROG.SEG in the previous example, execution is accomplished with

```
OK, SEG MYPROG
```

Execution of Other Runfiles

For runfiles whose names do not end in .SEG, execution is performed at the PRIMOS level using the SEG command

SEG *runfilename*

where *runfilename* is the pathname of a runfile created as described in the section titled The Older Loading Procedure above, but whose name does not end with .SEG.

Immediate Execution

A shortcut to saving and executing a loaded program is available. In the loading process described in the previous sections, immediately after receiving the LOAD COMPLETE message, enter EXECUTE. This command saves the loaded program and starts executing the program. The runfile is automatically saved. Use EXECUTE only within the SEG subprocessor environment (that is, when the prompt $ is displayed).

Upon completion of program execution, control returns to PRIMOS command level.

File Assignments With –FILE_ASSIGN

Interactive File Assignments

If the –FILE_ASSIGN option is used for compilation, interactive file assignments are made under the following conditions:

- No EXIT PROGRAM statement is included in the program.
- File Descriptions (FDs) are contained in the runfile.

In this case, immediately following the execute command, RESUME *runfilename*, a request for runtime file assignments is displayed.

```
ENTER FILE ASSIGNMENTS:
>
```

The format for a file assignment is

literal-1 = actual_filename

literal-1 is one of the following:

- The literal following the VALUE OF FILE-ID clause in the *file-description-entry*
- The contents of the *data-name* following the VALUE OF FILE-ID clause
- If there is no VALUE OF FILE-ID clause in the *file-description-entry*, the *file-name* used in the SELECT clause

literal-1 may not contain more than 32 characters. *actual_filename* may not contain more than 128 characters.

For files whose names within *literal-1* are not equal to the actual physical filename, enter the literal, followed by an equal sign, followed by the name of the physical file that is to be associated with the program filename.

The system displays the prompt character > while waiting for more user input. Make one entry for each FD whose FILE-ID is to be assigned. Syntax errors are generated during file

assignment for improper formats. When no file assignments remain to be entered, use a slash mark (/) to conclude the session.

Execution of the application program then begins. When files are accessed in the program, the system uses the actual filename supplied in order to identify the file. When VALUE OF FILE-ID IS *data-name* is used, the contents of the *data-name* at the time of the OPEN statement are used to find a match with *actual_filename*.

At program execution time, if *literal-1* entered during file assignment does not match any filename specified in the program, *actual_filename* is ignored and file access during OPEN uses the program filename. In other words, *literal-1* is not checked for a match against filenames within the program during file assignment.

If a VALUE OF FILE-ID is present and no interactive assignment is made, then the *actual_filename* is the name specified in VALUE OF FILE-ID. If the clause contains a *data-name*, then the COBOL85 program assigns the file pathname to that *data-name*.

Example

Suppose that in a COBOL85 program the following statements exist:

```
FD  DISK-FILE
    VALUE OF FILE-ID IS 'FILE1'.
FD  TAPE-FILE
    LABEL RECORDS ARE STANDARD,
    VALUE OF FILE-ID IS 'FILE2'.
FD  DISK-FILE-2
    VALUE OF FILE-ID IS 'FILE3'.
```

Then an appropriate runtime dialog is

```
ENTER FILE ASSIGNMENTS:
>FILE1 = MY_DIRECTORY>DATA>DISBURSE
>FILE2 = $MT0, S, MYNAME, T1
>/
```

The first response causes PRIMOS to search a directory called MY_DIRECTORY>DATA for a disk file called DISBURSE to use as DISK-FILE in the program.

The second response assumes that a tape drive is assigned as logical drive 0, with a mounted tape that contains a *volume-id* of T1 and an *owner-id* of MYNAME. You must assign the tape drive with the PRIMOS ASSIGN statement before you execute the program.

Because no file assignment entry is made for 'FILE3', DISK-FILE-2 is associated with 'FILE3'.

O

COBOL85 Sample Programs

This appendix contains source listings of two programs that process variable-length records, compiling and linking dialogs for the programs, a sample data file, and sample execution dialogs.

Contents of Data File

The data file used in the following two sample programs is a MIDASPLUS indexed file called CLASS.FILE.MIDAS. The template for the file is created using the MIDASPLUS utility CREATK. The CREATK dialog follows. For more information, see the *MIDASPLUS User's Guide*.

```
OK, CREATK
[CREATK Rev. 22.0 Copyright (c) 1988, Prime Computer, Inc.]

MINIMUM OPTIONS? YES

FILE NAME? CLASS.FILE.MIDAS
NEW FILE? YES
DIRECT ACCESS? NO

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: A
PRIMARY KEY SIZE = : B 6
DATA SIZE IN WORDS = : 0 34 359
SECONDARY INDEX

INDEX NO.? 1

DUPLICATE KEYS PERMITTED? NO

KEY TYPE: A
KEY SIZE = : B 40
SECONDARY DATA SIZE IN WORDS = : (CR)
```

```
INDEX NO.? 2

DUPLICATE KEYS PERMITTED? YES

KEY TYPE: A
KEY SIZE = : B 20
SECONDARY DATA SIZE IN WORDS = : (CR)

INDEX NO.? (CR)

SETTING FILE LOCK TO N READERS AND N WRITERS
OK,
```

The following data is loaded into CLASS.FILE.MIDAS by executing CLASS.BUILD.COBOL85, the first of the two programs below.

RECORD	COURSE ID	COURSE TITLE	INSTRUCTOR	NUMBER OF STUDENTS	STUDENT NAME	STUDENT ID
# 1 :	111111	COURSE 1	INSTRUCTOR 1	002	STUDENT 1	111111
					STUDENT 2	222222
# 2 :	222222	COURSE 2	INSTRUCTOR 2	001	STUDENT 1	111111
# 3 :	333333	COURSE 3	INSTRUCTOR 2	000		

Source Listing — CLASS.BUILD.COBOL85

The following is a source listing of the program CLASS.BUILD.COBOL85.

```
  IDENTIFICATION DIVISION.
  PROGRAM-ID.    CLASSBLD.
  AUTHOR.        PRIMATE.

  REMARKS.
*****************************************************************
*                                                               *
* THIS PROGRAM DEMONSTRATES THE ADDITION OF NEW RECORDS         *
* TO A MIDASPLUS INDEXED FILE THAT CONTAINS                     *
* VARIABLE-LENGTH RECORDS.                                      *
*                                                               *
*****************************************************************

  ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.

*****************************************************************
*                                                               *
* EACH RECORD IN THE CLASS FILE CONTAINS A UNIQUE PRIMARY KEY   *
* (COURSE ID), A UNIQUE SECONDARY KEY (COURSE TITLE), AND A     *
* NON-UNIQUE SECONDARY KEY (INSTRUCTOR NAME).                   *
*                                                               *
*****************************************************************
```

```
  SELECT CLASS-FILE
  ASSIGN TO MIDASPLUS
  ORGANIZATION IS INDEXED
  ACCESS MODE  IS DYNAMIC
  FILE STATUS  IS WS-CLASS-FILE-STATUS
  RECORD KEY   IS COURSE-ID
  ALTERNATE RECORD KEY IS COURSE-TITLE
  ALTERNATE RECORD KEY IS INSTRUCTOR-NAME WITH DUPLICATES.
  DATA DIVISION.
  FILE SECTION.

*****************************************************************
*                                                               *
* EACH RECORD MAY CONTAIN FROM 0 TO 25 STUDENTS DEPENDING UPON  *
* ENROLLMENT.                                                   *
*                                                               *
*****************************************************************

  FD CLASS-FILE
     RECORD IS VARYING FROM 68 TO 718 CHARACTERS
     VALUE OF FILE-ID IS WS-CLASS-FILE-NAME.

  01 CLASS-REC.
     05  COURSE-ID               PIC 9(06).
     05  COURSE-TITLE            PIC X(40).
     05  INSTRUCTOR-NAME         PIC X(20).
     05  NUMBER-OF-STUDENTS      PIC 9(04) COMP.
     05  STUDENT-RECORD          OCCURS 0 TO 25 TIMES
                                 DEPENDING ON
                                 NUMBER-OF-STUDENTS
                                 INDEXED BY STUD-INDX.
         10  STUDENT-ID          PIC 9(06).
         10  STUDENT-NAME        PIC X(20).


  WORKING-STORAGE SECTION.

*****************************************************************
*                                                               *
* EACH RECORD IS FIRST BUILT IN WORKING STORAGE AND THEN        *
* WRITTEN TO THE MIDASPLUS FILE.  NOTE THAT WS-STUDENT-RECORD   *
* ARRAY ALLOWS FOR A MAXIMUM OF 25 STUDENTS.                    *
*                                                               *
*****************************************************************

  01 WS-CLASS-REC.
     05  WS-COURSE-ID              PIC 9(06).
     05  WS-COURSE-TITLE           PIC X(40).
     05  WS-INSTRUCTOR-NAME        PIC X(20).
     05  WS-NUMBER-OF-STUDENTS     PIC 9(04) COMP.
     05  WS-STUDENT-RECORD         OCCURS 25 TIMES
                                   INDEXED BY WS-STUD-INDX.
         10  WS-STUDENT-ID         PIC 9(06).
         10  WS-STUDENT-NAME       PIC X(20).
```

```
01  WORK-FIELDS.
    05  WS-CLASS-FILE-NAME          PIC X(80)  VALUE SPACES.
    05  WS-CLASS-FILE-STATUS        PIC 9(02)  VALUE ZEROES.
    05  WS-CLASS-SW                 PIC X(01)  VALUE SPACES.
        88  END-OF-CLASSES                     VALUE 'Y'.
    05  WS-STUDENT-SW               PIC X(01)  VALUE SPACES.
        88  END-OF-STUDENTS                    VALUE 'Y'.
PROCEDURE DIVISION.


0000-MAINLINE.

    PERFORM 1000-INITIALIZE THRU 1000-EXIT.

    PERFORM 2000-PROCESS-CLASSES THRU 2000-EXIT
            UNTIL END-OF-CLASSES.

    PERFORM 3000-FINISH-UP THRU 3000-EXIT.

    STOP RUN.

0000-EXIT.
    EXIT.


1000-INITIALIZE.
     DISPLAY 'ENTER OUTPUT FILE NAME .....'.
    ACCEPT WS-CLASS-FILE-NAME.
    OPEN OUTPUT CLASS-FILE.
    MOVE 'N' TO WS-CLASS-SW.

1000-EXIT.
    EXIT.
2000-PROCESS-CLASSES.

*****************************************************************
*                                                               *
* DISPLAY PROMPTS AND PROCESS USER'S INPUT UNTIL USER QUITS.    *
*                                                               *
*****************************************************************

    MOVE SPACES TO WS-CLASS-REC.

    DISPLAY ' '.
    DISPLAY 'ENTER COURSE TITLE .....'.
    DISPLAY '(ENTER "QUIT" WHEN FINISHED)'.

    ACCEPT WS-COURSE-TITLE.

    IF WS-COURSE-TITLE = 'QUIT'
       MOVE SPACES TO WS-COURSE-TITLE
       MOVE 'Y' TO WS-CLASS-SW
       GO TO 2000-EXIT.

    DISPLAY ' '.
    DISPLAY 'ENTER COURSE NUMBER .....'.
    ACCEPT WS-COURSE-ID.
```

```
           DISPLAY ' '.
           DISPLAY 'ENTER INSTRUCTOR''S NAME .....'.
           ACCEPT WS-INSTRUCTOR-NAME.

           MOVE 'N' TO WS-STUDENT-SW.
           MOVE 0 TO WS-NUMBER-OF-STUDENTS.

           PERFORM 2100-PROCESS-STUDENTS THRU 2100-EXIT
                   VARYING WS-STUD-INDX FROM 1 BY 1
                       UNTIL END-OF-STUDENTS
                          OR WS-STUD-INDX > 25.

           IF WS-STUD-INDX > 25
              DISPLAY ' '
              DISPLAY '*** WARNING ***'
              DISPLAY 'ENROLLMENT LIMIT HAS BEEN REACHED'
              DISPLAY 'COURSE ' WS-COURSE-ID ' IS FULL '
              DISPLAY ' '.

           WRITE CLASS-REC FROM WS-CLASS-REC
                 INVALID KEY
                         DISPLAY ' '
                         DISPLAY '*** ERROR ***'
                         EXHIBIT WS-COURSE-ID
                         EXHIBIT WS-CLASS-FILE-STATUS
                         DISPLAY ' '.

       2000-EXIT.
           EXIT.
       2100-PROCESS-STUDENTS.

      ****************************************************************
      * DISPLAY PROMPTS AND PROCESS USER'S INPUT UNTIL USER QUITS     *
      * OR 25 STUDENTS HAVE BEEN ENROLLED IN A GIVEN COURSE.          *
      ****************************************************************
            DISPLAY ' '.
           DISPLAY 'ENTER STUDENT NAME .....'.
           DISPLAY '(ENTER "QUIT" WHEN FINISHED)'.
           ACCEPT WS-STUDENT-NAME (WS-STUD-INDX).

           IF WS-STUDENT-NAME (WS-STUD-INDX) = 'QUIT'
              MOVE SPACES TO WS-STUDENT-NAME (WS-STUD-INDX)
              MOVE 'Y' TO WS-STUDENT-SW
              GO TO 2100-EXIT.

           DISPLAY ' '.
           DISPLAY 'ENTER STUDENT NUMBER .....'.
           ACCEPT WS-STUDENT-ID (WS-STUD-INDX).

           ADD 1 TO WS-NUMBER-OF-STUDENTS.

       2100-EXIT.
           EXIT.
```

```
3000-FINISH-UP.

    CLOSE CLASS-FILE.

3000-EXIT.
    EXIT.
```

Compile and Link Dialog — CLASS.BUILD.COBOL85

The preceding program, stored as CLASS.BUILD.COBOL85, can be compiled and linked with the following dialog.

```
OK, COBOL85 CLASS.BUILD -LISTING -VARY
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]
[0 ERRORS IN PROGRAM: CLASS.BUILD.COBOL85]

OK, BIND
[BIND Rev. 22.0 Copyright (c) 1988, Prime Computer, Inc.]
: LO CLASS.BUILD
: LI COBOL85LIB
: LI
BIND COMPLETE
: FILE
OK,
```

Program Execution — CLASS.BUILD.COBOL85

The following is a sample execution dialog for the program CLASS.BUILD.COBOL85.

```
OK, RESUME CLASS.BUILD

ENTER OUTPUT FILE NAME .....
CLASS.FILE.MIDAS

ENTER COURSE TITLE .....
(ENTER "QUIT" WHEN FINISHED)
COURSE 1

ENTER COURSE NUMBER .....
111111

ENTER INSTRUCTOR'S NAME .....
INSTRUCTOR 1

ENTER STUDENT NAME .....
(ENTER "QUIT" WHEN FINISHED)
STUDENT 1

ENTER STUDENT NUMBER .....
111111
```

```
ENTER STUDENT NAME .....
(ENTER "QUIT" WHEN FINISHED)
STUDENT 2

ENTER STUDENT NUMBER .....
222222

ENTER STUDENT NAME .....
(ENTER "QUIT" WHEN FINISHED)
QUIT

ENTER COURSE TITLE .....
(ENTER "QUIT" WHEN FINISHED)
COURSE 2

ENTER COURSE NUMBER .....
222222

ENTER INSTRUCTOR'S NAME .....
INSTRUCTOR 2

ENTER STUDENT NAME .....
(ENTER "QUIT" WHEN FINISHED)
STUDENT 1

ENTER STUDENT NUMBER .....
111111

ENTER STUDENT NAME .....
(ENTER "QUIT" WHEN FINISHED)
QUIT

ENTER COURSE TITLE .....
(ENTER "QUIT" WHEN FINISHED)
COURSE 3

ENTER COURSE NUMBER .....
333333

ENTER INSTRUCTOR'S NAME .....
INSTRUCTOR 2

ENTER STUDENT NAME .....
(ENTER "QUIT" WHEN FINISHED)
QUIT

ENTER COURSE TITLE .....
(ENTER "QUIT" WHEN FINISHED)
QUIT
OK,
```

Source Listing — CLASS.INQUIRY.COBOL85

The following is a source listing of the program CLASS.INQUIRY.COBOL85.

```
  IDENTIFICATION DIVISION.
  PROGRAM-ID.    CLASSINQ.
  AUTHOR.        PRIMATE.

  REMARKS.
*****************************************************************
*                                                               *
* THIS INQUIRY PROGRAM DEMONSTRATES THE PROCESSING OF A         *
* MIDASPLUS INDEXED FILE THAT CONTAINS VARIABLE-                *
* LENGTH RECORDS.                                               *
*                                                               *
*****************************************************************

  ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.

*****************************************************************
*                                                               *
* EACH RECORD IN THE CLASS FILE CONTAINS A UNIQUE PRIMARY KEY   *
* (COURSE ID), A UNIQUE SECONDARY KEY (COURSE TITLE), AND A     *
* NON-UNIQUE SECONDARY KEY (INSTRUCTOR NAME).                   *
*                                                               *
*****************************************************************

  SELECT CLASS-FILE
         ASSIGN TO MIDASPLUS
         ORGANIZATION IS INDEXED
         ACCESS MODE  IS DYNAMIC
         FILE STATUS  IS WS-CLASS-FILE-STATUS
         RECORD KEY   IS COURSE-ID
         ALTERNATE RECORD KEY IS COURSE-TITLE
         ALTERNATE RECORD KEY IS INSTRUCTOR-NAME WITH DUPLICATES.

  DATA DIVISION.
  FILE SECTION.

*****************************************************************
*                                                               *
* EACH RECORD MAY CONTAIN FROM 0 TO 25 STUDENTS DEPENDING UPON  *
* ENROLLMENT.                                                   *
*                                                               *
*****************************************************************
       FD CLASS-FILE
      RECORD IS VARYING FROM 68 TO 718 CHARACTERS
      VALUE OF FILE-ID IS WS-CLASS-FILE-NAME.
```

```
01 CLASS-REC.
   05  COURSE-ID                PIC 9(06).
   05  COURSE-TITLE             PIC X(40).
   05  INSTRUCTOR-NAME          PIC X(20).
   05  NUMBER-OF-STUDENTS       PIC 9(04) COMP.
   05  STUDENT-RECORD           OCCURS 0 TO 25 TIMES
                                DEPENDING ON
                                NUMBER-OF-STUDENTS
                                INDEXED BY STUD-INDX.
       10  STUDENT-ID           PIC 9(06).
       10  STUDENT-NAME         PIC X(20).


WORKING-STORAGE SECTION.

    01  WORK-FIELDS.
    05  WS-INSTRUCTOR-NAME          PIC X(20)     VALUE SPACES.
    05  WS-CLASS-FILE-NAME          PIC X(80)     VALUE SPACES.
    05  WS-CLASS-FILE-STATUS        PIC 9(02)     VALUE ZEROES.
    05  WS-CLASS-SW                 PIC X(01)     VALUE SPACES.
        88  END-OF-CLASSES                        VALUE 'Y'.
    05  WS-OPTION                   PIC X(01)     VALUE SPACES.
        88  LIST-BY-COURSE-ID                     VALUE '1'.
        88  LIST-BY-COURSE-TITLE                  VALUE '2'.
        88  LIST-BY-INSTRUCTOR                    VALUE '3'.
        88  QUIT                                  VALUE '4'.

PROCEDURE DIVISION.

0000-MAINLINE.

    PERFORM 1000-INITIALIZE THRU 1000-EXIT.

    PERFORM 2000-PROCESS-INQUIRY THRU 2000-EXIT
            UNTIL QUIT.

    PERFORM 3000-FINISH-UP THRU 3000-EXIT.

    STOP RUN.

0000-EXIT.
    EXIT.


1000-INITIALIZE.

    DISPLAY ' '.
    DISPLAY 'ENTER CLASS FILE NAME .....'.
    ACCEPT WS-CLASS-FILE-NAME.
    OPEN INPUT CLASS-FILE.
```

```
1000-EXIT.
    EXIT.

2000-PROCESS-INQUIRY.

*****************************************************************
*                                                               *
* DISPLAY MENU AND PROCESS USER'S INQUIRIES UNTIL USER QUITS    *
*                                                               *
*****************************************************************

     DISPLAY ' '.
     DISPLAY '***** SELECT OPTION BY NUMBER ***** '.
     DISPLAY ' '.
     DISPLAY '1 : LIST STUDENTS ENROLLED BY COURSE ID  '.
     DISPLAY ' '.
     DISPLAY '2 : LIST STUDENTS ENROLLED BY COURSE TITLE  '.
     DISPLAY ' '.
     DISPLAY '3 : LIST STUDENTS ENROLLED BY INSTRUCTOR '.
     DISPLAY ' '.
     DISPLAY '4 : EXIT '.
     DISPLAY ' '.

   ACCEPT WS-OPTION.
   IF LIST-BY-COURSE-ID
        PERFORM 2100-LIST-BY-COURSE-ID THRU 2100-EXIT
        GO TO 2000-EXIT.

     IF LIST-BY-COURSE-TITLE
        PERFORM 2200-LIST-BY-COURSE-TITLE THRU 2200-EXIT
        GO TO 2000-EXIT.

     IF LIST-BY-INSTRUCTOR
        PERFORM 2300-LIST-BY-INSTRUCTOR THRU 2300-EXIT
        GO TO 2000-EXIT.

     IF QUIT
        DISPLAY ' '
        DISPLAY 'BYE NOW!!!'
        DISPLAY ' '
        GO TO 2000-EXIT.

2000-EXIT.
    EXIT.

2100-LIST-BY-COURSE-ID.
```

```
*****************************************************************
*                                                               *
* READ CLASS FILE USING COURSE-ID (PRIMARY KEY) AND LIST        *
* STUDENTS ENROLLED.                                            *
*                                                               *
*****************************************************************

      DISPLAY ' '.
      DISPLAY 'ENTER COURSE NUMBER .....'.
      DISPLAY '(ENTER SPACES TO RETURN TO MAIN MENU)'.

      ACCEPT COURSE-ID.
      IF COURSE-ID = SPACES
         GO TO 2100-EXIT.

      READ CLASS-FILE
           INVALID KEY
                   DISPLAY ' '
                   DISPLAY '*** ERROR ***'
                   EXHIBIT COURSE-ID
                   EXHIBIT WS-CLASS-FILE-STATUS
                   DISPLAY ' '
                   GO TO 2100-EXIT.

      DISPLAY ' '
      DISPLAY 'STUDENTS ENROLLED IN COURSE ' COURSE-ID.

      PERFORM 9000-EXHIBIT-CLASS-REC.

      2100-EXIT.
      EXIT.

  2200-LIST-BY-COURSE-TITLE.

*****************************************************************
*                                                               *
* READ CLASS FILE USING COURSE-TITLE (UNIQUE SECONDARY KEY)     *
* AND LIST STUDENTS ENROLLED.                                   *
*                                                               *
*****************************************************************

      DISPLAY ' '.
      DISPLAY 'ENTER COURSE TITLE .....'.
      DISPLAY '(ENTER SPACES TO RETURN TO MAIN MENU)'.

      ACCEPT COURSE-TITLE.
      IF COURSE-TITLE = SPACES
         GO TO 2200-EXIT.
```

```
           READ CLASS-FILE
                KEY IS COURSE-TITLE
                INVALID KEY
                        DISPLAY ' '
                        DISPLAY '*** ERROR ***'
                        EXHIBIT COURSE-TITLE
                        EXHIBIT WS-CLASS-FILE-STATUS
                        DISPLAY ' '
                        GO TO 2200-EXIT.

           DISPLAY ' '
           DISPLAY 'STUDENTS ENROLLED IN ' COURSE-TITLE.

           PERFORM 9000-EXHIBIT-CLASS-REC.

         2200-EXIT.
           EXIT.

         2300-LIST-BY-INSTRUCTOR.

      ****************************************************************
      *                                                              *
      * READ CLASS FILE USING INSTRUCTOR-NAME (NON-UNIQUE SECONDARY  *
      * KEY) AND LIST STUDENTS ENROLLED.                             *
      *                                                              *
      ****************************************************************

           DISPLAY ' '.
           DISPLAY 'ENTER INSTRUCTOR NAME .....'.
           DISPLAY '(ENTER SPACES TO RETURN TO MAIN MENU)'.

           ACCEPT INSTRUCTOR-NAME
           IF INSTRUCTOR-NAME = SPACES
              GO TO 2300-EXIT.

           START CLASS-FILE
                 KEY IS EQUAL TO INSTRUCTOR-NAME
                 INVALID KEY
                         DISPLAY ' '
                         DISPLAY '*** ERROR ***'
                         EXHIBIT INSTRUCTOR-NAME
                         EXHIBIT WS-CLASS-FILE-STATUS
                         DISPLAY ' '
                         GO TO 2300-EXIT.

           DISPLAY ' '
           DISPLAY 'STUDENTS ENROLLED IN COURSES TAUGHT BY '
              INSTRUCTOR-NAME.

           MOVE 'N' TO WS-CLASS-SW.
           MOVE INSTRUCTOR-NAME TO WS-INSTRUCTOR-NAME.
```

```
           PERFORM 2310-PROCESS-CLASSES THRU 2310-EXIT
                   UNTIL END-OF-CLASSES.

       2300-EXIT.
           EXIT.

       2310-PROCESS-CLASSES.
           READ CLASS-FILE NEXT RECORD
                AT END
                   MOVE ZEROES TO WS-CLASS-FILE-STATUS
                   MOVE 'Y' TO WS-CLASS-SW
                   GO TO 2310-EXIT.

          IF INSTRUCTOR-NAME = WS-INSTRUCTOR-NAME
             PERFORM 9000-EXHIBIT-CLASS-REC
          ELSE
             MOVE 'Y' TO WS-CLASS-SW.
       2310-EXIT.
           EXIT.

       3000-FINISH-UP.

           CLOSE CLASS-FILE.

       3000-EXIT.
           EXIT.

       9000-EXHIBIT-CLASS-REC.

      ****************************************************************
      *                                                              *
      * LIST 0 TO 25 STUDENTS DEPENDING UPON ENROLLMENT.             *
      *                                                              *
      ****************************************************************

           DISPLAY ' '.
           EXHIBIT COURSE-TITLE.
           EXHIBIT COURSE-ID.
           EXHIBIT INSTRUCTOR-NAME.
               IF NUMBER-OF-STUDENTS > 0
              PERFORM 9010-EXHIBIT-STUDENTS THRU 9010-EXIT
                      VARYING STUD-INDX FROM 1 BY 1
                      UNTIL STUD-INDX > NUMBER-OF-STUDENTS
           ELSE
              DISPLAY ' '
              DISPLAY 'NO STUDENTS ENROLLED IN THIS COURSE'
              DISPLAY ' '.

       9000-EXIT.
           EXIT.
            9010-EXHIBIT-STUDENTS.
```

```
            DISPLAY ' '.
            EXHIBIT STUDENT-NAME (STUD-INDX).
            EXHIBIT STUDENT-ID (STUD-INDX).
            DISPLAY ' '.

        9010-EXIT.
            EXIT.
```

Compile and Link Dialog — CLASS.INQUIRY.COBOL85

The preceding program, stored as CLASS.INQUIRY.COBOL85, can be compiled and linked with the following dialog.

```
OK, COBOL85 CLASS.INQUIRY  -LISTING -VARY
[COBOL85 Rev. 1.0-22.0 Copyright (c) Prime Computer, Inc. 1988]
[0 ERRORS IN PROGRAM: CLASS.INQUIRY.COBOL85]

OK, BIND
[BIND Rev. 22.0 Copyright (c) 1988, Prime Computer, Inc.]
: LO CLASS.INQUIRY
: LI COBOL85LIB
: LI
BIND COMPLETE
: FILE
OK,
```

Program Execution — CLASS.INQUIRY.COBOL85

The following is a sample execution dialog for the program CLASS.INQUIRY.COBOL85.

```
OK, RESUME CLASS.INQUIRY

ENTER CLASS FILE NAME .....
CLASS.FILE.MIDAS

***** SELECT OPTION BY NUMBER *****

1 : LIST STUDENTS ENROLLED BY COURSE ID

2 : LIST STUDENTS ENROLLED BY COURSE TITLE

3 : LIST STUDENTS ENROLLED BY INSTRUCTOR

4 : EXIT

1
```

```
ENTER COURSE NUMBER .....
(ENTER SPACES TO RETURN TO MAIN MENU)
111111

STUDENTS ENROLLED IN COURSE    111111

COURSE-TITLE = COURSE 1
COURSE-ID =    111111
INSTRUCTOR-NAME = INSTRUCTOR 1

STUDENT-NAME = STUDENT 1
STUDENT-ID =    111111


STUDENT-NAME = STUDENT 2
STUDENT-ID =    222222

***** SELECT OPTION BY NUMBER *****

1 : LIST STUDENTS ENROLLED BY COURSE ID

2 : LIST STUDENTS ENROLLED BY COURSE TITLE

3 : LIST STUDENTS ENROLLED BY INSTRUCTOR

4 : EXIT

1

ENTER COURSE NUMBER .....
(ENTER SPACES TO RETURN TO MAIN MENU)
222222

STUDENTS ENROLLED IN COURSE    222222

COURSE-TITLE = COURSE 2
COURSE-ID =    222222
INSTRUCTOR-NAME = INSTRUCTOR 2

STUDENT-NAME = STUDENT 1
STUDENT-ID =    111111


***** SELECT OPTION BY NUMBER *****

1 : LIST STUDENTS ENROLLED BY COURSE ID

2 : LIST STUDENTS ENROLLED BY COURSE TITLE

3 : LIST STUDENTS ENROLLED BY INSTRUCTOR
```

```
4 : EXIT

1

ENTER COURSE NUMBER .....
(ENTER SPACES TO RETURN TO MAIN MENU)
333333

STUDENTS ENROLLED IN COURSE    333333

COURSE-TITLE = COURSE 3
COURSE-ID =    333333
INSTRUCTOR-NAME = INSTRUCTOR 2

NO STUDENTS ENROLLED IN THIS COURSE


***** SELECT OPTION BY NUMBER *****

1 : LIST STUDENTS ENROLLED BY COURSE ID

2 : LIST STUDENTS ENROLLED BY COURSE TITLE

3 : LIST STUDENTS ENROLLED BY INSTRUCTOR

4 : EXIT

1

ENTER COURSE NUMBER .....
(ENTER SPACES TO RETURN TO MAIN MENU)
777777

*** ERROR ***
COURSE-ID =    777777
WS-CLASS-FILE-STATUS =    23


***** SELECT OPTION BY NUMBER *****

1 : LIST STUDENTS ENROLLED BY COURSE ID

2 : LIST STUDENTS ENROLLED BY COURSE TITLE

3 : LIST STUDENTS ENROLLED BY INSTRUCTOR

4 : EXIT

2
```

```
ENTER COURSE TITLE .....
(ENTER SPACES TO RETURN TO MAIN MENU)
COURSE 1

STUDENTS ENROLLED IN COURSE 1

COURSE-TITLE = COURSE 1
COURSE-ID =    111111
INSTRUCTOR-NAME = INSTRUCTOR 1

STUDENT-NAME = STUDENT 1
STUDENT-ID =    111111


STUDENT-NAME = STUDENT 2
STUDENT-ID =    222222


***** SELECT OPTION BY NUMBER *****

1 : LIST STUDENTS ENROLLED BY COURSE ID

2 : LIST STUDENTS ENROLLED BY COURSE TITLE

3 : LIST STUDENTS ENROLLED BY INSTRUCTOR

4 : EXIT

2

ENTER COURSE TITLE .....
(ENTER SPACES TO RETURN TO MAIN MENU)
COURSE 2

STUDENTS ENROLLED IN COURSE 2

COURSE-TITLE = COURSE 2
COURSE-ID =    222222
INSTRUCTOR-NAME = INSTRUCTOR 2

STUDENT-NAME = STUDENT 1
STUDENT-ID =    111111

***** SELECT OPTION BY NUMBER *****

1 : LIST STUDENTS ENROLLED BY COURSE ID

2 : LIST STUDENTS ENROLLED BY COURSE TITLE

3 : LIST STUDENTS ENROLLED BY INSTRUCTOR
```

```
4 : EXIT

2

ENTER COURSE TITLE .....
(ENTER SPACES TO RETURN TO MAIN MENU)
COURSE 3

STUDENTS ENROLLED IN COURSE 3

COURSE-TITLE = COURSE 3
COURSE-ID =    333333
INSTRUCTOR-NAME = INSTRUCTOR 2

NO STUDENTS ENROLLED IN THIS COURSE


***** SELECT OPTION BY NUMBER *****

1 : LIST STUDENTS ENROLLED BY COURSE ID

2 : LIST STUDENTS ENROLLED BY COURSE TITLE

3 : LIST STUDENTS ENROLLED BY INSTRUCTOR

4 : EXIT

2

ENTER COURSE TITLE .....
(ENTER SPACES TO RETURN TO MAIN MENU)
COURSE 4

*** ERROR ***
COURSE-TITLE = COURSE 4
WS-CLASS-FILE-STATUS =    23


***** SELECT OPTION BY NUMBER *****

1 : LIST STUDENTS ENROLLED BY COURSE ID

2 : LIST STUDENTS ENROLLED BY COURSE TITLE

3 : LIST STUDENTS ENROLLED BY INSTRUCTOR

4 : EXIT

3
```

```
ENTER INSTRUCTOR NAME .....
(ENTER SPACES TO RETURN TO MAIN MENU)
INSTRUCTOR 1

STUDENTS ENROLLED IN COURSES TAUGHT BY INSTRUCTOR 1

COURSE-TITLE = COURSE 1
COURSE-ID =    111111
INSTRUCTOR-NAME = INSTRUCTOR 1

STUDENT-NAME = STUDENT 1
STUDENT-ID =    111111

STUDENT-NAME = STUDENT 2
STUDENT-ID =    222222


***** SELECT OPTION BY NUMBER *****

1 : LIST STUDENTS ENROLLED BY COURSE ID

2 : LIST STUDENTS ENROLLED BY COURSE TITLE

3 : LIST STUDENTS ENROLLED BY INSTRUCTOR

4 : EXIT

3

ENTER INSTRUCTOR NAME .....
(ENTER SPACES TO RETURN TO MAIN MENU)
INSTRUCTOR 2

STUDENTS ENROLLED IN COURSES TAUGHT BY INSTRUCTOR 2

COURSE-TITLE = COURSE 2
COURSE-ID =    222222
INSTRUCTOR-NAME = INSTRUCTOR 2

STUDENT-NAME = STUDENT 1
STUDENT-ID =    111111


COURSE-TITLE = COURSE 3
COURSE-ID =    333333
INSTRUCTOR-NAME = INSTRUCTOR 2

NO STUDENTS ENROLLED IN THIS COURSE
```

```
***** SELECT OPTION BY NUMBER *****

1 : LIST STUDENTS ENROLLED BY COURSE ID

2 : LIST STUDENTS ENROLLED BY COURSE TITLE

3 : LIST STUDENTS ENROLLED BY INSTRUCTOR

4 : EXIT

3

ENTER INSTRUCTOR NAME .....
(ENTER SPACES TO RETURN TO MAIN MENU)
INSTRUCTOR 3

*** ERROR ***
INSTRUCTOR-NAME = INSTRUCTOR 3
WS-CLASS-FILE-STATUS =    23


***** SELECT OPTION BY NUMBER *****

1 : LIST STUDENTS ENROLLED BY COURSE ID

2 : LIST STUDENTS ENROLLED BY COURSE TITLE

3 : LIST STUDENTS ENROLLED BY INSTRUCTOR

4 : EXIT

4

BYE NOW!!!
```

P

Glossary

alphabet-name
A programmer-defined word in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION that assigns a name to a specific character set or collating sequence.

arithmetic expression (arith-expr)
A numeric *data-name*, a numeric *literal*, such *data-names* and *literals* separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses. Any arithmetic expression may be preceded by a unary operator.

assumed decimal point
A decimal point position that does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning but no physical representation.

AT END condition
A condition caused by one of the following:

1. During the execution of a READ statement for a sequentially accessed file, the last record of the file has been processed, or the number of significant digits in the relative record number is larger than the size of the relative key data item, or an optional input file is not present.
2. During the execution of a RETURN statement, no next logical record exists for the associated sort or merge file.
3. During the execution of a SEARCH statement, the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

BIND
Prime's linker for Executable Program Formats (EPFs).

block
A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block may contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical record(s) that are either continued within the block or that overlap the block. The term is synonymous with physical record.

called program
A program that is the object of a CALL statement combined at object time with the calling program to produce a run unit.

calling program
A program that executes a CALL to another program.

character
The basic indivisible unit of the language.

character position
A character position is the amount of physical storage required to store a single standard data format character whose usage is DISPLAY.

clause
An ordered set of consecutive COBOL85 *character-strings* whose purpose is to specify an attribute of an entry, or form a portion of a COBOL85 procedural statement.

collating sequence
The sequence in which the characters that are acceptable in a computer are ordered for purposes of sorting, merging, and comparing.

command line linking or loading
Use of BIND with all necessary options on one command line.

comment line
A source program line represented by an asterisk in the indicator area of the line and any characters from the computer's character set in Area A and Area B of that line. The comment line serves only for documentation in a program. A special form of comment line represented by a slash (/) in the indicator area of the line and any characters from the computer's character set in Area A and Area B of that line causes page ejection prior to printing the comment.

comment-entry
An entry in the IDENTIFICATION DIVISION that may be any combination of characters from the computer character set.

condition-name
A user-defined word assigned to a specific value, set of values, or range of values, within the complete set of values that a conditional variable may possess; or the user-defined word assigned to a status of an implementor-defined switch or device. *condition-names* are defined with *level-number* 88.

conditional variable
A data item that has one or more values to which a *condition-name* is assigned.

current record
The record that is available in the record area associated with the file.

current record pointer
A conceptual entity that is used to select the next record.

data-description-entry
An entry in the DATA DIVISION that is composed of a *level-number* followed by a *data-name*, or the reserved word FILLER, and then followed by a set of data clauses, as required.

data-name
A user-defined word that names a data item described in a *data-description-entry* in the DATA DIVISION. A *data-name* can be subscripted, indexed, or qualified, unless these attributes are specifically prohibited by the rules for that format.

declaratives
A set of one or more special-purpose sections, written at the beginning of the PROCEDURE DIVISION, the first of which is preceded by the keyword DECLARATIVES and the last of which is followed by the keywords END DECLARATIVES. A declarative is composed of a section header, followed by a USE sentence, followed by zero, one, or more associated paragraphs.

division header
A combination of words followed by a period and a space that indicates the beginning of a division. The division headers are

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```

dynamic access
An access mode in which records can be obtained from or placed into a mass storage file in a nonsequential manner (see Random Access) and obtained from a file in a sequential manner (see Sequential Access), during the scope of the same OPEN statement.

dynamic runfile
Also called an Executable Program Format (EPF); a file containing a description of a complete program or library that is assigned addresses at program runtime rather than at program link time. Thus, it may execute in more than one address in memory, and more than one dynamic runfile may exist in memory at once. BIND produces dynamic runfiles.

editing character
A single character or a fixed two-character combination used in a PICTURE clause to change output format. Editing characters are listed in the section called PICTURE in Chapter 7.

EPF
Executable Program Format. An EPF consists of binary object code for a complete program. This includes code, data, and stack allocation information added by BIND to enable you to execute the file with RESUME, and link to system subroutines at runtime.

elementary item
A data item that is described as not being further subdivided.

file-description-entry
An entry in the FILE SECTION of the DATA DIVISION that is composed of the level indicator FD, followed by a *file-name*, and then followed by a set of file clauses as required.

file-name
A user defined word that names a file described in a *file-description-entry* or a sort-merge *file-description-entry* within the FILE SECTION of the DATA DIVISION.

fixed-length record
A record associated with a file whose file description or sort-merge description entry requires that all records contain the same number of character positions.

fullword
A unit of address space four bytes (32 bits) in size.

halfword
A unit of address space two bytes (16 bits) in size.

imperative statement
A statement that begins with an imperative verb and specifies an unconditional action to be taken. An imperative statement may consist of a sequence of imperative statements.

index

1. A computer storage position or register, the contents of which represent the identification of a particular element in a table.
2. A key that identifies a record for a file whose organization is INDEXED.

index data item
A data item in which the value associated with an *index-name* can be stored.

index-name
A user-defined word that names an index associated with a specific table.

input procedure
A set of statements that are executed each time a record is released to a sort file.

key
A data item that identifies the location of a record, or a set of data items that serve to identify the ordering of data.

key of reference
The key, either primary or alternate, currently being used to access records within an indexed file.

level indicator
Two alphabetic characters that identify a specific type of file or a position in a hierarchy. A level indicator is found only in the DATA DIVISION and must be one of the following: FD, SD.

level-number
A user-defined word that indicates the position of a data item in the hierarchical structure of a logical record or that indicates special properties of a *data-description-entry*. A *level-number* is expressed as a one-digit or two-digit number. *level-numbers* in the range 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. *level-numbers* in the range 1 through 9 may be written either as a single digit or as a zero followed by a significant digit. *level-numbers* 66, 77, and 88 identify a *data-description-entry* with special properties.

linker
A utility that resolves external references within a runfile, so that different pieces of code (such as a calling program and a subroutine) can find each other at runtime. The Prime linkers, BIND, SEG, and LOAD, are all linkers in that they all resolve external references as they are given binary files. BIND, however, leaves all loading to be done dynamically at runtime by PRIMOS.

logical operator
One of the reserved words AND, OR, or NOT. In the formation of a condition, both or either of AND and OR can be used as logical connectives. NOT can be used for logical negation.

merge file
A collection of records to be merged by a MERGE statement. The merge file, identified by SD, is created and can be used only by the merge function.

MIDASPLUS
A Prime utility that handles indexed and relative file creation and access for COBOL85.

mnemonic-name
A user-defined word that is associated, in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION, with a specified *implementor-name*, such as CONSOLE, or a *switch-name*.

native character set
Prime's native character set is the ASCII set defined in Appendix B.

noncontiguous items
Elementary data items in the WORKING-STORAGE and LINKAGE SECTION that bear no hierarchical relationship to other data items.

output procedure
A set of statements to which control is given during execution of a SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function has selected the next record in merged order.

paragraph
In the PROCEDURE DIVISION, a *paragraph-name* followed by a period, a space, and zero, one, or more sentences. In the IDENTIFICATION and ENVIRONMENT DIVISION, a paragraph header followed by zero, one, or more entries.

paragraph header
A reserved word, followed by a period and a space that indicates the beginning of a paragraph in the IDENTIFICATION and ENVIRONMENT DIVISION. The permissible paragraph headers are

PROGRAM-ID.
AUTHOR.
REMARKS.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
FILE-CONTROL.
I-O-CONTROL.

paragraph-name
A user-defined word that identifies and begins a paragraph in the PROCEDURE DIVISION. *paragraph-names* must start in columns 8 through 11.

pathname
The name of a Prime file, including, if necessary, its disk name and the name of the directory and subdirectories containing it.

phrase
A phrase is an ordered set of one or more consecutive COBOL85 *character-strings* that form a portion of a COBOL85 procedural statement or of a COBOL85 clause.

primary index
For MIDASPLUS, the primary record key or relative key.

primary record key
A key whose contents uniquely identify a record within an indexed file.

PRISAM
The Prime Recoverable Indexed Sequential Access Method. Data management interface for handling indexed and relative files for COBOL85 in a transaction processing environment.

procedure-name
A *paragraph-name* (which may be qualified) in the PROCEDURE DIVISION, or a *section-name* in the PROCEDURE DIVISION.

punctuation character
A character that belongs to the following set:

, ; . " ' ( ), or the pseudo-text delimiter ==

qualified data-name
An identifier that is composed of a *data-name* followed by OF or IN and another *data-name* at a higher level of the same hierarchy. The second *data-name* may also be qualified.

qualifier

1. A *data-name* that is used in a reference together with OF or IN and another *data-name* at a lower level in the same hierarchy.
2. A *section-name* that is used in a reference together with OF or IN and a *paragraph-name* specified in that section.
3. A *library-name* that is used in a reference together with OF or IN and a *text-name* associated with that library.

random access
An access mode in which the value of a key data item identifies the record to be accessed in or written to a relative or indexed file.

record-description-entry
The total set of *data-description-entries* associated with a particular record.

reserved word
A COBOL85 word specified in the list of words in Table B-2 of Appendix B, and which must not appear in the programs as a user-defined word.

runfile or run unit
The PRIMOS file to be used for execution. It consists of one or more user object files plus any necessary library files.

section
In the PROCEDURE DIVISION, a *section-name* followed by the word SECTION, an optional segment number, a period, a space and zero, one or more paragraphs. In the ENVIRONMENT and DATA DIVISION, a section header followed by zero, one or more entries. Valid section headers are

```
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
```

section-name
A user-defined word that names a section in the PROCEDURE DIVISION. *section-names* must start within columns 8-11.

sentence
A sequence of one or more statements, the last of which is terminated by a period followed by a space.

sequential access
An access mode in which logical records are obtained from or placed into a file in a consecutive sequence determined by the order of records in the file.

sort file
A collection of records to be sorted by a SORT statement. The sort file, identified by SD, is created and can be used by the sort function only.

sort-merge file-description-entry
An entry in the FILE SECTION of the DATA DIVISION that is composed of the level indicator SD, followed by a *file-name*, and then followed by a set of file clauses as required.

statement
A syntactically valid combination of words and symbols written in the PROCEDURE DIVISION, beginning with a verb.

static runfile
A runfile that is assigned addresses when it is linked rather than at runtime. All static runfiles tend to use the same addresses so that one overwrites another. SEG and LOAD produce static runfiles. See Dynamic Runfile.

subscript
An occurrence number that identifies a particular element in a table. The occurrence number can be represented by either an integer, a *data-name* optionally followed by an integer with the operator + or -, an *index-name* optionally followed by an integer with the operator + or -, or an *arith-expr* optionally followed by an integer with the operator + or -.

table
A set of logically consecutive items, all of the same description, that are defined in the DATA DIVISION with the OCCURS clause.

table element
A data item in a set of repeated items comprising a table.

unary operator
A plus or a minus sign that precedes a variable or a left parenthesis in an arithmetic expression and that has the effect of multiplying the expression by +1 or -1, respectively.

variable-length record
A record associated with a file whose file description or sort-merge description entry permits records to contain a varying number of character positions.

variable occurrence data item
A variable occurrence data item is a table element that is repeated a variable number of times. Such an item must contain an OCCURS DEPENDING ON clause in its *data-description-entry*, or be subordinate to such an item.

word

1. In storage, 32 bits.
2. A COBOL85 word is a *character-string* of not more than 30 characters chosen from the following set of 64 characters:

0-9	(digits)
A-Z	(uppercase letters)
a-z	(lowercase letters)
-	(hyphen)
_	(underscore)

All words except *level-numbers*, *section-names*, *segment-numbers*, and *paragraph-names* must contain at least one alphabetic character or a hyphen. A word must not begin or end with a hyphen. It is delimited by a space, or by proper punctuation. A word may contain more than one embedded hyphen; consecutive embedded hyphens are also permitted.

Index

Index

D

E

F

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Surveys

Reader Response Form
COBOL85 Reference Guide
DOC10166-1LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate this document for overall usefulness?

   ☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

   _____
   _____
   _____
   _____
   _____
   _____

3. What faults or errors in this manual gave you problems?

   _____
   _____
   _____
   _____
   _____
   _____

4. How does this manual compare to equivalent manuals produced by other computer companies?

   ☐ *Much better* ☐ *Slightly better* ☐ *About the same*
   ☐ *Much worse* ☐ *Slightly worse* ☐ *Can't judge*

5. Which other companies' manuals have you read?

   _____
   _____

Name:_____
Position:_____
Company:_____
Address:_____
_____
_____Postal Code:_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

Prime™

Attention: Technical Publications
Bldg 10
Prime Park, Natick, Ma. 01760