HP COBOL II/XL Reference Manual

Series 900 HP 3000 Computer Systems

HP Part No. 31500-90001
Printed in U.S.A. July 1991

E0791

Notice

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DoD agencies, Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Printing History

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The dates on the title page change only when a new edition or a new update is published. No information is incorporated into a reprinting unless it appears as a prior update; the edition does not change when an update is incorporated.

The software code printed alongside the data indicates the version level of the software product at the time the manual or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

First Edition	November 1987	31500A.00.12
Second Edition	October 1988	31500A.01.06
Third Edition	July 1991	31500A.04.03

Preface

This reference manual documents the HP COBOL II language for programming on Hewlett-Packard computer systems. HP COBOL II is based on the 1974 and ANSI COBOL'1985 Standard X3.23-1985.

This manual is a reference text for programmers who have a working knowledge of COBOL. The objective of the *HP COBOL II Reference Manual* is to guide you in writing source programs in HP COBOL II, compiling them into object programs, preparing and executing them.

Note The information in the main body of this manual is generic. Machine-dependent information is in Appendix H, "MPE XL System Dependencies."

This manual is organized as follows:

Chapter 1	Introduces HP COBOL II. Summarizes standard capabilities, HP extensions, and new features.
Chapter 2	Describes constructs of COBOL.
Chapter 3	Describes COBOL program elements.
Chapter 4	Explains how data is described and referenced in COBOL.
Chapter 5	Describes the Identification Division.
Chapter 6	Describes the Environment Division.
Chapter 7	Describes the Data Division.
Chapter 8	Gives a general description of the Procedure Division.
Chapter 9	Describes specific Procedure Division statements.
Chapter 10	Describes all the COBOL functions and how to call them.
Chapter 11	Explains interprogram communication in COBOL.
Chapter 12	Describes SORT-MERGE operations.
Chapter 13	Describes the COBOL debugging facility.
Chapter 14	Describes source-text manipulation statements.
Appendix A	Lists the error messages produced by the COBOL compiler and by COBOL programs.
Appendix B	Lists preprocessor commands and $CONTROL options.
Appendix C	Describes differences between COBOL'85 and COBOL'74 and lists incompatibilities and obsolete features of the language.
Appendix D	Defines the ASCII and EBCDIC character sets.
Appendix E	Contains the COBOL glossary.

Appendix F	Lists COBOL reserved words.
Appendix G	Describes the COBEDIT program and COPY libraries.
Appendix H	Describes machine-dependent information for MPE XL systems.

Additional Documentation

More information on HP COBOL II/XL is in the following manuals:

- *HP COBOL II/XL Programmer's Guide* (31500-90002)
- *HP COBOL II/XL Quick Reference Guide* (31500-90003)

This manual references the following manuals:

- *HP Toolset/XL Reference Manual* (36044-90001)
- *HP Symbolic Debugger/XL Reference Manual* (31508-90003)
- *Using KSAM/XL* (32650-90168)
- *KSAM/3000 Reference Manual* (30000-90079)
- *MPE XL Intrinsics Reference Manual* (32650-90028)
- *MPE XL Commands Reference Manual* (32650-90003)
- *Account Structure and Security Reference Manual* (32650-90041)
- *Native Language Programmer's Guide* (32650-90022)
- *System Startup, Configuration, and Shutdown Reference Manual* (32650-90042)
- *HP Screen Management Intrinsic Library Reference Manual* (32424-90002)
- *HP System Dictionary/XL General Reference Manual* (32256-90004)
- *HP SQL/XL COBOL Application Programming Guide* (36216-90006)
- *Link Editor/XL Reference Manual*(32650-90030)
- *System Debug Reference Manual* (32650-90013)
- *TurboIMAGE/XL Reference Manual* (30391-90001)
- *Compiler Library/XL Reference Manual* (32650-90029)
- *Trap Handling Programmer's Guide* (32650-90026)

The following book, not available from HP, contains more information about the COBOL functions:

- *COBOL Functions: An Introduction*, by Donald A. Sordillo, published in 1990 by Prentice Hall, Inc.

Information on migrating COBOL programs from HP COBOL II/V to HP COBOL II/XL is in the following manual:

- *HP COBOL II/XL Migration Guide* (31500-90004)

What's New in This Release

This section briefly describes what is new in this release of HP COBOL II/XL and where to find more information.

Addendum to ANSI COBOL'85: Built-in COBOL Functions

This version of the HP COBOL II/XL compiler introduces the 42 built-in COBOL functions recently defined by Addendum 1 of the ANSI COBOL'85 standard. Chapter 10, "COBOL Functions," describes all the functions.

Compatibility and the $CONTROL POST85 Option

To use the new COBOL functions, you must use the $CONTROL option POST85.

This option was added because the COBOL functions introduce a new reserved word, FUNCTION. If your existing COBOL programs use the word FUNCTION as an identifier, those programs will continue to compile without $CONTROL POST85. However, if you want to use the new COBOL functions in a program that uses the word FUNCTION, you must change the word to another word and use $CONTROL POST85. For more information, see Chapter 10, “COBOL Functions.”

The TZ Environment Variable

Use the TZ environment variable to set the time zone. The COBOL functions CURRENT-DATE and WHEN-COMPILED use the value of this variable and the value of the hardware clock when reporting their results. For more information, see Chapter 10, "COBOL Functions."

HP COBOL II/XL Reference Manual Supplement

The *HP COBOL II/XL Reference Manual Supplement* (31500-90005) is no longer a separate manual. It has been moved into Appendix H of this manual.

Previous HP COBOL II/XL Documents

Information from the *HP COBOL II/XL Technical Addendum*, published in April, 1990 for MPE XL Release 2.1 and the *HP Communicator* article, published for MPE XL Release 3.0 have been incorporated into this manual.

These documented the following features:

- The $CONTROL NLS option. See Appendix H, "MPE XL System Dependencies," for more information.
- Dynamic file assignment with the USING phrase of the ASSIGN clause. See the "ASSIGN Clause" in chapter 6 for more information.
- An additional position in the COBRUNTIME variable for handling run-time errors. See Appendix H, "MPE XL System Dependencies" for more information.
- The RETURN-CODE special register. See Chapter 11 "Interprogram Communication" for more information.

Appendix G, "Summary of COBOL II Syntax," in the previous edition of this manual was duplicated in the *HP COBOL II/XL Quick Reference Guide*. It has been removed. See the *Quick Reference Guide* for this information.

Appendix H, "HPTOOLSET Program Development System," in the previous edition of this manual has been moved to the *HP COBOL II/XL Programmer's Guide*.

Acknowledgement

At the request of the American National Standards Institute (ANSI), the following acknowledgement is reproduced in its entirety:

> Any organization interested in reproducing the COBOL standard and specifications in whole or in part, using ideas from this document as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgement paragraphs in their entirety as part of the preface to any such publication (any organization using a short passage from this document, such as in a book review, is requested to mention "COBOL" in acknowledgement of the source, but need not quote the acknowledgement):
>
> COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.
>
> No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.
>
> The authors and copyright holders of the copyrighted material used herein have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.
>
> FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the Univac++ I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F 28-8013, copyrighted 1959 by IBM, FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

Conventions

Notation	Description
▌	Change bars in the margin show where substantial changes have been made to this manual since the last edition. (Chapter 10 and Appendix H are new to this manual, but are not marked with change bars.)
UPPERCASE and UNDERLINING	Within syntax statements, characters in uppercase must be entered in exactly the order shown. Uppercase words that are underlined are keywords that are always required when the clause or statement in which they appear is used in your program. Uppercase words that are *not* underlined are optional, and may be included or omitted. They have no effect on program execution and serve only to make source program listings more readable. The following example illustrates this: `[FILE STATUS IS` *stat-item*`].` `STATUS` must be entered, `FILE` may be either included or omitted. See also "Underlining in dialog" on the following page.
italics	Within syntax statements, a word in italics represents a formal parameter, argument, or literal that you must replace with an actual value. In the following example, you must replace *filename* with the name of the file you want to release: `RELEASE` *filename*
punctuation	Within syntax statements, punctuation characters (other than brackets, braces, vertical parallel lines, and ellipses) must be entered exactly as shown.

{ }	Within syntax statements, when several elements within braces are stacked, you must select one. In the following equivalent examples, you select `ON` or `OFF`:
	`{ON }`
	`SETMSG {OFF}`
	`SETMSG { ON  }`
	`{ OFF }`

{| |}	Within syntax statements, bars in braces are choice indicators. One or more of the items within the choice indicators must be specified, but a single option may be specified only once.

`[ ]`

Within syntax statements, brackets enclose optional elements. In the following example, brackets around `,TEMP` indicate that the parameter and its delimiter are not required:

```
PURGE filename[,TEMP]
```

When several elements within brackets are stacked, you can select any one of the elements or none. In the following equivalent examples, you can select *devicename* or *deviceclass* or neither:

```
          [devicename]
SHOWDEV   [deviceclass]
```

```
SHOWDEV  [ devicename  ]
         [ deviceclass ]
```

Underlining in dialog

When it is necessary to distinguish user input from computer output, the input is underlined. See also underlining on the previous page.

```
NEW NAME?  ALPHA
```

`[ ] ...`

Brackets followed by a horizontal ellipsis indicate either that a previous bracketed element may be repeated zero or more times, or that elements have been omitted from the description.

```
[WITH DUPLICATES] ...
```

The ellipsis shows that the preceding clause may be repeated indefinitely.

`{ } ...`

Braces followed by a horizontal ellipses indicate either that the item within braces may be repeated one or more times, or that elements have been omitted from the description.

`␣`

Within syntax statements, the space symbol ␣ shows a required blank. In the following example, you must separate *modifier* and *variable* with a blank:

```
SET [(modifier)]␣(variable);
```

`<, >, =, <=, >=, <>`	These symbols are used in conditional statements to represent the keywords LESS THAN, GREATER THAN, EQUAL TO, LESS THAN OR EQUAL TO, GREATER THAN OR EQUAL TO, and NOT EQUAL TO, respectively. Although these symbols represent keywords, they are *not* underlined.
;	The semicolon is used only to improve readability and is always optional.
,	The comma is used only to improve readability, and is always optional.
.	The period is a terminator or delimiter that is always required where shown; it must always be entered at the end of every division name, section name, paragraph name, and sentence.
^	The caret is occasionally used in examples to represent an implied decimal point in computer memory.
`Shading`	Features that are part of the 1985 ANSI standard are `shaded`. They are accessible through the ANSI85 entry point.
LG200026_198	In some diagrams and tables, a number appears in the lower left corner. This number is for HP control purposes only and should not be interpreted as part of the diagram or table.

Contents

6. ENVIRONMENT DIVISION

10. COBOL Functions

Figures

Tables

1

Introduction

COBOL (Common Business Oriented Language) is the most widely used programming language for commercial applications. Hewlett-Packard's COBOL II is based on ANSI COBOL as specified in the American National Standard Programming Language COBOL (ANSI X3.23a-1989). It offers the following important features:

- Compatibility with American National Standards Institute (ANSI) COBOL at the highest level of all the required COBOL modules. The optional Report Writer and Communication modules are not supported. See "ANSI Standard Compliance" later in this chapter.
- Communication with programs written in other languages, including RPG, FORTRAN, SPL, C, and Pascal.
- Communication with other operating subsystems through intrinsic calls, including data management (TurboIMAGE), screen management (VPLUS), graphics (DSG), and program development tools (HPToolset/Dictionary).

ANSI Standard Compliance

The standards for COBOL were originally developed and defined by a national committee of computer manufacturers and users known as the Conference On Data Systems Languages (CODASYL). In 1960, under the guidance of this committee, the first official version of COBOL was designed, called COBOL'60. Since then subsequent versions were developed that significantly extended the power of the language. Later versions, ANSI COBOL'68, ANSI COBOL'74, and ANSI COBOL'85, followed the standards published and sanctioned by ANSI. In 1989, ANSI published an addendum that adds intrinsic functions to the language.

ANSI COBOL'85 and the functions addendum are organized on the basis of one nucleus and 11 functional processing modules. These elements are summarized in Table 1-1. Each module contains either two or three functional levels. In all cases, the lower levels are proper subsets of the higher level within the same module. The lowest levels supply elements needed for basic or elementary operations; the higher levels supply more extensive or sophisticated capabilities. The full ANSI COBOL is composed of the highest level of the nucleus and of each module.

Table 1-1. ANSI COBOL'85 Organization

Module	Function
Nucleus	Contains language elements necessary for internal processing.
Sequential I/O Module	Provides language elements for definition and access of sequentially organized external files.
Relative I/O Module	Provides capability for defining and accessing mass storage files in which records are identified by relative record number.
Indexed I/O Module	Provides capability for defining and accessing mass storage files in which records are identified by the value of a key and accessed through an index.
Sort/Merge Module	Provides sorting and merging operations in a COBOL program.
Report Writer Module	Provides for semi-automatic production of printed reports. Not implemented in HP COBOL II.
Segmentation Module	Provides overlaying of PROCEDURE DIVISION sections at object time.
Source Text Manipulation	Allows you to include predefined COBOL text into your program.
Debug Module	Offers a way to specify a debugging algorithm—the conditions under which data or procedure items are monitored during the execution of the program.
Interprogram Communication Module	Allows a program to communicate with other programs.
Communication Module	Provides ability to access, process, and create messages or portions of messages and to communicate through a message control system with local and remote communication devices. Not implemented in HP COBOL II.
Intrinsic Function Module	Provides 42 intrinsic or "built-in" COBOL functions including date and string functions, financial and statistical functions, trigonometric and other arithmetic functions. (These are not MPE intrinsics.)

Hewlett-Packard's COBOL II compiler conforms to the high level of the ANSI COBOL X3.23-1974 specification and to the high level of the ANSI COBOL X3.23a-1989 specification. It meets the corresponding level of the United States Government Federal Information Processing Standard as described in FIPS PUB 21-3. Some of the individual modules meet the high level requirements as described in Table 1-2.

Table 1-2. HP COBOL II Compiler Conformity Levels

ANSI COBOL'85 Module	Level Supported by HP COBOL II	FIPS PUB 21-3 Requirements for High Level
Nucleus	2	2
Sequential I/O	2[1]	2
Relative I/O	2	2
Indexed I/O	2	2
Source Text Manipulation	2	2
Interprogram Communication	2	2
Sort/Merge	1	1
Debug	1	2[2]
Segmentation	2	2[2]
Report Writer	Not supported	1[2]
Communication	Not supported	2[2]
Intrinsic Functions	1	1

1 Exceptions are PADDING CHARACTERS and RECORD DELIMITER.

2 These are optional modules that are not required for high-level implementation.

The HP COBOL II compiler is two compilers in one. That is, it contains two entry points: ANSI74 and ANSI85. The ANSI74 entry point accepts ANSI COBOL'74 syntax and semantics. The ANSI85 entry point provides an ANSI COBOL'85 compiler. To use any of the ANSI COBOL'85 features, use the ANSI85 entry point.

Note

Use the option $CONTROL POST85 for intrinsic functions.

Hereafter in this manual, *intrinsic functions* will be referred to as *COBOL functions*.

Table 1-3 lists the terms used in this manual.

Table 1-3. Terms Used in This Manual

This Term	Refers To
ANSI COBOL'74	The 1974 ANSI COBOL standard.
ANSI COBOL'85	The 1985 ANSI COBOL standard.
ANSI74	The compiler entry point used to invoke COBOL features of ANSI COBOL'74 plus HP extensions.
ANSI85	The compiler entry point used to invoke COBOL features of ANSI COBOL'85 plus COBOL functions and HP extensions.
HP COBOLII	Refers to the HP compiler that implements ANSI COBOL'74, ANSI COBOL'85, and HP extensions on MPE systems.

ANSI COBOL'85 Features in HP COBOL II

Features in the X3.23a-1989 American National Standard Programming Language COBOL that are in HP COBOL II are shown below. These features provide structured programming capabilities to make coding and maintenance easier. They also provide capabilities to enhance the manipulation of data initialization and extend I/O status codes for reporting I/O errors.

- EVALUATE statement.
- In-line PERFORM.
- Scope-delimited statements.
- CONTINUE statement.
- Optional FILLER clause.
- INITIALIZE statement.
- Reference modification.
- De-edited MOVE operations.
- BY CONTENT argument passing.
- Symbolic characters.
- CLASS clause.
- SET statement.
- NOT phrases.
- REPLACE statement.
- Alphabetic tests.
- EXTERNAL clause.
- GLOBAL clause.

Note Use the ANSI85 entry point to the HP COBOL II compiler whenever you use any ANSI COBOL'85 features. Throughout this manual, `shading` identifies those features that are available through the ANSI85 entry point.

ANSI Features Added Since ANSI COBOL'85

42 built-in functions have been added to ANSI standard COBOL. For more information on the COBOL functions, see Chapter 10, "COBOL Functions."

Note Use the option $CONTROL POST85 for the COBOL functions.

Compatibility Considerations

There are several compatibility issues to consider when using the HP COBOL II compiler.

Compatibility between ANSI COBOL'74 and ANSI COBOL'85

The HP COBOL II compiler is compatible with the ANSI COBOL'74 standard. Through the use of entry points to the compiler you can choose which standard to execute, thereby avoiding any conversion issues between the two standards. Some incompatibilities between ANSI COBOL'74 and ANSI COBOL'85, such as reserved words, allow you to preserve existing source code while doing new development using the ANSI COBOL'85 standard. Or if you choose to move all of your applications to the ANSI COBOL'85 standard, the compiler flags the syntax changes, allowing you to modify, recompile, and execute using the ANSI COBOL'85 entry point.

For a complete list of the incompatibilities between ANSI COBOL'74 and ANSI COBOL'85, as well as obsolete features to consider when upgrading, refer to Appendix C.

Compatibility of COBOL Functions

The COBOL functions add the reserved word FUNCTION to the COBOL language. HP COBOL II maintains compatibility with existing programs. Your existing programs will continue to compile without change. If you want to use the COBOL functions, you must use $CONTROL POST85 and you must not use the word FUNCTION as an identifier anywhere in your program. If you have used the word FUNCTION as an identifier, you must change it to another word before you can call any COBOL functions. Otherwise, the compiler gives an error message.

HP Extensions

HP extensions are features added to HP COBOL II that are not part of the ANSI standard. These features make COBOL easier to use on the MPE operating system and ease the conversion from previous versions of the ANSI standard.

You can use the HP extensions with either the ANSI74 or ANSI85 entry points unless one of the following is true:

- The HP extension if part of a ANSI COBOL'85 feature.
- The HP extension contains a ANSI COBOL'85 reserved word.
- The description of the HP extension specifically mentions that it can be used only with the ANSI85 entry point.

$CONTROL STDWARN flags features that are HP extensions. HP extensions are generally not portable to other systems. For a list of HP extensions, see the *HP COBOL II/XL Programmer's Guide*.

Portability to HP from Other Vendors

If you are transferring COBOL source programs and data to an HP computer system from another system, ANSI standard features are compatible. Any extension to accommodate vendor operating systems and file systems need to be examined for conversion efforts. Consult your HP representative for assistance and advice.

Portability between HP COBOL II/V and HP COBOL II/XL

For portability from HP COBOL II/V to HP COBOL II/XL, compiler options provide 16-bit or 32-bit alignment for synchronized items and allow passing of parameters on byte boundaries, as well as word boundaries. These features allow compatibility with 32-bit or 16-bit architectures. For more information, refer to the *HP COBOL II/XL Migration Guide*.

2

Program Structure

COBOL is similar to the English language in both structure and content. Structurally, for example, COBOL programs are made up of such familiar constructs as paragraphs, sentences, statements, and clauses. These constructs, in turn, contain such elements as words, names, verbs, and symbols. Program constructs are described in this chapter of the manual; program elements are described in Chapter 3.

Within the context of COBOL, constructs and elements all have very specific meanings. In this manual, all such terms are defined at or near the point where they are introduced. For additional convenience, their definitions appear in the glossary in Appendix E.

The concept of program modules is also described in this chapter. These modules make up a superset into which all other constructs fall; they contain almost all of the program constructs.

Structural Hierarchy

All COBOL programs are organized in a structure that consists of divisions, sections, paragraphs, sentences, statements, clauses, and phrases. This structure is hierarchical—that is, as a general rule, a COBOL program is made up of divisions; a division is made up of sections; a section is made up of paragraphs; a paragraph is made up of either sentences or clauses (depending upon the division); a sentence can contain one or more statements; a statement or clause can contain one or more phrases. The general hierarchy appears schematically in Figure 2-1. Those COBOL constructs with English language counterparts (paragraphs, sentences, clauses, and phrases) generally resemble their corresponding counterparts. From the standpoint of the compiler, each construct is treated as a logical entity within your program.

In describing the COBOL constructs, this manual begins with the highest level construct within a program, and proceeds to the lowest level.

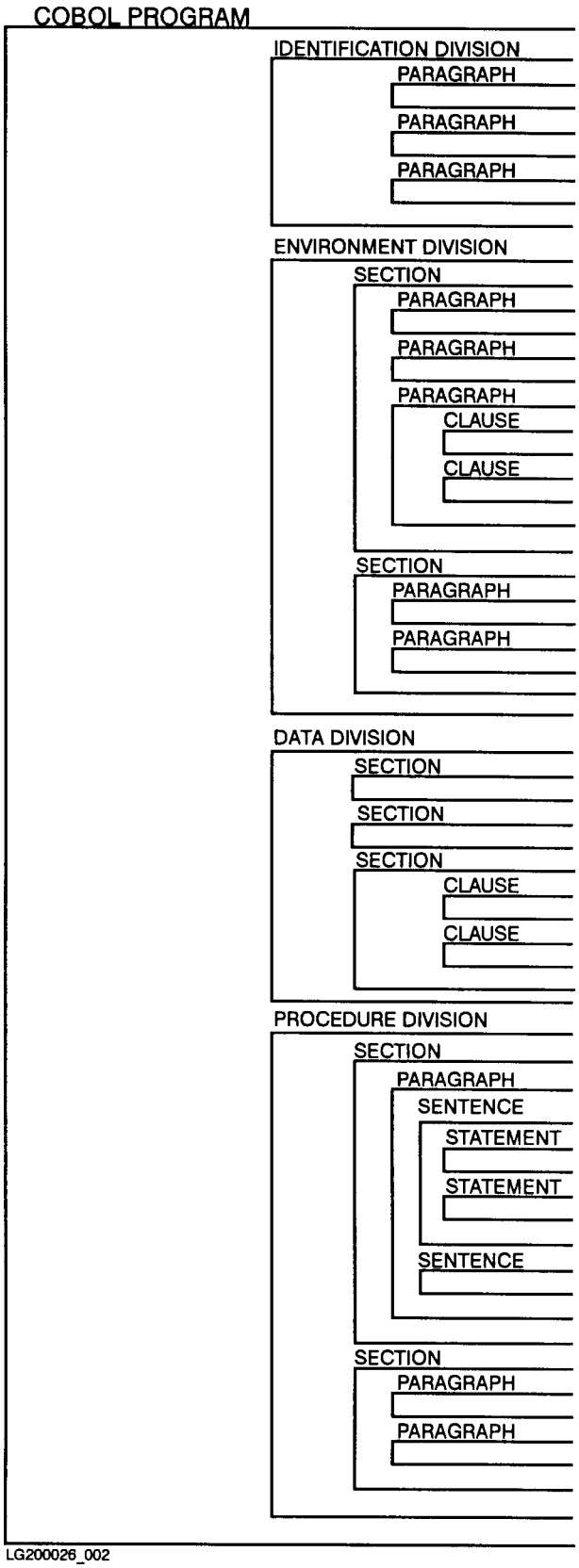


Figure 2-1. COBOL Structure Hierarchy

Figure 2-2 is an example of the COBOL program structure. The numbers indicate specific parts of the program. They are described in more detail later in this chapter as "item". For example, under the section "Division Header Format", the phrase "items 1 through 4" refers to the circled numbers 1 through 4.

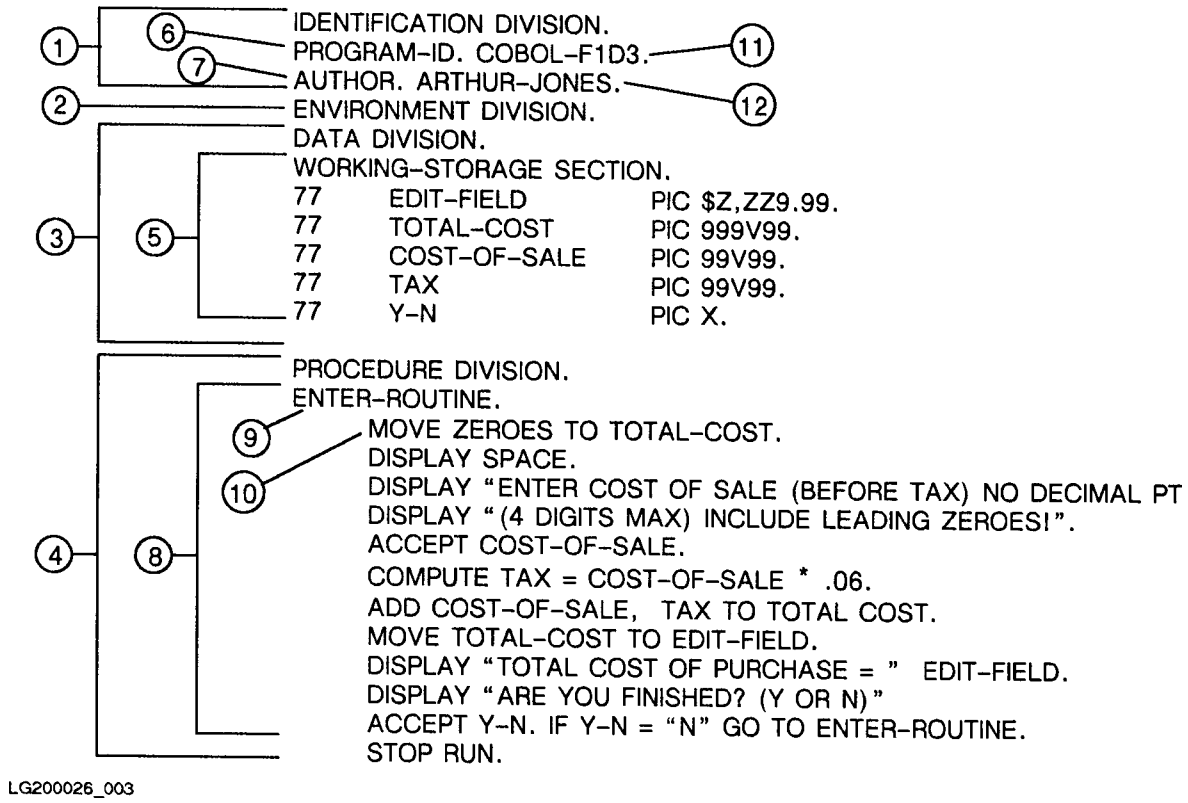


Figure 2-2. Program Structure Example

The items shown in Figure 2-2 are:

1. Identification Division.
2. Environment Division.
3. Data Division.
4. Procedure Division.
5. Working-Storage Section.
6. Program-Id Paragraph.
7. Author Paragraph.
8. A paragraph in the Procedure Division.
9. A user-defined paragraph name.
10. A sentence in the Procedure Division.
11. The program name.
12. The author's name.

Divisions

A division is the first level (highest) construct in a COBOL program. COBOL programs are partitioned into the following four divisions, which appear in the order listed:

- IDENTIFICATION DIVISION: Specifies the program name and other items used to uniquely identify the program. This division is required in every COBOL program.
- ENVIRONMENT DIVISION: Describes the computer and peripheral devices used to compile and execute the program, and the data files used by the program. `This division is optional.`
- DATA DIVISION: Describes and defines the data items referenced by the program, including their names, lengths, decimal point locations (if applicable), and storage formats. `This division is optional.`
- PROCEDURE DIVISION: Specifies the operations that the program must perform, describing how the data defined in the DATA DIVISION should be processed. `This division is optional.`

More information about the functions of these divisions appears later in this manual, where the divisions are described individually.

Division Format

Each division begins with a header entry, which is sometimes followed by one or more sections (in the ENVIRONMENT, DATA, and PROCEDURE DIVISIONs) or paragraphs (in the IDENTIFICATION DIVISION) called the division body. A division is terminated by the next division header in the program, or by the end of the program in the PROCEDURE DIVISION. The IDENTIFICATION DIVISION requires a body that specifies the name of the program.

Division Header Format

The division header consists of the division name, followed by the word DIVISION, followed by a period and a space. In the PROCEDURE DIVISION only, the optional USING phrase may also appear in the header between the word DIVISION and the period. In any COBOL program, only the following division headers are allowed:

```
IDENTIFICATION DIVISION.

ENVIRONMENT DIVISION.

DATA DIVISION.

PROCEDURE DIVISION [ USING {data-name-1...}].
```

Note Remember that the terminating periods shown in construct format descriptions must be included. Otherwise, the compiler generally misinterprets the construct.

The IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE DIVISIONs, including appropriate headers, appear in Figure 2-2 as items 1 through 4, respectively.

Sections

A section is the second level construct in a COBOL program. In the source program, sections allow you to group logically related items together within a division. In the PROCEDURE DIVISION, you can organize logically related functions into the same sections in such a way that often used routines reside in main memory for longer periods of time than routines used infrequently. This minimizes the total number of input-output operations that the operating system must perform on the code segments belonging to the program. It also facilitates program debugging. In other divisions, the features that a program uses determine which sections must be specified in the program.

Sections are optional in the PROCEDURE DIVISION. If you do not specify sections in a division, the entire division is treated as a single section. Sections are not used, however, in the IDENTIFICATION DIVISION.

Section Format

Each section begins with a header entry that is optionally followed by zero, one, or more paragraphs (in the ENVIRONMENT or PROCEDURE DIVISION) or clauses (in the DATA DIVISION). The paragraphs or clauses comprise the section body. A section is terminated by the next section header, the next division header, the END DECLARATIVES keywords (in the declarative portion of the PROCEDURE DIVISION), or the physical end of the program.

Section Header Format

In the ENVIRONMENT and DATA DIVISIONs, the section header consists of a COBOL reserved word that identifies the section, followed by the word SECTION, followed by a period and a space. In the ENVIRONMENT DIVISION, only the following section headers are permitted:

```
CONFIGURATION SECTION.

INPUT-OUTPUT SECTION.
```

In the DATA DIVISION, only the following section headers are allowed:

```
FILE SECTION.

WORKING-STORAGE SECTION.

LINKAGE SECTION.
```

In the PROCEDURE DIVISION, the section header consists of a user-defined section name that identifies the section, followed by the word "SECTION", followed by an optional segment number, followed by a period and a space. In the PROCEDURE DIVISION, unlike the ENVIRONMENT and DATA DIVISIONs, names are not restricted to specific words, so you can supply any section names you desire. Here are some examples of section headers that might be used in the PROCEDURE DIVISION:

```
INITIALIZATION SECTION.

HOUSEKEEPING SECTION 3.
```

In the second example above, the number 3 represents the segment number. This number is used in program segmentation (partitioning of a program into distinct code segments.) Segmentation is described in Chapter 8.

For clarity, programmers usually write a section header on a line by itself, although you are not formally required to do so.

In Figure 2-2, the WORKING-STORAGE SECTION (item 5) appears in the DATA DIVISION. Because no section is specified in the PROCEDURE DIVISION, the whole division is regarded as a section by the compiler.

Paragraphs

A paragraph is the third level construct in a COBOL program. Paragraphs allow you to break your program into even more elementary units. One paragraph (the PROGRAM-ID paragraph) is required in the IDENTIFICATION DIVISION. Paragraphs are optional in the ENVIRONMENT and PROCEDURE DIVISIONs. They are not used in the DATA DIVISION.

Paragraph Format

In the IDENTIFICATION and ENVIRONMENT DIVISIONs, each paragraph begins with a header entry, optionally followed by one or more words or clauses that comprise the paragraph body. In the PROCEDURE DIVISION, a paragraph begins with a paragraph name, optionally followed by one or more sentences comprising the paragraph body. In any division, a paragraph is terminated by one of the following:

- The next paragraph header or name.
- The next section or division header.
- The physical end of the program in the PROCEDURE DIVISION.
- The words END DECLARATIVES in the PROCEDURE DIVISION.
- `The words END PROGRAM in the PROCEDURE DIVISION.`

Paragraph Header and Name Format

The paragraph header, used in the IDENTIFICATION and ENVIRONMENT DIVISIONs, consists of a COBOL reserved word identifying the paragraph, followed by a period and a space. In the IDENTIFICATION DIVISION, only the following headers are permitted:

```
PROGRAM-ID.

AUTHOR.

INSTALLATION.

DATE-WRITTEN.

DATE-COMPILED.

SECURITY.
```

`REMARKS`. (This is an HP extension to the ANSI COBOL standard.)

In the ENVIRONMENT DIVISION, only these headers are allowed:

```
SOURCE-COMPUTER.

OBJECT-COMPUTER.

SPECIAL-NAMES.

FILE-CONTROL.

I-O-CONTROL.
```

The paragraph name, used in the PROCEDURE DIVISION, is a user-defined word that identifies the paragraph and is always terminated by a period and a space. It must be unique within a section or in a program if no sections are defined. If sections are used, however, the same paragraph name may appear in different sections. When referencing such a paragraph, you can use the section name to qualify the paragraph name, and you must do so if you are referencing it from within a section other than the section in which it is defined.

The paragraph header or name must be the first item on a coding line, but may be followed by other items on the same line.

In Figure 2-2, the IDENTIFICATION DIVISION contains paragraphs identified by the headers PROGRAM-ID (item 6) and AUTHOR (item 7). The PROCEDURE DIVISION includes one paragraph (item 8) identified by the user-defined name `ENTER-ROUTINE` (item 9).

Sentences, Statements, and Clauses

Within a paragraph, sentences (in the PROCEDURE DIVISION) and entries (in the IDENTIFICATION, ENVIRONMENT, and DATA DIVISIONs) may appear. Within a sentence, in turn, one or more statements can be written. These items provide a further syntactic breakdown of your program. In structure, they closely resemble their English language counterparts.

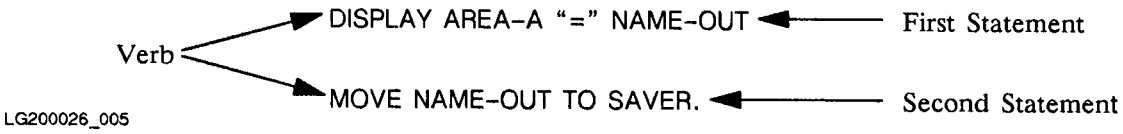
A *sentence* is a sequence of one or more statements, with the last statement terminated by a period followed by a space.

A *statement* is a syntactically valid combination of words and symbols, beginning with a verb such as ADD, READ, or DISPLAY.

An *entry* is any descriptive set of consecutive clauses terminated by a separator period and written in the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, or DATA DIVISION of a COBOL program.

All of these items are explained further in the descriptions of the divisions in which they appear.

In Figure 2-2, the PROCEDURE DIVISION begins with a sentence that contains a single statement: `MOVE ZEROS TO TOTAL COST.` (item 10). Another example is the sentence shown below, which contains two statements:



A *clause* is an ordered set of character strings (sequences of characters) that specify an attribute of an entry in the program.

In Figure 2-2, the IDENTIFICATION DIVISION contains the clauses `COBOL-F1D3` (item 11) and `ARTHUR-JONES` (item 12).

Statements and entries may also include phrases and clauses. A *phrase* is a sequence of one or more consecutive character strings that form a portion of a statement or clause. In the example below, the characters `RED-DATA OF COLOR-DATA` form a phrase:

```
MOVE RED-DATA OF COLOR-DATA TO FORM-DATA.
```

3

Program Elements

All COBOL language constructs are made up of basic elements comprised of character strings and separators. A *character string* is a character or sequence of characters that forms a COBOL word, literal, PICTURE character string, or comment entry (as defined later in this chapter). Every character string is enclosed by a separator that is either a single special character (such as a period, comma, semicolon, or blank) or a sequence of special characters. The characters that you can use in character strings and separators are selected from the COBOL character set, described in this chapter under "Character Set."

Character Strings

Character strings may form:

- Words.
- Literals.
- PICTURE character strings.
- Comment entries.

Words

In COBOL, a *word* is generally the name of some entity such as a function, paragraph, register, section, data item, constant, or other syntactical term used in a format description. There are four types:

- Reserved words.
- User-defined words.
- System names.
- Function-Names.

Each word is limited to a maximum length of 30 characters. Certain types of words, such as user-defined words, may be restricted to a shorter length.

Reserved Words

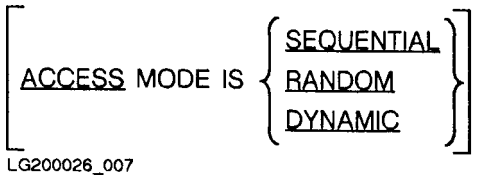
A COBOL *reserved word* is a word that has a predefined meaning (see Appendix F for a complete list) that is always consistent within all COBOL programs. Thus, reserved words are always interpreted in the same way by the compiler. For instance, the reserved word SECTION always denotes a COBOL section header. As a programmer, you may not define your own reserved words. You can use the following types of reserved words in your program:

- Keywords.
- Optional words.
- Special register words.
- Figurative constant words.
- Special character words.

Keywords. A *keyword* is a reserved word that is required in a statement or clause. You must enter the keywords where they are used. In the division header below, the words PROCEDURE, DIVISION, and USING are keywords. (In the format descriptions in this manual, all keywords are denoted by underlined upper case letters.) The USING clause is optional, as indicated by the brackets. But if you use this clause in your program, you must include the keyword USING.

PROCEDURE DIVISION [USING {*data-name-1*} ...].

Optional Words. An *optional word* is a reserved word that you can include in or omit from a statement or clause. It has no effect on program execution and serves only to make source program listings more readable. In the following clause, the words MODE and IS are optional. (In the format descriptions, all *optional* reserved words are denoted by upper case letters that are not underlined.)



Special Register Words. A special register is a storage area in main memory that contains information primarily used in connection with specific COBOL features. The content of this area is generated automatically by the compiler. In a COBOL program, such an area is referenced by a *special register word*. Those special register words that are part of ANSI COBOL are indicated in Table 3-1.

Table 3-1. Special Register Words

Word	Contents
LINAGE-COUNTER	An unsigned number used to keep track of the number of lines written to each page of a printed report. It is generated for each output file whose description in the DATA DIVISION contains a LINAGE clause (which defines the number of lines permitted per page). The register is initialized to zero and then updated each time a line is written with the WRITE statement in the PROCEDURE DIVISION. When this value exceeds the number specified by *integer-1* or *data-name-1* in the LINAGE clause, the program skips to the next page and resets the register to one.
DEBUG-ITEM	A data item used in support of the COBOL DEBUG facility. The compiler automatically generates one DEBUG-ITEM register for each program.

Table 3-2 lists the HP COBOL II special register words that are an HP extension to the ANSI COBOL standard.

Table 3-2. Extensions to Special Register Words

Word	Contents
TALLY	A 5-digit unsigned integer typically used to store information produced by the EXAMINE statement in the PROCEDURE DIVISION. (This statement counts the occurrences of a particular character within a data item and optionally replaces all instances of that character with another character.) This register may also be used as a data name for an unsigned numeric value with no decimal positions, for instance, as a subscript.
CURRENT-DATE	An 8-digit alphanumeric item used only as the sending field in a MOVE or DISPLAY statement in the PROCEDURE DIVISION. These statements send data to another field or to an output device, respectively. This item is always stored in this format: `mm/dd/yy` Here, *mm* indicates the month, *dd* indicates the day of the month, and *yy* indicates the last two digits of the year. The slash marks are automatically included in the data; you need not insert them. The date and time are obtained from the software clock. The date and time obtained by the COBOL function CURRENT-DATE are obtained from the hardware clock. See Chapter 10 for more information about the COBOL function CURRENT-DATE.
WHEN-COMPILED	An 18-character alphanumeric item that represents the date and time that the program is compiled. It may be used only in MOVE and DISPLAY statements of the PROCEDURE DIVISION. This field is automatically stored as follows, with slash marks inserted: `mm/dd/yy␣␣hh:mm:ss` As in the CURRENT-DATE format, *mm* means current month, *dd* means day of month, and *yy* indicates the year. The hh:mm:ss means hours, minutes and seconds, as in TIME-OF-DAY. The date and time are obtained from the software clock. The date and time obtained by the COBOL function WHEN-COMPILED are obtained from the hardware clock. See Chapter 10 for more information about the COBOL function WHEN-COMPILED.

Table 3-2. Extensions to Special Register Words (continued)

Word	Contents
RETURN-CODE	A predefined numeric data name in the PROCEDURE DIVISION of a subprogram, RETURN-CODE is used to pass a value back to the calling program. For complete information, see "GIVING Phrase (COBOL Subprograms)" in Chapter 11.

TIME-OF-DAY	A six-character numeric item accessed only as the transmitting field of a MOVE or DISPLAY statement in the PROCEDURE DIVISION to access the time of day. This data may be used to determine the clock time required to run a COBOL program; this is done by printing the contents of this register at the beginning and end of the program. Remember however, that clock time in a multiprogramming environment is not necessarily related to the central-processor time used by the program; it varies according to the current mix of active programs. The data is always stored in this format:
	
	`hhmmss`
	
	hh indicates the current hour, *mm* the current minute, and *ss* the current second, relative to midnight. The data is unedited. However, the statement DISPLAY TIME-OF-DAY results in the edited format:
	
	`hh:mm:ss`

Figurative Constants. Figurative constants are values that have been used so often that they have been assigned fixed data names within the COBOL language. For example, the figurative constant consisting of a string of zeros is the figurative constant ZERO. The values for figurative constants are generated automatically by the compiler. The figurative constants that you can use are shown in Table 3-3. Singular and plural forms of these words are identical.

Table 3-3. Figurative Constant Words

Word	Constant Value
ALL *literal*	The character string denoted by the variable *literal*. This string may be either a nonnumeric literal (as defined later in this chapter), or another figurative constant (such as ZERO). If a literal is used, it must be enclosed in quotation marks. If a figurative constant is used, the word ALL is redundant.
HIGH-VALUE HIGH-VALUES	One or more occurrences of the character with the highest possible value in the program collating sequence. The default program collating sequence is the ASCII Collating Sequence. The ASCII equivalent of this character is not used on the HP computers, but this bit configuration is equivalent to the hexadecimal character FF. (all eight bits on)
LOW-VALUE LOW-VALUES	One or more occurrences of the character with the lowest possible value in the program collating sequence. This is the nonprinting character NULL. The default program collating sequence is the ASCII Collating Sequence. (all eight bits off)
QUOTE QUOTES	One or more quotation marks. This constant is used to code the quotation mark as a literal in statements such as MOVE QUOTES. However, the word QUOTE or QUOTES cannot be used in place of an explicit quotation mark (") to delimit a nonnumeric literal. Thus, QUOTE ABD QUOTE cannot be substituted for the nonnumeric literal "ABD".
SPACE SPACES	One or more spaces.
ZERO ZEROS ZEROES	One or more occurrences of the digit zero.
`[ALL]` `symbolic-character`	User-defined figurative constants which are defined using the SYMBOLIC CHARACTERS clause of the ENVIRONMENT DIVISION.

Figurative constants are not enclosed in quotation marks or apostrophes. The number of characters for a figurative constant is determined by the size of the field to which the constant is moved or with which it is associated, as follows.

1. When the constant is associated with another data item, as in a VALUE clause or when the constant is moved to or compared with another item, the constant assumes the same length as the associated item. The string of characters represented by the constant is repeated, character by character, until the size of the resultant string equals that of the associated data item. Thus, when the constant word group ALL literal is used, the literal specified is repeated until the associated data item is filled with the value of the literal. For example, if `FIELD-A` is defined as a ten-character item, the statement:

   ```
   MOVE ALL "123" TO FIELD-A
   ```

   produces the following result:

   ```
   1231231231
   ```

2. When the constant is not associated with some other item, as when used in a DISPLAY, STRING, UNSTRING, EXAMINE, or STOP statement, the constant assumes a length of one character or the length of the literal.

A figurative constant may be referenced wherever a literal appears in a format description, except that literals restricted to numeric characters only may be replaced by the figurative constant words ZERO, ZEROS, OR ZEROES only.

Use of figurative constant words is demonstrated in Table 3-4 with MOVE and STRING statements, which transmit the value of the constants referenced to the storage areas denoted by STORE-*n*.

Table 3-4. Figurative Constants Examples

Example	Comment
`MOVE QUOTES TO STORE-1`	Suppose `STORE-1` is an area six character positions long. When this statement is executed, `STORE-1` contains: `" " " " " "`
`MOVE ALL "NEGATIVE" TO STORE-2`	Suppose `STORE-2` is twelve positions long. It contains: `NEGATIVENEGA`
`MOVE SPACES TO STORE-3`	Suppose `STORE-3` is nine positions long. It contains all spaces.
`STRING QUOTE "BETA" QUOTE DELIMITED BY SIZE INTO STORE-4`	Suppose `STORE-4` is six positions long. It contains: `"BETA"` Note that quotation marks delimit the literal value `BETA` in the program. These, however, are not transmitted to `STORE-4`. Instead, the quotation marks in `STORE-4` are supplied by the figurative constant QUOTE.

Special Character Words. A *special character* word is a reserved word, grouping of reserved words, or character that represents an arithmetic or relational operator. These words are listed in Table 3-5.

Table 3-5. Special Character Words

Arithmetic Operators	Relational Operators
+	IS [NOT] GREATER THAN IS [NOT] >
-	IS [NOT] LESS THAN IS [NOT] <
*	IS [NOT] EQUAL TO IS [NOT] =
/	IS <> (<> is an HP extension to the ANSI COBOL standard.) IS GREATER THAN OR EQUAL TO
**	IS >= IS LESS THAN OR EQUAL TO IS <=

User-Defined Words

A *user-defined word* is a word that you must supply to satisfy the format of a statement or clause. Such words act as arbitrary variables that name or identify various program items. These words include such elements as program names, section names, paragraph names, and data names. In the PROCEDURE DIVISION header below, *data-name-1* is a user-defined word. (In the format descriptions, all user-defined words are denoted by italic lower case letters.)

PROCEDURE DIVISION [USING {*data-name-1*} ...].

The following shows a PROCEDURE DIVISION header with the data names `ALPHA`, `BETA`, and `GAMMA`:

```
PROCEDURE DIVISION USING ALPHA, BETA, GAMMA.
```

User-defined data names, procedure names, and section names can contain up to 30 characters. These include letters (A through Z), digits (0 through 9) and the hyphen (-). Except for paragraph names, section names, segment numbers, and level numbers, all user-defined words must contain at least one alphabetic character (letter). However, user-defined words cannot begin or end with a hyphen, include an embedded space, or have the same spelling as any reserved word.

Note For more information on internal naming conventions, refer to "System Dependencies" in Appendix H.

In specific formats, the rules covering user-defined words may be more restrictive. Where such rules apply, they are explained in the description of the statement or clause in which the word appears.

In ANSI COBOL'85, 15 types of user-defined words are permitted. These are defined in Table 3-6. Fourteen of these word types are implemented in HP COBOL II. *routine-name* is not, but it is accepted by the compiler and treated as a comment.

Table 3-6. User-Defined Word Types

Word Type	Definition
Alphabet-name	Word that identifies (names) a specific character set or collating sequence to be used by the program. Defined in SPECIAL-NAMES paragraph of ENVIRONMENT DIVISION; used in CODE-SET clause of DATA DIVISION and in COLLATING SEQUENCE phrase of SORT and MERGE statements in PROCEDURE DIVISION. Must also be named in the PROGRAM COLLATING SEQUENCE clause of the CONFIGURATION SECTION in the ENVIRONMENT DIVISION in order to specify a collating sequence to be used throughout your program.

Condition-name	Word that identifies a specific value, or subset or range of values, within a complete set of values that a data item may assume. (This data item is called a conditional variable.) It may be defined in the DATA DIVISION, where the condition name is preceded by the level number 88 and followed by a VALUE clause. In the following example, the condition names `FIRST-CONST`, `SECOND-CONST`, and `THIRD-CONST` appear:
	`02 CONST PICTURE 99.`
	`88 FIRST-CONST VALUE IS 10.`
	`88 SECOND-CONST VALUE IS 20.`
	`88 THIRD-CONST VALUE IS 30.`
	A condition name may also appear in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION, where it is assigned to denote the status of switches or as an abbreviation for specific conditions.

Data-name	Word that identifies a data item. Defined in data description entries in the DATA DIVISION. Cannot be subscripted, indexed, or qualified unless specifically permitted by the format description in which it appears.
File-name	Word that identifies a data file. Defined in a file description entry or a sort-merge file description entry in DATA DIVISION.
Index-name	Word that identifies an index associated with a specific table, and used to select an item from that table. Used in DATA and PROCEDURE DIVISIONs.

Table 3-6. User-Defined Word Types (continued)

Word Type	Definition
Level-number	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. Level numbers 1 through 49 indicate the position in a record structure; level numbers 66, 77, and 88 identify special properties. In the example that appears in the *condition-name* description above, level numbers 02 and 88 are used. In single-digit level numbers, a leading zero may be optionally added. Used in the DATA DIVISION.
Library-name	Word that identifies a COBOL library (containing source text) used as input by the compiler during a particular compilation. Used in all divisions.
Mnemonic-name	Word equated to a name that identifies a special feature of the computer system on which the program is compiled or run. This relationship is established in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.
Paragraph-name	Word that identifies and begins a paragraph in the PROCEDURE DIVISION.
Program-name	Word that identifies a COBOL language source program. Used in the IDENTIFICATION DIVISION.
Record-name	Word that identifies a logical record in a data file. Used in a record description entry in the DATA DIVISION and WRITE statement in the PROCEDURE DIVISION.
Routine-name	Word that identifies a procedure written in a language other than COBOL. Used in PROCEDURE DIVISION in ANSI COBOL, but HP COBOL II treats it as a comment.
Section-name	Word that identifies and begins a section in the PROCEDURE DIVISION.
Segment-number	Word that classifies sections in the PROCEDURE DIVISION for purposes of program segmentation. Must be one of the numbers zero through 99; leading zeros are optional.
Text-name	Word that identifies text within a source library. Used in all divisions.

All user-defined words within the same program, except segment numbers and level numbers, can belong to only one of the following disjoint sets.

- Alphabet names.
- Condition names, data names, and record names.
- File names.
- Index names.
- Library names.
- Mnemonic names.
- Paragraph names.
- Program names.
- Section names.
- Text names.

For example, if the word `TEST-1` is used as a program name, it cannot also be used as a routine name. Furthermore, all such words must be unique within a disjoint set, either because no other user-defined word in the set is spelled and punctuated the same way or because uniqueness is ensured by qualification. In other words, a program cannot include two paragraphs both named `PAR-A` unless special qualification is made.

Note The general term *procedure-name* is often used to refer to either a section name or a paragraph name in the PROCEDURE DIVISION.

Following are examples of user-defined names:

```
END-OF-SCHOOL-AVERAGE
PAGECTR
123B
```

System Names

This is an obsolete feature of the 1985 ANSI COBOL standard.

A *system name* is a word that is used to define the operating environment in which the COBOL program is compiled or run. It permits communication between the program and this environment. There are two types of system names:

- *Computer name*, used to identify the computer on which the program is to be compiled or run. This name appears in the CONFIGURATION SECTION of the ENVIRONMENT DIVISION.
- *Language name*, used to specify the language in which the program is written. This name is used in the ENTER statement of the PROCEDURE DIVISION.

In HP COBOL II, all system names are treated as comments. They appear on source program listings but do not affect compilation or execution. Nevertheless, when present, system names can only contain letters (A through Z), digits (0 through 9), or hyphens. The first character of a system name must be alphabetic and the last character in the name cannot be a hyphen.

Function-Names

A *function-name* is a word that names a function you can call from your COBOL source program. Except for CURRENT-DATE, LENGTH, RANDOM, and WHEN-COMPILED, which are already reserved words, you can use function-names in a different context as user-defined words or system-names. See Chapter 10, "COBOL Functions," for more information about the COBOL functions.

Literals

A *literal* is a character string that defines itself, rather than representing some other value. The value of the literal is the character string composing the literal.

Literals are always constant values that cannot be changed during program execution. Because literals are self-defining, you do not define them in the DATA DIVISION. Instead, you code them directly into your program.

In COBOL, two types of literals are used, numeric and nonnumeric.

Numeric Literals

A *numeric literal* is essentially a number (numeric value) that is specified directly in a program. It is comprised of characters selected from the digits 0 through 9, the plus sign (+), the minus sign (-), and a decimal point (.). As an example, the literal `1` appears in the ADD statement below:

```
ADD 1 TO PAGE-NUMBER.
```

The specific value of the literal is the algebraic quantity represented by the characters that compose the literal. In the format descriptions, literals are indicated by the word *literal-n*. For example, *literal-1* and *literal-2*. Every numeric literal must contain:

- At least one digit.
- No more than 18 digits.
- No more than one arithmetic sign (+ or -). If a sign is used, it must appear as the leftmost character in the literal. If no sign is used, the literal is treated as a positive value by the compiler.
- No more than one decimal point. If a decimal point is used, it may appear anywhere in the literal except as the rightmost character. (Any decimal point used as the rightmost character is interpreted as a period that terminates a sentence.) If no decimal point is used, the literal is treated as an integer.

Note If a character string follows the rules for the formation of a numeric literal, but is enclosed in quotation marks, it is treated by the compiler as a nonnumeric literal. (See "Nonnumeric Literals" below.) As such, it cannot be used in arithmetic operations.

Following are examples of numeric literals:

```
1670037627          3.415
+2                  -30.06
```

Octal Literals

Octal literals are an HP extension to the ANSI COBOL standard.

You can use octal literals in your program. Octal literals are always preceded by a percent sign (%). Following are examples of octal literals:

```
%17          %37777777777          %456
```

Tip

Try using the SYMBOLIC CHARACTERS clause to define constants instead of using octal literals.

Caution

For octal literals used in VALUE clauses, it is recommended that you only use nonnumeric items or items with USAGE BINARY. For other types that are used in level 88 values, the octal literal is converted to decimal before it is used. Otherwise, no conversion is done for octal literals. When no conversion is done, the octal literal is right-justified with NULL fill.

No conversion is done for octal literals in the PROCEDURE DIVISION except in arithmetic statements. You must make sure the octal literal is a valid value for the particular way you are using it.

In IF statements, a nonnumeric compare is performed without conversion of the octal literals. The DISPLAY, EXAMINE, INSPECT, SEARCH ALL, STRING, and UNSTRING statements interpret the octal constants as nonnumeric literals.

Examples. The following program uses octal literals:

```
WORKING-STORAGE SECTION.
01  ITEM-ALPHA     PIC XX        VALUE %40502.    Octal literal is "AB".
01  ITEM-NUMERIC   PIC 99 BINARY VALUE %47.       Octal literal is 39.
01  ITEM-NUM       PIC 99        VALUE %30462.    Octal literal is ASCII 12.
     :
DISPLAY %54131.                                  Octal literal is "XY".
DISPLAY ITEM-ALPHA.                              Displays "AB".
DISPLAY ITEM-NUMERIC.                            Displays 39.
DISPLAY ITEM-NUM.                                Displays the value 12.
ADD %23 TO ITEM-NUM.                             Adds octal 23 (decimal 19).
DISPLAY ITEM-NUM.                                Displays the sum, 31.
```

When the above program runs, it displays the following:

```
XY
AB
39
12
31
```

See also the SYMBOLIC CHARACTERS clause in Chapter 6, "ENVIRONMENT DIVISION."

Nonnumeric Literals

A *nonnumeric literal* is a character string containing letters, digits, or special characters that is coded directly into a program. It is formed by entering:

- A quotation mark or apostrophe that denotes the beginning of the literal.
- The character string that comprises the literal.
- A matching quotation mark or apostrophe that delimits the end of the literal.

A character string may be from 1 to 255 characters long (an HP extension to the ANSI COBOL standard), and may consist of any of the characters from the ASCII collating sequence. (The ANSI COBOL standard allows a maximum length of 160 characters.) See Appendix D for a list of these characters.

Note The delimiting quotation marks or apostrophes are not considered part of the literal.

All punctuation marks within a nonnumeric literal are treated as ordinary punctuation marks, not as delimiters or separators.

Single and Double Quotation Marks in Nonnumeric Literals

You can use either quotation marks or apostrophes as delimiters. There is no restriction on which set of delimiters can be used at a given time. This allows you a great amount of freedom in forming nonnumeric literals. For example, to display the following message on your terminal screen:

```
PLEASE ENTER "AGE UNDETERMINED" IF UNSURE
```

You can use the following DISPLAY statement:

```
DISPLAY 'PLEASE ENTER "AGE UNDETERMINED" IF UNSURE'
```

As an example of invalid usage, the character string, `'I DON'T KNOW'`, is interpreted by the HP COBOL II compiler as being the string, `'I DON'`, followed by the characters, `T KNOW'`. In this case, a syntax error would be generated. Since you can use quotation marks and apostrophes throughout your program to delimit nonnumeric literals, the above string could be made valid by using quotation marks:

```
DISPLAY  "I DON'T KNOW"
```

Embedded Quotation Marks in Nonnumeric Literals

You can use two consecutive quotation marks, or two consecutive apostrophes, within the characters of a nonnumeric literal to represent a single quotation mark or apostrophe. For example, the DISPLAY statement above could also have been written as follows:

```
DISPLAY 'I DON''T KNOW'
```

The results of executing this statement would be the following message on your terminal screen or line printer:

```
I DON'T KNOW
```

If you use double apostrophes in a nonnumeric literal, and the literal is bounded by quotation marks, then both apostrophes are used as part of the literal. The opposite is also true. For example:

```
DISPLAY "DOUBLE APOSTROPHES, '', ARE PART OF THIS LITERAL"
```

The above statement results in the following:

```
DOUBLE APOSTROPHES, '', ARE PART OF THIS LITERAL
```

Following is another example:

```
DISPLAY 'DOUBLE QUOTES, "", ARE PART OF THIS LITERAL'
```

The above statement results in the following:

```
DOUBLE QUOTES, "", ARE PART OF THIS LITERAL
```

Notice that the figurative constant words `QUOTE` and `QUOTES` cannot be used to supply delimiting quotation marks for nonnumeric literals.

For how to continue nonnumeric literals onto a second line, see "Continuation Lines" later in this chapter.

PICTURE Character Strings

The PICTURE character string appears in the PICTURE clause of the DATA DIVISION. This clause describes the characteristics and editing requirements of data that is typically destined for some external output device such as a terminal or line printer. Specifically, the PICTURE clause determines the appearance of the field that is actually output by specifying:

- The size of the field.
- The class (type) of data that can be written into the field (alphabetic, numeric, or alphanumeric).
- The appearance of a numeric sign, if any, in the field.
- The position of the decimal point, if any, in the field.
- The editing required to insert, suppress, or replace characters in the field.

The PICTURE clause supplies currency signs, leading or trailing zeros, commas, plus or minus signs, and other punctuation stated in the PICTURE string. Often, for example, it is used to suppress leading zeros on checks, replacing them with asterisks or spaces. As an example, if the data to be printed was comprised of the digits 8765432123 and you wished to print it as a dollar value with appropriate punctuation, you could specify the following PICTURE character string in the PICTURE clause:

```
$99,999,999.99
```

In this string, the "9" digits are used as special symbols to specify the character positions that are filled with numeric data. The dollar sign, commas, and decimal point indicate the positions of the punctuation characters. The PICTURE character string is superimposed on the output data so that the following information is printed:

```
$87,654,321.23
```

Complete details about PICTURE character strings appear in Chapter 7.

Comment Entries

This is an obsolete feature of the 1985 ANSI COBOL standard.

You should use comment lines instead of comment entries. A *comment entry* is used in the IDENTIFICATION DIVISION to include comments or remarks in the program. These comments appear on the source program listing but do not affect program compilation or execution. They denote such items as program author, installation name, date written, date compiled, security requirements, and other general remarks. They may include any printable character from the ASCII Character Set.

All comment entries are optional. When included, however, they must also conform to the rules for paragraph and sentence structure described in this manual.

Comment Lines

A *comment line* is any line with an asterisk in the continuation indicator area (column 7) of the line.

A comment line can appear anywhere in your program following the IDENTIFICATION DIVISION header. It can be made up of any combination of characters from the ASCII collating sequence, with all characters (except the asterisk in column 7) contained in columns 8 through 72 of the line.

Additionally, you can use a special form of comment line to cause page ejection prior to printing the comment. This special comment line is the same as the general one described above, except you put a slash character (/) in column 7 instead of an asterisk.

Separators

A *separator* is a punctuation character that delimits a character string. Separators include:

- Spaces (one or more).
- A comma or semicolon immediately followed by a space, except when the comma is used in a PICTURE character string.
- A period that is followed by a space. The period must be used only to indicate the end of a sentence, or as shown in formats.
- Left and right parentheses. These must only appear as balanced pairs used to delimit subscripts, indices, arithmetic expressions, conditions, reference-modifiers, or a list of function arguments.
- Quotation marks or apostrophes. These delimit nonnumeric literals, and must appear as balanced pairs (except as noted under "Continuation Lines" later in this chapter). An opening quotation mark or apostrophe must be immediately preceded by a space or left parenthesis. A closing quotation mark or apostrophe must be immediately followed by a space, comma, semicolon, period, or right parenthesis.
- Sets of two contiguous equal signs (==), used to delimit pseudo-text. (Pseudo-text is text incorporated into, or replaced in, a COBOL program by the COPY or {REPLACE}} statement.) An opening delimiter must be immediately preceded by a space; a closing delimiter must be immediately followed by a space, comma, semicolon, or period. These delimiters must appear in balanced pairs.

Any of the above separators may, at your option, be immediately preceded by one or more spaces, except if specifically prohibited by format rules. (A space preceding a closing quotation mark is treated as part of the literal enclosed by this and the preceding quotation mark.)

Any of the above separators, except the opening quotation mark, may be optionally followed immediately by one or more spaces. (A space following an opening quotation mark is considered as part of the literal enclosed by this and the next following quotation mark.)

Note The above rules do not apply to punctuation characters within nonnumeric literals, comment entries, comment lines, or PICTURE character strings. Those characters are not regarded as separators.

Character Set

Most character strings and all delimiters in a COBOL program are formed from characters selected from the COBOL character set. This character set includes upper case letters A through Z, lower case letters a through z, digits 0 through 9, and certain special characters:

Special Character	Meaning	
+	Plus sign	
-	Minus sign	
*	Asterisk	
/	Slash	
=	Equal sign	
$	Currency sign	
,	Comma	
;	Semicolon	
:	Colon	
.	Period (decimal point)	
”	Quotation mark	
’	Apostrophe	This is an HP extension to the ANSI COBOL standard.
(	Left parenthesis	
)	Right parenthesis	
>	Greater than	
<	Less than	
	Space	
%	Percent sign	This is an HP extension to the ANSI COBOL standard.
@	At sign	This is an HP extension to the ANSI COBOL standard.
\	Back slash	This is an HP extension to the ANSI COBOL standard.

In the case of nonnumeric literals, comment entries, and comment lines, additional characters may be used. These characters are selected from the ASCII Character Set listed in Appendix D, (which includes the COBOL Character Set).

Note When lower case letters appear outside of literals, the HP COBOL II compiler automatically converts them to upper case letters.

Each character in the ASCII Character Set has a unique value that establishes its order in the collating sequence of this character set. In Appendix D, the characters are listed in order of ascending value.

Note When COBOL programs originally written for other systems are run on an HP Computer System, variations between the collating sequences for both systems may cause variations in output. For example, the Binary Coded Decimal (BCD) and Extended Binary Coded Decimal Information Interchange Code (EBCDIC) both collate the letters of the alphabet before (lower than) the digits 0 through 9. This differs from the ASCII Collating Sequence, where digits are collated before letters. In addition, several special characters collate differently in various sequences. To permit valid processing when a different collating sequence is used, you may specify the appropriate sequence in the ALPHABET and PROGRAM COLLATING SEQUENCE clauses of the ENVIRONMENT DIVISION.

Coding Rules

The following paragraphs state the standards that you should follow when coding COBOL source programs. These coding rules are known in COBOL as the *reference format*.

Sequence Number (Columns 1 through 6)

Sequence numbers appear in columns 1 through 6 of the source record. They identify the order of the record with respect to other records in your program. When you enter a program through EDIT/3000, the sequence numbers are supplied automatically by that subsystem. When you write the program on a coding form, however, these numbers are optional. When you choose to use them, you enter them in columns 1 through 6 of the coding sheet. Any character may be used in the sequence numbers, though sequential numbers are recommended.

When you compile your program, you may request the compiler to use the sequence numbers to check the sequence of the source statements in the program. You may also request the compiler to renumber these statements. You select these options with the $EDIT command (refer to Appendix B for descriptions of the $EDIT options).

Sequence numbers are also useful if you must recompile the program. They allow you to merge new text with the originally compiled text stored on disk, according to sequence number. Thus, you need to enter only additional or changed statements for this compilation.

Note If you intend to use this feature, increment the sequence numbers by 10 or 100 to allow space for possible new statements.

Program Text (Columns 8 through 72)

Program text appears in columns 8 through 72. This group of columns is divided into Areas A and B. The coding sheet provides both column and area headings. All Area A and Area B coding conventions presented throughout this manual represent the ANSI standard specifications for COBOL. In many instances the HP COBOL II compiler allows variations from these standards. In order to enhance the readability of source programs, and ensure compatibility with standard ANSI COBOL, you are encouraged to follow the rules presented here.

In Area A (columns 8 through 11), you begin all division headings, section headers, paragraph headers, paragraph names, level indicators FD and SD, and level numbers 01 and 77. These entries may, where necessary, be continued into Area B.

In Area B (columns 12 through 72), you enter all other COBOL text. For example, the following elements must appear in Area B: all sentences and procedural statements; data description entries (including their names), whether or not associated with level indicators and numbers.

Continuation Lines

Any sentence or entry that requires more than one line, must be continued in Area B of the next line. Your program can contain any number of continuation lines.

When a word or numeric literal is broken from one line to the next, you must enter a hyphen (-) in column 7 of the continuation line. This hyphen indicates that the first nonspace character in Area B is part of the word or literal broken on the previous line.

When a nonnumeric literal is broken from one line to the next, you again must place a hyphen in column 7, and you must enter a quotation mark or apostrophe before the continuation of the literal. In any case, the continuation of the word or literal can begin anywhere within Area B of the continuation line. All spaces at the end of the continued line are considered part of the end of the word or literal. In any continuation line, Area A must contain spaces only.

Examples

The following example shows a continuation line where the data item ITEM-NUMBER-FOUR is continued onto the second line:

```
     ADD ITEM-NUMBER-ONE, ITEM-NUMBER-TWO, ITEM-NUMBER-THREE TO IT
-      EM-NUMBER-FOUR.                                           ↑
↑      ↑                                                  Column 72
Columns 7 and 14.
```

All characters of the first line up to column 72 are considered part of the line. On the second line, the continuation hyphen is in column 7. The continuation text begins in column 14.

The following example shows a nonnumeric literal continued onto a second line. The literal is:

```
"Name 1A Number 1A Name 2A Number 2A Name 3A Number 3A Name 4A Number 4A"
```

```
     MOVE "Name 1A Number 1A Name 2A Number 2A Name 3A Number 3A N
-    "ame 4A Number 4A" TO RECORD-ITEM.                          ↑
↑    ↑                                                  Column 72
Columns 7 and 12.
```

All characters of the first line up to column 72 are considered part of the literal. Notice that there is no closing quotation mark on the first line. Once again, the continuation hyphen is in column 7, and the continuation text begins in column 12. The continuation of the literal must start and end with quotation marks or apostrophes.

The following example shows that the continued line can start anywhere in Area B. It also shows apostrophes instead of quotation marks. This example is equivalent to the previous example:

```
     MOVE 'Name 1A Number 1A Name 2A Number 2A Name 3A Number 3A N
-         'ame 4A Number 4A' TO RECORD-ITEM.                     ↑
↑         ↑                                              Column 72
Column 7. Continuation text starts in column 17.
```

The following example shows that all characters of the first line are used in the nonumeric literal:

```
                                  Columns 68 through 72 are blank
                                  and considered part of the literal.
                                                          ↓   ↓
       MOVE "Name 1A Number 1A Name 2A Number 2A Name 3A Number
-      "3A Name 4A Number 4A" TO RECORD-ITEM.                    ↑
↑      ↑                                                  Column 72.
Columns 7 and 12.
```

When the above MOVE statement is executed, the following literal is moved to RECORD-ITEM:

```
"Name 1A Number 1A Name 2A Number 2A Name 3A Number     3A Name 4A Number 4A"
```

There are five spaces between "`Number`" and "`3A`" in the literal, which correspond to the five spaces in columns 68 through 72 of the first line. This literal contains a total of 75 characters.

Debugging Lines

A *debugging* line is any line with a "D" in column 7. Refer to Chapter 13, for more information on rules and use of debugging lines.

Identification Code (Columns 73-80)

An optional identification code may appear in columns 73 through 80. This feature can be used to identify different versions of a program. It also serves as the library name for source statements placed in a COBOL copy library. All statements in such a library require an identification code.

4

Describing and Referencing Data

The data used by a COBOL program is defined and described in the DATA DIVISION, and referenced and operated upon in the PROCEDURE DIVISION. This data is stored in, read from, and written to files (collections of information) that reside on various peripheral devices. For instance, a payroll processing program might accept input from a file that contains wage and salary information for all employees on the company payroll; this program might also write new output to this same file during updating operations.

Within a file, all information is organized into units of related data called logical records. These records are similar in form, purpose, and content. For example, in a payroll file, each logical record could contain the wage and salary data related to a particular employee. In other words, there would be one record for each employee.

Within each record, individual elements of data, or groups of such elements, are called data items. As an example, a payroll record for an employee might contain the following data items: the employee's name, social security number, marital status, gross pay, tax exemptions, individual deductions, and net pay. The individual deductions data item might itself contain subordinate data items, such as federal income tax, state income tax, insurance premiums, bond payments, and charity contributions.

Files

In COBOL, a *file* is a collection of records that is identified by a unique name and is currently recognizable by your program. The name allows you to reference the file in your program. Other specifications define how records are organized within the file with respect to the physical device on which the file is stored. The specifications for the file are defined in the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION and the FILE SECTION of the DATA DIVISION.

Records

Each logical record constitutes a group of related information, uniquely identifiable and treated as a unit. A record is actually the most inclusive data item in a file. Each input-output statement in the PROCEDURE DIVISION accesses one logical record, although it can also extract subordinate data items from that record.

Logical Versus Physical Records

A physical record is one or more logical records and is commonly called a block. A block is the physical unit used by the operating system to read data from a file, or write data to it; it is the basic unit transferred between the device on which the file resides and main memory each time a program executes an input or output operation.

You can use the BLOCK CONTAINS clause to specify the size of (that is, the number of logical records contained in) a physical record. For files on magnetic tape or disk, a block consists of either one logical record or a group of several logical records. For instance, 2, 16, or 256 logical records could be grouped into one block. For tape files, blocking is normally done to improve execution time or to conserve file space by reducing the number of gaps on the tape. For files on card readers and punches, line printers, and terminals, each block is identical to each logical record, and its length is determined by the type of device. Thus, each block/logical record read from a card reader consists of one 80-character punched card; each block/logical record written to a line printer consists of one line of print; typically 132 characters. The size of a block has no relation to the size of the data file contained on the device to or from which the block is transferred.

A single storage device can hold one or more logical records.

Note

In this manual, the term record refers to logical records unless the term block or physical record is specifically used.

COBOL allows you to define logical records in main memory as well as in files stored on peripheral devices. This definition is done through the WORKING-STORAGE SECTION of the DATA DIVISION (refer to Chapter 7).

Record Descriptions

Each record in a file is defined by a record description entry in the DATA DIVISION. This entry, in turn, consists of one or more data description entries that collectively define the characteristics of the record. Each data description entry consists of the following elements in the order listed:

- *Level number* that indicates a subdivision or portion of the logical record.
- *Data name* that allows you to identify and reference the data item.
- *Independent* clauses that describe the attributes of the data item.

To reference portions of the information in a logical record, you must subdivide the record into corresponding data items. You must also identify each data item that you wish to reference with a name. Once you specify any data item, you can further subdivide it into subordinate data items to permit more detailed data reference. You can also reference data using `reference modification` (described later in this chapter). The level number indicates the hierarchical order of a data item within the record structure. Figure 4-1 contains some examples. Since a record is the most inclusive data item your program can reference, it is assigned the level number 01. Less inclusive data items are assigned numerically higher level numbers, ranging from 02 through 49. These numbers need not be successive.

The most basic subdivisions of a record (those data items that have no further subdivisions) are called *elementary items*. Items with subdivisions are called *group items*, or simply *groups*.

Within the record description entry, each group includes all following group and elementary items until an item with a level number greater than or equal to the level number of that group is encountered.

A record is considered a single elementary item if it is not subdivided; otherwise, it is regarded as a sequence of elementary items that may or may not be organized into groups. Because of the hierarchical structure of the record, a basic element can belong to its immediate group and higher level groups that contain that group. In the PROCEDURE DIVISION, your program can refer to the entire record, to any group of any level within that record, or to an elementary item.

In Figure 4-1, a record named PERSONNEL-RECORD (line 11) is defined in the DATA DIVISION. This record is divided into the various group items:

- Two main group items, named EMPLOYEE-ID (line 12) and ADDRESS (line 15).
- The EMPLOYEE-ID group item is subdivided into two elementary items: EMPLOYEE-NUMBER (line 13) and SOCIAL-SECURITY-NUMBER (line 14). The ADDRESS group item is subdivided into three items: STREET (line 16), LOCATION (line 17), and ZIP (line 20).
- The LOCATION group item is further subdivided into two elementary items: CITY (line 18) and STATE (line 19).

In this example, the following data items are all elementary items: EMPLOYEE-NUMBER, SOCIAL-SECURITY-NUMBER, STREET, CITY, STATE, and ZIP. If your program accesses the group item ADDRESS, it implicitly accesses STREET, LOCATION, CITY, STATE, and ZIP.

Notice that the level numbers used in this example are not successive, and that the descriptions of all elementary items include PICTURE clauses. The first entry in this example begins with the word FD, which is a level indicator that indicates the entire file; this entry is a file description entry, which must always precede any group of record description entries in the FILE SECTION. File description entries are described completely in Chapter 7.

```
           ⋮
0010  FD PAYROLL-FILE.
0011  01 PERSONNEL-RECORD.
0012      03 EMPLOYEE-ID.
0013          05 EMPLOYEE-NUMBER                    PIC 9(5).
0014          05 SOCIAL-SECURITY-NUMBER             PIC 9(9).
0015      03 ADDRESS.
0016          05 STREET                             PIC X(20).
0017          05 LOCATION.
0018              07 CITY                           PIC X(20).
0019              07 STATE                          PIC X(20).
0020          05 ZIP                                PIC 9(5).
                  ⋮
```

Figure 4-1. Record Desctiption Entry

Level 66, 77, and 88 Items

Programs can contain special level numbers that do not actually apply to hierarchical levels. Instead, they indicate special properties of entries in the DATA DIVISION. These level numbers are described below:

Level Number	Purpose
66	specifies group or elementary items introduced by a RENAMES clause. This clause permits the regrouping of data.
77	specifies noncontiguous data items that are not subdivisions of other items and are not themselves subdivided. These items are defined in the WORKING-STORAGE SECTION and typically reference internal counters and accumulators.
88	specifies condition names associated with particular values of a conditional variable.

Refer to the description of the DATA DIVISION in Chapter 7 for specific rules on coding the above entries.

Data Items - Classes and Categories

Data items in a COBOL program are specified and referenced very precisely. The various restrictions governing data items are outlined below.

Classes of Data Items

COBOL has three general classes of data items:

- **Alphabetic**, which can contain letters (A through Z) or spaces, in any combination.
- **Numeric**, which can contain digits (0 through 9) in any combination, optionally including an operational sign. This is the only class of data item that can be used in arithmetic operations.
- **Alphanumeric**, which can contain any characters from the ASCII character set, in any combination.

Note For complete compatibility with all ANSI COBOL compilers, use only members of the COBOL character set.

Categories of Data Items

The three classes of data items are subdivided into five categories:

- **Alphabetic**, which is synonymous with the alphabetic class.
- **Numeric**, which is synonymous with the numeric class.
- **Alphanumeric**, which can contain any characters from the ASCII character set in any combination, not edited by a PICTURE clause.
- **Alphanumeric-edited**, which can contain any characters from the ASCII character set in any combination, plus editing symbols supplied by a PICTURE clause.
- **Numeric-edited**, which can contain any digits (0 through 9), plus editing symbols supplied by a PICTURE clause.

Note More precise definitions of these categories appear under the description of the PICTURE clause in Chapter 7.

These classes and categories are independent of the external or internal storage formats of the data items. The relation of classes to categories are summarized in Table 4-1. For alphabetic and numeric data items, the classes and categories are synonymous. For alphanumeric data items, the relation of class to category depends on the level (group or elementary) of the data item within the record structure. Every elementary item (except an index data item) belongs to one of these classes and categories. During program execution, every group item is treated as an alphanumeric item regardless of the class of the elementary items subordinate to that group item.

Table 4-1. Data Item Classes and Categories

Level of Item	Class	Category
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Numeric Edited Alphanumeric Edited Alphanumeric
Group	Alphanumeric	Alphabetic Numeric Numeric Edited Alphanumeric Edited Alphanumeric

Algebraic Signs

Numeric data can have two types of algebraic signs: operational signs and editing signs.

Operational Signs

These signs are associated with signed numeric data items and signed numeric literals, to indicate their algebraic properties. In regular (default) internal format, they are represented in storage as noted in the description of the USAGE clause of the DATA DIVISION (see Chapter 7). However, you can optionally override this format by explicitly specifying the location of the signs with the SIGN clause of the DATA DIVISION.

Note Using the SIGN clause to force operation signs into a representation different from the regular (default) format typically causes the compiler to create less efficient object code.

Editing Signs

These signs typically appear on edited reports, and are used to denote the positive or negative value of data. They are inserted into the data through sign/control symbols in the PICTURE clause.

Data Alignment

In a COBOL program, data is moved from one area of storage (sending data item) to another (receiving data item) through use of the MOVE, ACCEPT, STRING, UNSTRING, arithmetic, or other statements in the PROCEDURE DIVISION. When aligning data within the receiving item, the compiler follows specific rules. These rules depend upon the category of that item, as noted below. Table 4-2 contains some examples.

- For data items in the numeric category, the compiler aligns the data by the decimal point and places it in the receiving data item. The compiler also truncates excess characters on either end of the sending data item, and fills unused positions in the receiving data item with zeros.

Note

The decimal point is never actually stored in a numeric data item; instead, the compiler defines and keeps track of an assumed decimal point that appears in the data item only when it is read or written or output. Any stored data item that contains a combination of digits and editing characters such as the decimal point, comma, and so forth, does not belong to the numeric category and cannot be used in arithmetic operations except as a receiving field.

When your program does not explicitly specify a decimal point for a numeric data item, the compiler defines an assumed decimal point immediately after the rightmost digit and aligns the item as described above.

- For data items in the numeric-edited category, the compiler aligns the data by decimal point with zero-fill or truncation at either end (as with numeric data items), except where editing replaces leading zeros with another character.

Note

The decimal point in a numeric-edited data item, unlike that in a numeric data item, is actually stored in the item.

- For data items in the alphabetic, alphanumeric, and alphanumeric-edited categories, the compiler aligns the data at the leftmost character position. It also truncates excess characters to the right of the sending item and fills unused positions to the right of the receiving item with spaces.

Note

If your program specifies the JUSTIFIED clause for the receiving data item in the DATA DIVISION, the above rules are modified as directed by that clause.

Level-01 (record) and level-77 data items are always aligned on word boundaries unless they are in the LINKAGE SECTION. Refer to "System Dependencies" in Appendix H for more information. In Table 4-2, a space character is represented by the symbol ␣, and an assumed decimal point position is represented by the symbol ∧. The receiving data items for alphabetic and alphanumeric data items are each 11 positions long. The receiving data item for the alphanumeric-edited data item is six positions long. The receiving data items for numeric and numeric-edited data items may each contain up to 18 digits. The specifications for the PICTURE format are explained in Chapter 7.

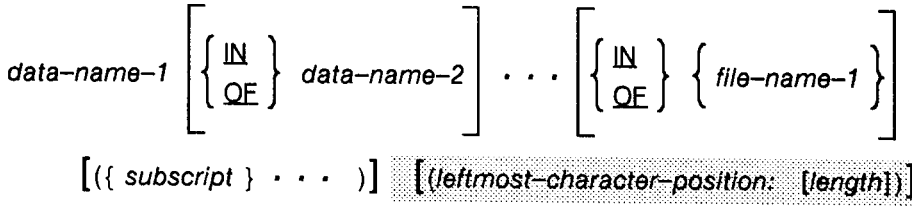
Table 4-2. Data Alignment

Category	Data to be Stored	Receiving Item before Transfer	Receiving Field after Transfer	
			PICTURE Clause	Content
Alphabetic	ABC	PQRSTUVWXYZ	A(11)	ABC␣␣␣␣␣␣␣␣
	ABCDEFGHIJK	PQRSTUVWXYZ	A(11)	ABCDEFGHIJK␣
	ABCDEFGHIJKLMN	PQRSTUVWXYZ	A(11)	ABCDEFGHIJK
Numeric	1∧2	987654	9(3)V9(3)	001∧200
	123∧456	987654	9(3)V9(3)	123∧456
	12345∧67890	987654	9(3)V9(3)	345∧678
Numeric Edited	1.2	987.654	9(3).9(3)	001.200
	123.456	987.654	9(3).9(3)	123.456
	12345.67890	987.654	9(3).9(3)	345.678
Alphanumeric Edited	ABCDE	ZZZ/ZZZ	XXX/XXX	ABC/DE␣
Alphanumeric	A2C	PQRSTUVW123	X(11)	A2C␣␣␣␣␣␣␣␣␣
	A2C?E!G*I5K	PQRSTUVW123	X(11)	A2C?E!G*I5K␣
	A2C?E!G*I5K@M7	PQRSTUVW123	X(11)	A2C?E!G*I5K

Identifiers

Data names that are uniquely identified by qualifiers, subscripts, indexes, or `reference modifiers` are known collectively as *identifiers*. Function-identifiers are also *identifiers*.

The syntax for identifiers is summarized in the general format:



LG200026_008a

The syntax for a function-identifier is:

```
FUNCTION function-name-1 [({parameter-1} ... )] [reference-modifier]
```

Function-identifiers are described later in this chapter.

The following are restrictions on the use of identifiers:

- Where subscripting is prohibited, indexing is also prohibited.
- You can alter the value of an index only by using the SET, SEARCH, and PERFORM statements in the PROCEDURE DIVISION.
- You can store values referenced by index names into data names that are described by the USAGE IS INDEX clause of the DATA DIVISION. These data items are *index data items*.

Uniqueness of Reference

To ensure that the basic elements defined in the various program divisions can be properly referenced in the PROCEDURE DIVISION, the compiler places several restrictions on some of these elements. These restrictions cover qualifiers, subscripts, indexes, and identifiers as they apply to data names, condition names, paragraph names, and text names.

Qualifiers

Each data name, condition name, paragraph name, and text name must be unique within the program in which it appears. Such a name is unique if either of the following conditions applies:

- No other name in the program has the same spelling and hyphenation.
- If the same name is used for two different elements in a program, it must be made unique through qualification. For instance, if two paragraphs are both identified by the name `PAR-MSG`, they must be members of different sections, called perhaps `SEC-1` and `SEC-2`. In the PROCEDURE DIVISION, you might then reference one of these paragraphs by specifying:

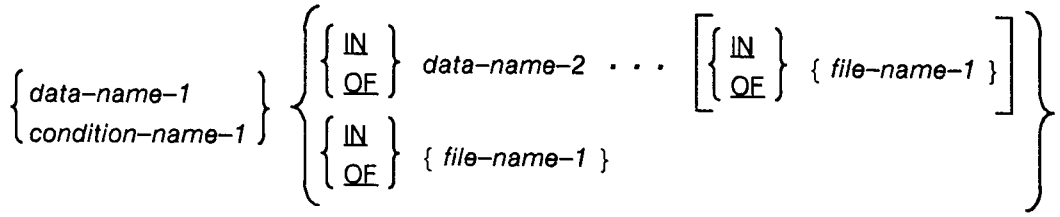
```
PAR-MSG OF SEC-1.
```

In a hierarchy of names such as this, the higher-level names are called qualifiers. Specifically, a qualifier is one of the following:

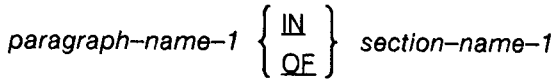
- A data name used in a reference together with another data name or with a condition name at a lower level in the same hierarchy.
- A section name used in a reference together with a paragraph name specified in that section.
- A library name used in a reference together with a text name associated with that library.

In a program, you qualify a data name, condition name, paragraph name, or text name by entering one or more phrases composed of the following: the name to be qualified, followed by the reserved word IN or OF, followed by a qualifier. Four formats are possible, depending on the type of name:

Format 1

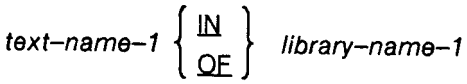


Format 2

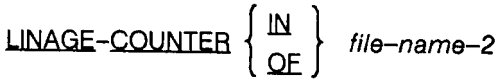


LG200026_009a

Format 3



Format 4



LG200026_010

Note In these format descriptions, the keywords OF and IN are logically equivalent and can be used interchangeably.

In all cases, you must use sufficient qualifiers to make the name unique. It is not always necessary, however, to mention all levels in a particular hierarchy.

Within the DATA DIVISION, all data names used as qualifiers must be associated with a level indicator or level number. Thus, you cannot specify two identical data names as subordinate entries in a group item unless you can make them unique through qualification. In the qualification hierarchy, names associated with a level indicator are the most significant; names associated with level number 01 are the next most significant; names associated with level numbers 02 through 49 are then ranked in descending order of significance.

In the PROCEDURE DIVISION, two identical paragraph names must not appear in the same section. A section name is the highest and only qualifier for a paragraph name.

Note The most significant name in any hierarchy must be unique and cannot be qualified.

Subscripted or indexed data names and conditional variables can be made unique through qualification. The name of a conditional variable can be used as a qualifier for any of its condition names. Regardless of the available qualification, no name can be both a data name and a procedure name.

In using qualification, the following specific rules apply:

For all names:

- You must make sure that each qualifier belongs to a successively higher level and falls within the same hierarchy as the name it qualifies.
- You cannot use the same name at two or more levels in the same hierarchy.
- You can use any combination of qualifiers to reference the name if more than one combination of qualifiers ensures uniqueness.
- You can qualify a name even if it does not require qualification.

For data names and condition names:

- If you assign a data name or condition name to more than one data item, you must qualify this name each time it is referenced in the ENVIRONMENT, DATA, and PROCEDURE divisions.

Note This rule does not apply to the REDEFINES clause of the DATA DIVISION, where qualification is prohibited.

For data names:

- You cannot subscript a data name that is used as a qualifier.
- You cannot specify, as the complete set of qualifiers for one data name, a partial set of qualifiers used for another data name.
- You must qualify data names if more than one file contains a LINAGE clause.

For paragraph names:

- You cannot duplicate a paragraph name within a section.
- You cannot use the reserved word SECTION in the qualification of a paragraph when the paragraph name is qualified by a section name.
- You need not qualify a paragraph name that is referenced within the same section.

For text names:

- You must qualify all text names each time they are referenced in the program if more than one COBOL library is used during compilation.

Example

The following example illustrates qualification. The name `DATA-GRAY` is duplicated in the program, where it actually refers to two different data items. In each case, the name can be qualified as `DATA-GRAY OF DATA-BLACK` or `DATA-GRAY OF DATA-GREEN`.

```
DATA DIVISION.
       ⋮
01 RECORD-1.
   03 DATA-BLACK.
      05 DATA-GRAY.
         10 DATA-BLUE           PIC X(06).
         10 DATA-YELLOW         PIC X(06).
      05 DATA-BROWN             PIC X(12).
   03 DATA-WHITE.
      05 DATA-GREEN.
         10 DATA-GRAY           PIC X(06).
             ⋮
```

Tables

Quite frequently in business applications, data is arranged in the form of tables. This is because of the logical arrangement of data and because it is easier to both describe and select elements of a table than it is to write all the components of the table as one record.

Tables composed of contiguous data items are defined in COBOL by using the OCCURS clause in a record description entry. The OCCURS clause states how many elements there are in a table, gives these elements a common name, tells whether the elements are arranged in ascending or descending order, and whether to use subscripting or indexing to access elements of the table.

You must use subscript or index names to access table elements because they share the same name.

Defining a Table

In HP COBOL II, you can define a table containing up to `seven dimensions`. This is accomplished by using the OCCURS clause once for each dimension within different level numbers (other than 01) of the description. In ANSI COBOL'74, the maximum number of dimensions is three.

To define a one dimensional table, use an OCCURS clause as part of the data description for the table itself. If you do not use the OCCURS clause as part of the first level description following the table name, the elements described before the OCCURS clause are not part of the table. For example, in the program fragment below, `TABLE1-HEADER` is not a table element, whereas `ELEMENT`, `TIME-ADJUSTER`, and `DATE-ADJUSTER` are table elements.

```
01 TABLE-1.
   02 TABLE1-HEADER                PIC X(20) VALUE "TABLE ONE".
   02 ELEMENT OCCURS 100 TIMES.
      03 TIME-ADJUSTER             PIC X(10).
      03 DATE-ADJUSTER             PIC X(10).
```

To define a two dimensional table, you must build it from a one dimensional table. That is, to define a two dimensional table, you must use an OCCURS clause twice, once in an element of the first table, and a second time in the description of a group item containing that table element. For example:

```
01 SHOW-TABLE.
   11 FIRST-DIM OCCURS 10 TIMES.
      22 DIM1-HEAD               PIC X(20)
      22 SECOND-DIM OCCURS 10 TIMES.
         25 TWODIM-ELEMENTS.
            30 MORE-ONE          PIC X(10).
            30 MORE-TWO          PIC X(07).
            30 MORE-THREE        PIC X(16).
```

The table element, `SECOND-DIM`, of the first table dimension (`FIRST-DIM`) uses the OCCURS clause to define a second dimension of `SHOW-TABLE`, while the group item `FIRST-DIM` defines the first dimension.

Defining a three dimensional table is analogous to defining a two dimensional table. Simply extend the table elements of the two dimensional table to include an element that uses the OCCURS clause. For example:

```
01 SALES-ORGANIZATION-TABLE.
   11 REGION-TABLE  OCCURS 4 TIMES.
      22 SALES-REGION                PIC X(05).
      22 STATE-TABLE OCCURS 13 TIMES.
         33 STATE                    PIC X(20).
         33 REP-TABLE OCCURS 4 TIMES.
            34 REP-INFO.
               35 REPRESENTATIVE     PIC X(20).
               35 LOCATION-INFO      PIC X(60).
```

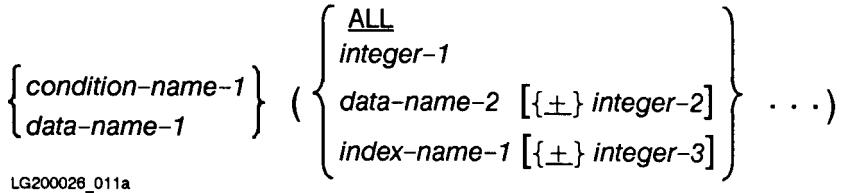
The table above, named SALES-ORGANIZATION-TABLE, has one dimension for REGION-TABLE, a second dimension for STATE-TABLE, and a third dimension for REP-TABLE.

SALES-ORGANIZATION-TABLE contains 472 data items. There are four data items for REGION-TABLE, 52 data items for STATE-TABLE (4 times 13), and 416 data items for REP-TABLE (52 times 4, twice, one for each element name).

Referencing Table Items with Subscripting

Elements in a table of like elements can be uniquely referenced through subscripts. A subscript is an integer that corresponds to a specific element in the table. You can only use subscripts for elements that have not been assigned individual data names. The lowest possible subscript value is 1, which identifies the first element in the table. The next ascending values (2, 3, 4 ...) point to the second, third, and fourth elements, and so forth. The highest possible value is specified by the OCCURS clause of the DATA DIVISION.

The table is identified by a table element data name or referenced by a condition name. Individual elements in the table are identified by one, two, three, or up to seven subscripts. The subscript, or set of subscripts, is enclosed in a pair of balanced parentheses following any qualification for the table element data name. (Any data name written in this format is called a *subscripted data name*.) When two or more subscripts are used, they are written in order of successively decreasing inclusiveness. Subscripted data names have the following format:



LG200026_011a

The subscript can be represented by the reserved word ALL, a numeric literal, a data name, or an index name.

The subscript ALL can be used only when the subscripted identifier is a parameter to a COBOL function. ALL cannot be used with *condition-name-1*. ALL specifies that each table element associated with that subscript position is a parameter to the function. See Chapter 10, "COBOL Functions", for more information on calling COBOL functions.

Literals and data names must represent an integer, optionally preceded by a plus (+) sign. If a data name is used, it must specify an elementary item; the data name can be qualified but cannot itself be subscripted. The plus or minus sign must be preceded and followed by a space.

An index is a special register containing a binary value that corresponds to an occurrence number of an element of the table to which it is associated. The implementation of an index is machine dependent for efficiency.

The index is defined for the table and assigned an index name through the INDEXED BY phrase in the table definition in the DATA DIVISION. In the PROCEDURE DIVISION, you use the index name to reference the index. The index name must correspond to a data description entry in the hierarchy of the table being referenced that contains an INDEXED BY phrase specifying that index name. Before you can use the index as a table reference, however, you must assign the index an initial value. You do this by using the SET, SEARCH ALL, or PERFORM statement.

```
Indices, data-names, integers, and the word ALL can be combined to
reference a multidimensional table.
```

Two subscripting techniques are available:

- *Direct subscripting*, where the element desired is specified by the contents of the subscript.
- *Relative subscripting*, where the element desired is specified by the contents of the subscript plus or minus a specific value.

Direct subscripting is specified by using a subscript after the table element data name. For example:

```
DATA-1 (INDEX-A)
```

Relative subscripting is specified by using the subscript, followed by a plus or minus sign, followed by an unsigned integer specified as a numeric literal, all enclosed in balanced parentheses and following the table element data name.

The compiler determines the sequential location of the element in the table by incrementing (for the plus sign) or decrementing (for the minus sign) the value in the index or data item by the value of the literal.

The following illustrates direct subscripted data items.

```
DAY (1)

DATE (1 2)

BENCHMARK (ALPHA BETA GAMMA)

CANDY (20  FILEZ-01  +3)
```

The following illustrates relative subscripting.

```
TABLE-1 (BETA + 1)

TABLE-2 (THETA - 1  MU)

ARRAY-3 (STORE-1 + 1  STORE-2 + 1  STORE-3 + 1)
```

In the first of the above examples, if `TABLE-1` is the table name and the value of `BETA` is 15, the program accesses the 16th element. The second example demonstrates that both direct and relative subscription are permitted in the same subscripted data name.

Subscripting uses an occurrence number (that is, the number of where in a particular dimension an element occurs) for each dimension of a table. To illustrate, using the three dimensional table `SALES-ORGANIZATION-TABLE` described under "Defining a Table", the following refers to the third occurrence of `REPRESENTATIVE` in the first state of the fourth sales region:

```
REPRESENTATIVE (4, 1, 3)
```

Thus, if the fourth sales region is the western sales region, and the first state in that region is California, then the name of the third representative for that state is accessed by the above subscripted reference.

Of course, if you wish only to access a state entry, you can do so by using only two subscripts. For example, the following references the 12th state in the fourth sales region:

```
STATE(4, 12)
```

Note that data names could as easily have been used to perform all or just part of the subscripting above. Generally, these data items are defined in working storage, and have no restrictions on them except that they cannot be index data names.

Referencing Table Items with Indexing

Indexing requires more coding in the OCCURS clause, since you must specify at least one name to be used for the indexing. To illustrate, the SALES-ORGANIZATION-TABLE has been modified for indexing:

```
01 SALES-ORGANIZATION-TABLE.
   11 REGION-TABLE OCCURS 4 TIMES.
      22 SALES-REGION                    PIC X(05).
      22 STATE-TABLE OCCURS 13 TIMES.
         33 STATE                        PIC X(20).
         33 REP-TABLE OCCURS 4 TIMES
                        INDEXED BY RPINDX.
            34 REP-INFO.
               35 REPRESENTATIVE         PIC X(20).
               35 LOCATION-INFO          PIC X(60).
```

Once the index names have been defined, you must use the SET statement of the PROCEDURE DIVISION to initialize the index names to a value within the range of from 1 to the highest occurrence number associated with the dimension in which the index name was defined. See "OCCURS Clause" in Chapter 7 for more details.

Once an index name has been set, you can use it to access table elements. Assuming that RPINDX has been set to 2, the following example accesses the information about the second representative in the first state of the fourth sales region:

```
REP-INFO(4, 1, RPINDX)
```

Referring to the use of this same data base in the subscript example above, note that this accesses the information about the second sales representative in California.

You can use index names in conjunction with the SEARCH statement of the PROCEDURE DIVISION to search for occurrences of table items within a given table. For information and restrictions on searching tables, refer to "SEARCH Statement" in Chapter 9.

Condition Names

Condition names made unique through qualification, indexing, or subscripting have the same overall syntax for identifiers as the two formats above. In these formats, however, the user-defined word *condition-name-1* replaces *data-name-1*.

The restrictions that apply to the combined use of qualification, subscripting, and indexing of identifiers also apply to condition names. In addition, these further restrictions apply:

1. If a condition name is made unique through qualification, you can use either the hierarchy of names associated with the related conditional variable or the conditional variable itself as the first qualifier.
2. If references to a conditional variable require indexing or subscripting, you must use the same indexing or subscripting for references to any condition name associated with that variable.

Note In the format descriptions appearing throughout this manual, *condition-name* refers to a condition name that can be qualified, indexed, or subscripted as necessary.

Function-Identifiers

A function-identifier is a syntactically correct combination of character strings and separators that references a function.

Syntax

The format of a function-identifier is:

```
FUNCTION function-name-1 [({parameter-1} ... )] [reference-modifier]
```

Parameters

function-name-1	Any of the COBOL functions listed in Chapter 10, "COBOL Functions."
parameter-1	Must be an identifier, a literal, or an arithmetic expression. Specific rules for parameters are listed with each function in Chapter 10.
reference-modifier	A reference-modifier can only be used with alphanumeric functions. See "Reference Modification" later in this chapter for more information.

Description

You can use a function-identifier anywhere an identifier of the same class and category is allowed except where it is specifically disallowed. However, a function-identifier cannot be used as a receiving item in any statement. An integer or numeric function identifier can only be used in an arithmetic expression.

Evaluation of Function Parameters. When you call a function, its parameters are evaluated individually from left to right. A parameter can itself be a function-identifier or an expression containing a function-identifier.

For more information on the COBOL functions, see Chapter 10, "COBOL Functions."

Reference Modification

`Reference modification` is a feature of the 1985 ANSI COBOL standard.

Reference modification is a method of referencing data by specifying a leftmost character and length for the data item.

Syntax

The format of a *reference modifier* is:

(*leftmost-character-position*: [*length*])

You use a *reference modifier* with a data item or a function identifier. The general format for reference modification is:

data-name-1 (*leftmost-character-position*: [*length*])

The format for reference modification with COBOL functions is:

FUNCTION *function-name-1* [({*parameter-1*} ...)] (*leftmost-character-position*: [*length*])

Parameters

data-name-1	must reference a data item whose usage is DISPLAY. *data-name-1* can be qualified or subscripted.
leftmost-character-position	must be an arithmetic expression.
length	must be an arithmetic expression.
function-name-1	must be an alphanumeric COBOL function. For a list of COBOL functions, see Chapter 10, "COBOL Functions."
parameter-1	is any parameter to the function *function-name-1*.

Reference modification is allowed wherever an identifier referencing an alphanumeric data item or alphanumeric function is permitted, except in *identifier-3* of the STRING statement and *identifier-1* of the UNSTRING statement.

Reference Modification Rules

The following rules apply when using reference modification:

- Each character of a data item referenced by *data-name-1* or by *function-name-1* and its parameters, if any, is assigned an ordinal number according to its position. The leftmost position is assigned the number one and the number of each successive position to the right is incremented by one. If the data description entry for *data-name-1* contains a SIGN IS SEPARATE clause, the sign position is assigned an ordinal number within that data item.
- If *data-name-1* or *function-name-1* is numeric, numeric-edited, alphabetic, or alphanumeric-edited, reference modification operates upon the item as if it were redefined as an alphanumeric data item of the same size.
- For an operand, reference modification is evaluated as follows:
  - If you specify subscripting for an operand, the reference modification is evaluated immediately after the subscripts. If you specify an ALL subscript for an operand, the reference-modifier is applied to each element of the table. (Using ALL as a subscript is only allowed when the operand is a parameter to a function.)
  - If subscripting is not specified for the operand, the reference modification is evaluated at the time subscripting would be evaluated if subscripts had been specified.
  - If you specify reference modification for a function reference, the reference modification is done immediately after the function is evaluated.
- Reference modification creates a unique data item that is a subset of the data item referenced by *data-name-1* or *function-name-1* and its arguments, if any.

  This unique data item is defined as follows:
  - The *leftmost-character-position* specifies the ordinal position of the leftmost character of the unique data item in relation to the leftmost character of *data-name-1* or *function-name-1*. Evaluation of the *leftmost-character-position* must result in a positive nonzero integer less than or equal to the number of characters in the data item.
  - The *length* specifies the size of the data item to be used in the operation. The result must be a positive nonzero integer. The sum of *leftmost-character-position* and *length* minus 1 must be less than or equal to the number of characters in the data item *data-name-1* or *function-name-1*.
  - If *length* is not specified, the unique data item extends from the *leftmost-character-position* through the rightmost character of the data item.
- The unique data item is considered an elementary data item without the JUSTIFIED clause. When a function is referenced, it has the class and category of alphanumeric. When *data-name-1* is specified, the unique data item has the same class and category as *data-name-1* except that the categories numeric, numeric-edited, and alphanumeric-edited are considered class and category alphanumeric.

Examples

Based upon the following example, Table 4-3 shows the result of reference modification upon four statements:

```
01 TAB.
   05  ELEMENT  PIC  X(5)
               OCCURS 5 TIMES VALUE "12345".
01 X    PIC  X(3).
```

Table 4-3. Reference Modification Results

Statement	Result
`MOVE "ABC" TO ELEMENT (3) (2:)`	Changes the third element of `ELEMENT` table to `1ABC␣`.
`MOVE "ABC" TO ELEMENT (2) (4:)`	Changes the second element of `ELEMENT` table to `123AB`.
`MOVE "ABC" TO ELEMENT (1) (1:4)`	Changes the first element to `ABC␣5`.
`MOVE ELEMENT (5) (2:2) TO X.`	Changes `X` to `23␣`.

Based on the following example, Table 4-4 shows the result of reference modification without subscripting upon three statements:

```
01 Y     PIC XXXX VALUE SPACES.
```

Table 4-4. Reference Modification Without Subscripting

Statement	Result
`MOVE "AB" TO Y(2:)`	`␣AB␣`
`MOVE "XYZ" TO Y(2:)`	`␣XYZ`
`MOVE "F" TO Y(2:1)`	`␣FYZ`

The following program shows reference modification on function calls:

```
10   $CONTROL POST85
11    IDENTIFICATION DIVISION.
12    PROGRAM-ID.  FUNC-EXAMPLE.
13    ENVIRONMENT DIVISION.
14    DATA DIVISION.
15    WORKING-STORAGE SECTION.
16    01  TAB.
17        05  ELEMENT        PIC X(5) USAGE DISPLAY
18                             OCCURS 5 TIMES VALUE "12345".
19    PROCEDURE DIVISION.
20    FIRST-PARA.
21       DISPLAY FUNCTION WHEN-COMPILED
22       DISPLAY FUNCTION WHEN-COMPILED (9:2)
23       DISPLAY FUNCTION CURRENT-DATE
24       DISPLAY FUNCTION CURRENT-DATE (1:4)
25       MOVE "93843" TO ELEMENT (2)
26       MOVE "38103" TO ELEMENT (3)
27       MOVE "49382" TO ELEMENT (4)
28       MOVE "78397" TO ELEMENT (5)
29       DISPLAY FUNCTION MAX ( ELEMENT (ALL) )
30       DISPLAY FUNCTION MAX ( ELEMENT (ALL) ) (2:2)
31       STOP RUN.
```

The above program displays the following kind of output:

```
1991011612272700-0500    Output from WHEN-COMPILED, line 21.
12                       Output from reference-modified WHEN-COMPILED, line 22.
1991011612283000-0500    Output from CURRENT-DATE, line 23.
1991                     Output from reference-modified CURRENT-DATE, line 24.
93843                    Output from MAX, line 29.
38                       Output from reference-modified MAX, line 30.
```

5

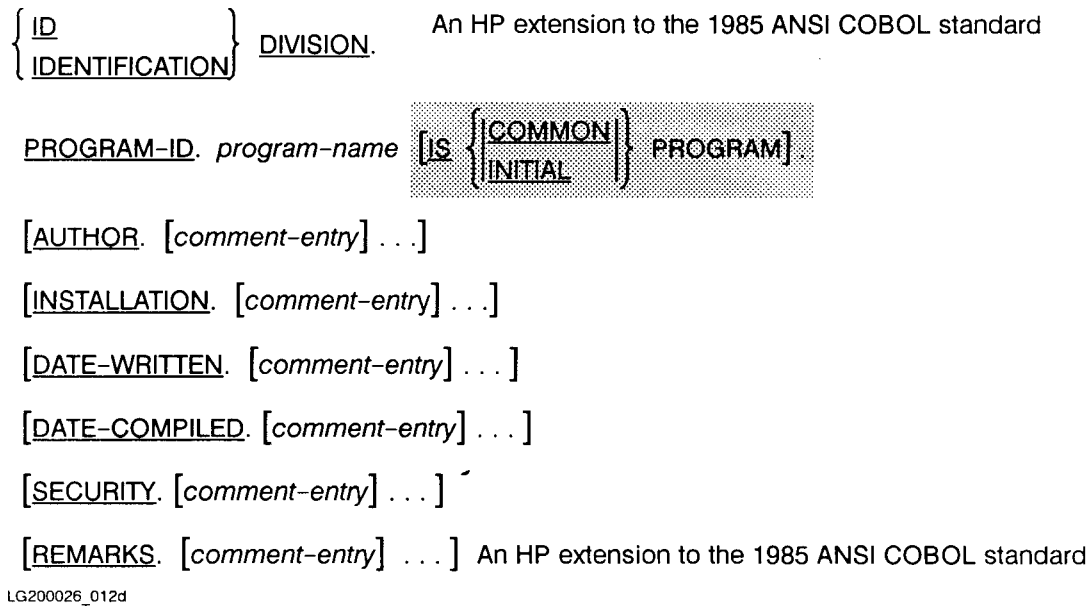
IDENTIFICATION DIVISION

Every HP COBOL II program begins with the IDENTIFICATION DIVISION. This division specifies information that identifies both the source program and related listings produced by the HP COBOL II compiler. Among this information, you must always include the name of your program. In addition, you may optionally identify:

- The author of the program.
- The installation where the program is compiled.
- The date that the program is written.
- The date that the program is compiled.
- Any security restrictions governing the program.

IDENTIFICATION DIVISION Format

The IDENTIFICATION DIVISION has the following format:



In this format, the paragraph headers identify the kind of information that each paragraph contains. Thus, in the PROGRAM-ID paragraph, you specify the name of your program; in the AUTHOR paragraph, you generally enter your own name.

The IDENTIFICATION DIVISION can not be abbreviated to ID DIVISION in the contained programs within a nested program. Only the outermost containing program can abbreviate IDENTIFICATION to ID.

IDENTIFICATION DIVISION Syntax Rules

The PROGRAM-ID paragraph is always required, but all other paragraphs are optional. When any optional paragraphs are included, they must always appear in the order shown in the format description.

Begin the division header in Area A of the first line. Begin each paragraph header in Area A of a new line. In each paragraph, begin the paragraph body (*program-name* or *comment-entry*) either on the same line as the paragraph header or in Area B of a new line following the header. When you must continue a lengthy entry, begin the continuation in Area B of the next available line. The PROGRAM-ID paragraph must be terminated by a period followed by a space.

Paragraphs

The IDENTIFICATION DIVISION contains the following paragraphs: PROGRAM-ID, AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, and SECURITY. A description of these paragraphs follow. All paragraphs in the IDENTIFICATION DIVISION except the PROGRAM-ID paragraph are obsolete features of the 1985 ANSI COBOL standard.

PROGRAM-ID Paragraph

This paragraph must appear in every program and must include the program's name. This name identifies your source program and appears on the listings associated with it. It must be a unique name with respect to all program units (HP COBOL II main program or subroutines) compiled in a particular instance. The name must begin with a letter and cannot contain more than 30 characters, including hyphens.

```
The INITIAL clause specifies that when the program is called, that program
and any other programs it contains are in their initial state.  The initial
state of a program is the state of the program the first time it is called
as a run unit.  Using the INITIAL clause sets $CONTROL DYNAMIC and is only
useful for subprograms.
```

For a description of the `COMMON` clause, see Chapter 11, "Interprogram Communication."

The program names of a run unit do not need to be unique. When two program names in a run unit are the same, at least one of the two programs must be directly or indirectly contained within another separately compiled program that does not contain the other of the two programs.

Example

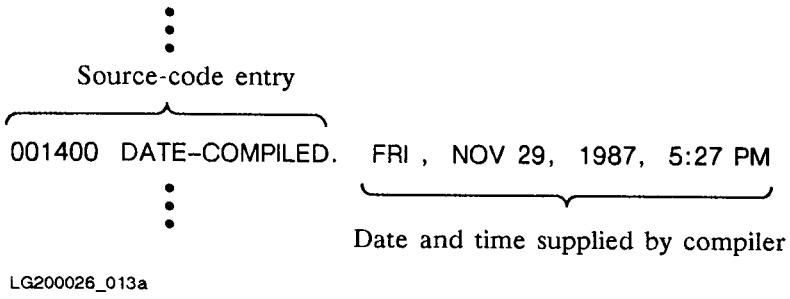
The following example shows parts of several programs. Only the IDENTIFICATION DIVISION, PROGRAM-ID, and END PROGRAM are shown for each program. For clarity, the lines are appropriately indented. However, in practice, IDENTIFICATION DIVISION, PROGRAM-ID, and END PROGRAM must start in margin A.

Program Structure	Scope
`IDENTIFICATION DIVISION.`	
`PROGRAM-ID. A.`	*Callable by separately compiled program (G below).*
⋮	
`IDENTIFICATION DIVISION.`	
`PROGRAM-ID. B.`	*Callable by A.*
⋮	
`END PROGRAM B.`	
`IDENTIFICATION DIVISION.`	
`PROGRAM-ID. C COMMON.`	*Callable by B, A, D and E.*
⋮	
`END PROGRAM C.`	
`IDENTIFICATION DIVISION.`	
`PROGRAM-ID. D.`	*Callable by A.*
⋮	
`IDENTIFICATION DIVISION.`	
`PROGRAM-ID. E.`	*Callable by D.*
⋮	
`END PROGRAM E.`	
`END PROGRAM D.`	
`END PROGRAM A.`	
`IDENTIFICATION DIVISION.`	
`PROGRAM-ID. F.`	*Callable by separately compiled programs (A above or G below).*
⋮	
`IDENTIFICATION DIVISION.`	
`PROGRAM-ID. A.`	*Callable by F. Note that a call to A from F will call the nested program A, not the separately compiled program A (above).*
⋮	
`END PROGRAM A.`	
`IDENTIFICATION DIVISION.`	
`PROGRAM-ID. B.`	*Callable by F.*
⋮	
`END PROGRAM B.`	
`END PROGRAM F.`	
`IDENTIFICATION DIVISION.`	
`PROGRAM-ID. G.`	
⋮	
`END PROGRAM G.`	

DATE-COMPILED Paragraph

The DATE-COMPILED paragraph is an obsolete feature of the 1985 ANSI COBOL standard.

When you enter the DATE-COMPILED paragraph in your source program, the compiler prints the current date and time on the first line of this paragraph as it appears on the source program listing. Generally, you include only the paragraph header (and do not specify a body) when you enter the source program. On the source listing, the date and time appear in the following format:



No additional data, either on the same line as the paragraph header or on subsequent lines, is printed on the source listing.

Other Paragraphs

The remaining paragraphs of the IDENTIFICATION DIVISION, AUTHOR, INSTALLATION, DATE-WRITTEN, and SECURITY, are obsolete features of the 1985 ANSI COBOL standard.

In the other paragraphs of the IDENTIFICATION DIVISION, which are optional, the paragraph bodies are treated as comments. Thus, you may enter any information you wish in any of them. For example, you may use the AUTHOR paragraph for data other than someone's name.

To continue any of these comment entries onto two or more lines, simply enter the desired information in Area B of the necessary lines. However, in this case, do not enter the hyphen continuation indicator in column 7.

The following illustrates a complete IDENTIFICATION DIVISION, showing all required and optional paragraphs:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  EXAMPLE-ID-SECTION.
AUTHOR. MANUAL-J-WRITER.
INSTALLATION. GSD-CUST-TRAINING-AND-DOC-GROUP.
DATE-WRITTEN. 07/11/86.
DATE-COMPILED.
SECURITY. NONE.
```

6

ENVIRONMENT DIVISION

The ENVIRONMENT DIVISION allows you to define those aspects of a data processing application that depend on the physical characteristics of the processing environment. `The ENVIRONMENT DIVISION is optional.` It always follows the IDENTIFICATION DIVISION.

This division consists of two sections:

- CONFIGURATION SECTION, for specifying the hardware characteristics of the system on which you will compile and run your program.
- INPUT-OUTPUT SECTION, for specifying the data files used by your program, and various input-output control elements.

Each of these sections may contain several paragraphs.

ENVIRONMENT DIVISION Format

The ENVIRONMENT DIVISION has the following general format:

GENERAL FORMAT FOR ENVIRONMENT DIVISION

```
[ENVIRONMENT DIVISION.

[CONFIGURATION SECTION.

[SOURCE-COMPUTER.  [source-computer-entry]]

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

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

[INPUT-OUTPUT  SECTION.

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

[I-O-CONTROL.  [input-output-control-entry]]]]
```

LG200026_014

ENVIRONMENT DIVISION Syntax Rules

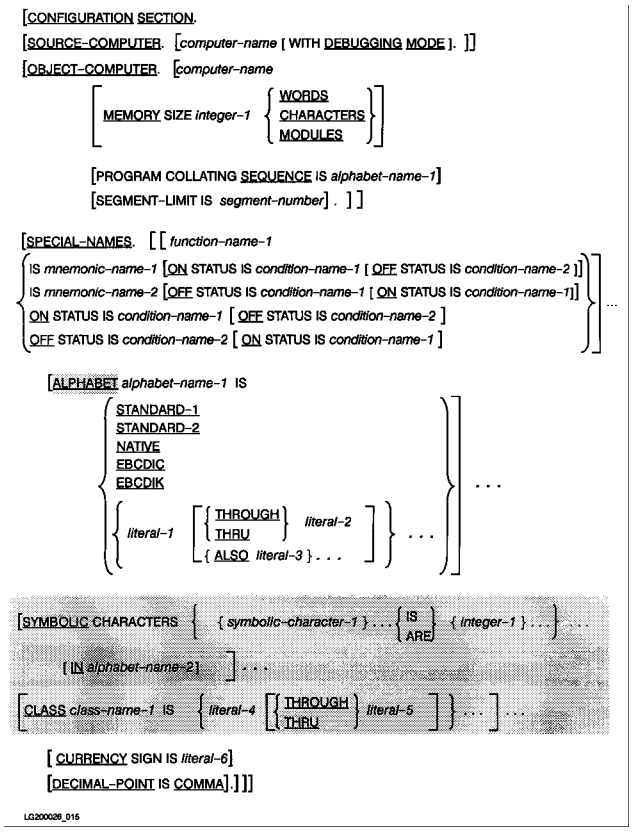
The CONFIGURATION and INPUT-OUTPUT SECTIONs are optional. In nested programs, the CONFIGURATION SECTION is not allowed, because the information in the containing program is global across the contained programs. In separate programs, or in concatenated programs when included, they must appear in the order shown in the format description.
`Within each section, all paragraphs are optional.`

CONFIGURATION SECTION

The CONFIGURATION SECTION may include the following paragraphs:

- SOURCE-COMPUTER paragraph, for defining the characteristics of the computer system on which your HP COBOL II source program will be compiled.
- OBJECT-COMPUTER paragraph, for specifying the characteristics of the computer system on which the resulting object program will be run.
- SPECIAL-NAMES paragraph, for specifying symbolic characters and the relationship of special function names to mnemonic names appearing in the source program.

The CONFIGURATION SECTION format is shown on the next page, and each paragraph is described on the following pages.


[CONFIGURATION SECTION.
[SOURCE-COMPUTER. [computer-name [WITH DEBUGGING MODE].]]
[OBJECT-COMPUTER. [computer-name
[MEMORY SIZE integer-1 { WORDS / CHARACTERS / MODULES }]
[PROGRAM COLLATING SEQUENCE IS alphabet-name-1]
[SEGMENT-LIMIT IS segment-number] .]]
[SPECIAL-NAMES. [[function-name-1
{ IS mnemonic-name-1 [ON STATUS IS condition-name-1 [OFF STATUS IS condition-name-2]]
IS mnemonic-name-2 [OFF STATUS IS condition-name-1 [ON STATUS IS condition-name-1]]
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 / EBCDIK
{ literal-1 [{ THROUGH / THRU } literal-2] } ... } ...
[SYMBOLIC CHARACTERS { { symbolic-character-1 } ... { IS / ARE } { integer-1 } ... } ...
[IN alphabet-name-2]] ...
[CLASS class-name-1 IS { literal-4 [{ THROUGH / THRU } literal-5] } ...] ...
[CURRENCY SIGN IS literal-6]
[DECIMAL-POINT IS COMMA].]]]
LG200026_015

SOURCE-COMPUTER Paragraph

The SOURCE-COMPUTER paragraph denotes the computer system on which you plan to compile your source program, and whether or not to set the debugging mode switch on at compile time.

Syntax

This paragraph has the following format:

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

where *computer-name* is any valid user-defined COBOL word. That is, any combination of alphanumeric characters and hyphens you choose, with the restriction that the first character must be alphabetic and that there must be no blanks between the first and last characters.

HP COBOL II assumes that all programs are compiled on an HP computer system. If you specify a *computer-name* in the SOURCE-COMPUTER paragraph, the compiler treats this name as a comment.

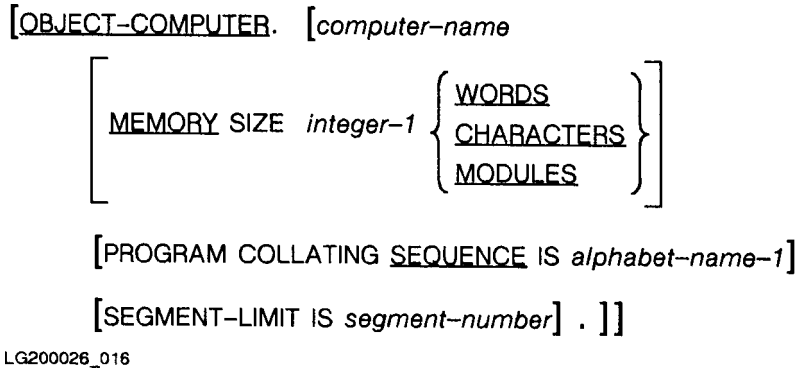
For further details on the WITH DEBUGGING MODE clause, refer to Chapter 13.

OBJECT-COMPUTER Paragraph

The OBJECT-COMPUTER paragraph denotes the computer system on which the object program is executed. HP COBOL II assumes that all COBOL programs are executed on an HP computer system.

The only clause in the OBJECT-COMPUTER paragraph that is not treated as a comment is the PROGRAM COLLATING SEQUENCE clause.

Syntax



LG200026_016

Parameters

computer-name	any combination of alphanumeric characters and hyphens you choose, with the restriction that the first must be alphabetic, and that there must be no blanks between the first and the last characters in the name.
integer-1	any positive integer.
alphabet-name-1	any name you choose, with the same rules and restrictions as *computer-name* above. This name must appear in the alphabet clause of the SPECIAL-NAMES paragraph.
segment-number	any nonnegative integer in the range 1 to 49.

MEMORY-SIZE Clause

The MEMORY-SIZE clause is an obsolete feature of the 1985 ANSI COBOL standard.

The MEMORY-SIZE clause specifies the amount of main memory required by your program. In HP COBOL II, however, memory is allocated automatically through the operating system. Thus, any entry in this clause is treated as a comment.

PROGRAM COLLATING SEQUENCE Clause

On an HP computing system, the following operations are performed on the basis of the ASCII collating sequence:

- Determining the truth value of nonnumeric comparisons explicit in relation or condition name conditions.
- Using nonnumeric sort or merge keys (unless the COLLATING SEQUENCE clause of the respective SORT or MERGE statement is specified in the PROCEDURE DIVISION, and the alphabet name used in it specifies a non-ASCII collating sequence).

The COLLATING SEQUENCE clause can be used in relation with the SPECIAL-NAMES paragraph to define a different collating sequence to be used in these operations.

That is, in the SPECIAL-NAMES paragraph, you can relate *alphabet-name* to the specific collating sequence desired.

An example of the COLLATING SEQUENCE clause is shown under the SPECIAL-NAMES paragraph later in this chapter.

The PROGRAM COLLATING SEQUENCE clause applies only to the program in which it appears. If you omit this clause, the ASCII collating sequence is used.

SEGMENT-LIMIT Clause

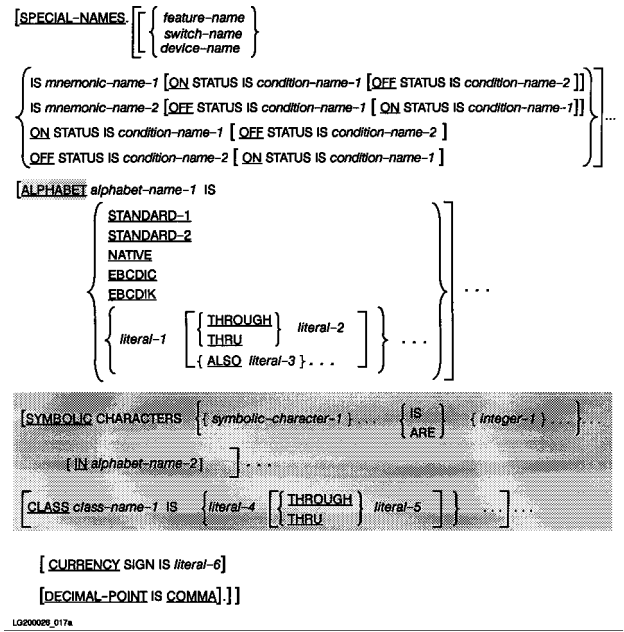
The SEGMENT-LIMIT clause is an obsolete feature of the 1985 ANSI COBOL standard.

The SEGMENT-LIMIT clause is used to define the number of permanent segments in a COBOL program. However, since the concept of a permanent segment has no meaning on an HP computer system, this clause, if specified, is treated as a comment.

SPECIAL-NAMES Paragraph

The SPECIAL-NAMES paragraph allows you to relate certain COBOL-supplied feature, switch, or device names to mnemonic names. By specifying these mnemonic names in your program, you can invoke these functions. It allows you to relate the alphabet name specified in the PROGRAM COLLATING SEQUENCE clause to a particular collating sequence or character set, and to define other collating sequences with different alphabet names. The SPECIAL-NAMES paragraph also allows you to specify symbolic characters and to relate class names to sets of characters.

Syntax



Parameters

feature-name-1	a COBOL reserved word having a specific meaning. Table 6-1 lists these feature names and describes their meanings.
switch-name-1	a COBOL reserved word having a specific meaning. Table 6-1 lists these switch names and describes their meanings.
device-name-1	a COBOL reserved word having a specific meaning. Table 6-1 lists these device names and describes their meanings.
mnemonic-name-1 and *mnemonic-name-2*	names you choose to represent *feature-name-1*, *switch-name-1*, or *device-name-1* within your program. These names can be any valid user-defined COBOL words.
condition-name-1 and *condition-name-2*	each are any valid user-defined COBOL words.
alphabet-name-1	either the name (if any) specified in the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph, or any valid user-defined COBOL word.
literal-1 through *literal-5*	are each nonnumeric or numeric literals. If any such literal is a numeric literal, it must be a positive integer from the range 1 to 256. If any nonnumeric literal is used in a THROUGH or ALSO phrase, it must be only one character long. Note that no ASCII character can be specified more than one time as a literal in any given ALPHABET clause. Literals 1 through 5 must not be nonnumeric figurative constants. `ANSI COBOL'85 allows literals 1 through 5 to be figurative constants.`
`STANDARD-1` and `STANDARD-2`	represent the ASCII collating sequence.
`NATIVE`	currently defined as representing the ASCII collating sequence. However, it may be changed to represent another character set (for example, the KATAKANA character set).
`EBCDIC`	specifies that the EBCDIC collating sequence is to be used. Note that this does not enable any conversion of data. It only allows data to be (for example) sorted or merged according to the EBCDIC collating sequence.
`EBCDIK`	specifies the Japanese version of the EBCDIC collating sequence.

literal-6	a single character, chosen from a specific set. This set is shown later in this chapter, when the CURRENCY SIGN clause is described.
`THROUGH` and `THRU`	are equivalent and may be used interchangeably.
symbolic- character-1	a name you choose to represent the user-defined constant within your program. This name can be any valid user-defined COBOL word.
integer-1	the ordinal position specified by *integer-1* must exist in the ASCII character set. If the IN phrase is specified, the ordinal position must exist in the character set named by *alphabet-name-2.*
alphabet-name-2	the name specified in the ALPHABET clause of the SPECIAL-NAMES paragraph.
class-name-1	a name you choose to represent a user-defined class; this name can be any valid user-defined COBOL word.

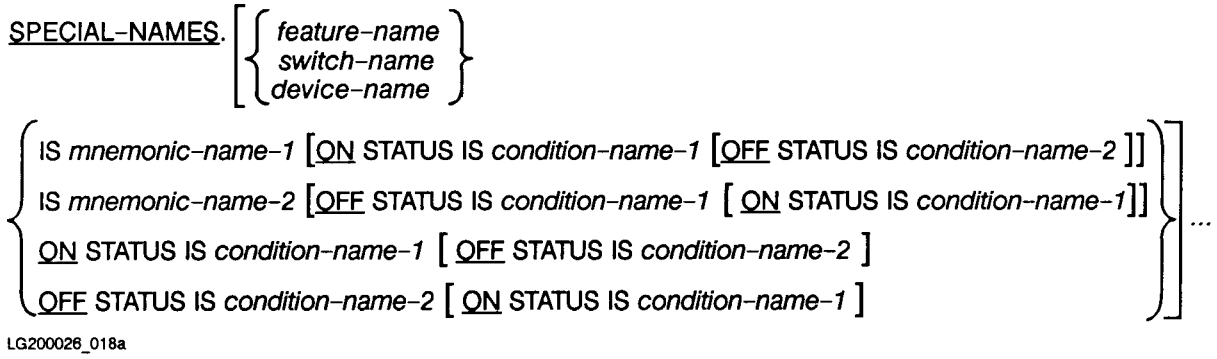
Each clause of the SPECIAL-NAMES paragraph is described on the following pages.

Note The SPECIAL-NAMES clause must follow the order shown in the syntax diagram.

Feature-name, Switch-name, or Device-name Clause

In the Feature-name, Switch-name, or Device-name clause, you relate various COBOL functions to mnemonic names used in your program. You do this by equating the specific function name to the desired mnemonic name.

Syntax



LG200026_018a

Parameters

feature-name-1	a COBOL reserved word having a specific meaning. Table 6-1 lists these feature names and describes their meanings.
switch-name-1	a COBOL reserved word having a specific meaning. Table 6-1 lists these switch names and describes their meanings.
device-name-1	a COBOL reserved word having a specific meaning. Table 6-1 lists these device names and describes their meanings.
mnemonic-name-1 and *mnemonic-name-2*	names you choose to represent *feature-name-1*, *switch-name-1*, or *device-name-1* within your program. These names can be any valid user-defined COBOL words.
condition-name-1 and *condition-name-2*	each are any valid user-defined COBOL words.

Description

The feature, switch, and device names and corresponding functions are listed in Table 6-1. All names except the software switches (SW0 through SW15) and the CONDITION-CODE feature may be referenced by using the assigned *mnemonic-name* in the ACCEPT, DISPLAY, or WRITE statements of the PROCEDURE DIVISION. The CONDITION-CODE feature is related to the special MPE intrinsic relation condition. Refer to the description of relation conditions in Chapter 8 for more information.

Table 6-1. HP COBOL II Feature, Switch, and Device Names

Feature Name	**Function**
CONDITION-CODE	Refers to condition codes returned by operating system intrinsics when they have been called through the CALL statement.
NO SPACE CONTROL	When included in the ADVANCING clause of the WRITE statement, this prevents the line printer from advancing vertically or horizontally.
TOP	When included in the ADVANCING clause of the WRITE statement, the mnemonic name assigned to TOP causes the line printer to perform a page eject.
C01 through C16	Used in the ADVANCING clause of the WRITE statement for sequential files. Each directs the line printer to skip to a particular channel (1 through 16) on the carriage control tape. Refer to Chapter 9 for details.
Switch Name	**Function**
SW0 through SW15	Refer to software switches associated with condition names. (Software switches are described in the next section of this chapter.)
Device Name	**Function**
SYSIN	Refers to the operating system standard input device. In an interactive session, this is your terminal. In a batch job, it is either the card reader or operator's console.
SYSOUT	Refers to the operating system standard output device. In an interactive session, this is your terminal. In a batch job, it is the line printer
CONSOLE	Refers to the computer operator's console (not your terminal).

Software Switches

The software switches are external switches known to COBOL, the status of which is available to each object program functioning within the entire run unit.

Each of the 16 software switches (SW0 through SW15) used in a given program has at least one condition name associated with it.

A condition name and the SET statement are the means by which you can reference one of these switches. For example,

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HPCOMPUTER
OBJECT-COMPUTER. HPCOMPUTER
SPECIAL-NAMES.
    SW0 IS SORT-SWITCH ON STATUS IS SORT-ON.


PROCEDURE DIVISION.
      ⋮
    IF SORT-ON
       THEN CALL "SORTER".
            ⋮
```

In the above example, the actual reference to SW0 is done in the IF statement in the PROCEDURE DIVISION, where `SORT-ON` is tested in a condition name condition. (Chapter 8 has information on condition name conditions.) If `SORT-ON` is in an "on" condition, the subprogram `SORTER` is called. If it is in an "off" condition, the next executable sentence is executed.

Software switches are always considered to be OFF at the beginning of the execution of a program unless you turn them on before execution begins. To turn switches on or off after execution begins, use the SET statement. Refer to Appendix H, "MPE XL System Dependencies", for information on setting software switches before execution begins.

Line Printer Features

The TOP, NO SPACE CONTROL, and C01 through C16 switches are all related to line printer control. Since there is no way for a COBOL II program to explicitly check for the condition of a line printer, you cannot use condition names with these functions. You must, however, specify mnemonic names for these functions if you intend to reference them later in your program. For example:

```
SPECIAL-NAMES.
     TOP IS TOP-OF-FORM, C09 IS TO-END-OF-FORM.
               ⋮
PROCEDURE DIVISION.
               ⋮
    WRITE REC-OUT FROM FOOT-NOTE
       AFTER ADVANCING TO-END-OF-FORM.
               ⋮
    WRITE REC-OUT FROM TITLE
       AFTER ADVANCING TOP-OF-FORM.
               ⋮
```

CONDITION-CODE Features

The CONDITION-CODE function allows you to check the condition code returned by MPE intrinsics. This function is itself a form of conditional variable, with an integer value. Thus, no condition name can be associated with it, and a mnemonic name must be used if you wish to check condition codes returned by intrinsics called from your program. Refer to the section on relation conditions in Chapter 8 for an example and more information.

SYSIN, SYSOUT, and CONSOLE Devices

The SYSIN, SYSOUT, and CONSOLE functions are used in the ACCEPT and DISPLAY statements of the PROCEDURE DIVISION. Since these names refer to terminals, line printers, card readers, and the operator's console, you cannot associate condition names with them. Also, because of the formats of the ACCEPT and DISPLAY statements, you need not specify mnemonic names for them in the SPECIAL-NAMES paragraph. You may, however, choose to do so. Refer to the ACCEPT and DISPLAY statement descriptions in Chapter 9 for more information.

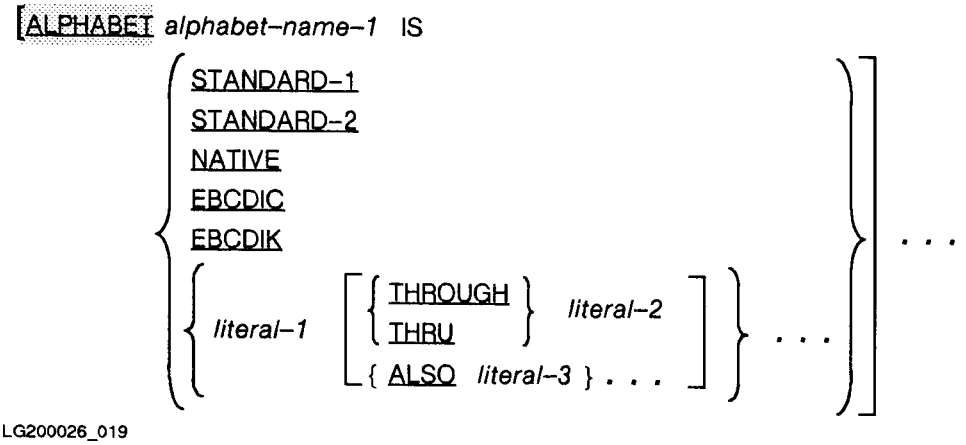
ALPHABET Clause

The ALPHABET clause has three functions:

1. To define a program collating sequence to be used in sort and merge operations, and nonnumeric comparisons.
2. To define an alphabet name and relate this name to either the ASCII, EBCDIK, or the EBCDIC collating sequence. This alphabet name can later be used in the CODE-SET clause of the DATA DIVISION to specify whether records of a sequential file are written in ASCII or EBCDIC code.
3. To define an alternate collating sequence separate from the program collating sequence to be used (optionally) in sort and merge operations.

Syntax

The ALPHABET clause has the following format:



Parameters

alphabet-name-1	either the name (if any) specified in the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph, or any valid user-defined COBOL word.
literal-1 through *literal-3*	each are alphabetic or numeric literals. If any such literal is a numeric literal, it must be a positive integer from the range 1 to 256. If any nonnumeric literal is used in a THROUGH or ALSO phrase, it must be only one character long. Note that no ASCII character can be specified more than one time as a literal in any given ALPHABET clause.
`STANDARD-1` and `STANDARD-2`	represents the ASCII collating sequence.
`NATIVE`	represents the ASCII collating sequence.
`EBCDIC`	specifies that the EBCDIC collating sequence is to be used. Note that this does not enable conversion of data. It only allows data to be

	(for example) sorted or merged according to the EBCDIC collating sequence.
`EBCDIK`	specifies the Japanese version of the EBCDIC collating sequence.
`THROUGH` and `THRU`	equivalent and may be used interchangeably.

To define a program collating sequence, you must relate the *alphabet-name* specified in the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph to either the words NATIVE, STANDARD-1, STANDARD-2, EBCDIC, EBCDIK, or a list of literals.

STANDARD-1 and STANDARD-2 refer to the ASCII collating sequence, which is used on all HP computer systems. Therefore, you need not specify *alphabet-name* in either the PROGRAM COLLATING SEQUENCE clause or the SPECIAL-NAMES paragraph if this is your choice for a program collating sequence.

STANDARD-1, STANDARD-2 and NATIVE Phrases

To specify the ASCII collating sequence used on HP computer systems, enter either the STANDARD-1, STANDARD-2, or NATIVE phrase in the ALPHABET clause. For example,

```
SPECIAL-NAMES.
   ALPHA-NAME IS STANDARD-1.
```

Note Although NATIVE is currently defined as being the ASCII collating sequence, it may be changed in future releases of HP COBOL II. Thus, to avoid the possibility of having to change your program in the future, you should always use the STANDARD-1 or STANDARD-2 phrase, rather than the NATIVE phrase.

EBCDIC and EBCDIK Phrases

To specify the EBCDIC collating sequence, enter EBCDIC in the ALPHABET clause:

```
SPECIAL-NAMES.
   ALPHA-NAME IS EBCDIC.
```

To specify the Japanese version of the EBCDIC collating sequence, use EBCDIK.

LITERAL Phrase

The literal phrase allows you to rearrange the ASCII collating sequence to suit your needs. However, if you specify the literal phrase in an ALPHABET clause, you may not reference that name in a CODE-SET clause.

Defining Your Own Collating Sequence

To define your own collating sequence, you must follow the rules listed below.

1. If you specify a nonnumeric literal, it represents the equivalent character in the ASCII collating sequence. For instance, the literal "A" represents an ASCII A.

   If the literal consists of several characters, each character is assigned successive ascending positions in the collating sequence, beginning with the leftmost character. For example,

   ```
   ALPHA-NAME IS "0A9B8C7D".
   ```

   Results in the following collating sequence:

   ```
   0
   A
   9
   B
   8
   C
   7
   D
   ⋮
   ```

2. If you specify a numeric literal, it represents the *ordinal* number of the corresponding character in the ASCII collating sequence. Therefore, if you specify 66, this represents the 66th character in the ASCII collating sequence—an uppercase A.

3. The order in which the literals appear in the ALPHABET clause determines, in ascending sequence, the ordinal numbers of the corresponding characters in the new collating sequence.

4. Any ASCII characters not explicitly specified in the literal phrase assume positions in the new collating sequence numerically higher than any explicitly specified characters. The relative order of the unspecified characters, with respect to each other, is the same as in the ASCII collating sequence.

5. If you use the THROUGH phrase, the new collating sequence consists of contiguous characters from the ASCII character set, beginning with the value of *literal-1* and ending with the value of *literal-2*. These characters are assigned successive ascending positions in the new character set. As an example, the following ALPHABET clause creates a collating sequence consisting of all uppercase alphabetic characters (letters) from the ASCII collating sequence:

   ```
   ALPHA-NAME IS "A" THROUGH "Z"
   ```

   With the THROUGH phrase, you can also specify and assign ASCII characters in descending sequence:

   ```
   ALPHA-NAME IS "Z" THROUGH "A".
   ```

Note If a nonnumeric literal is used in a THROUGH phrase, it must be only one character long.

---

6. If you use the ALSO phrase, the ASCII characters specified by *literal-1* and *literal-3* are assigned to the same relative positions in the new collating sequence. If you use a nonnumeric literal in an ALSO phrase, it must be only one character long.
7. The character with the highest ordinal position in the new collating sequence may be referenced by the figurative constant, HIGH-VALUE. If more than one character shares the highest ordinal position, the last character specified in the ALPHABET clause is referenced by HIGH-VALUE.
8. The character with the lowest ordinal position in the new collating sequence may be referenced by the figurative constant LOW-VALUE. If more than one character shares the lowest ordinal position, the first character specified in the ALPHABET clause is referenced by LOW-VALUE.
9. Within the SPECIAL-NAMES paragraph, the figurative constants HIGH-VALUE and LOW-VALUE are those positions in the native collating sequence, ASCII. If you redefine HIGH-VALUE or LOW-VALUE in the SPECIAL-NAMES paragraph, the new values will not take effect until after the SPECIAL-NAMES paragraph.

Example

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
OBJECT-COMPUTER.  HP3000
     PROGRAM COLLATING SEQUENCE IS ASCII.
SPECIAL-NAMES.
     ASCII IS STANDARD-1
     IBMCODE IS EBCDIC
     SORT-SEQ IS "A" THROUGH "Z"
                 "a" THROUGH "z".
DATA DIVISION.
FILE SECTION.
FD EBCDICIN
     CODE-SET IS IBMCODE.
           ⋮
FD EBCDOUT
     CODE-SET IS IBMCODE.
           ⋮
PROCEDURE DIVISION.
     SORT SFILE ON ASCENDING LEFT-CHR
          COLLATING SEQUENCE IS SORT-SEQ
          USING INFILE
          GIVING OUTFILE.
     SORT SFILE ON ASCENDING LEFT-CHAR
          COLLATING SEQUENCE IS IBMCODE
          USING EBCDICIN
          OUTPUT PROCEDURE IS SORT-OUT-PARA THROUGH END-OUT.
                    ⋮
SORT-OUT-PARA.
GET-NEXT-REC.
     RETURN SFILE INTO CHK-REC AT END GOTO END-OUT.
     IF DATA-FIELD-2 OF CHK-REC IS NOT ALPHABETIC
```

```
           THEN DISPLAY "ERROR IN SORTED RECORD "
                DISPLAY CHK-REC
           ELSE MOVE CHK-REC TO EBCDOUT.
        GO GET-NEXT-REC.
   END-OUT.
```

In the above example, the program collating sequence is specified as STANDARD-1. Because this is the default, the PROGRAM COLLATING SEQUENCE clause and the ASCII IS STANDARD-1 phrase serve only as documentation.

The *alphabet-name*, `IBMCODE` is used in two file description entries to indicate that the records of the files `EBCDICIN` and `EBCDOUT` are in EBCDIC code. Thus, when the records are read in, they are translated to ASCII, and when they are written out, they are translated back to EBCDIC.

The use of `IBMCODE` in the second SORT statement causes the ASCII records of `SFILE` (obtained from `EBCDICIN`) to be sorted using the EBCDIC collating sequence. Since `EBCDOUT` also names `IBMCODE` as its code-set, the sorted records are translated back from ASCII to EBCDIC when they are written to `EBCDOUT`. Thus, the result of this sorting operation is an EBCDIC file sorted with the EBCDIC collating sequence. It is not necessary to translate the records from EBCDIC to ASCII, unless you want the ability to display an erroneous record. The results of sorting the records without using a translation are the same. However, since, in the output procedure, any erroneous record is displayed, and since it appears as the ASCII equivalent of EBCDIC characters, the CODE-SET clause is required in the file description of `EBCDICIN` to translate the records into ASCII.

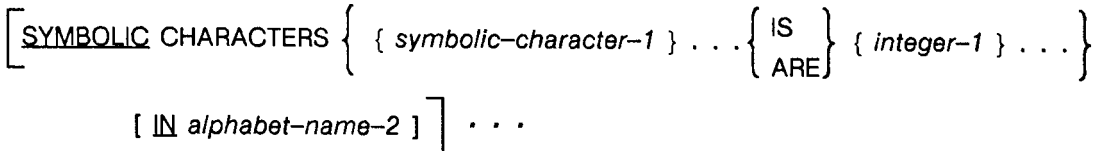
In the first SORT statement, the collating sequence is specified as `SORT-SEQ`. The result of this is that the records of `OUTFILE` are arranged in such a way that all records containing alphabetic characters in their leftmost positions precede records containing nonalphabetic characters in corresponding positions. This is different from the standard ASCII collating sequence, since, in the standard sequence, all numerals and 55 other characters precede the letters of the alphabet name.

SYMBOLIC CHARACTERS Clause

The `SYMBOLIC CHARACTERS` clause is a feature of the 1985 ANSI COBOL standard.

A symbolic character is a user-defined word that specifies a user-defined figurative constant. This feature is useful for unprintable characters. For example, a symbolic character can be used to produce audible output from the terminal.

Syntax



Parameters

symbolic-character-1	name you choose to represent a user-defined figurative constant. The same *symbolic-character-1* must appear only once in a SYMBOLIC CHARACTERS clause.
integer-1	must be a positive integer in the range of 1 to 256.
alphabet-name-2	must be an alphabet name specified in the ALPHABET clause.

The following rules apply to the SYMBOLIC CHARACTERS clause:

- The relationship between each *symbolic-character-1* and the corresponding *integer-1* is by position in the SYMBOLIC CHARACTERS clause. The first *symbolic-character-1* is paired with the first *integer-1*, the second *symbolic-character-1* is paired with the second *integer-1*, and so on.
- There must be a one-to-one correspondence between occurrences of *symbolic-character-1* and *integer-1*.
- The ordinal position specified by *integer-1* must exist in the ASCII character set. If the IN phrase is specified, *integer-1* specifies the ordinal position of the character set named by *alphabet-name-2*.
- If the IN phrase is not specified, *symbolic-character-1* represents the character whose ordinal position in the ASCII character set is specified by *integer-1*.
- The internal representation of *symbolic-character-1* is the internal representation of the character that is represented in the ASCII character set.

For example, the following SYMBOLIC CHARACTERS clause declares the words BELL, CARRIAGE-RETURN, and ESCAPE:

```
SYMBOLIC CHARACTERS BELL IS 8, CARRIAGE-RETURN IS 14, ESCAPE IS 28.
```

With the above declaration you can use the following DISPLAY statements. The first DISPLAY makes a sound on the terminal before displaying the message. The second displays the message in inverse video on certain terminals:

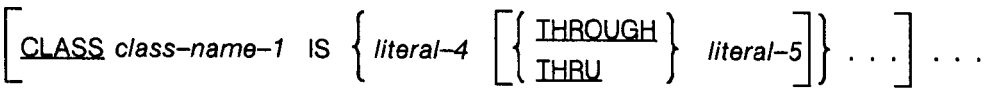
```
DISPLAY BELL " JOB COMPLETED ".
DISPLAY ESCAPE "&dB" "Enter a number: ".
```

CLASS Clause

The `CLASS` clause is a feature of the 1985 ANSI COBOL standard.

The CLASS clause is used to create a user-defined class and provides a means for relating a name to the specified set of characters it lists.

Syntax



LG200026_021a

Parameters

class-name-1	can only be referenced in a class condition. (Refer to Chapter 8 for more information about class conditions.)
literal-4 and *literal-5*	specify values that define the exclusive set of characters contained in *class-name-1*.
`THROUGH` and `THRU`	equivalent.

The following rules apply to the literal specified in the CLASS clause:

- A numeric literal must be an unsigned integer and must have a value within the range of one through the maximum number of characters in the ASCII character set (256).
- A nonnumeric literal that is associated with a THROUGH phrase must be one character in length. If a nonnumeric contains multiple characters, each character is included in *class-name-1*.
- If the THROUGH phrase is specified, the contiguous characters in the ASCII character set beginning with *literal-4* and ending with *literal-5*, are included in *class-name-1*. In addition, the contiguous characters specified by a given THROUGH phrase may specify the characters of the ASCII character set in either ascending or descending sequence.

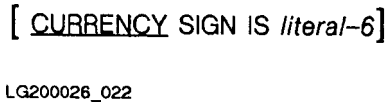
Example

```
SPECIAL-NAMES.
   CLASS VALID-GRADE IS "A" "B" "C" "D" "F".
        ⋮
WORKING-STORAGE.
01 GRADE-LIST.
   05 CLASS-GRADES PIC X OCCURS 5 TIMES.
        ⋮
IF GRADE-LIST IS NOT VALID-GRADE THEN
   PERFORM ERROR-ROUTINE.
```

CURRENCY SIGN IS Clause

The CURRENCY SIGN IS clause is used to specify a literal whose value is later referenced in the PICTURE clause of the DATA DIVISION to denote a currency symbol.

Syntax



Parameters

literal-6 a single character, chosen from a specific set. The literal cannot be any of the following characters:

- The digits 0 through 9.
- The letters A B C D P R S V X Z a b c d p r s v x z
- A space

Note ANSI COBOL'85 allows the letter L, but does not allow lowercase letters.

- The special characters:

  * (asterisk)
  + (plus sign)
  - (minus sign)
  , (comma)
  . (period or decimal point)
  ; (semicolon)
  ((left parentheses)
) (right parentheses)
  " (quotation mark)
  / (slash mark)
  = (equal sign)

  The literal must not be a figurative constant.

Description

If the CURRENCY SIGN IS clause is omitted, the dollar sign ($) must be used as the currency symbol in the PICTURE clause.

Example

To specify the percent sign (%) as the currency symbol, enter the following clause:

```
CURRENCY SIGN IS "%"
```

DECIMAL POINT IS COMMA Clause

The DECIMAL POINT IS COMMA clause allows you to request the exchange of the function of the comma and decimal point (or period) in numeric literals or PICTURE-CLAUSE character strings.

Syntax

[DECIMAL-POINT IS COMMA].

Description

This clause has some effect on pictures for edited data items. Refer to Chapter 7, under "PICTURE-CLAUSE," for details. This clause also has an effect on the ACCEPT FREE verb. When entering data for the ACCEPT FREE verb, use a comma instead of a decimal point.

INPUT-OUTPUT SECTION

The INPUT-OUTPUT SECTION allows you to specify information needed to control the transmission and handling of data between the object program and various input-output devices. Specifically, it permits you to define the names of data files, and the devices on which they reside, and special control techniques to be used in the object program.

The INPUT-OUTPUT SECTION can include the following paragraphs:

- FILE-CONTROL paragraph (required), for specifying the names of files used by your program and other file-related information.
- I-O-CONTROL paragraph (optional), for defining special storage techniques.

The INPUT-OUTPUT SECTION has the following format:

INPUT-OUTPUT SECTION Format

```
[INPUT-OUTPUT SECTION.

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

[I-O-CONTROL.

    [ [       [ RECORD     ]                                                ]
    [ [ SAME  [ SORT       ]  AREA FOR file-name-3  { file-name-4 } . . .   ] . . .
    [ [       [ SORT-MERGE ]                                                ]

    [MULTIPLE FILE TAPE CONTAINS { file-name-5 [POSITION integer-3 ]} . . . ] . . .   . ]]]
```

LG200026_024b

Each paragraph of the INPUT-OUTPUT SECTION is described on the following pages.

FILE-CONTROL Paragraph

The FILE-CONTROL paragraph is used to name files to be used in your program, and to define certain properties of these files that are necessary for their use by your program. Each file named in the FILE-CONTROL paragraph must be described in the DATA DIVISION of your program. Conversely, each file described in the DATA DIVISION must be named once in the FILE-CONTROL paragraph.

An overview of the types of files that can be used in HP COBOL II is presented on the following pages. Following this overview, the various clauses of the FILE-CONTROL paragraph are described.

In HP COBOL II, there are five ways to access and use files:

- Sequential access
- Random access
- Relative access
- Indexed access
- Sort-merge access

Each type of file described must be named, and certain features specified, in the INPUT-OUTPUT SECTION. The organization of the files and their logical records must be described in an SD entry (for sort-merge files), or an FD entry (for any other type of files) in the FILE SECTION of the DATA DIVISION.

Sequential Files

Sequential files are generally files residing on, or being written to, a serial access device (such as a magnetic tape or a serial disk). Of course it is possible to access disk files sequentially also.

A sequentially accessed file means the records of that file can only be accessed in the order in which the records were written to the file. Because of the nature of serial access devices, these types of files can only be written to or read from in a single operation. However, on direct access discs, files being accessed sequentially can be read from and written to at the same time, as well as have a record brought in, modified, and returned to the same storage area.

Random Access Files

Random access files must reside on disk. Through the use of a key, you can read or write a record anywhere within a random access file, regardless of whether data has been written on previous records.

The only limitation on where you can write a record is the externally defined boundaries of the file. For example, if a random access file has been defined to contain a maximum of three thousand records, you cannot write data to record number 3010, although you can write data to record number 2000 without having written data to any preceding records.

Record access of random access files is controlled by the data item defined in the ACTUAL KEY clause of the FILE-CONTROL paragraph. This data item is described anywhere other than in records associated with this file. It is most efficient if defined as a signed integer of five to nine digits whose usage should be described as COMPUTATIONAL

SYNCHRONIZED. However, any numeric data item of a sufficient size for the records in a file can be used.

The ACTUAL KEY data item is used by placing a number into it that corresponds to the logical record number in the file. Logical records in a random access file begin with record zero. Thus, to access the tenth record in a random access file, your program must move the integer 9 into the ACTUAL KEY data item, and then execute the input-output statement.

Execution of an input-output statement for random access files does not update the ACTUAL KEY data item. For example, if the ACTUAL KEY data item contains a value of one before reading or writing takes place, the second record is accessed when the READ or WRITE statement is executed. Following execution of the statement, any subsequent WRITE statement or READ statement (without the NEXT phrase) also accesses record one, unless the value in the ACTUAL KEY data item has been changed.

When your program writes data to a record of a random access file, and previous record areas have had no data written to them, these records are filled with blanks or zeroes. Blank fill is used when the records of the file are designated as ASCII records. Zero fill is used when the records are designated as binary records.

The implication of this blank/zero filling is that you can read records of the file for which no WRITE statement has been executed. In such a case, the data moved into your program is either a blank record, or a zero-filled record. This capability does not exist for any other type of file.

Relative Files

Relative files are similar to random access files in that you access records of such a file through the use of a record number. The only real difference in the two keys used for random access and relative files is that record numbers on a relative file begin with one, rather than with zero as in random access files.

The major functional difference between random access and relative files is that you can always reuse record areas in a random access file by simply writing new data in them. In relative files, you must use the DELETE statement to purge data from record areas. Once a record has been deleted, you can no longer access the area it occupied except to write a record into it again.

Relative files opened in dynamic mode use a data item named in the RELATIVE KEY clause of the FILE-CONTROL paragraph to access records. This data item is later described in the WORKING-STORAGE SECTION. Although the ANSI standard allows only an unsigned integer value for this data item, it is most efficient if defined as a signed integer of five to nine digits whose usage should be described as COMPUTATIONAL SYNCHRONIZED. However, any numeric data item of a sufficient size for the records in a file can be used.

Relative files opened in sequential mode do not need to use the RELATIVE KEY data item to access records. You can simply execute input-output operations on them as though they were sequential files. If, however, you wish to position the file by using the START statement, you must specify the RELATIVE KEY data item, since the START statement uses this data item to find the record you want.

Your program can access a relative file in one of the following ways.

Sequential Access

Sequential WRITE statements for a relative file release data to the file, starting with the first record on the file, and proceeding to the second, third, and so forth in turn. If the RELATIVE KEY data item has been specified, it is updated each time a sequential WRITE statement is executed.

Note Even though you may already have data on these records, a sequential WRITE statement will cause the new data to replace it.

Sequential READ statements for relative files start with a particular record, read it, and proceed to the next existing record.

The record that is read depends upon the type of the last input-output statement executed before the READ statement is encountered. If a DELETE or READ statement is executed, the READ statement reads the next existing record following the record just read or deleted. If an OPEN statement is executed, the first existing record is read. If a successful START statement is executed, the record pointed to is read.

Sequential DELETE statements require the use of the READ statement to position the file to the record to be deleted. This READ statement must be the last input-output operation performed on the file before the DELETE statement is encountered.

Sequential REWRITE statements require the use of the READ statement to position the file to the record to be rewritten. This READ statement must be the last input-output statement performed on the file before the REWRITE statement is encountered.

Random Access

Random access input-output statements use the required RELATIVE KEY data item to select the record for the READ, WRITE, REWRITE, and DELETE statements. Thus, to perform a random access input-output operation on a relative file, you must place the number of the record to be accessed into the RELATIVE KEY data item before executing the input-output statement.

When the random access input-output operation is executed, an implicit seek is performed to find the record, and the specified input-output operation is performed if possible.

Dynamic Access

A relative file open in dynamic access mode allows you to access your file in either random or sequential mode.

The only permissible sequential access that can be performed on a relative file opened in dynamic access mode is the READ NEXT form of the READ statement. This statement allows you to access the records of the file, starting with the record pointed to (if valid) by the current record pointer. If the record is invalid (that is, has been deleted), the next valid record in the file is read.

Indexed Files

Indexed files use data items that are integral parts of the records to control accessing of the records.

For a given indexed file, each record contains a single prime record key, and zero or more alternate record keys. Each key must be described as alphanumeric within the record description entry of the associated indexed file. To indicate which key is the prime record key and which, if any, are the alternate keys, you must specify their names in the RECORD KEY and ALTERNATE RECORD KEY clauses, respectively, in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION.

The prime record key is used in writing, deleting, or updating records of an indexed file. Alternate record keys are used only in reading records.

The values of both prime and alternate record keys may be duplicated within an indexed file. Note however, that since the prime record key must be used for all input-output operations except reading, you should be very careful to make sure that if you are deleting a record whose prime key has duplicates, it is the record you wish to delete, and not some other record with the same prime record key value.

When a prime record key has duplicates, and you use such a key to access a record on an indexed file, the file is searched in a chronological order. That is, the file is searched according to which record was written first. Thus, the record accessed is the first record containing the specified record key value that is still active.

As with relative files, an indexed file can be accessed in random, dynamic, or sequential mode. The actions taken for a specific access mode and a specific input-output operation are listed below.

Sequential Access

A sequential access READ statement for an indexed file uses the current record pointer to read records from the file. The record selected to be read is determined in essentially the same way as for relative files opened in sequential mode. See the description of sequential access of relative files, above.

A sequential access WRITE statement for an indexed file uses the prime record key to place records in the file. Records must be written in ascending order according to these keys. Since the prime record key (and all alternate keys as well) must be alphanumeric, this means that the records must be written using the ASCII collating sequence to determine ascending order.

A sequential access REWRITE or DELETE statement requires that a READ statement be the last executed input-output statement for the file being referenced.

Random Access

The READ ... KEY IS form requires that you place a key value into one of the record key data items (prime or alternate). This data item is then specified in the READ statement, and the indexed file is searched until a record having the same value in the same record key is found. This record is then brought into your program.

A random access WRITE, REWRITE, or DELETE statement uses the contents of the RECORD KEY data item to select the record to be written or deleted.

Dynamic Access

When dynamic access is used for indexed files, you can use either the READ NEXT or the READ ... KEY IS form of the READ statement.

The READ NEXT form is used when you wish to read records in a sequential manner. This statement allows you to access the records of the file starting with the record pointed to by the current record pointer (if valid). If the record is invalid, the next valid record in the file is read.

A dynamic access WRITE, REWRITE, or DELETE statement uses the contents of the RECORD KEY data item to select the record to be written or deleted.

As an extension to ANSI COBOL'85, HP COBOL II allows the use of alphanumeric, computational, numeric display (without the optional SIGN clause) and COMPUTATIONAL-3 data types for RECORD KEY and ALTERNATE RECORD KEY clauses respectively, in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION.

Indexed files are processed with Hewlett-Packard's KSAM subsystem in HP COBOL II. Refer to the *KSAM/XL Reference Manual* for information such as creation and deletion of this file type. Programmatic creation and deletion of these files can be accomplished by calling the appropriate intrinsics described in the above manual using the CALL statement. Opening, closing, and reading of these files is accomplished through the normal COBOL OPEN, CLOSE, and READ statements.

Sort-Merge Files

The ability to arrange records in a particular order is often required in COBOL applications. This ability is provided by the sort and merge features of HP COBOL II.

The sort facility allows you to arrange the records of one or more files in a specified sequence. The merge facility merges two or more previously sorted files.

Sort-merge files are the files acted upon by the sort and merge operations. These files can never be accessed directly, except in input and output routines associated with a SORT or MERGE statement.

When a SORT or MERGE statement is issued, the records are taken from the named input file or procedure and then placed in the sort/merge workfile. Finally, the sorted/merged records are placed in the output file, files, or procedure.

Refer to Chapter 12, "SORT-MERGE Operations", for more information on sort-merge files and operations.

File Status

Every type of file described above (except sort-merge files) may have a file status data item associated with it. This data item can be used to check on the success or failure of an input-output operation involving the file with which it is associated. File status data items are described in more detail later in this chapter.

Fixed Length Records

Fixed length records must contain the same number of character positions for all the records in the file. Only one record size can be processed by all input-output operations on the file. Fixed length records may be explicitly selected by specifying format 1 of the RECORD clause in the file description entry for the file, regardless of the individual record descriptions. Refer to Chapter 7 for more information on the RECORD clause.

Variable Length Records

Variable length records can contain differing numbers of character positions among the records on the file. To define variable length records explicitly, specify the VARYING phrase in the RECORD clause of the file description entry or the sort-merge file description entry for the file. The length of a record is either affected by: the data item referenced in the DEPENDING phrase of the RECORD clause, the DEPENDING phrase of an OCCURS clause, or the length of the record description entry for the file.

File Control Clauses

The various clauses of the FILE-CONTROL paragraph can be specified in five separate formats, depending upon the type of file being described. These formats follow:

File Control Format

Format 1 – For Sequential Files

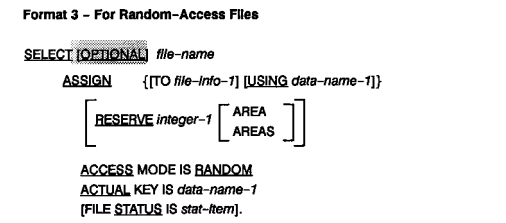
```
SELECT [OPTIONAL] file-name-1
    ASSIGN {[TO file-info-1] [USING data-name-1]}
    [RESERVE integer-1 [AREA|AREAS]]
    [[ORGANIZATION IS] SEQUENTIAL]
    [ACCESS MODE IS SEQUENTIAL]
    [FILE STATUS IS stat-item].
```

Format 2 – For Relative Files

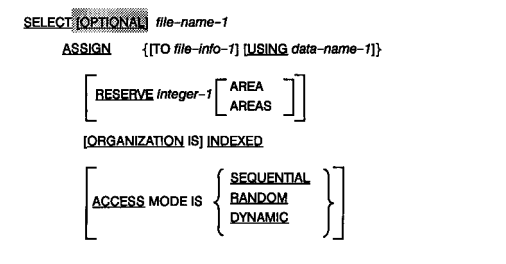
```
SELECT [OPTIONAL] file-name-1
    ASSIGN {[TO file-info-1] [USING data-name-1]}
    [RESERVE integer-1 [AREA|AREAS]]
    [ORGANIZATION IS] RELATIVE
    [ACCESS MODE IS {SEQUENTIAL [RELATIVE KEY IS data-name-1]
                    {{RANDOM|DYNAMIC} RELATIVE KEY IS data-name-1}}]
    [FILE STATUS IS stat-item].
```

LG200026_025d

Format 3 – For Random-Access Files



Format 4 – For Indexed Files



LG200026_029a

Format 5 – For Sort–Merge Files

SELECT *file–name–1*

ASSIGN {[TO *file–info–1*] [USING *data–name–1*]}

LG200026_027a

In the FILE-CONTROL paragraph body, the SELECT clause must be specified first. The remaining clauses may appear in any order.

Each of the clauses of the FILE-CONTROL paragraph are described on the following pages in alphabetical order, except the SELECT and ASSIGN clauses. Since these clauses must be specified first for any file type, they are discussed first.

SELECT Clause

The SELECT clause is used to identify a file to be used in your program.

Syntax

The SELECT clause has two formats:

Format 1 – For Sort–Merge Files

SELECT *file–name–1*

Format 2 – For All Other Files

SELECT [OPTIONAL] *file–name–1*

LG200026_028

Parameters

file-name-1 any valid user-defined COBOL word.

File-name-1 is the name you use later in OPEN, CLOSE, USE, and other statements in the PROCEDURE DIVISION. It must also be used in an FD or an SD entry in the DATA DIVISION.

OPTIONAL Phrase

The purpose of the OPTIONAL phrase is to allow you to specify an input file in the SELECT statement that may not be present during a particular execution of the program in which it is named. If the file is not present and the OPTIONAL phrase has been specified, when the first READ statement naming that file is executed, the imperative statement in the associated AT END phrase is executed. If no AT END phrase has been specified, then a USE procedure must be defined, either explicitly or implicitly, and this procedure is executed.

```
When the file is not present and it is opened in I-O or extend mode with the
OPTIONAL phrase, a new file is created. Otherwise, file status 35 is returned.
```

ASSIGN Clause

The ASSIGN clause associates a file with the storage medium on which the file resides. With the USING phrase, you can assign logical files to physical files dynamically (that is, at run time).

Syntax

ASSIGN { [TO *file-info-1*] [USING *data-name-1*] }

Parameters

file-info-1 a nonnumeric literal of the form:

name `[,[`*class*`] [,[`*recording mode*`] [, [`*device* `[(CCTL)] ]`
`[,[`*file-size*`] [, [`*formsmessage.*`] [,L]]]]]]`

The meanings of each parameter of *file-info-1* are given below.

name the operating system file designator. Refer to Appendix H, "MPE XL System Dependencies", for detailed information about this file designator.

If you use the USING phrase of the ASSIGN clause, this parameter is ignored. The operating system file designator must be supplied in *data-name-1* instead.

class the class of device on which the file resides. This parameter is not used by the file system and is ignored if it is specified. However, if specified, the class parameter can be one of the following three mnemonics:

DA, implying a mass-storage device.

UT, implying a utility device such as a tape drive.

UR, implying a unit-record device, such as a card reader.

If the class parameter is omitted, DA is assigned by default.

recording mode the recording mode of the file. It may be either ASCII or binary. If the file is an ASCII file, recording mode must be A. If the file is binary, the recording mode must be B. If the recording mode parameter is omitted, ASCII is assigned by default.

device the type of device on which the file resides. Refer to Appendix H, "MPE XL System Dependencies", for further details.

`CCTL` the carriage control option for an output file, indicating that carriage control directives are supplied in write operations referencing line printer files. If omitted, your program uses the file system default for the device or file.

file-size the number of records in the file. Refer to Appendix H, "MPE XL System Dependencies", for further details.

formsmessage for a listing device, a request for the operator to provide special forms, such as blank checks or inventory report forms, on the printer. For any other device, this parameter is ignored. This entry may contain a maximum of 49 characters and must be terminated with a period.

`L` enables your program to dynamically lock and unlock a disk file. However, this feature is provided only to assist in the conversion of COBOL'68 programs to COBOL II. It is recommended that the EXCLUSIVE and UN-EXCLUSIVE statements be used instead for file locking and unlocking. `L` is not a required parameter if the EXCLUSIVE and UN-EXCLUSIVE statements are used. Otherwise, it enables your program to dynamically lock and unlock a disk file.

data-name-1 an alphanumeric data item containing the operating system file designator (see *name* above). *data-name-1* must not be subordinate to the file description entry for the file described in the enclosing SELECT clause. If the TO phrase is also specified, all the information in *file-info-1* except the file designator, *name*, is used when the file is opened.

See Appendix H, "MPE XL System Dependencies," for detailed information about this file designator.

The USING *data-name-1* form of the ASSIGN clause is an HP extension to the ANSI COBOL standard.

Description

You associate a file with a storage medium in the ASSIGN clause. When you use the TO phrase, this association occurs at compile time and cannot be changed unless you modify your program and recompile it. When you use the USING phrase, the association occurs at run time. You can change the association in the PROCEDURE DIVISION by changing the contents of *data-name-1*. *data-name-1* contains the operating system file designator for the file. See the *HP COBOL II/XL Programmer's Guide* for an example of the USING clause.

If you use both the TO and USING phrases, the *name* parameter of *file-info-1* is ignored. The name specified in *data-name-1* is used instead. Any other parameters specified in *file-info-1* also apply to the file named in *data-name-1*.

File Status Code

File status code 31 indicates a permanent error where an OPEN, SORT, or MERGE of a file specified in *data-name-1* has failed. The operation may have failed because the contents of *data-name-1* were not consistent with the contents of *file-info-1* in the TO phrase. This error also occurs if *data-name-1* contains an invalid file name.

ACCESS MODE Clause

The ACCESS MODE clause specifies the way in which your program is to access the associated file.

Syntax

There are four formats of this clause, depending upon the type of file being described. This clause cannot be used for sort-merge files. The four formats are shown below.

Format 1 – Sequential

```
[ACCESS MODE IS SEQUENTIAL]
```

Format 2 – Relative

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

Format 3 – Random

```
ACCESS MODE IS RANDOM
```

Format 4 – Indexed

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

LG200026_029

Parameters

data-name-1 a data item that must not be defined in the record description entry for the relative file being described. Furthermore, the item must be defined as an unsigned integer. Such a data item might be defined as the following where n is an integer in the range 5 to 9:

```
USAGE COMPUTATIONAL SYNCHRONIZED PIC 9(n),
```

Note The ANSI standard defines the relative key value as an unsigned integer data item; however, HP COBOL II extends the standard and allows the use of signed integers for efficiency of code. Because a 32-bit integer is used for relative record numbers, a definition of "USAGE COMPUTATIONAL SYNCHRONIZED PIC S9(9)" would allow for maximum code efficiency and record access.

Even if you want to access a relative file sequentially, you must specify the RELATIVE KEY phrase in order to reference the associated file in a START statement.

For three of the four file types to which this clause pertains (sequential, relative, and indexed), you must specify an ACCESS MODE clause or sequential access is assumed. You must specify the ACCESS MODE clause exactly as it is shown for random access files.

The three access modes are defined as follows:

- Sequential access means that existing records are accessed in ascending order. The relative key is used for relative files, and a prime or alternate record key is used for indexed files. Random access files may not be accessed sequentially.
- Dynamic access means that your program may alternate between sequential and random access modes by selectively using different forms of various input-output statements. This type of access may only be used for relative and indexed files.
- Random access means that the records are accessed directly by using a record key data item (for indexed files) or by using the relative record numbers of records (for relative and random access files).

For details on how sequential, dynamic, and random access is performed on the various file types, refer to the overview on preceding pages of this chapter.

ACTUAL KEY Clause (an HP extension to the ANSI COBOL standard)

The ACTUAL KEY clause names the data item to be used in accessing records of a random access file and applies only to random access files.

Syntax

This clause has the following format:

ACTUAL KEY IS *data-name-1*

Parameters

data-name-1 an integer item of one to nine digits.

Example

For greatest efficiency, the variable *data-name-1* should have a PICTURE clause in the following form:

```
PIC S9(9) USAGE COMPUTATIONAL SYNCHRONIZED.
```

This data item must be defined in either the FILE SECTION or the WORKING-STORAGE SECTION of the DATA DIVISION. It corresponds to a relative record number. Record numbers in random access files begin with 0. To ensure the accessibility of all records, the data item must be large enough to contain the greatest record number in the file.

ALTERNATE RECORD KEY Clause

Each use of the ALTERNATE RECORD KEY clause names an alternate record key for an indexed file. The number of alternate keys for a file must be the same as the number used when the file was created.

Syntax

This clause has the following format:

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

Parameters

data-name-2 — the name of a data item described as alphanumeric in a record description entry for the file with which it is associated. It must not reference an item whose first character begins in the same position as the first character of any other key, whether alternate or prime.

DUPLICATES Phrase

The DUPLICATES phrase specifies that the named alternate key may be duplicated within any of the records of the file. If this phrase is unused, the contents of the data item referenced by *data-name-2* must be unique among records of the file.

As an extension to ANSI COBOL'85, HP COBOL II also allows the use of computational, numeric display (without the optional SIGN clause), COMPUTATIONAL-3, `BINARY, and PACKED-DECIMAL` data types for *data-name-2*.

FILE STATUS Clause

The FILE STATUS clause allows you to name a data item to be used in obtaining information about the success or failure of input-output operations performed using the file being described. This clause is optional, and may be used for all types of files except sort-merge files.

Syntax

The FILE STATUS clause has the following format:

[FILE STATUS IS *stat-item*].

Parameters

stat-item — a two character alphanumeric data item defined in the WORKING-STORAGE SECTION of the DATA DIVISION.

Description

When an input or output operation has been performed on a file that has a FILE STATUS *data-item* associated with it, the data item is updated with two characters that indicate the status of the operation. These two characters are the file status code. The leftmost character of this data item is called *status-key-1*. The rightmost character is called *status-key-2*. The values that can be placed in *status-key-1* and *status-key-2*, and their meanings, are shown in Table 6-2 and Table 6-3.

Table 6-2 and Table 6-3, respectively contain information about the ANSI COBOL'85 and ANSI COBOL'74 I-O status-codes. Key terms used in those tables are defined below.

`EOF` — The program attempted to read a record following the last record in the file.

`AT END` — A sequential READ statement was unsuccessfully executed as a result of an AT END condition.

`INVALID KEY` — The input-output operation failed because a duplicate key existed, a boundary violation occurred, the record sought could not be found, or a sequence error occurred (for indexed files only).

`PERMANENT ERROR` — The input-output statement was unsuccessfully executed as the result of an error that precluded further processing of the file.

`LOGIC ERROR` — The input-output statement was unsuccessfully executed as a result of an improper sequence of input-output operations that were performed on the file, or as a result of violating a user-defined limit.

`IMPLEMENTOR DEFINED` — The implementor defined codes are 9*x*. When *status-key-1* is set to 9, an unexpected error has occurred. In this case, the value placed in *status-key-2* is a binary integer quantity corresponding to a file system error. Since this quantity can range from 0 to 255, and since the status key item is alphanumeric, your program will interpret this integer as some character from the ASCII collating sequence. For an example, see the section "File Status Codes" in Chapter 5 of the *HP COBOL II/XL Programmer's Guide*.

`FIXED FILE ATTRIBUTE` Information about a file that is established when a file is created and cannot subsequently be changed during the file's existence. These attributes include the following:

- Organization of the file (sequential, relative, or indexed).
- Prime record key.
- Alternate record keys.
- Code-set.
- Minimum and maximum record size.
- Record type (fixed or variable).
- Collating sequence of the keys for indexed files.
- Blocking factor.
- Padding character.
- Record delimiter.

The ANSI COBOL'85 file status codes are potentially incompatible with the ANSI COBOL'74 status codes.

- Some 9*x* values have been changed to other codes. For example, the ANSI COBOL'85 file status 38 was previously a 9*x* code.
- In order to provide more information to you, some additional values have been specified for *status-key-2*.

Table 6-2. ANSI COBOL'85 File Status Codes

	SEQUENTIAL	RANDOM ACCESS or RELATIVE	INDEXED
SUCCESSFUL	00-Successful. No more information available. 04-READ length of record doesn't match file. 05-OPEN. Optional file not present, created. 07-File is not a tape as the OPEN/CLOSE phrase implies.	00-Successful. No more information available. 04-READ length of record doesn't match file. 05-OPEN. Optional file not present, created.	00-Successful. No more information available. 02-READ current key = next key value -WRITE or REWRITE creates duplicate key for alternate key in which duplicates are allowed. 04-READ length of record doesn't match file. 05-OPEN. Optional file not present, created.
AT END	10-EOF or optional file not present on READ.	10-EOF or optional file not present on READ. 14-Record number too big for relative key data item on READ.	10-EOF or optional file not present on READ.
INVALID KEY		22-WRITE a duplicate key. * 23-Record does not exist. -START OR READ on missing optional file. 24-WRITE beyond file boundary. -Sequential WRITE record number too big for relative key data item.	21-Sequence error. 22-WRITE OR REWRITE a duplicate key. 23-Record does not exist. -START OR READ on missing optional file. 24-WRITE beyond file boundary.

* Does not apply to random files.

Table 6-2. ANSI COBOL'85 File Status Codes (continued)

	SEQUENTIAL	RANDOM ACCESS or RELATIVE	INDEXED
PERMANENT ERROR	30-No more information available. 31-OPEN, SORT, or MERGE of dynamic file failed due to file attribute conflict. 34-Boundary violation. 35-Nonoptional file not present for OPEN. 37-EXTEND or OUTPUT on unwritable file. -I-O for file that does not support it. -INPUT on invalid device for input. 38-OPEN on file closed with LOCK. 39-OPEN unsuccessful due to fixed file attribute conflict.	30-No more information available. 31-OPEN, SORT, or MERGE of dynamic file failed due to file attribute conflict. 35-Nonoptional file not present for OPEN. 37-EXTEND or OUTPUT on unwritable file. -I-O for file that does not support it. -INPUT on invalid device for input. 38-OPEN on file closed with LOCK. 39-OPEN unsuccessful due to fixed file attribute conflict.	30-No more information available. 31-OPEN, SORT, or MERGE of dynamic file failed due to file attribute conflict. 35-Nonoptional file not present for OPEN. 37-EXTEND or OUTPUT on unwritable file. -I-O for file that does not support it. -INPUT on invalid device for input. 38-OPEN on file closed with LOCK. 39-OPEN unsuccessful due to fixed file attribute conflict.
LOGIC ERROR	41-OPEN on file that is already open. 42-CLOSE for file not open. 43-No READ before REWRITE. 44-Boundary violation. -Record too big or too small. -Rewrite record not same size. 46-READ after AT END or after unsuccessful READ. 47-READ on file not open for input. 48-WRITE on file not open for output. 49-REWRITE on file not open for I-O.	41-OPEN on file that is already open. 42-CLOSE for file not open. 43-No READ before REWRITE/DELETE. 44-Boundary violation. -Record too big or too small. 46-READ after AT END or after unsuccessful READ or START. 47-READ or START on file not open for input or I-O. 48-WRITE on file not open for output or I-O. 49-REWRITE or DELETE on file not open for I-O.	41-OPEN on file that is already open. 42-CLOSE for file not open. 43-No READ before REWRITE/DELETE. 44-Boundary violation. -Record too big or too small. 46-READ after AT END or after unsuccessful READ or START. 47-READ or START on file not open for input or I-O. 48-WRITE on file not open for output or I-O. 49-REWRITE/DELETE on file not open for I-O.

Table 6-3. ANSI COBOL'74 File Status Codes

	SEQUENTIAL	RANDOM ACCESS or RELATIVE	INDEXED
SUCCESSFUL	00-Successful. NO more information available.	00-Successful. No more information available.	00-Successful. No more information available. 02-READ current key = next key value. -WRITE or REWRITE creates duplicate key for alternate key in which duplicates are allowed.
AT END	10-EOF or optional file not present.	10-EOF or optional file not present.	10-EOF or optional file not present.
INVALID KEY		22-WRITE a duplicate key. 23-Record does not exist. -START OR READ on missing optional file. 24-WRITE beyond file boundary. -Sequential WRITE record number too big for relative key data item.	21-Sequence error. 22-WRITE OR REWRITE a duplicate key. 23-Record does not exist -START OR READ on missing optional file. 24-WRITE beyond file boundary.
PERMANENT ERROR	30-No more information available. 34-Boundary violation.	30-No more information available.	30-No more information available.

ORGANIZATION Clause

The ORGANIZATION clause specifies the logical structure of the file being described. It can be used in sequential, relative, and indexed files.

Syntax

The three formats of the ORGANIZATION clause are shown below.

[ORGANIZATION IS] SEQUENTIAL

[ORGANIZATION IS] RELATIVE

[ORGANIZATION IS] INDEXED

Description

The ORGANIZATION clause is required for relative and indexed files. It is optional for sequential files. This clause cannot be used for sort-merge and random access files.

RECORD KEY Clause

The RECORD KEY clause is used for indexed files only. It is required, because it provides the means (*data-name-1*) by which the associated indexed file is accessed.

Syntax

RECORD KEY IS *data-name-1* [WITH DUPLICATES]

The WITH DUPLICATES phrase in the record key clause is an HP extension to the ANSI COBOL standard.

Parameters

data-name-1 — the name of an alphanumeric data item defined in a record description entry associated with the file being described. As an HP extension to ANSI COBOL'85, HP COBOL II also allows the use of computational, numeric display (sign overpunched on least significant digit), COMPUTATIONAL-3, `BINARY, and PACKED-DECIMAL` data types for *data-name-1*.

Data-name-1 is the prime record key for the file. The data-description for *data-name-1*, and its relative position within a record must be the same as that used when the file was created.

DUPLICATES Phrase

The DUPLICATES phrase specifies that the named prime record key may be duplicated within any of the records of the file. If, however, you do not specify that duplicates may exist, then the value of the key must be unique among records of the file.

Note If an indexed file has the DUPLICATES phrase specified for its primary key, the REWRITE/DELETE statement should be used only when the indexed file is in sequential access mode. This is because a REWRITE/DELETE statement issued for a file whose access mode is dynamic or random only rewrites/deletes the first record of a DUPLICATE primary key chain.

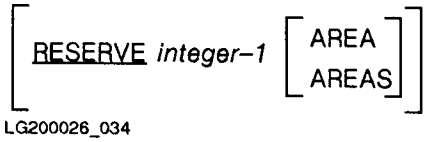
RESERVE Clause

The RESERVE clause allows you to indicate the number of input-output buffers to be allocated for the file being described. Its use is optional. If you do not specify it, the number of buffers allocated is the operating system default.

This clause may be used for sequential, relative, indexed, and random access files. It may not be used for sort-merge files.

Syntax

The format of the RESERVE clause is shown below.



LG200026_034

Parameters

integer-1 a nonnegative integer in the range zero to 16.

`AREA` and `AREAS` each are equivalent, and can be used interchangeably.

Description

If you specify zero, the operating system still allocates the default number. Also, although you can specify more than two, any more than three buffers does not usually increase input-output efficiency. See your operating system reference manual for more information on buffers.

Example

The following example shows a FILE-CONTROL paragraph for an indexed file and a sequential file, as well as important data items associated with the files.

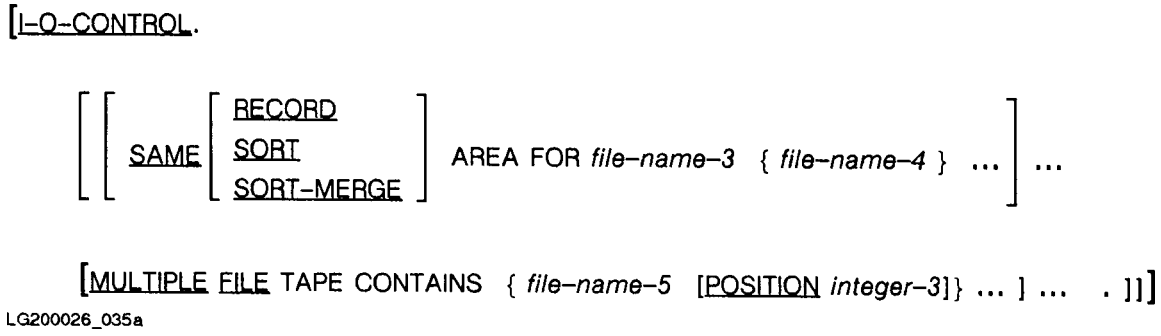
```
ENVIRONMENT DIVISION.
       ⋮
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT INDXFILE
  ASSIGN TO "KFILE,DA,A,DISC,5000,,L"
  RESERVE 3 AREAS
  ORGANIZATION IS INDEXED
  ACCESS MODE IS DYNAMIC
  RECORD KEY IS FIRST-CHARS
   ALTERNATE RECORD KEY IS SECOND-CHARS WITH DUPLICATES
  FILE STATUS IS CHECK-KFILE.

SELECT TAPEIN
  ASSIGN TO "READTAPE,UT,TAPE,,HANG TAPE 001."
  RESERVE 2 AREAS
  FILE STATUS IS CHECK-TAPE.
       ⋮
DATA DIVISION.
FILE SECTION.
FD INDXFILE
    LABEL RECORDS ARE OMITTED.
01 RECORD-IN.
   02 FIRST-CHARS                  PIC X(8).
   02 SECOND-CHARS                 PIC X(24).
       ⋮
FD TAPEIN
    LABEL RECORDS ARE OMITTED.
01 TAPE-REC                        PIC X(80).
       ⋮
WORKING-STORAGE SECTION.
77 CATCHNUM         PIC 9(4)  USAGE DISPLAY.
01 CHECK-KFILE.
   02  STAT-KEY-1   PIC X.
   02  STAT-KEY-2   PIC X.
01 CHECK-TAPE.
   02 STAT-KEY-1    PIC X.
   02 STAT-KEY-2    PIC X.
```

I-O-CONTROL Paragraph

The I-O-CONTROL paragraph is an optional paragraph that specifies the areas of memory (buffers) to be shared by different files and the locations of files on a multi-file tape reel.

Syntax

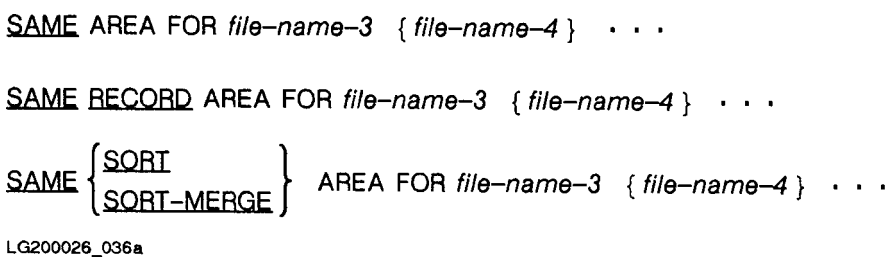
This paragraph has the following format:



SAME Clause

The SAME clause has three formats, whose meanings and restrictions are described below.

Syntax



SAME AREA Clause

The SAME AREA clause allows you to conserve main memory space by permitting two or more non-sort or non-merge files to use the same area of main memory for processing the file.

Because the area shared includes all storage areas assigned to the files specified, only one file can be open at any given time.

Also, a file name can appear in only one SAME AREA clause within a program. However, this does not exclude the possibility of the file name appearing in a SAME RECORD AREA, SAME SORT AREA, or SAME SORT-MERGE AREA clause.

The restrictions on file names appearing in more than one SAME RECORD, SAME SORT AREA, or SAME SORT-MERGE AREA are listed on the following pages under the headings SAME RECORD AREA clause, SAME SORT AREA, and SAME SORT-MERGE AREA clauses.

The files referenced in the SAME AREA clause need not all have the same organization or access.

An external file can not be referenced in a SAME AREA clause.

To specify that `FILEA`, `FILEB`, and `FILEC` share the same processing area, enter the following:

```
SAME AREA FOR FILEA, FILEB, FILEC.
```

SAME RECORD AREA Clause

The SAME RECORD AREA clause specifies that two or more files (of any kind) be allowed to share the same area of main memory for processing the current logical record.

If none of the files in this clause appears in a SAME AREA clause, then all of the files can be open at the same time.

A logical record in this shared record area is considered a logical record of each open output file named in the SAME RECORD AREA clause. It is also considered a record of the most recently opened input file named in the SAME RECORD AREA clause. This is equivalent to an implicit redefinition of the shared area. That is, records are aligned on the leftmost character position.

If any file appears in the SAME RECORD AREA clause and in the SAME AREA clause, then all other files appearing in the SAME AREA clause must also appear in the SAME RECORD AREA clause. Of course, files not named in the SAME AREA clause can also appear in the SAME RECORD clause. Because of this restriction, and since only one file named in a SAME AREA clause can be open at any given time, if any file in the SAME RECORD AREA clause also appears in a SAME AREA clause, then the rule that only one file at one time may be open takes precedence over the rule that all files named in the SAME RECORD AREA clause can be open at the same time.

As with files named in the SAME AREA clause, files named in the SAME RECORD AREA clause can have different organizations and access modes. Also, if a file name appears in a SAME RECORD AREA clause, it can not appear in any other SAME RECORD AREA clause within the program.

External files can not appear in a SAME RECORD AREA clause.

SAME SORT AREA and SAME SORT-MERGE AREA Clauses

The SAME SORT AREA and SAME SORT-MERGE AREA clauses are equivalent. Both specify that the area used in main memory for sorting or merging sort-merge files is shared.

No sort or merge file can appear in a SAME AREA clause; However, it is not necessary for all files named in a SAME SORT AREA or SAME SORT-MERGE AREA clause to be sort or merge files. Only one must be. Furthermore, any sort or merge file that appears in a SAME SORT AREA or SAME SORT-MERGE AREA clause can not appear in another SAME SORT or SAME SORT-MERGE AREA clause within the same program.

If a non-sort or non-merge file appears in a SAME AREA clause and in one or more SAME SORT AREA or SAME SORT-MERGE AREA clauses, then all files named in that SAME AREA clause must appear in the SAME SORT AREA or SAME SORT-MERGE AREA clauses.

During the execution of a SORT or MERGE statement that refers to a file named in a SAME SORT AREA or SAME SORT-MERGE AREA clause, those files that are not sort or merge files, but are named in the SAME SORT AREA or SAME SORT-MERGE AREA clauses, must not be open.

The files named in a SAME SORT AREA or SAME SORT-MERGE AREA clause need not have the same organization or access mode.

External files can not appear in a SAME SORT AREA clause or a SAME SORT-MERGE AREA clause.

Note Because only one file can be OPEN at any given time, there is no implied redefinition of the record storage area, unless the SAME RECORD AREA clause is also used. Therefore, any access of the record area through non-OPEN'ed file data items yields undefined results.

MULTIPLE FILE Clause

The MULTIPLE FILE clause is an obsolete feature of the 1985 ANSI COBOL standard.

The MULTIPLE FILE clause specifies the location of files on a multiple file reel.

Syntax

[MULTIPLE FILE TAPE CONTAINS {*file-name-5* [POSITION *integer-3*] } ...] ...

Description

When the file referenced by your program shares a labeled tape with other files, you must enter the MULTIPLE FILE clause. Regardless of the number of files on the reel, you need specify only those used by your program. If you specify the files in chronological order, you need only enter the file names in the MULTIPLE FILE clause:

```
MULTIPLE FILE TAPE CONTAINS FILEA, FILEC, FILEF
```

But if you specify the files in random order, you must indicate their positions by using the POSITION clause:

```
MULTIPLE FILE TAPE CONTAINS FILEC POSITION 3
    FILEF POSITION 6, FILEA POSITION 1
```

In the second example, the first position, in relation to the beginning of the reel, is position one.

No more than one file on the same tape device can be open at the same time.

The files specified in this clause cannot be external files.

Note The MULTIPLE FILE clause applies to labeled sequential files only. This is because of the intrinsically sequential nature of magnetic tape devices.

7

DATA DIVISION

The DATA DIVISION describes all data that your program is to use. That is, it describes any data to be read from or written to a file, data developed internally and held in temporary or working storage, and any constants that are used. `The DATA DIVISION is optional.` When included, however, it must follow the ENVIRONMENT DIVISION.

There are three sections within the DATA DIVISION:

- FILE SECTION: defines the structure of data files.
- WORKING-STORAGE SECTION: describes data items used as constants by the object program, as well as records and noncontiguous data items that are developed and used internally, and are not part of external data files.
- LINKAGE SECTION: describes data items within subprograms that are referenced by both the calling and the called program. This section appears only in programs that will be called by some other program. Its format is the same as the format of the WORKING-STORAGE SECTION.

DATA DIVISION Format

The DATA DIVISION has the following general format;

GENERAL FORMAT FOR DATA DIVISION

```
[DATA DIVISION.

[FILE SECTION.

[ file-description-entry   { record-description-entry } . . .                    ]. . . .]
[ sort-merge-file-description-entry  { record-description-entry } . . .          ]

[WORKING-STORAGE SECTION.

[ 77-level-description-entry ] . . .]
[ record-description-entry   ]

[LINKAGE SECTION.

[ 77-level-description-entry ] . . .]]
[ record-description-entry   ]
```

LG200026_038

The format of each section is described separately on the following pages, followed by an explanation of record description entries.

DATA DIVISION Syntax Rules

All sections within the DATA DIVISION are optional. When included, however, they must appear in the order shown in the format description.

Each DATA DIVISION entry begins with a level indicator or a level number, followed by a space, the name associated with the level indicator or level number, and a sequence of independent description clauses. The last clause must be terminated by a period.

The division header must begin in Area A (the eighth through eleventh character positions of each record). It consists of the words DATA DIVISION and is followed by a period.

FILE SECTION

The FILE SECTION defines the structure of any sequential, indexed, relative, random access, or sort-merge file appearing in your program.

Each file is defined by a file description entry and one or more record descriptions.

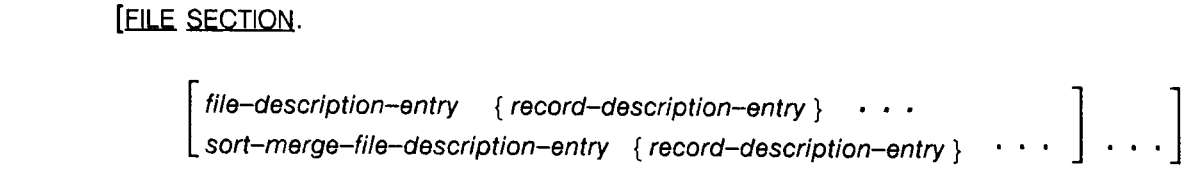
A file description entry always begins with the letters SD (for sort-merge files), or FD (for any other type of file as mentioned above) in Area A. It is followed by a space, the name of the file being described, and a sequence of input-output description clauses.

An FD file description furnishes information concerning the physical structure, identification, and record names pertaining to any type of file except a sort-merge file.

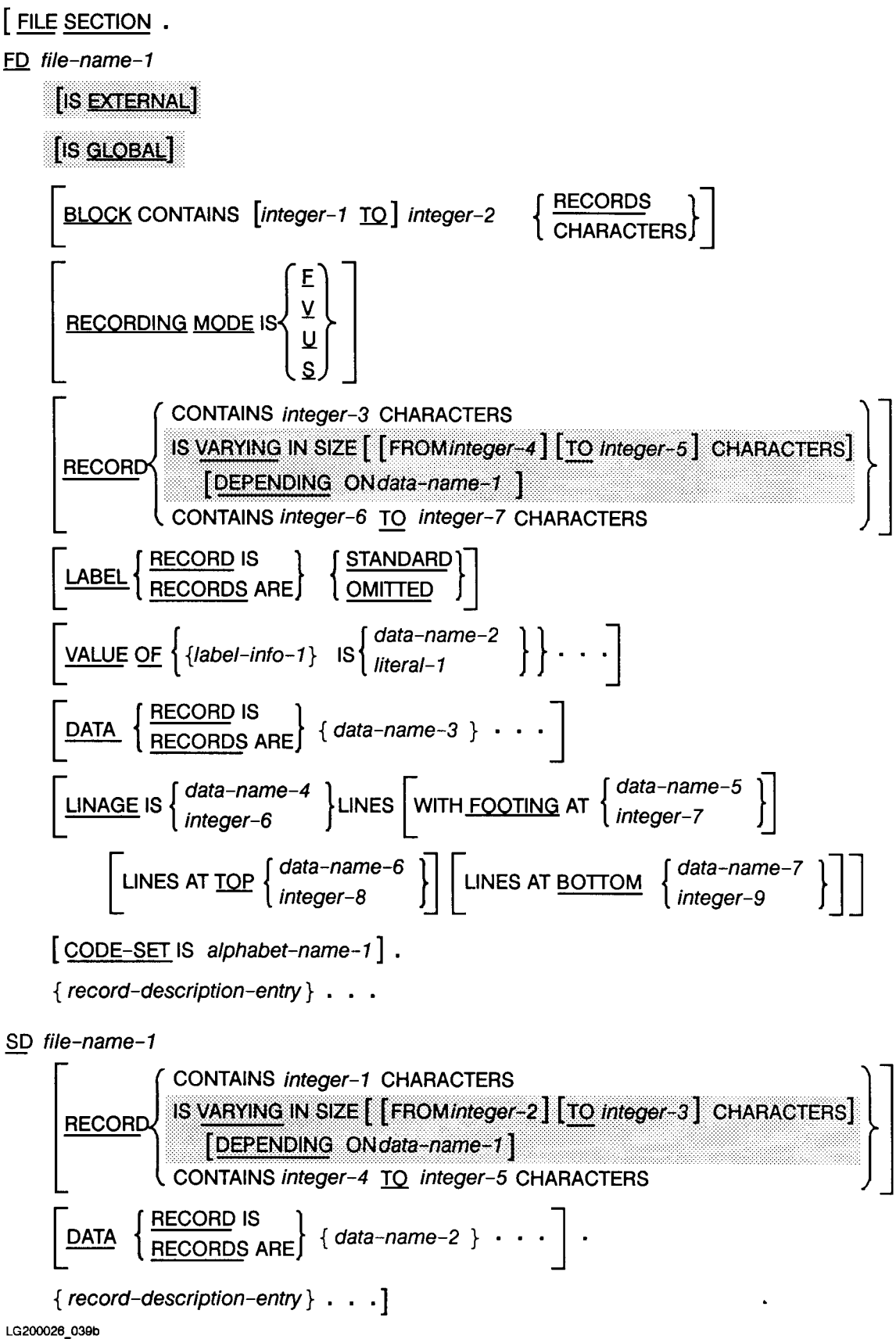
An SD file description provides information about the size and names of the data records in a file to be sorted or merged. Because you cannot control label procedures and because the blocking for the internal storage of sort-merge records is controlled by the SORT and MERGE operations, the only two clauses allowed in a file description for a sort-merge file are the RECORD CONTAINS and DATA RECORD IS clauses.

A record description always begins with a level number which, in the FILE SECTION, may be any number from 1 to 49, or the numbers 66 and 88. Level numbers of 66 and 88 have special usages associated with them, as described in Chapter 4. Record description entries are described in Chapter 4.

Note that record description entries can be used in every section and must be used in the FILE SECTION, following each file description appearing in that section. For a complete description of the FILE SECTION format refer to the "File Description Clauses" section later in this chapter.



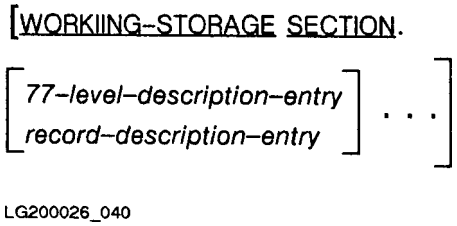
LG200026_197



LG200026_039b

WORKING-STORAGE SECTION

The WORKING-STORAGE SECTION consists of a section header, "WORKING-STORAGE SECTION", and one or more data description entries for noncontiguous data items, as well as record description entries. The general format of the WORKING-STORAGE SECTION is:



LG200026_040

Each data description entry within WORKING-STORAGE allocates memory space for the data item and associates a data name with the item.

You can use the WORKING-STORAGE SECTION to assign initial values to data items, define report headings, set up tables with initial values, define counters and accumulators, and so forth.

Level 77 and other data description entries are described in the following paragraphs.

LINKAGE SECTION

The LINKAGE SECTION consists of the header, "LINKAGE SECTION", and one or more data description entries for noncontiguous data items, as well as record description entries.

The LINKAGE SECTION in a program is meaningful only if the object program is to be called from another object program through a CALL statement, and the CALL statement in the calling program contains a USING phrase. See Chapter 11, "Interprogram Communication", for more information on how calling programs and called programs operate.

The LINKAGE SECTION is used to describe the data that is available from the calling program, but is to be used in both the calling program and the called program.

No space is allocated in the program containing a LINKAGE SECTION entry for the data items described in such an entry. PROCEDURE DIVISION references to these data items are resolved at object time by equating the reference in the called program to the location used in the calling program.

In the case of index names, no such correspondence is established. Index names are always local to their programs. Therefore, such names in the called and calling program always refer to separate indices. If index names are to be passed between programs, they should first be saved in a data item described in the WORKING-STORAGE SECTION as an index data item. A SET statement in the PROCEDURE DIVISION accomplishes this. Next, the index data item should be passed to the called program by the USING option of the CALL statement in the PROCEDURE DIVISION. Once the index data item is passed, a SET statement in the called program can be used to set the index value to the index name declared in the LINKAGE SECTION of the called program.

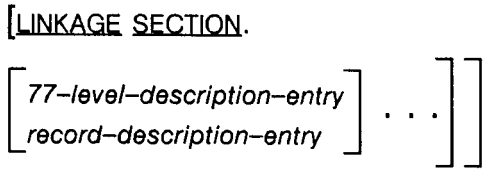
Data items defined in the LINKAGE SECTION of the called program may be referenced within the PROCEDURE DIVISION of the called program only under the following conditions:

- If they are specified as operands of the USING phrase of the PROCEDURE DIVISION header or ENTRY statement or are subordinate to such operands.
- If the object program is under the control of a CALL statement that specifies the USING phrase.

All data description clauses may be used in the LINKAGE SECTION, with the restriction that a VALUE clause can only be used in a condition name (level 88) entry.

The VALUE clause cannot be used in any other type of entry because no memory space is allocated for LINKAGE SECTION data items.

The LINKAGE SECTION has the following format:



LG200026_041

DATA DIVISION Clauses

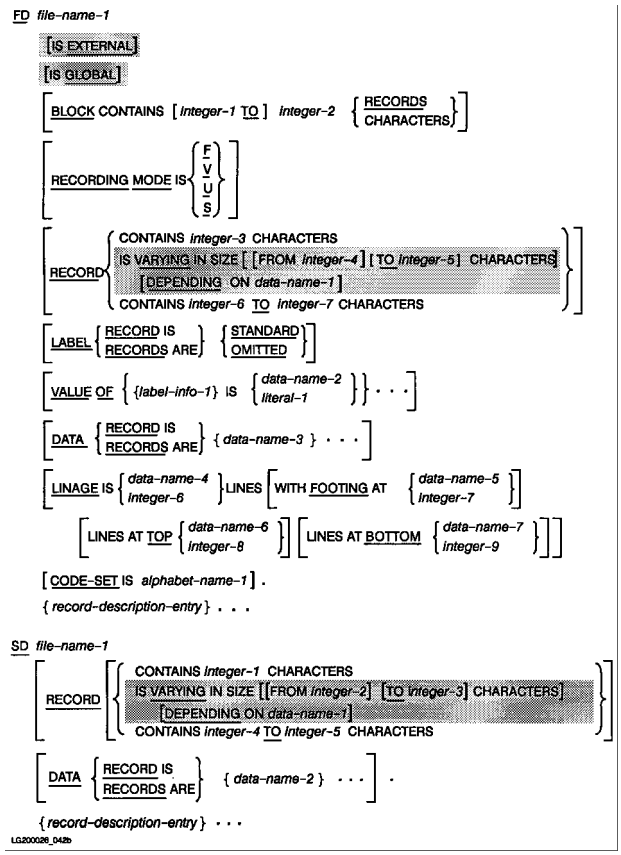
There are two distinct sets of clauses in the DATA DIVISION: file description clauses and data description clauses. File description clauses apply only to data files (FD level entries) and sort-merge files (SD level entries). Data description clauses can be used in any of the three sections allowed by HP COBOL II within the DATA DIVISION. They must be used in conjunction with each file description entry.

File Description Clauses

Each clause described on the following pages, if used, must be part of a file description entry. However, several of the clauses do not apply to sort-merge files (that is, to SD level indicator entries). Those that do not apply to SD level entries are noted as they are described.

The FILE SECTION must begin in Area A with the words FILE SECTION followed by a period. The header is followed by a level indicator (either FD or SD) and the name of the file being described. One or more record description entries must follow each file description entry. A file description entry is terminated by a period.

The general formats for the file description clauses are:


FD file-name-1
[IS EXTERNAL]
[IS GLOBAL]
[BLOCK CONTAINS [integer-1 TO] integer-2 {RECORDS / CHARACTERS}]
[RECORDING MODE IS {F / V / U / S}]
[RECORD {CONTAINS integer-3 CHARACTERS / IS VARYING IN SIZE [[FROM integer-4] [TO integer-5] CHARACTERS] [DEPENDING ON data-name-1] / CONTAINS integer-6 TO integer-7 CHARACTERS}]
[LABEL {RECORD IS / RECORDS ARE} {STANDARD / OMITTED}]
[VALUE OF {{label-info-1} IS {data-name-2 / literal-1}} ...]
[DATA {RECORD IS / RECORDS ARE} {data-name-3} ...]
[LINAGE IS {data-name-4 / integer-6} LINES [WITH FOOTING AT {data-name-5 / integer-7}]
[LINES AT TOP {data-name-6 / integer-8}] [LINES AT BOTTOM {data-name-7 / integer-9}]]
[CODE-SET IS alphabet-name-1].
{record-description-entry} ...
SD file-name-1
[RECORD {CONTAINS integer-1 CHARACTERS / IS VARYING IN SIZE [[FROM integer-2] [TO integer-3] CHARACTERS] [DEPENDING ON data-name-1] / CONTAINS integer-4 TO integer-5 CHARACTERS}]
[DATA {RECORD IS / RECORDS ARE} {data-name-2} ...].
{record-description-entry} ...
LG200026_042b

FD Level Indicator - For Data File Descriptions

The FD level indicator names the data file being described. It must be the first clause in a data file description entry.

Syntax

The data file description clause has the following format:

FD *file-name-1*

Parameters

`FD`	indicates that the clauses that follow are data file description clauses.
file-name-1	the name of the data file being described.

Description

The characters `FD` must begin in Area A. These characters are followed by a space and the name of the data file being described. Following the name, several clauses, mostly optional, are used to describe the file. At least one record description entry must follow the file description entry.

SD Level Indicator - For Sort File Descriptions

An SD level indicator names the file to be sorted or merged. It must be the first entry in a sort-merge file description entry.

Syntax

The format of the SD level indicator is:

SD *file-name-1*

Parameters

`SD`	indicates that the clauses that follow are used to describe a file to be sorted and/or merged.
file-name-1	the name of the file being described.

Description

The characters `SD` must begin in Area A. These characters are followed by a space and the name of the data file being described. The name may then be followed by a RECORD CONTAINS clause or DATA RECORDS clause, and must be followed by at least one record description entry.

BLOCK CONTAINS Clause

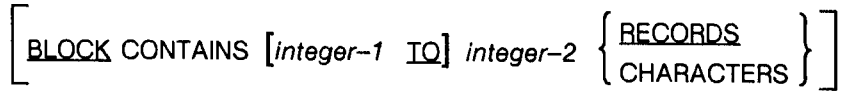
The BLOCK CONTAINS clause allows you to specify the blocking factor of the file being described. This clause should be used if the actual blocking factor of the file being described cannot be determined by the operating system.

For example, this clause is optional when the physical record contains exactly one complete logical record or, for sequential files only, when the physical device associated with the file is a unit-record device (such as a card reader).

When the BLOCK CONTAINS clause is omitted, the default blocking factor from the operating system is automatically assigned.

This clause does not apply to sort-merge files (SD level descriptions).

Syntax



LG200026_044

Parameters

integer-1	optional and must be positive. It refers to the minimum blocking factor size or to the minimum size of the physical record, depending upon whether the keyword RECORDS or CHARACTERS is used. Due to the way in which the operating system determines file attributes, this phrase is treated as a comment.
integer-2	required and must be positive. If used in conjunction with *integer-1*, it specifies the maximum size of the physical record; used alone, however, it specifies the exact size of the physical record, or the exact blocking factor.
`RECORDS`	specifies that the physical record size of the file is determined by its blocking factor.
`CHARACTERS`	when specified, is used to determine the blocking factor by dividing the value of *integer-2* by the logical record size. Refer to the RECORD CONTAINS clause description in this chapter.

Description

When the word CHARACTERS is used, the physical record size should be specified as a multiple of the maximum logical record size. Note that this logical record size must include any slack bytes generated by the compiler. Refer to the SYNCHRONIZED clause description in this chapter.

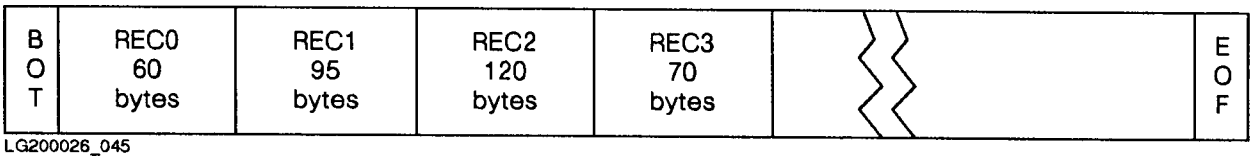
If logical records of differing size are grouped into one physical record, they are treated differently, according to the file's organization:

- Sequential or indexed file - The size of a logical record is variable, and is equal to the size of the record currently being accessed.

- Random and relative files require fixed length records.

To illustrate the use of the BLOCK CONTAINS clause:

- A magnetic tape file contains variable length logical records with a maximum of 120 characters, blocked with a minimum of four logical records per physical record. There is a maximum of 480 characters (bytes) per physical record, and a minimum of 60 bytes per physical record.



LG200026_045

In this case, you can use either:

```
BLOCK CONTAINS 4 RECORDS.
```

or:

```
BLOCK CONTAINS 60 TO 480 CHARACTERS.
```

Note The figures specified above are only used as estimates by the operating system. The actual blocking factor varies depending on the logical records, and the physical record size for variable records contains control information. Therefore, it is larger than specified in the BLOCK CONTAINS clause. This applies to relative files as well. Refer to the *MPE File System Manual* for more information on blocking factors.

- A random access disk file contains fixed length logical records of 206 bytes (103 words) each. Therefore, there are 50 unused bytes per sector. To minimize this waste, a blocking factor of six records can be used:

  ```
  BLOCK CONTAINS 6 RECORDS.
  ```

- A serial access disk file contains variable length logical records, ranging from 256 to 2560 bytes per record. The blocking factor is 10.

  You can use either:

  ```
  BLOCK CONTAINS 256 TO 2560 CHARACTERS.
  ```

  or:

  ```
  BLOCK CONTAINS 10 RECORDS.
  ```

CODE-SET Clause

The CODE-SET clause specifies the character code convention that represents data stored in sequential files. This clause may be specified for all files with sequential organization. This clause is optional, with ASCII being the default if it is not specified.

Syntax

```
[CODE-SET IS alphabet-name-1].
{record-description-entry} . . .
```

LG200026_046

Parameters

alphabet-name-1 — a previously defined name related to either EBCDIC, EBCDIK, STANDARD-1, STANDARD-2 or NATIVE. It must be specified in the ALPHABET clause of the SPECIAL-NAMES paragraph in the ENVIRONMENT DIVISION.

Description

If the CODE-SET clause is specified, *alphabet-name-1* indicates the character code convention used to represent the data on the related file. It further indicates the conversion routine to be used in translating the data into ASCII (when reading it) and translating data back into its original character code (when writing it from your program).

When the CODE-SET clause is used for a file, all data in that file must be described as USAGE IS DISPLAY, and any signed numeric data must be described with the SIGN IS SEPARATE clause.

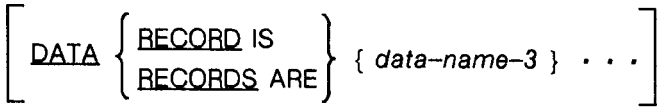
Note The HP utility FCOPY can be used to translate EBCDIC files containing records with elements that are other than USAGE DISPLAY. Refer to the *FCOPY Reference Manual* for more information.

DATA RECORDS Clause

The DATA RECORDS clause is an obsolete feature of the 1985 ANSI COBOL standard.

For any type of file (random, sequential, sort-merge, and so forth) the DATA RECORDS clause serves only to document the names of data records associated with a file. This clause is, therefore, optional for both FD and SD level file descriptions.

Syntax



LG200026_047

Parameters

data-name-2 and its subsequent occurrences are the names of data records.

Description

Use of more than one data name in this clause indicates that the file contains more than one type of data record. For instance, they might be of different size or format.

EXTERNAL Clause

The `EXTERNAL` clause is a feature of the 1985 ANSI COBOL standard.

The EXTERNAL clause specifies the external attributes of a file connector, the associated data records, and the associated data items.

Syntax

```
IS EXTERNAL
```

Description

If the file description entry for a sequential file contains the LINAGE clause and the EXTERNAL clause, the LINAGE-COUNTER data item is an external data item. FILLER cannot be specified for any record descriptions associated with a file description entry that contains the EXTERNAL clause. See Chapter 11, "Interprogram Communication", for more detailed information.

GLOBAL Clause

The `GLOBAL` clause is a feature of the 1985 ANSI COBOL standard.

The GLOBAL clause specifies that the file connector, the data records and associated data items are available to the contained programs within a nested program in which the file is declared global.

Syntax

```
IS GLOBAL
```

Description

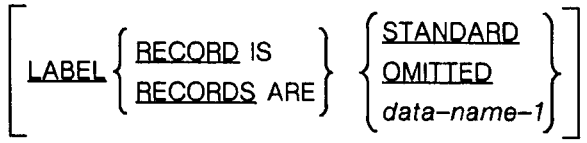
The GLOBAL clause can only be specified in the FD clause of the FILE SECTION. Refer to Chapter 11, "Interprogram Communication," for more detailed information.

LABEL RECORDS Clause

The LABEL RECORDS clause is an obsolete feature of the 1985 ANSI COBOL standard.

The LABEL RECORDS clause specifies whether one or more labels exist on the file and, optionally, the names of the records containing the label. This clause does not apply to sort-merge files. This clause is optional.

Syntax



LG200026_048

Parameters

`OMITTED` specifies that no explicit labels exist for the file or the device to which the file is assigned.

`STANDARD` specifies that labels exist for the file or the device to which it has been assigned, and that the labels conform to the operating system's label specification.

data-name-1 the name of the label record which must be described in a record description entry associated with the file. This record must not appear in the DATA RECORDS clause associated with the file. Use of this option indicates that user labels, as well as standard labels, are to be processed. All PROCEDURE DIVISION references to these names, or to any subordinate items, must appear within USE procedures. Label records for all files share the same area of memory.

Data-name-1 cannot be an external record. The *data-name-1* parameter is an HP extension to the ANSI COBOL standard.

Description

With HP COBOL II, it does not matter whether you specify that labels are STANDARD or OMITTED because the operating system processes standard labels before making the associated file available to your COBOL program.

If the file being described is an external file, all programs describing this file must have consistent LABEL RECORDS clauses.

LINAGE Clause

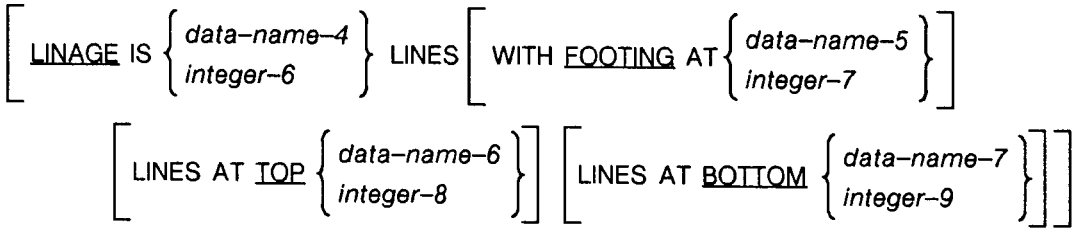
The purpose of the LINAGE clause is to describe the format of a logical page. It is used in conjunction with sequential files opened for output. Although there is not necessarily any relation between a logical and a physical page, it is advisable (particularly when writing a logical page to the line printer) to consider the size of a physical page when you are defining a logical one.

The LINAGE clause applies only to sequential files. It has no meaning for relative, random, indexed, or sort-merge files.

Its use is optional with sequential files, but it must be used if you intend to write records to the file using the END-OF-PAGE (or EOP) phrase of the WRITE statement.

A logical page consists of a top margin, page body, footing area, and bottom margin. Within a file, logical pages are contiguous. Figure 7-1 shows the concept of a logical page.

Syntax



LG200026_049

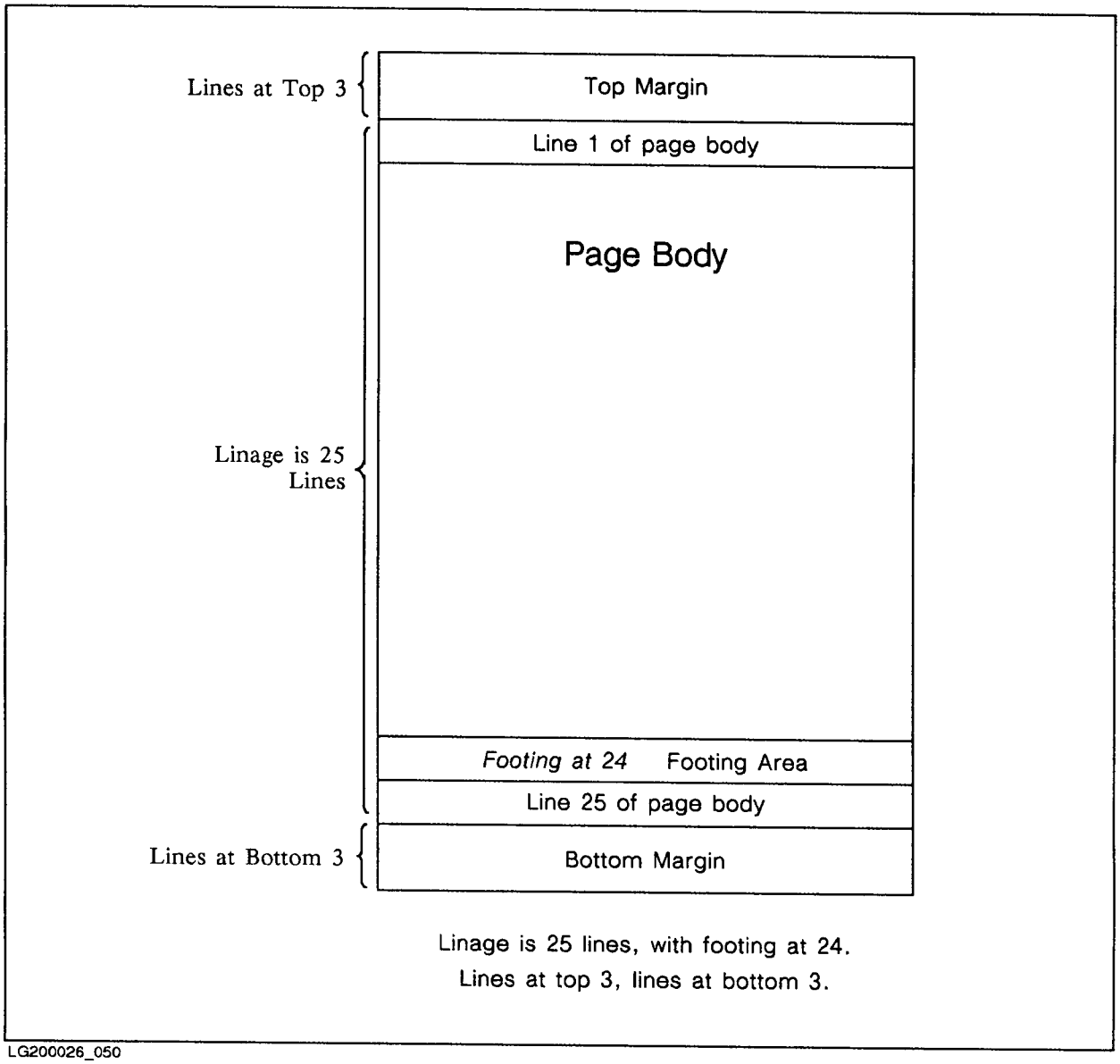
Parameters

data-name-4 through *data-name-7*	each reference an elementary unsigned integer data item.
integer-6	greater than zero.
integer-7	greater than or equal to zero and not greater than integer-6.
integer-8 and *integer-9*	each greater than or equal to zero.

Description

The LINAGE clause uses *data-name-4* or *integer-6* to define the number of lines in the page body. The page body is the area of the logical page in which lines can be written or spaced. Because a page size is being defined, *integer-6* (or the value associated with *data-name-4*) must be greater than 0.

If the file being described is an external file, and any file description entries for this file have a LINAGE clause, all file description entries in the run unit that describe this file must have a LINAGE clause. Also, the parameters must be either constants or external data items. Corresponding parameters must be the same in all LINAGE clauses for this file.



LG200026_050

Figure 7-1. Example of the LINAGE Clause and its Logical Representation

FOOTING Phrase

The FOOTING phrase is optional. If specified, however, it uses *integer-7* or *data-name-5* to define the FOOTING AREA of the page body.

The entire page body can be the footing area. That is, *integer-7* (or, the value of *data-name-5*) may be 1, in which case the footing area is all of the page body.

The footing area is used in conjunction with the END-OF-PAGE phrase of the WRITE statement. It signifies that the end of the logical page has been reached.

If you do not use the FOOTING phrase, then the only way that an end of page condition can occur is for a WRITE statement to attempt to write a record beyond the end of the logical page body (that is, when a page overflow condition exists).

LINES AT TOP and LINES AT BOTTOM Phrases

The LINES AT TOP and LINES AT BOTTOM phrases are optional. They are used to specify a top margin and a bottom margin, respectively, for the logical page. If neither phrase is used, the margins are assumed to be zero.

If THE LINES AT TOP phrase is specified, it uses *integer-8* or *data-name-6* to specify the number of lines in the top margin.

If THE LINES AT BOTTOM phrase is specified, it uses *data-name-7* or *integer-9* to specify the number of lines in the bottom margin.

The top and bottom margins are distinct from the page body; therefore, no data may be written into them.

Use of Data Names Versus Use of Integers

The use of integers in a LINAGE clause allows for less flexibility than does the use of data names.

When an integer (either *integer-6, integer-7,* or *integer-8*) is specified, it is used when the file associated with the LINAGE clause is opened for output. This value is used for every logical page written for the file and cannot change during a particular execution of the program in which it appears.

The values of data names, on the other hand, can vary during the execution of a program. Therefore, the values are checked and used not only when the associated file is open for output, but also whenever a WRITE statement containing an ADVANCING PAGE phrase is executed, or a WRITE statement is executed and a page overflow condition occurs.

Taking each of these cases in turn:

- When the file is opened for output, the current values of *data-name-4, data name-5*, *data-name-6*, and *data-name-7* are used to define their associated sections of the FIRST logical page only.
- When a WRITE statement is executed, and the ADVANCING PAGE phrase is activated, the current values of *data-name-4, data name-6* and *data-name-7* are used to define the page body, top and bottom margins of the next logical page.
- If a footing area has been defined, the ADVANCING PAGE phrase is activated when the WRITE statement in which it appears attempts to write data into the footing area. In this case, the data is written into the footing area of the current logical record and the current value of *data-name-5* is then used to define the footing area for the next logical page.
- When a WRITE statement is executed and a page overflow condition occurs, thus forcing an end-of-page condition, the current values of *data-name-4, data name-6*, and *data-name-7* are used to define their associated parts of the next logical page.

This type of end-of-page condition implies that either the value of *data-name-5* is the same as that of *data-name-4* or that a footing area was not defined (the two are equivalent).

In either case, the data to be written is placed in the first available line of the next logical record (depending upon whether the BEFORE or AFTER ADVANCING phrase was used in the WRITE statement).

If a footing area has been defined, the current value of *data-name-5* is then used to define the footing area of this logical record.

LINAGE-COUNTER

Any time a LINAGE clause is specified for a file, a LINAGE-COUNTER is supplied for the file.

Because you can have more than one file whose description contains a LINAGE clause, you must qualify the LINAGE-COUNTER of each file by using the file name.

The value of a LINAGE-COUNTER at any given time is the current line number of the associated page body. This value ranges from one, for the first line of a page body, to *integer-6* (or the value of *data-name-4*). You can reference a LINAGE-COUNTER in the PROCEDURE DIVISION, but cannot change it.

Each time a record is written to a logical page, the associated LINAGE-COUNTER is incremented according to the following rules:

- When the file associated with LINAGE-COUNTER is first opened, LINAGE-COUNTER is set to one.
- If the ADVANCING phrase of the WRITE statement is not specified, LINAGE-COUNTER is incremented by one when the WRITE statement is executed.
- If the ADVANCING phrase is used with a WRITE statement, and
  - is of the form, ADVANCING *integer-1* or ADVANCING *identifier-2*, LINAGE-COUNTER is incremented by *integer-1* (or the value of *identifier-2*) when the WRITE statement is executed.
  - is of the form ADVANCING PAGE, LINAGE-COUNTER is reset to one.
- If a new logical page is to be written upon, LINAGE-COUNTER is reset to one.

RECORD CONTAINS Clause

The RECORD CONTAINS clause specifies the size, in characters, of data records in a file. Because each data record of a file is completely defined in a record description entry, this clause is optional for any file description entry.

Syntax

The RECORD CONTAINS clause has the following three formats:

Format 1

RECORD CONTAINS *integer–1* CHARACTERS

Format 2

RECORD IS VARYING IN SIZE [[FROM *integer–2*] [TO *integer–3*] CHARACTERS]
[DEPENDING ON *data–name–1*]

Format 3

RECORD CONTAINS *integer–4* TO *integer–5* CHARACTERS

LG200026_051

Parameters

integer-1	specifies the number of characters contained in each record of the file.
integer-2	specifies the minimum number of character positions to be contained in any record of the file.
integer-3	specifies the maximum number of character positions to be contained in any record of the file.
integer-4	specifies the minimum number of characters in the smallest size data record.
integer-5	specifies the maximum number of characters in the largest size data record.
data-name-1	must be an elementary unsigned integer in the WORKING-STORAGE or LINKAGE section.

Description

The size of a record is determined by taking the sum of the numbers of all characters in all fixed length elementary items, and adding to that sum the maximum number of characters in any variable length item subordinate to the record.

This sum may differ from the actual size of the record because of slack bytes. Refer to the SYNCHRONIZED and USAGE clause descriptions appearing later in this chapter.

If the RECORD clause is not specified in all formats, the size of each data record is completely defined in the record description entry.

Fixed Length Records

Format 1 is used to specify fixed length records. *integer-1* specifies the number of character positions contained in each record in the file.

Variable Length Records

Format 2 is used to specify variable length records.

The number of character positions associated with a record description is determined by the sum of the number of character positions in all elementary data items excluding redefinitions and renamings, plus any implicit FILLER due to synchronization.

If a table is specified, the minimum or maximum number of table elements described in the record is used in the summation above to determine the minimum or maximum number of character positions associated with the record description.

If *integer-2* is not specified, the minimum number of character positions to be contained in any record of the file is equal to the least number of character positions described for a record in that file.

If *integer-3* is not specified, the maximum number of character positions to be contained in any record of the file is equal to the greatest number of character positions described for a record in that file.

If *data-name-1* is specified, the number of character positions in the record must be placed into the data item referenced by *data-name-1* before any RELEASE, REWRITE, or WRITE statement is executed for the file.

If *data-name-1* is specified, the execution of a DELETE, RELEASE, REWRITE, START, or WRITE statement or the unsuccessful execution of a READ or RETURN statement does not alter the content of the data item referenced by *data-name-1*.

During execution of a RELEASE, REWRITE, or WRITE statement, the number of character positions in the record is determined by the following three conditions:

- If *data-name-1* is specified, by the content of the data item referenced by *data-name-1*.
- If *data-name-1* is not specified and the record does not contain a variable occurrence data item, by the number of character positions in the record.
- If *data-name-1* is not specified and the record does contain a variable occurrence data item, by the sum of the fixed portion and that portion of the table described by the number of occurrences at the time of execution of the output statement.

If *data-name-1* is specified, after the READ or RETURN statement for the file successfully executes, the contents of the data item referenced by *data-name-1* indicate the number of character positions in the record just read.

If the INTO phrase is specified in the READ or RETURN statement, the number of character positions in the current record that participate as the sending data items in the implicit MOVE statement are determined by the following two conditions:

- If *data-name-1* is specified, by the content of the data item referenced by *data-name-1*.
- If *data-name-1* is not specified, by the value that would have been moved into the data item referenced by *data-name-1*.

Format 2 is the preferable way to specify variable length records. If format 3 is used, the RECORDING MODE clause must also be specified.

In format 3 of the RECORD clause, the size of each data record is completely defined in the record description entry.

The size of each data record is specified in terms of the number of character positions required to store the logical record, regardless of the types of characters used to represent the items within the logical record. The size of a record is determined by the sum of the number of characters in all fixed length elementary items plus the sum of the maximum number of characters in any variable length item subordinate to the record. This sum can be different from the actual size of the record.

Example

The following example illustrates use of the RECORD VARYING clause.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBVAR.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT IFILE  ASSIGN TO "IFILE".
SELECT IFILE2 ASSIGN TO "IFILE".
DATA DIVISION.
FILE SECTION.
FD IFILE
   RECORD IS VARYING FROM 10 TO 50 DEPENDING ON LEN.
01 IREC.
  05 FILLER  PIC X OCCURS 10 TO 50 TIMES DEPENDING ON LEN.
FD IFILE2
   RECORD IS VARYING FROM 10 TO 50.
01 IREC2     PIC X(50).
WORKING-STORAGE SECTION.
01 LEN       PIC S9(4) BINARY.
01 LEN-ED    PIC ++++9.
01 WREC      PIC X(50).
PROCEDURE DIVISION.
P1.
  DISPLAY "EXAMPLE 1 ODO REC"
  OPEN INPUT IFILE
  PERFORM UNTIL LEN = -1
    READ IFILE
      AT END MOVE -1 TO LEN
      NOT AT END
        DISPLAY IREC
        MOVE LEN TO LEN-ED
        DISPLAY "Length is ", LEN-ED
    END-READ
  END-PERFORM
  CLOSE IFILE
  DISPLAY SPACE

  DISPLAY "EXAMPLE 2 FIXED REC"
  OPEN INPUT IFILE2
  MOVE ALL "X" TO IREC2
  READ IFILE2 AT END MOVE -1 TO LEN
  END-READ
  DISPLAY IREC2
  DISPLAY SPACE

  DISPLAY "EXAMPLE 3 READ INTO WREC"
  MOVE ALL "X" TO IREC2 WREC
  READ IFILE2 INTO WREC AT END MOVE -1 TO LEN
  END-READ
```

```
    DISPLAY IREC2
    DISPLAY WREC
    CLOSE IFILE2.
```

If IFILE contains the following data:

```
1234567890
123456789*123456789*
123456789*123456789*123456789*
```

The program will produce the following output:

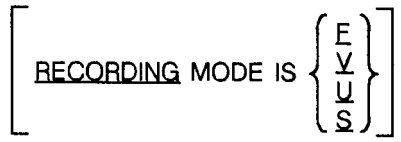
```
EXAMPLE 1 ODO REC
1234567890
Length is   +10
123456789*123456789*
Length is   +20
123456789*123456789*123456789*
Length is   +30

EXAMPLE 2 FIXED REC
1234567890XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

EXAMPLE 3 READ INTO WREC
123456789*123456789*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
123456789*123456789*
```

RECORDING MODE Clause

The RECORDING MODE clause is an HP extension to the ANSI COBOL standard.

The RECORDING MODE clause specifies how logical records are contained in the file, and whether or not the logical record being read or written spans more than one physical record (generally because of hardware constraints or for I-O efficiency). This clause does not apply to sort-merge files. This clause is optional.

Syntax



LG200026_052

Parameters

F specifies fixed length logical records. This implies that no OCCURS DEPENDING on clause can be associated with any record description entry for the file. Also, if more than one record description entry is supplied for the file, all record lengths calculated from the record descriptions must be the same. This option is the only one that is valid for random access and relative files.

V specifies variable length logical records.

U specifies undefined length logical records. This kind of file cannot be blocked. Therefore, the BLOCK CONTAINS clause need not be used for this kind of file.

S enables the MULTI-RECORD (or more accurately, "multi-block") option. This option allows the reading or writing of a single logical record across more than one physical record.

Description

If none of the above codes is specified, an F is the default value.

Logical records are contained in files as either fixed, variable, or undefined in length.

A fixed length record file contains logical records whose lengths are all the same.

A variable length record file contains logical records whose lengths may vary. In such a file, each record is preceded by a two-byte count, which specifies the length of that particular record.

An undefined length record file contains logical records of undetermined length. In such a file, each logical record is equivalent to one physical record, and a physical record is as long as the longest possible logical record in the file.

To clarify the case of logical records spanning more than one physical record, assume you want to read logical records of 128 characters each from a card reader.

Each card represents a physical record of 80 characters. Therefore, to read one logical record, you must read two physical records.

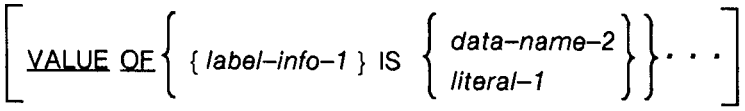
In such a case, you could specify the recording mode as equal to S, the operating system's multi-record option.

VALUE OF Clause

The VALUE OF clause is an obsolete feature of the 1985 ANSI COBOL standard.

The VALUE OF clause allows you to access existing files on labeled tapes or to create a new labeled tape. A label contains identification, whether the label is in IBM or ANSI standard format, the expiration date of the file protection, and the position of the file on the tape. This clause does not apply to sort-merge files.

Syntax



LG200026_053

Parameters

label-info-1 specifies one of the following fields: VOL, LABELS, SEQ, or EXDATE. Each of these fields is described in Table 7-1 on the following page.

data-name-2 must be described in the WORKING-STORAGE SECTION. This name can be qualified, but cannot be subscripted, indexed, or described with the USAGE IS INDEX clause in a data description. It is used to specify the value of the associated label-info entry. The possible values are shown in Table 7-1 on the following page.

literal-1 a COBOL literal or a figurative constant.

Description

The data name associated with VOL can specify a data item of any category, but must consist of a maximum of six characters or digits. The data name associated with SEQ can also specify a data item of any category, but must consist of a maximum of four digits or characters. The data names associated with LABELS and EXDATE must name alphanumeric data items; the respective picture strings for LABELS must describe data that is three characters long, EXDATE that is eight characters long and of the form, mm/dd/yy.

All VALUE OF clauses for each external file in the run unit must be consistent.

Table 7-1.
Values of the LABEL INFO and DATA NAME Parameters in the VALUE OF Clause

label-info-n	Meaning	data-name-n or literal-n
VOL	Volume identification.	Any combination of one to six characters from the set A through Z, and 0 through 9.
LABELS	ANSI standard or IBM format.	ANS or IBM.
SEQ	Relative position of file on a magnetic tape.	0 to 9999, NEXT, or ADDF.
EXDATE	Date when file may be written over. Until that time, the file is protected.	Date, in the form month/day/year. The default is 00/00/00.

Example

```
DATA DIVISION.

FILE SECTION.
FD TAPEFL
VALUE OF VOL IS "JTAPE1", LABELS IS "ANS",
         SEQ IS 10, EXDATE IS "02/25/85".
               ⋮
FD NEW-TAPE
VALUE OF VOL IS "JTAPE2", LABELS IS "ANS",
         SEQ IS "ADDF", EXDATE IS "02/25/85".
               ⋮
PROCEDURE DIVISION.
    DISPLAY "PLEASE MOUNT NEW TAPE FOR JTAPE2" UPON CONSOLE.
    OPEN OUTPUT NEW-TAPE, INPUT TAPEFL.
                .
                .
                .
```

Assuming that `NEW-TAPE` names a new file, when the OPEN statement above is executed, the information given in the VALUE OF clause is used to write a label for the tape volume.

When `TAPEFL` is opened, the specification of 10 for the `SEQ` value causes the tape to automatically be placed at the beginning of the tenth file on the volume named `JTAPE1`.

Note that a message requesting that the volumes `JTAPE2` and `JTAPE1` be mounted is displayed on the operator's console. Because `JTAPE2` is a new volume, the DISPLAY statement above was used to tell the operator that `JTAPE2` does not already exist.

Data Description Entries

A data description entry is composed of a level number followed by a data name, and then followed by a set of data clauses.

The level numbers can be 01 to 49 for record description entries, 77 for unrelated data items, 66 for alternative groupings of elementary items in the preceding record description entry, and 88 for condition names.

A level number is required for each data description entry, and must be the first element of such an entry.

The level numbers 01 and 77 must begin in Area A. Other level numbers may begin in either Area A or Area B.

All data description entries are discussed in the following pages, beginning with level 77 entries.

77 Level Description Entries

Noncontiguous data items are data items or constants that are not subdivided and bear no hierarchical relationship to one another. That is, noncontiguous data items are unrelated data items.

These data items are only defined in the WORKING-STORAGE and LINKAGE SECTIONs, and have level numbers of 77. Recall, however, that those items defined in the LINKAGE SECTION do not have any storage allocated for them.

Each name used for a noncontiguous data item must be unique since it cannot be qualified. For example,

```
WORKING-STORAGE SECTION.

      77 COUNTER                  PICTURE 9(4) VALUE 0.
      77 MID-TOTALS               PICTURE 9(6)V99 VALUE 0.
```

`COUNTER` is an accumulator, and `MID-TOTALS` is an intermediate storage variable. Because they are not subdivided, and they have no immediate relationship to any other data items, they are described using level 77 entries.

Record Description Entries

At least one record description entry must follow each FD or SD file description entry. This discussion of record description entries is applicable to either type of file description, as well as for the LINKAGE and WORKING-STORAGE SECTIONS.

A record description consists of one or more data description entries.

Associated with each data description entry is a level number chosen from the set 01 to 49, 66, or 88. The level number for the first data description entry of any record description must be 01 (or simply 1). Succeeding level numbers of data description entries for the same record description may range from 01 to 49, or may be 66. If, however, multiple 01 entries are used for a given level indicator (FD, or SD) they represent implicit redefinitions of the same area. The 02 to 49 level numbers are used to specify subsets of the characters of a record.

The level number 66 is used only when the data description uses the RENAMES clause to regroup data items. Refer to the description of the RENAMES clause later in this chapter.

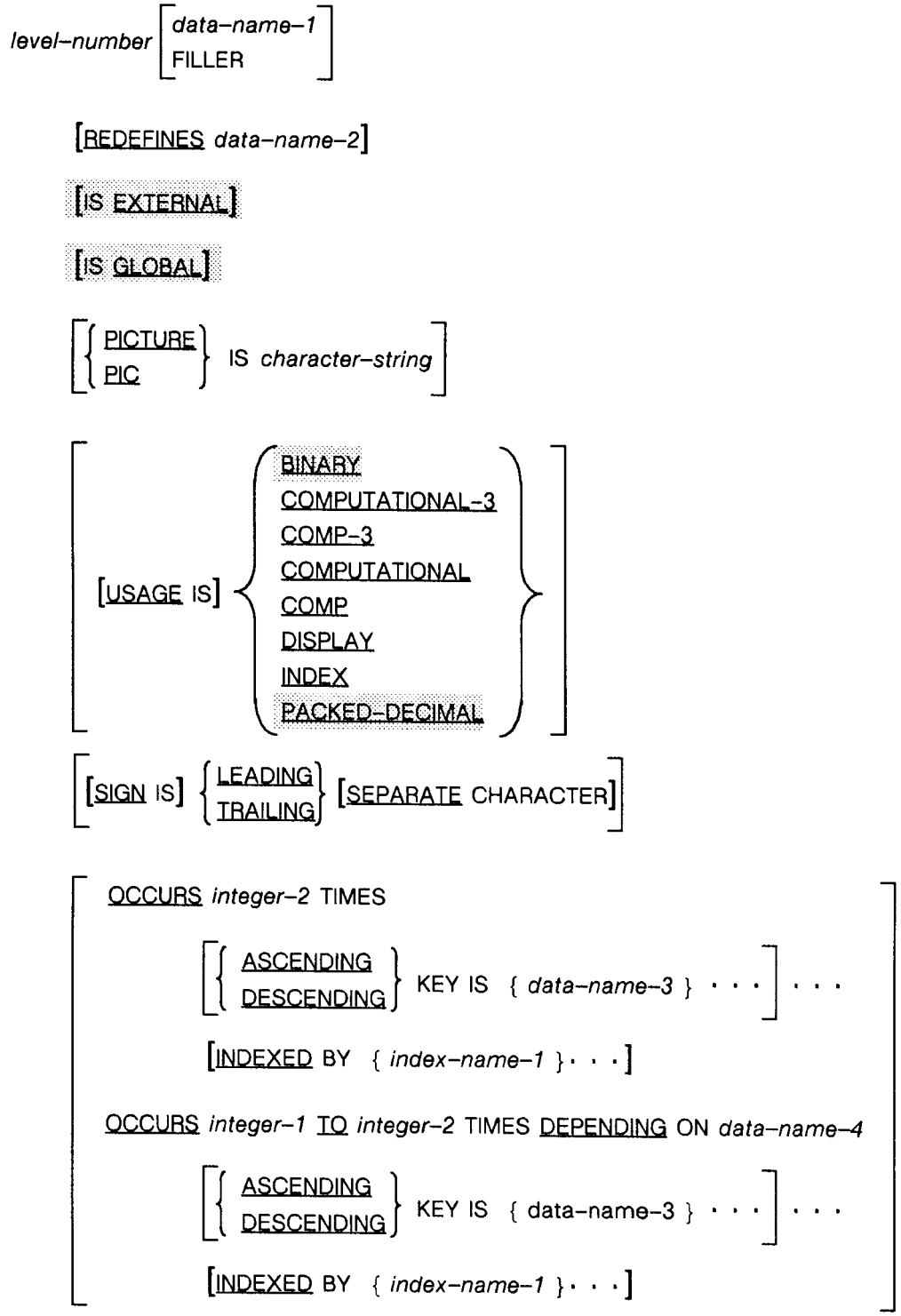
The level number 88 is used only for a condition name data description. It is always associated with a VALUE clause. Refer to "Condition Names" later in this chapter.)

Data names subordinate to record names can be nonunique, provided they can be made unique by qualification.

The three general formats for data description entries are shown below and described in the following paragraphs.

Data Description Format

Format 1



LG200026_054a

```
[ { SYNCHRONIZED } [ LEFT  ] ]
[ { SYNC        } [ RIGHT ] ]

[ { JUSTIFIED } RIGHT ]
[ { JUST      }       ]

[BLANK WHEN ZERO]

[VALUE IS literal-1].
```

Format 2

```
66 data-name-1 RENAMES data-name-2 [ { THROUGH } data-name-3 ] .
                                   { THRU    }               
```

Format 3

```
88 condition-name-1 { VALUE IS  } { literal-1 [ { THROUGH } literal-2 ] } . . .  .
                    { VALUES ARE }            [ { THRU    }           ]
```

LG200026_055

In the first format, the level number may be any number from 01 to 49, or 77. It cannot be 66 or 88.

If it is used, the data name or FILLER clause must immediately follow the level number, and the REDEFINES clause (if used) must immediately follow the *data-name-1* clause. Except for these two conditions, all other clauses, if used, may be written in any order.

The PICTURE clause must be used for every elementary item except an index data item. A PICTURE clause must not be used for an index data item.

Data elements and constants bearing a definite hierarchic relationship to one another must be grouped into records.

The initial value of any data item except an index data item is specified by using the VALUE clause in the description of that item. Because the VALUE clause does not apply to index data items, the initial value of any such item is unknown.

You may not use the SYNCHRONIZED, PICTURE, JUSTIFIED, or BLANK WHEN ZERO clauses for any but elementary data items.

Except for the data name or filler clause, which is described first, all other data description clauses are described in alphabetical order on the following pages.

Data Name or FILLER Clause

A data name clause specifies the name of the data being described. The keyword, FILLER, implies that you are specifying an elementary item of the logical record being described that cannot be referenced explicitly. If this clause is omitted, the record is treated as though FILLER had been specified.

In the FILE, WORKING-STORAGE, and LINKAGE SECTIONS, a data name must be the first word following the level number in each data description entry.

Syntax

level-number [*data-name-1* | FILLER]

Parameters

data-name-1 must be a valid user-defined COBOL word.

Description

Although you may not refer to a FILLER item explicitly, the keyword FILLER may be used as a conditional variable (format 3) because the use of it in this manner does not require explicit reference to the FILLER item, but to its value.

BLANK WHEN ZERO Clause

The BLANK WHEN ZERO clause causes a numeric or numeric-edited data item to be filled with spaces when the value of the data item is zero.

This clause is optional. If used, it may refer only to a numeric or numeric-edited elementary data item. When this clause is used with a numeric data item, the category of the data item is considered to be numeric-edited.

Syntax

```
[BLANK WHEN ZERO]
```

Note This clause cannot be used for a numeric-edited data item whose PICTURE uses asterisks for zero suppression and replacement.

EXTERNAL Clause

The `EXTERNAL` clause is a feature of the 1985 ANSI COBOL standard.

The EXTERNAL clause specifies external attributes of a WORKING-STORAGE SECTION data description entry, the associated data record, and its subordinate data items.

Syntax

```
IS EXTERNAL
```

Description

The EXTERNAL clause can only be specified in data description entries in the WORKING-STORAGE SECTION whose level number is 01.

The EXTERNAL clause and the REDEFINES clause must not be specified in the same data description entry.

FILLER cannot be specified for any entry containing the EXTERNAL clause. Refer to Chapter 11, "Interprogram Communication", for more detailed information.

GLOBAL Clause

The `GLOBAL` clause is a feature of the 1985 ANSI COBOL standard.

The GLOBAL clause specifies that a 01 record description and its subordinate data items are available to all contained programs within a nested program in which the record is declared global. Refer to Chapter 11, "Interprogram Communication," for more detailed information.

Syntax

```
IS GLOBAL
```

Description

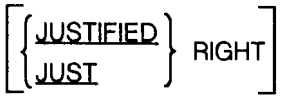
The GLOBAL clause can be specified in data description entries in the WORKING-STORAGE SECTION where the level number is 01.

FILLER cannot be specified for any entry containing the EXTERNAL clause. Refer to Chapter 11, "Interprogram Communication", for more detailed information.

JUSTIFIED Clause

The JUSTIFIED clause allows you to right-justify alphabetic or alphanumeric data items. It cannot be used with numeric or edited data items, and only applies to elementary data items being used to receive data. This clause is optional.

Syntax



LG200026_058

Parameters

JUST is an abbreviation for JUSTIFIED.

Description

Data is moved from a sending data item to the right justified receiving data item starting with the rightmost character of the sending data item. The rightmost character is placed in the rightmost character of the receiving data item. The next rightmost data item of the sending data item is then moved to the next rightmost character of the receiving data item. This process continues until either all of the sending data item has been moved, or the receiving data item is full. Note that a space in the sending data item is considered a valid character, no matter where it is within the sending data item. That is, spaces are not stripped from the sending item, even if they are in the rightmost positions of the sending item.

When a receiving data item is described using this clause, and the sending data item is larger than the receiving item, the leftmost characters are truncated.

When the receiving data item is longer than a sending item, the data is aligned at the rightmost character position in the receiving field, and unused characters to the left are filled with spaces.

If the JUSTIFIED clause is not used, standard rules for aligning data within an elementary item are used.

Example

Sending data item: `HEWLETT-PACKARD COBOL II␣␣`

Receiving data item: `01 INFIRST  PIC X(29)  JUSTIFIED RIGHT.`

Resulting data item: `␣␣␣␣␣HEWLETT-PACKARD COBOL II␣␣`

Assuming the same sending data item, but using a new receiving data item:

Receiving data item: `01 NEWIN PIC X(14) JUSTIFIED RIGHT.`

Resulting data item: `KARD COBOL II␣␣`

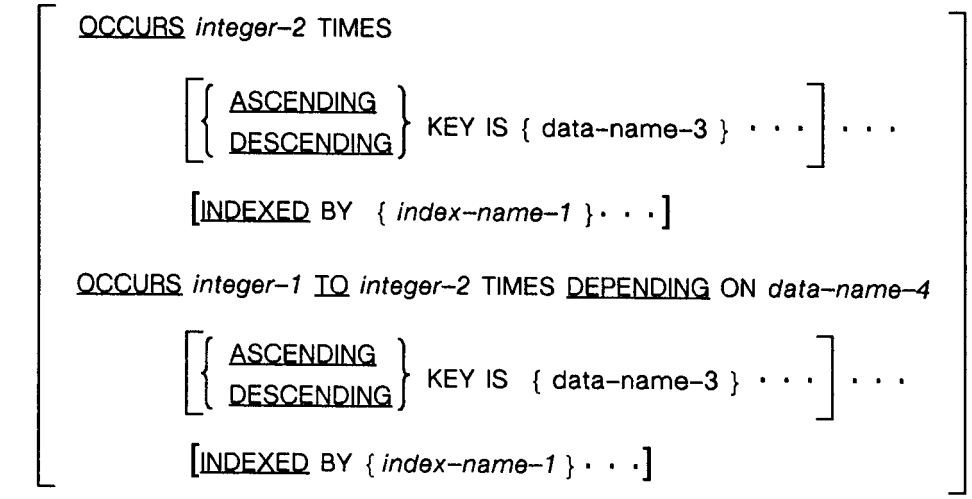
OCCURS Clause

The OCCURS clause is used to define a table containing up to seven dimensions. Its use eliminates the need for separate entries to describe repeated data items, and provides information required for the application of subscripts or indices.

Refer to Chapter 4, "Tables", for a description of the formation and use of tables.

Syntax

This clause has the following two general formats:



LG200026_059

The two formats of the OCCURS clause are described in the following paragraphs. The first format of the OCCURS clause is used to define a table of fixed length data items.

Parameters to Format 1

integer-2	specifies the exact number of occurrences of the item being described.
KEY IS	phrase indicates that the repeated data is arranged in ASCENDING or DESCENDING order according to the values contained in *data-name-3*.
data-name-3	must either be the name of the entry containing the OCCURS clause or the name of an entry subordinate to the entry containing the OCCURS clause (when first specified). Subsequent specification of *data-name-3* must be subordinate to the entry containing the OCCURS clause. *Data-name-3* may be qualified.
INDEXED BY	specifies one or more index names to be used when reference to the subject of this entry or items subordinate to it is done by indexing.
index-name-1	not defined elsewhere in a program, and cannot be associated with any data hierarchy. Index names must be unique within a given program.

The data names referred to by *data-name-3* must be listed in their descending order of significance. If *data-name-3* is not the same name as the item being described, then all of

the items identified by data names in this phrase must be within the group item that is the subject of this entry, and must not contain an OCCURS clause.

There must not be any entry containing an OCCURS clause between the items identified in the KEY IS phrase and the subject of this entry.

The second format of the OCCURS clause is used to define a variable length table.

Parameters to Format 2

integer-1 represents the minimum number of occurrences of the subject of the OCCURS clause. *Integer-1* must be less than *integer-2*.

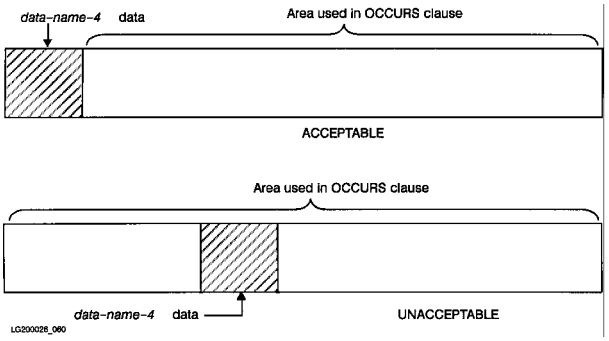
integer-2 represents the maximum number of occurrences. This implies that only the number of occurrences is variable and not the length of the data item.

data-name-4 represents an integer used to determine the number of occurrences of data items within the table. Therefore, the current value of the data item referenced by *data-name-4* represents the number of occurrences.

A data description entry containing this format of the OCCURS clause can only be followed, within the one record description, by data description entries subordinate to it.

Data-name-4 must not be contained in that part of the record being described that starts at the first character of the first element of the table and continues to the end of the record.

To illustrate, the two representations of records below show an allowable, and an unacceptable placement of the item referenced by *data-name-4* in a record.



Data-name-4, and its contents, must be described in a separate data description entry, and may be qualified.

If the OCCURS clause is specified in a data description entry included in a record description entry containing the EXTERNAL clause, *data-name-4*, if specified, must reference a data item possessing the external attribute that is described in the same DATA DIVISION.

If the OCCURS clause is specified in a data description entry subordinate to one containing the GLOBAL clause, *data-name-4*, if specified, must be a global name and must reference a data item that is described in the same DATA DIVISION.

Because the current value of the data item referenced by *data-name-4* represents the number of occurrences of a data item, it must be an integer in the range of values from *integer-1* to *integer-2*.

If the value of the integer represented by *data-name-4* is reduced from a previous value, the contents of those data items whose occurrence numbers exceed this new value are unpredictable.

When a group item has a subordinate entry that specifies format 2 of the OCCURS clause, only that part of the group item specified by the value of *data-name-4* is used.

The KEY IS and INDEXED BY phrases follow the rules given for format 1.

The OCCURS clause cannot be specified in a data description entry having an 01, 66, 77, or 88 level number.

The OCCURS clause cannot be used in a description entry for an item whose size is variable. The size of an item is variable if the data description of any subordinate item contains format 2 of the OCCURS clause. In other words, this restriction means that no OCCURS clause using the DEPENDING ON phrase can be used in the description of an item subordinate to an item that also uses either format of the OCCURS clause.

Examples

For example, the following two records are allowed:

```
01  ROAD.
    02  SURFACE OCCURS 1 TO 12 TIMES
        DEPENDING ON SIZER.
        03  SIDE OCCURS 10 TIMES PIC X(9).      Allowed.
01  ROAD.
    02  SURFACE OCCURS 10 TIMES.
        03  SIDE OCCURS 4 TIMES PIC X(9).
```

But, the following two records are not allowed:

```
01  ROAD.                                    Not allowed.
    02  SURFACE OCCURS 10 TIMES.
        03  SIDE OCCURS 1 TO 8 TIMES
            DEPENDING ON SIZER PIC X(9).
01 ROAD.
   02  SURFACE OCCURS 1 TO 10 TIMES
           DEPENDING ON SIZER.               Not allowed.
       03  SIDE OCCURS 1 TO 8 TIMES
           DEPENDING ON SIZEUP PIC X(9).
```

The name of the data item being described must be either subscripted or indexed whenever it is referred to in any way other than with the SEARCH or USE FOR DEBUGGING statements. Also, if the name of the data item being described is the name of a group item, then all data names belonging to the group must be subscripted or indexed whenever they are used as operands, except as the object of a REDEFINES clause.

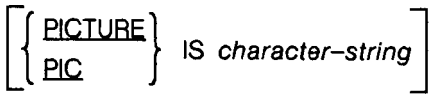
Except for the OCCURS clause itself, all data description clauses associated with an item whose description includes an OCCURS clause apply to every occurrence of the item being described.

If a data item possessing the global attribute includes a table accessed with an index, that index also possesses the global attribute. See Chapter 11, "Interprogram Communication," for more detailed information.

PICTURE Clause

The PICTURE clause describes the category, length, and editing requirements for an elementary item. It applies only to elementary data items and must be used for every elementary data item. It is not allowed for an index data item.

Syntax



LG200026_061

Parameters

`PICTURE` and `PIC`	equivalent.
character-string	a set of up to 30 characters, arranged in certain allowable combinations; these combinations determine the category of the elementary item.

Description

You can define any of five categories of data with the PICTURE clause. The five categories of data are:

- Alphabetic
- Numeric
- Alphanumeric
- Alphanumeric-edited
- Numeric-edited

Alphabetic Data

Alphabetic data consists of upper and lowercase letters from the English alphabet, and one or more blanks. When you wish to define the characteristics of an alphabetic data item, the character string must consist of a combination of the letters A and/or B and, optionally, one or more nonnegative integers in parentheses.

The letter A represents a character of the alphabet or a space. The letter B represents a blank (or a space).

The integers are repetition factors and are used to specify one or more occurrences of A or B in the picture.

To describe an alphabetic data item that consists of six alphabetic characters, three spaces, and then twelve more alphabetic characters (or blanks), the following PICTURE clauses could be used, and are equivalent:

```
PICTURE IS AAAAAABBBAAAAAAAAAAAA

PIC   A(6)B(3)A(12)
```

Numeric Data

Numeric data consists of a combination of the Arabic numerals 0 through 9, optionally including the positive or negative sign, or a representation of an operational sign as defined in the SIGN clause. Note that a decimal point is not part of the possible set of characters allowed in forming numeric data for a COBOL program.

You must specify the data without a decimal point. Use the V character of the PICTURE clause to indicate where the decimal point belongs.

The size of a numeric data item can be from one to 18 digits long.

When you wish to define the characteristics of a numeric data item, the picture clause you use must consist of only the symbols 9, P, S, V, and (in conjunction with the 9 and P symbols only) one or more repetition factors as described under the heading "Alphabetic Data", above.

The symbol 9 represents the character position that is to contain a numeral.

The symbol S indicates the presence of an operational sign, but not necessarily its position within a numeric data item. It must be the leftmost character in the picture of the data item being described and can appear only once in a given picture clause.

The S symbol is not counted in determining the size of an elementary data item unless the data item is subject to a SIGN clause using the SEPARATE CHARACTER phrase.

The symbol V is used to indicate the location of the assumed decimal point in numeric data. Like the S symbol, it may only appear once in any given picture clause.

Because the V does not represent a character position, it is not counted in the size of the elementary data item. Note also that if the V appears as the rightmost character in a character string, it is redundant.

The symbol P indicates an assumed decimal scaling position and is used to specify the location of an assumed decimal point when the point is not within the number appearing in the data item.

That is, for each appearance of a P in the PICTURE of a data item, the data item is assumed to be multiplied by 10, or by one tenth, depending upon whether the P symbol is on the right, or on the left of the character string used to define the field.

The P in a character string does not occupy a position in memory and does not add to the size of the item. However, the P has the effect of the digit 0 whenever the item is accessed. When the data item is used for arithmetic operations, the P must be counted as a digit to determine whether the field exceeds the 18-digit maximum size for a numeric field.

One or more P symbols can only be placed at either the left or the right end of a character string. Because P represents a decimal position, the use of the P symbol and the V symbol as the left or rightmost symbol in the same string is redundant.

Therefore, if the V symbol is used with the P symbol, it is meaningful only if it appears as the leftmost or rightmost symbol in the character string.

The following illustrates the use of the P symbol:

- Input data: `241000`
- PICTURE clause: `999P(3)`
- Data stored as: `241`
- Data accessed as: `241000.`

Using the following PICTURE clause for the number 00000241

```
PICTURE  P(5)999
```

results in the data being stored as 241, but being accessed as .00000241.

A repetition factor may be used with the P symbol. Therefore, `PPPPP99` is equivalent to `P(5)99`.

All numeric literals used in a VALUE clause must have a value that is within the range of values indicated in the PICTURE clause. For example, the range of values permitted for an item with the PICTURE PPP999 are .000000 through .000999.

Alphanumeric Data

Alphanumeric data is made up of any valid character used on an HP computer.

To define an alphanumeric data item, you can use the symbols A, 9, or X. The PICTURE clause for this type of data, however, must contain at least one X symbol, or a combination of the A and 9 symbols to indicate that it represents an alphanumeric data item.

The A symbol can be used to represent alphabetic characters or a space, while the 9 symbol can be used to represent numerals. However, the entire PICTURE clause is treated as if it consists entirely of X symbols, where each X symbol can represent any single character used on an HP computer.

Repetition factors may be used with all these symbols. For example,

```
77 FINISHER  PICTURE A(5)9(8).
```

is equivalent to:

```
77 FINISHER   PICTURE X(13).
```

Alphanumeric-Edited Data

Alphanumeric data is data consisting of any set of characters available on an HP computer. This type of data can be edited by specifying where one or more spaces, strokes, or zeros are to appear as part of the receiving data item.

You can use the A, X, 9, B, 0 (zero), and / symbols to define a data item to receive an edited alphanumeric data item. The A, X, 9, and B symbols are explained under the headings "Alphabetic Data," "Numeric Data," "Alphanumeric Data," and "Numeric-edited," respectively.

The 0 symbol is used to specify where in the character string the numeral 0 is to be inserted.

The / (stroke) symbol represents where in the character string a stroke symbol is to be inserted.

All of the symbols used in an alphanumeric-edited PICTURE clause may use a repetition factor.

The PICTURE character string of an alphanumeric-edited data item is restricted to certain combinations of the following symbols:

`A, X, 9, B, 0,` and `/`

The PICTURE clause must contain at least one A or X, and must contain at least one B, 0 (zero), or / (stroke).

For example,

Sending data item: `010685`

PICTURE of receiving data item: `99/99/99`

Receiving data item: `01/06/85`

Sending data item: `NAMEADDRESSPHONENUMBERZIPCODE`

PICTURE of receiving data item: `PICTURE A(4)B(5)A(7)B(5)A(5)BA(6)/A(7)`

Receiving data item: `NAME␣␣␣␣␣␣ADDRESS␣␣␣␣␣␣PHONE␣NUMBER/ZIPCODE`

Numeric-Edited Data

In standard data format, numeric-edited data consists of a combination of the numerals zero through 9 and an optional decimal point.

Editing of this type of data in HP COBOL II consists of leading zero suppression, filling or replacement, placement and alignment of a decimal point and a currency symbol, insertion of a sign, commas, blanks, or strokes. This is accomplished through use of the following symbols:

```
P,  V,  9,  B,  /,  r,  Z,  (,),  (.),  *,  +,  -,  CR,  DB
```

and the currency symbol as defined in the CURRENCY SIGN clause of the SPECIAL-NAMES paragraph in the ENVIRONMENT DIVISION. To distinguish this type of data from unedited numeric data, at least one of the above symbols (except the 9) must appear in the PICTURE clause.

A maximum of 18 digit positions can be represented in this type of PICTURE.

The first three of the above symbols are described under the heading, "Numeric Data", and the second set of three are described under the heading, "Alphanumeric-Edited Data". The remaining symbols are described below.

The Z symbol is used for the suppression of leading zeros in the receiving data item. It can only be used to represent the leftmost leading numeric character positions.

The comma symbol (,) represents where in the character string of the receiving data item a comma is to be inserted. It cannot be the rightmost character in the PICTURE clause unless followed by the period symbol (.).

The period symbol (.) represents the decimal point for aligning the sending and receiving data items and also represents a character position into which a period (decimal point) is to be inserted. It may not be used if the V symbol is used, and may only appear once in a given PICTURE clause if the DECIMAL POINT IS COMMA clause is not specified in the SPECIAL-NAMES paragraph. Also, it may not appear as the rightmost element in the PICTURE clause unless followed by the decimal point (.). The P symbol and the decimal point (.) cannot be used in the same PICTURE character string.

If the DECIMAL POINT clause is specified, the roles of the commas and period symbols are reversed. Therefore, in such a case, only one comma symbol may appear in a numeric-edited PICTURE clause, but several periods may appear. The plus (+), minus (-), CR (for CRedit), and DB (for DeBit) are used as editing sign control symbols. Only one of these symbols may appear in any given PICTURE clause, and when used, specify the position in the receiving data item into which the editing sign control symbol will be placed.

The asterisk (*) symbol is used for replacing leading zeros. Each leading zero in the sending data item is replaced in the receiving data item by an asterisk if there is an asterisk in the PICTURE clause for the receiving data item whose position corresponds to the position of the zero in the sending data item.

This symbol may not appear in a PICTURE clause for a data item which has the BLANK WHEN ZERO clause specified for it.

The appearance of a currency symbol in a PICTURE clause represents the position into which a currency symbol is to be placed. If you do not specify an alternative currency symbol through the CURRENCY SIGN clause of the SPECIAL-NAMES paragraph, the dollar ($) symbol is used.

This symbol must appear as the leftmost symbol in the character string, except that it may be preceded by a plus (+) or minus (-) symbol.

With the exception of the symbols V, CR, DB, and period (.), all of the symbols described above may be specified using a repetition factor.

To illustrate edited numeric data:

Sending data item: `-1234.59`

PICTURE clause of the receiving data item: `PICTURE 99,999.99DB`

Receiving data item: `01,234.59DB`

Note in the second example that the DECIMAL-POINT IS COMMA clause is assumed.

Sending data item: `345777.78`

PICTURE clause of the receiving data item: `PICTURE +$ZZZ,ZZZ,ZZZ,ZZZ.99`

Receiving data item: `+$       345.777,78`

Table 7-2 summarizes the editing picture characters and their function.

Table 7-2. Editing Picture Characters

Picture Characters	Symbol Definition	Editing Function
`B`	Letter B	Inserts a blank.
`/`	Slash	Inserts a slash character.
`0`	Zero	Inserts a zero digit.
`.`	Decimal point	Inserts a decimal point.
`,`	Comma	Inserts a comma.
`+`	Plus sign	Inserts `+` or `-` sign.
`-`	Minus sign	Inserts `-` or blank.
`CR`	Credit sign	Inserts `CR`.
`DB`	Debit sign	Inserts `DB`.
`$`	Dollar sign	Inserts currency symbol.
`Z`	Letter Z	Zero suppression by blank.
`*`	Asterisk	Zero suppression by `*`.

Size of Elementary Data Items

The size of an elementary data item is defined as being the number of character positions occupied by the elementary item in standard data format. This size is determined by counting the number of allowable symbols used to represent character positions within a PICTURE clause for that item.

With the exception of the S, V, period (.), CR and DB symbols, all symbols in the PICTURE clause may use repetition factors. These repetition factors are represented by an integer enclosed by parentheses following the symbol to which they pertain, and indicate the number of consecutive occurrences of the symbol.

Furthermore, if the SEPARATE CHARACTER phrase of the SIGN clause is not specified, the V symbol and the S symbol do not participate in the count when you are determining the size of the data item.

DB and CR each represent two character positions.

When you count occurrences of characters in an elementary data item description, you must only count those symbols that appear without repetition factors, and add them to the sum of all integers appearing in repetition factors for that PICTURE clause.

Examples

The size of the data item represented by the following PICTURE clause is 11 characters.

```
PICTURE ZZZ,999V99CR
```

The size of the data item represented by the next PICTURE clause is 17 characters.

```
PICTURE A(10)B(5)XX
```

In the next PICTURE clause, assume that the SIGN IS SEPARATE clause is NOT specified for the data item represented by the PICTURE clause below.

```
PICTURE S9(5)V99
```

The size of the item described above is 7 characters.

Editing Rules

There are two general methods for performing editing in the PICTURE clause: *insertion*, and *zero suppression and replacement*.

Editing takes place only when data is moved into an elementary data item whose PICTURE clause specifies editing (that is, whose PICTURE clause is alphabetic, alphanumeric-edited or numeric-edited). Therefore, data moved into a numeric field is not edited.

You may perform simple insertion editing for an item belonging to the alphabetic or alphanumeric-edited categories.

Three other types of insertion, as well as suppression and replacement, may be performed on numeric-edited data: *special*, *fixed*, and *floating insertion*.

Table 7-3 below summarizes the type of editing permitted for each category.

Table 7-3. Allowable Types of Editing For Categories of Data Items

Category	Type of Editing
Alphabetic	Simple insertion `B` only
Numeric	None
Alphanumeric	None
Alphanumeric-Edited	Simple insertion (`0`), (`,`), (`B`), and (`/`)
Numeric-Edited	All

Simple Insertion Editing. The comma (,), space (B), zero (0), and stroke (/) symbols are used in simple insertion editing. These insertion characters are counted in the size of the receiving item and represent the position in the receiving item into which the character is inserted.

Simple insertion editing is so called because, other than inserting the particular symbol, no other editing is done and the data sent is unaffected except for the placement of the simple insertion characters between, before, or after the other characters received from the sending data item.

An example of simple insertion editing is shown in the illustration under the heading, "Alphanumeric-Edited Data", on the preceding pages.

Special Insertion Editing. The period (.) is used in special insertion editing. In addition to being an insertion character, it is also used for alignment purposes when the sending data item is numeric and contains a decimal point. The result of this form of editing is the appearance of the period in the same position as it appears in the PICTURE clause for the item.

When data is moved to an item defined with the special insertion character, COBOL automatically provides truncation and zero fill to both the left and the right of the decimal point. However, if zero suppression or floating insertion editing is included in the PICTURE clause of the receiving data item, zero fill normally produced by special insertion editing is overridden.

The following illustrates special editing:

Sending data item: `12345.678`

PICTURE clause of receiving data item: `9(5).99`

Receiving data item: `12345.67`

Note that the rightmost digit, 8, was truncated. This was caused by the alignment of decimal points.

Sending data item: `.001`

PICTURE clause of receiving data item: `9,999.9999`

Receiving data item: `0,000.0010`

Finally:

Sending data item: `658456.995`

PICTURE clause of receiving data item: `999.99`

Receiving data item: `456.99`

Fixed Insertion Editing. Fixed insertion editing uses the currency symbol and the editing sign control symbols, +, -, CR, and DB.

Only one currency symbol and one editing sign control symbol can be used in a given PICTURE clause when you wish to use fixed insertion editing. When the CR or DB symbol is used, each represents two characters, and must be in the rightmost character positions counted in determining the size of the receiving item.

When the + or - symbol is used, it must be either the leftmost or rightmost character in the PICTURE clause for the receiving item, and is counted in the size of that item.

The currency symbol must be the leftmost character position to be counted in the size of the item, except that it may be preceded by a + or a - symbol.

Fixed insertion editing results in the insertion character occupying the same character position in the receiving item as it does in the character string used in the PICTURE clause.

The sign control symbols produce different results, depending upon whether the sending data item is positive or negative. These differing results are shown in Table 7-4 below.

Table 7-4. Effects of Sign Control Symbols on Receiving Items

Editing Symbol in Character String	Result	
	Data Item Positive	Data Item Negative
+	+	-
-	space	-
CR	2 spaces	CR
DB	2 spaces	DB

Floating Insertion Editing. Floating insertion editing uses the currency, +, or - symbol. Each symbol is mutually exclusive of the others when you wish to perform this type of editing.

Also, zero suppression and replacement cannot be used in the same character string of a PICTURE clause using floating insertion editing.

You can represent floating insertion editing in one of two ways. The first is to represent any or all of the leading numeric positions to the left of the decimal point with your chosen floating insertion character. The second way is to represent every numeric character position, on both sides of the decimal point, with the insertion character.

Floating insertion editing is indicated by an occurrence of the same symbol used at least twice in the same string. This is the major distinction between fixed and floating insertion editing.

Between or to the right of this floating insertion string, can be any of the simple insertion characters. If such is the case, the simple insertion characters are a part of the floating insertion character string.

The bounds of the floating insertion string (including the simple insertion characters, as noted above) are formed by the leftmost and the rightmost elements of the floating string.

Nonzero numeric data can be stored in the receiving data item starting at the first character to the right of the leftmost character in the floating string, and proceeding through the entire floating string.

If the floating insertion characters are only to the left of the decimal point, insertion takes place in a fashion analogous to the following algorithm:

1. The leading character of the sending data item is checked to see if it has a zero value. If it is zero, the floating insertion character is inserted in the corresponding character position of the receiving data item and the preceding character of this data item is replaced with a space.
2. The next character of the sending data item is then checked for a zero value, and if it is zero, the action described in step 1 is repeated.
3. The process continues either until no numeral in the sending data item is nonzero (in which case all the positions corresponding to the floating insertion string in the receiving data item are replaced with spaces), or some nonzero numeral is found in the sending data item and this numeral appears to the left of the decimal place.

If any simple insertion character appears as part of the floating insertion string, and no nonzero character is encountered in the sending data item before the next floating insertion character position is considered, one simple insertion character is replaced by the floating insertion character, and the preceding floating insertion character is replaced by a space.

When a nonzero numeral is encountered in the data item, that numeral and all following it are replaced in the positions corresponding to their positions in the floating insertion string.

If the floating insertion characters correspond to every numeric character position, including those to the right of the decimal point, the algorithm is the same as above, with one exception.

The exception is when the original data item is zero. In this case, the result of floating insertion editing is that the data item referenced by the PICTURE clause contains only spaces.

Note that to avoid truncation, your character string in the PICTURE clause for the receiving data item must be, minimally, the size of the number of characters in the sending data item, plus the number of nonfloating insertion characters being inserted into the receiving data item, plus one for the first floating insertion character. To illustrate floating insertion editing, the following example uses a sending data item that is 00123.45, in standard data format and uses the PICTURE clause to describe the receiving data item:

```
PICTURE   $$$,$$$.99
```

Using the algorithm described above, the steps taken appear as follows:

1. First character equal to 0? Yes. Therefore, receiving data item appears as ␣$.
2. Second character equal to 0? Yes. Therefore, receiving data item appears as ␣␣␣$ (Because the comma preceded the first occurrence of a nonzero numeral).
3. Third character equal to 0? No. Therefore, receiving data item appears as ␣␣␣$123.45

Result: ␣␣␣$123.45

Note that if the PICTURE clause had been of the form:

```
PICTURE $$$.99
```

the result would be `$23.45` because of truncation of the sending data item to allow for insertion of the floating character, `$`, in the receiving data item.

Zero Suppression Editing. Zero suppression editing allows you to replace leading zeros of the sending data item with either spaces or asterisks in the receiving data item.

You can replace one or more leading zeros with spaces by placing a Z in the corresponding positions of the PICTURE character string used to represent the receiving data item.

If you wish to replace leading zeros with asterisks (*), use a string of asterisks rather than Z's in the PICTURE for the receiving data item.

You may use either the Z symbol or the * symbol, but not both, in any one PICTURE clause.

The algorithm used in zero suppression and replacement is essentially the same as the algorithm used for floating insertion editing. That is, any simple insertion symbols may appear between the first and last symbol or to the right of the last suppression symbol, and are included as part of the suppression string.

Furthermore, if the suppression symbols appear only to the left of the decimal point, any leading zero in the sending data item that corresponds to a suppression symbol is replaced by that suppression symbol. Suppression terminates with the first occurrence of a nonzero numeral in the sending data item or with the decimal point, whichever occurs first.

If all numeric character positions are represented by suppression symbols and the sending data item is zero, the entire receiving data item consists of spaces (if Z's are used) or asterisks (if asterisks are used), except for the decimal point.

If all numeric characters are represented in the receiving data item by suppression or replacement symbols, and the sending data item is not zero, suppression and replacement take place in the same manner as if the suppression symbols appeared only to the left of the decimal place.

Note that you may not use floating insertion editing in the same PICTURE clause in which you are using zero suppression and replacement. The following illustrates zero suppression and replacement:

Sending data item in standard data format: `004053.67`

Picture of receiving data item: `PICTURE $ZZZ,ZZZ.ZZ`

Result: `$␣␣4,053.67`

Using the same sending data item, but with the following picture of the receiving data item:

```
PICTURE $***,**9.99
```

results in:

```
$**4,053.67
```

Precedence Rules. Table 7-5 shows the order of precedence when using insertion, suppression, or replacement symbols in a character string of a PICTURE clause.

An x at an intersection indicates that the symbol at the top of the column may precede the symbol at the left of the row. Multiple symbols in a box (except for A and X) are mutually exclusive. The leters "cs" indicate the currency symbol.

The +, -, Z, *, cs, and P symbols appear twice in the Non-Floating and Floating insertion symbols sections of Table 7-5. The lefthand column and the upper row of each of these pairs represents the use of the symbol to the *left* of the decimal point. The righthand column and lower row represent the use to the *right* of the decimal point.

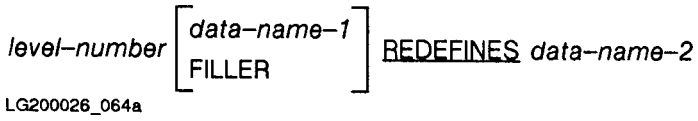
Table 7-5. PICTURE Character Precedence Chart

Second Symbol	First Symbol																			
	Non-Floating Insertion Symbols									Floating Insertion Symbols						Other Symbols				
	B	0	/	,	.	+ -	+ -	CR DB	cs	Z *	Z *	+ -	+ -	cs	cs	9	A X	S	V	P	P
B	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
0	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
/	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
,	x	x	x	x	x	x			x	x	x	x	x	x	x	x			x		x
.	x	x	x	x		x			x	x		x		x		x					
+ -																					
+ -	x	x	x	x	x				x	x	x			x	x	x			x	x	x
CR DB	x	x	x	x	x				x	x	x			x	x	x			x	x	x
cs						x															
Z *	x	x	x	x		x			x	x											
Z *	x	x	x	x	x	x			x	x	x								x		x
+ -	x	x	x	x					x			x									
+ -	x	x	x	x	x				x			x	x						x		x
cs	x	x	x	x		x								x							
cs	x	x	x	x	x	x								x	x				x		x
9	x	x	x	x	x	x			x	x		x		x		x	x	x	x		x
A X	x	x	x													x	x				
S																					
V	x	x	x	x		x			x	x		x		x		x		x		x	
P	x	x	x	x		x			x	x		x		x		x		x		x	
P						x			x									x	x		x

REDEFINES Clause

The REDEFINES clause allows you to define the same storage area in main memory for different data items whose lengths are not described as variable in an OCCURS clause.

Syntax



Parameters

level-number and *data-name-1* and FILLER	level number and data name of the data item being described; these are not part of the REDEFINES clause. When the REDEFINES clause is used in a data description entry, it must be immediately preceded by *level-number* and *data-name-1*.
data-name-2	data name used in a different data description entry; it must have the same level number associated with it as does *data-name-1*. The level number must not be 66 or 88, nor can it be 01, if the REDEFINES clause is used in the FILE SECTION.

Description

Redefinition of storage area begins at the *data-name-1* entry and continues until a level number less than or equal to that of *data-name-1* (or *data-name-2* since they are the same) is found.

Because *data-name-1* is a redefinition of the storage area for *data-name-2*, no entry having a level number numerically lower than the level number of *data-name-2* may occur between the data description entries of *data-name-2* and *data-name-1*.

Furthermore, the description entry for *data-name-2* cannot contain a REDEFINES or an OCCURS clause, but may be subordinate to an entry that contains one of these clauses.

If the data description entry for an item to which *data-name-2* is subordinate contains an OCCURS clause, the reference to *data-name-2* in the REDEFINES clause must not be subscripted or indexed.

If the level number of *data-name-1* and *data-name-2* is other than 01 or *data-name-2* is an external record, the description of *data-name-1* must specify `less than or` the same number of character positions as specified for the data item referenced by *data-name-2*.

Multiple redefinitions of the same character positions are permitted in COBOL. However, multiple redefinitions of the same character positions must all use the data name of the entry that was originally used to define the area. The entries providing the new descriptions must immediately follow the entries used to define the area currently being redefined and the new entries must not (except in the case of condition name entries) contain any VALUE clauses.

Examples

```
01 RECORD-IN      PICTURE X(80).
01 RECORD-PARTS REDEFINES RECORD-IN.
    02 NAME           PICTURE X(30).
    02 STREET         PICTURE X(20).
    02 CITY           PICTURE X(20).
    02 STATE          PICTURE X(10).

01 PARTS-TABLE.
    02 PART OCCURS 35 TIMES.
       03 NAME              PIC X(10).
       03 QUANTITY          PIC 9(04).
       03 UNIT-PRICE        PIC 9(06).
       03 LOCALE            PIC X(10).
       03 SITE-INFO REDEFINES LOCALE.
          04 BUILDING-NO    PIC X(03).
          04 FLOOR-NO       PIC X(02).
          04 SECTION-NO     PIC X(02).
          04 BIN-NO         PIC X(03).
```

The above two uses of the REDEFINES clause are permissible, whereas the following two are not.

```
05 VOCABULARY OCCURS 2000 TIMES    PIC X(100).
05 WORDLIST REDEFINES VOCABULARY.
    20 INITIAL                     PIC X(20).
    20 SECOND                      PIC X(30).
    20 THIRD                       PIC X(30).
    20 FOURTH                      PIC X(20).

01 RECORD-IN.
    02 FIRST-FIELD.
       03 SUB-AA                   PIC X(15).
    02 SECOND-FIELD.
       03 SUB-BB REDEFINES SUB-AA  PIC X(15) VALUE SPACES.
       03 SUB-BB1                  PIC X(05).
```

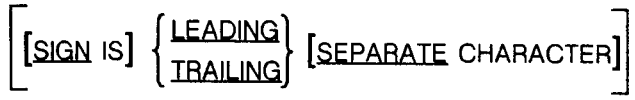
The first unacceptable usage above is because of the use of an OCCURS clause in the description of `VOCABULARY`. The second is unacceptable because of the 02 level entry between `SUB-AA` and `SUB-BB`. The new entry may not contain any value clauses.

SIGN Clause

The SIGN clause is only used with a signed numeric data description item or description entry whose usage is DISPLAY, or a group item containing at least one such data description entry.

It states the position of the sign, whether leading or trailing, as well as whether the sign was formed by overpunching in the first or last character of the data item (see the USAGE IS DISPLAY clause, below) or was formed separately.

Syntax



LG200026_065

Parameters

`LEADING` and `TRAILING`	indicates that the sign is at the beginning or end, respectively, of the item.
`SEPARATE`	indicates that the sign is not overpunched; that is, the sign exclusively occupies the first or last character of this item.

Description

Only one sign clause may be used per given numeric data description entry.

Also, if the CODE-SET clause is specified in a file description, any signed numeric data description entry associated with that file must be described with the SIGN IS SEPARATE clause.

Valid signs for data items, and their representations when overpunching is used, are shown in the table under the USAGE IS DISPLAY heading on the following pages.

For a SIGN IS SEPARATE designation, the two valid operational signs (whether LEADING or TRAILING) are + or - for positive and negative quantities, respectively.

For a signed numeric data description entry having no SIGN clause associated with it, the default is equivalent to SIGN IS TRAILING. That is, the sign is assumed to be overpunched in the last character of the item.

In either the default case, or the case when the optional SEPARATE CHARACTER phrase is not used, the letter "S" in the PICTURE CLAUSE is not counted in determining the size of the item when represented in standard data format. To illustrate the SIGN clause:

- The data to be entered is `123489F`

  In this case, no SIGN clause is required because the default is SIGN IS TRAILING. However, if the PICTURE clause for this data item is `PICTURE S9(7)`, the size of the data item is seven characters.

- The data to be entered is `+1409748`

  In this case, the SIGN clause should be: `SIGN IS LEADING SEPARATE CHARACTER`

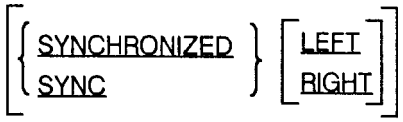
  Also, since the sign is separate, the PICTURE clause for this data item, `PICTURE S9(7)` defines the data item to be eight characters long in order to hold the separate sign.

SYNCHRONIZED Clause

The SYNCHRONIZED clause is used to align items defined as USAGE IS COMPUTATIONAL or BINARY on word boundaries in order to facilitate arithmetic operations. A word size is defined by the operating system environment (a word is 32 bits on MPE XL).

All other items are aligned on byte boundaries. Because the character (byte) is the smallest directly addressable unit within the COBOL language, the SYNCHRONIZED clause has no meaning when applied to an item with any usage other than COMPUTATIONAL. It is treated as a comment for items described as DISPLAY, INDEX, COMPUTATIONAL-3, or PACKED-DECIMAL.

Syntax



LG200026_066

Description

Because of the word structure used on HP computers, the LEFT and RIGHT options are irrelevant and are treated as comments by the compiler.

The words SYNCHRONIZED and SYNC are equivalent.

The compiler always aligns all level 01 and level 77 items on word boundaries except in the LINKAGE section (see the section "OPTFEATURES" in Appendix G for details).

When the SYNCHRONIZED clause is specified for a data item whose description also contains an OCCURS clause, or in a data description entry of a data item subordinate to a description entry containing an OCCURS clause, each occurrence of the data item is synchronized. Any implicit filler (see "Slack Bytes", below) generated for other data items within that same table are generated for each occurrence of those data items.

Slack Bytes

The SYNCHRONIZED clause specifies that the data described is to be aligned on word boundaries. If the SYNCHRONIZED item does not fall naturally on a word boundary, the compiler assigns the next highest boundary address to the item.

The effect of adding a byte (or bytes) is equivalent to providing extra FILLER characters, known as slack bytes, just before the SYNCHRONIZED item.

These slack bytes are not used for any other data item and are not counted in the size of the items. They are, however, included in the size of any group item or items to which the elementary item belongs, and are included in the character positions redefined when the SYNCHRONIZED item is the object of a REDEFINES clause. Therefore, when you use the REDEFINES clause in a data description that also contains a SYNCHRONIZED clause, you must ensure that the redefined item has the proper boundary alignment for the item that redefines it.

The computation of boundary addresses is affected by the $CONTROL SYNC option. This option changes the alignment of SYNCHRONIZED data items, which affects the number of slack bytes generated in a record. Items with the SYNC clause are aligned along 16-bit (2 characters) boundaries, if SYNC16 is in effect. Items with the SYNC clause are aligned along 32-bit (4 character) boundaries, if SYNC32 is in effect. (Refer to the description of DATA DIVISION language dependencies and $CONTROL SYNC16/SYNC32 in "MPE XL System Dependencies" in Appendix H for synchronization alignment specifics.) This option may be coded more than once in a program to align one record along 16-bit boundaries, and another record along 32-bit boundaries. Alignment cannot be changed within a record, only between records.

This option is especially useful to develop files to be used on other computer architectures, or to read files developed on other architectures.

Whenever a SYNCHRONIZED item is referenced in your program, the original size of the item, as shown in the PICTURE clause, is used in determining any action that depends on size. Such actions include justification, truncation, or overflow.

If the SYNCHRONIZED clause is not used, no space is reserved for slack bytes, and when a computation is performed on a data item described as COMPUTATIONAL, the compiler provides the code and space required to move the data item from its storage area to a work area. This work area has the alignment required to perform the computation.

As an illustration of slack bytes (assuming 16-bit synchronization), consider the following data description entries:

```
01 ITEM-LIST.
   02 ITEM-NUMBER     PICTURE X(3).
   02 ITEM-1          PICTURE X(4).
   02 ITEM-2 REDEFINES ITEM-1 PICTURE S9(6) USAGE COMP SYNC.
```

The above is an example of *not* taking into account the slack byte required because of the REDEFINES clause. To correct it, the description of `ITEM-LIST` should include an extra byte prior to `ITEM-1`:

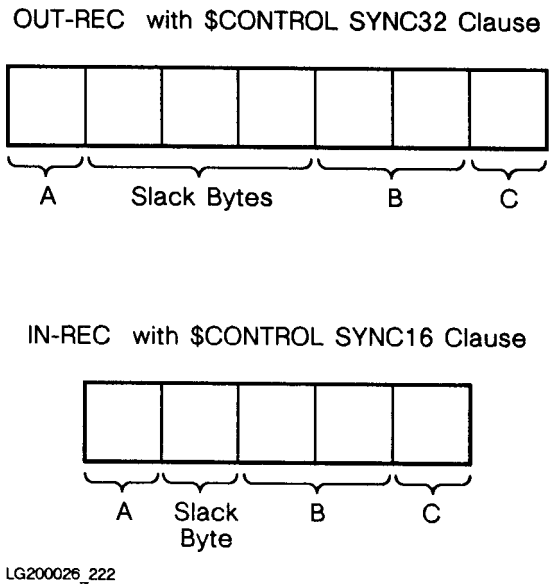
```
01 ITEM-LIST.
   02 ITEM-NUMBER    PICTURE X(3).
   02 SLACK-BYTE     PICTURE X.
   02 ITEM-1         PICTURE X(4).
   02 ITEM-2 REDEFINES ITEM-1 PICTURE S9(6) USAGE COMP SYNC.
```

This change is all that was needed, since all 01 level entries are aligned on word boundaries except in the LINKAGE section (see the section "OPTFEATURES" in Appendix H for details).

Example

The following illustrates the use of $CONTROL SYNC:

```
$CONTROL SYNC32

01 OUT-REC.

   05 A  PIC  X.
   05 B  PIC S9999  BINARY SYNC.
   05 C  PIC  X.

$CONTROL SYNC16

01 IN-REC.

   05 A  PIC  X.
   05 B  PIC S9999  BINARY SYNC.
   05 C  PIC  X.
```



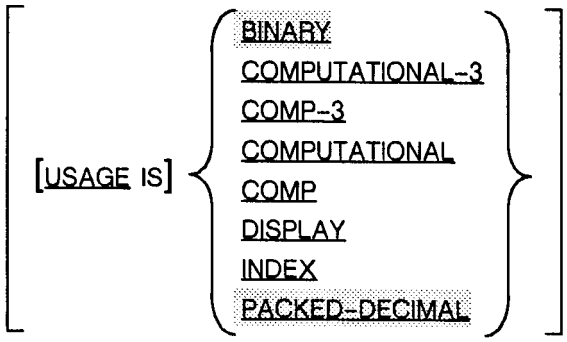
In the above example, three slack bytes are inserted before `B` of `OUT-REC`. One slack byte is inserted before `B` of `IN-REC`.

Note Due to the $CONTROL SYNC option, boundary alignment within a record is constant. It only changes between records.

USAGE Clause

The USAGE clause specifies how the data item being described is stored internally. When used with index, the USAGE clause specifies that the data item being described contains a value equal to the value of an index name associated with an occurrence number for a table element.

Syntax



LG200026_067

Parameters

`PACKED-DECIMAL`, `COMPUTATIONAL-3`, and `COMP-3`	equivalent; they specify packed-decimal format. The words COMPUTATIONAL-3 and COMP-3 are an HP extension to the 1985 ANSI COBOL standard.
`BINARY`, `COMPUTATIONAL`, and `COMP`	equivalent; they specify two's complement binary integer format.
`DISPLAY`	the default usage if no USAGE clause is specified. It specifies that data is to be stored internally as ASCII characters.
`INDEX`	specifies an index data item.

Description

This clause is optional, with a USAGE IS DISPLAY being used by your program for any data item having no USAGE clause as part of its description, or as part of the description of a data item to which it is subordinate.

You may use the USAGE clause at any level of organization. However, if written at a group level, the USAGE clause applies to each elementary item in the group, and any USAGE clause specified at the elementary level must be the same as at the group level.

Although a USAGE clause does not affect the use of a data item, some of the statements in the PROCEDURE DIVISION may restrict the USAGE clause of the operands used.

USAGE IS DISPLAY

When usage of a data item is defined (implicitly or explicitly) as DISPLAY, the data is stored internally as ASCII characters.

This means that each character of data is stored as an 8-bit byte.

If you are using the data item in a noncomputational manner (that is, printing or displaying it), this is the appropriate type of usage to be specified.

However, for optimum use of your COBOL program, you should specify USAGE IS COMPUTATIONAL, or `BINARY`, COMPUTATIONAL-3, or `PACKED-DECIMAL` for data items intended for use in computations. This is because data items described as USAGE IS DISPLAY must be converted to two's-complement binary or packed-decimal format before they can be used in computations, and this conversion takes time. Also, if you intend to use signed numeric data items for computational purposes, you must specify a sign (by using the S symbol) in the PICTURE clause for that item (see USAGE IS COMPUTATIONAL on the following page), whether its usage is specified as DISPLAY or otherwise.

An unsigned numeric data item whose description specifies the USAGE IS DISPLAY clause is assumed to be positive.

Numeric DISPLAY items without a clause that designates SIGN IS SEPARATE are represented in ASCII coded (8-bit) decimal digits (0 through 9) except for the units digit which carries the sign of the data item. The units digit, with the sign of its associated number being positive, negative, or no sign (absolute value) respectively, is represented in ASCII code as shown in Table 7-6.

Note that using signed numeric DISPLAY data items for computational purposes is more efficient than using unsigned numeric data items.

Table 7-6.
Overpunch Characters for Rightmost Digit in ASCII Coded Decimal Numbers

Units Digit	Internal Representation (ASCII)		
	Positive	Negative	No Sign
0	{	}	0
1	A	J	1
2	B	K	2
3	C	L	3
4	D	M	4
5	E	N	5
6	F	O	6
7	G	P	7
8	H	Q	8
9	I	R	9

Signed decimal fields entered through punched cards are known as zone-signed fields. To represent a positive value, an overpunch is placed in the 12-zone above the rightmost digit of the field. To represent a negative value, an overpunch is placed in the 11-zone above the rightmost digit of the field. If no sign is desired, only the digits need be punched.

Zone signs cause the signed digit to have the same punch configuration as certain other characters. This is the purpose of the S symbol in the PICTURE clause; it informs the compiler that the last digit in the field is to be interpreted as a number and a sign, and not as the character that it would otherwise represent. Table 7-6 shows the data character equivalents to each possible rightmost digit sharing a zone sign.

USAGE IS BINARY or COMPUTATIONAL

When usage of a data item is defined as `BINARY` or COMPUTATIONAL, the data must be numeric. It is stored in two's-complement binary integer form, consisting of either two, four, or eight bytes each. The number of bytes used depends upon the size of the data item, as shown in Table 7-7 below.

Table 7-7. Number of Bytes Used to Contain a `BINARY` Data Item

PICTURE	Number of Bytes
S9 to S9(4)	2
S9(5) to S9(9)	4
S9(9) to S9(18)	8

A data item whose usage is defined as `BINARY` or COMPUTATIONAL must have an unedited numeric PICTURE clause associated with it. It may contain up to 18 digits plus a sign. Also, if a group item is described as `BINARY` or COMPUTATIONAL, all of the elementary items in the group are computational and may be used in computations. However, the group item itself may not be used in computations because it is considered alphanumeric. A numeric data item that does not have a sign associated with it is assumed to be positive.

As with numeric DISPLAY data items, a signed numeric data item whose USAGE IS `BINARY` is more efficient than an unsigned numeric data item with the same USAGE.

USAGE IS PACKED-DECIMAL or COMPUTATIONAL-3

Data items described as `PACKED-DECIMAL` or COMPUTATIONAL-3 are subject to the same restrictions and are used in the same way as data items described as COMPUTATIONAL. Such items are, however, stored in packed-decimal format. In this format, there are two digits per byte, with a sign in the low order 4-bits of the rightmost byte.

Each `PACKED-DECIMAL` or COMPUTATIONAL-3 item may contain up to 18 digits plus a sign. If the picture for the item does not contain a sign, the sign position in the data field is occupied by a bit configuration that is interpreted as positive. Table 7-8 illustrates the bit configurations used to represent signs in packed-decimal fields. Notice that the bit configuration 1100 specifies a positive value and that the 4-bit configuration 1111 represents the unsigned (assumed positive) value when an unsigned picture is specified. For negative values, the 4-bit configuration is 1101.

Table 7-8. COMPUTATIONAL-3 or PACKED-DECIMAL Sign Configuration

Sign	Bit Configuration	Hexadecimal Value
+	1100	C
-	1101	D
Unsigned	1111	F

Table 7-9 gives a graphic illustration of packed-decimal fields as they might appear in memory or in a file. Notice that these items follow the normal rules for truncation, even though the field may include an unused half-byte position. The contents of this half-byte are unpredictable when data is interchanged with other computer systems. In the table below, each box in the result column represents a byte.

Table 7-9. PACKED-DECIMAL Fields in Memory or in a File

Value to be Stored	PICTURE of Result	Result			
+1234.	S9999		01	23	4C
+12345.	S99999		12	34	5C
12345	S9999		12	34	5F
-1.2	S999V999	00	01	20	0D
-.5	S999V999	00	00	50	0D
+1.22172	S999V999	00	01	22	1C
-12345.	99999		12	34	5F

Note that the third and last number in the table were stored as unsigned (assumed positive) numbers because the receiving field is unsigned according to its PICTURE.

USAGE IS INDEX

An elementary data item whose usage is defined as INDEX is called an index data item. Its purpose is to hold the contents of a table index while the table is being processed. Therefore, any value within the index data item must correspond to an occurrence number of an element in a table. An index data item is stored as a synchronized unsigned computational integer, both internally and externally. (Refer to "System Dependencies" in Appendix H for the correct size of an index data item.) An index data item cannot be a conditional variable, and can only be referenced explicitly in a SEARCH or SET statement, a relation condition, the USING phrase of the PROCEDURE DIVISION header, or USING phrase of a CALL statement.

Do not use the SYNCHRONIZED, JUSTIFIED, PICTURE, VALUE, and BLANK WHEN ZERO clauses to describe a group of elementary items whose usage is defined as INDEX. Index data items are automatically SYNCHRONIZED.

In ANSI COBOL'85, if a group item is described with a USAGE IS INDEX clause, all of its elementary items are index data items, but the group itself is not an index data item and cannot be used in the SEARCH or SET statements, or in an alphanumeric comparison in the PROCEDURE DIVISION. As an HP extension to the ANSI COBOL standard, HP COBOL II allows a group item described with USAGE IS INDEX to be used in an alphanumeric comparison.

VALUE Clause

The VALUE clause is used to define the values of constants and to initialize the values of WORKING-STORAGE data items. For information on VALUE clauses in condition names, refer to "Condition Names" later in this chapter.

Syntax

[VALUE IS *literal-1*]

Parameters

literal-1 the value assigned to the data item being described.

Description

The above format for the VALUE clause can only be used in the WORKING-STORAGE SECTION. If used, the VALUE clause causes the item to which it is associated to assume the specified value at the start of the object program, irrespective of any BLANK WHEN ZERO or JUSTIFIED clause. If the VALUE clause is not used in an item's description, the initial value of the item is undefined.

```
A VALUE clause may be used with data items (or descendants of data items)
containing an OCCURS clause to initialize tables.

A VALUE clause (VALUE is literal-1) 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 specific value.

If a VALUE clause is specified in a data description entry of a data item
that is associated with a variable occurrence data item, {{the initialization of the da
DEPENDING ON phrase in the OCCURS clause is set to the maximum number
of occurrences specified by that OCCURS clause.
```

A data item is associated with a variable occurrence data item in any of the following cases:

- When it is a group data item that contains a variable occurrence data item.
- When it is a variable occurrence data item.
- When it is 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 considered to be placed in the data item after the variable occurrence data item is initialized.

Restrictions on the Use of the VALUE Clause

The following restrictions apply to the use of the VALUE clause. They also apply to the VALUE clause in condition names.

- The VALUE clause cannot be used in a data description entry containing a REDEFINES clause (except when used with a condition name), or in an entry subordinate to an entry containing a REDEFINES clause. Nor can it be used for a group containing items with descriptions including JUSTIFIED, SYNCHRONIZED, or USAGE (except USAGE IS DISPLAY).
- The VALUE clause must not conflict with any other clauses in the data description of the item, or in the data description within the hierarchy of the item.
- The VALUE clause must not be used with any external record, except for condition-name entries associated with external records.

Literals in the VALUE Clause

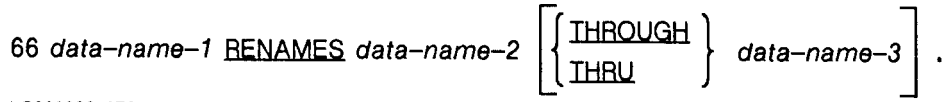
The literals used in the VALUE clause are subject to the following rules:

- Figurative constants may be substituted for literals.
- A signed numeric literal must have a signed numeric PICTURE and character string associated with it.
- All numeric literals must have a value within the range of values indicated by the PICTURE clause, and must not have a value which would require truncation of nonzero digits. Nonnumeric literals must not exceed the size indicated by the PICTURE clause.
- If the category of the item being described is numeric, all literals in the VALUE clause must be numeric. If the literal defines the value of a working-storage item, the literal is aligned in the data item according to the standard alignment rules.
- If the category of the item being described is any other than numeric, all literals in the VALUE clause must be nonnumeric. The literal is aligned in the data item as if the data item had been described as alphanumeric. Editing characters in the PICTURE clause are included in determining the size of the data item, but have no effect on initialization. Therefore, the VALUE for an edited item must be presented in an edited form.
- If the VALUE clause is used in an entry at the group level, the literal must be a figurative constant or a nonnumeric literal, and the group area is initialized without consideration for the individual elementary or group items contained within the group. A VALUE clause cannot be used for elements of a group that has a VALUE clause assigned to it at the group level.

RENAMES Clause

The RENAMES clause permits alternative, possibly overlapping groupings of elementary items. This clause is always associated with a 66 level entry.

Syntax



The level number (66) and *data-name-1* are not part of the RENAMES clause, but are used to clarify the purpose of the clause.

Parameters

`THROUGH` and `THRU` equivalent.

data-name-1 the name used to rename the item or items referenced by *data-name-2* and *data-name-3*. It cannot be a qualifier, and can only be qualified by the names of the associated 01, FD, or SD level entries.

data-name-2 and *data-name-3* must be names of elementary items or groups of elementary items in the same logical record. They must not be the same name, and neither may have an OCCURS clause in its data description, or be subordinate to an item that has an OCCURS clause in its description. Furthermore, no item within the range of the portion of the logical record being renamed can be variable in size, or can contain such an item. *Data-name-2* and *data-name-3* may be qualified.

If *data-name-2* is used alone (that is, the optional THROUGH phrase is unused), *data-name-2* can be either a group or an elementary item.

Description

When *data-name-2* is a group item, *data-name-1* is treated as a group item; when *data-name-2* is an elementary item, *data-name-1* is treated as an elementary item.

If the THROUGH phrase is used, *data-name-1* is a group item that includes all elementary items starting from *data-name-2* and concluding with elementary item *data-name-3*. Or, if *data-name-2* and *data-name-3* are also group names, *data-name-1* is a group name that begins with the first elementary item in *data-name-2* and concludes with the last elementary item in *data-name-3*.

Because of the way in which *data-name-1* is defined, there are restrictions on the area described by *data-name-2* and *data-name-3*. That is, the area described by *data-name-3* must not begin to the left of the first character in the area described by *data-name-2*, and it must end to the right of the last character of the area of *data-name-2*.

Example

```
01 STUDENT-REC.
   03 NAME           PIC X(20).
   03 ID-NO          PIC X(9).
   03 MAJOR          PIC X(3).
   03 CLASSES        OCCURS 5 TIMES.
      05 CLASS-ID    PIC X(3).
      05 DEPT        PIC X(3).
   66 STUD-INFO      RENAMES NAME THRU MAJOR.
   66 MAJOR-DEPT     RENAMES MAJOR.
```

In the above example, `CLASSES`, `CLASS-ID`, and `DEPT` may not be named in a RENAMES clause because of the OCCURS clause.

Note The above paragraph implies that *data-name-3* cannot be subordinate to *data-name-2*.

You can use more than one RENAMES entry for a single logical record. However, all RENAMES entries referring to data items within a given logical record must immediately follow the last data description entry of the associated record description entry. You cannot use a level 66 entry to rename another level 66 entry or a 77, 88, or 01 level entry.

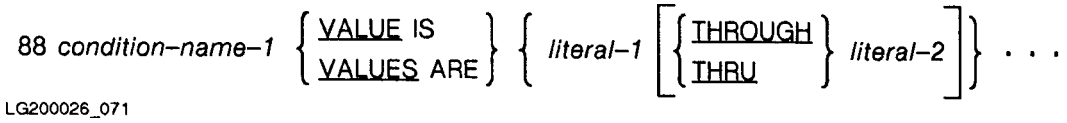
Condition Names

A *condition name* is always subordinate to another data item called the *conditional variable*. The level number 88 is used to describe it. The characteristics of a condition name are implicitly those of its conditional variable. This must be reflected in the value or values assigned to the condition name.

A condition name is assigned one or more values. The condition name can later be used to specify comparison with the conditional variable (see Chapter 8, "PROCEDURE DIVISION", for information on condition name conditions).

Syntax

The VALUE clause in a condition name has the following format:



LG200026_071

Parameters

condition-name-1	any valid user-defined COBOL word.
literal-1 and *literal-2*	the values assigned to the condition name.
`THROUGH` and `THRU`	equivalent and can be used interchangeably.

Description

The VALUE clause and the condition name itself are the only two clauses permitted in the data description entry.

The VALUE clause can be used in any of the sections of the DATA DIVISION and must be used for condition names.

Wherever the THROUGH phrase is used, *literal-1* must be less than *literal-2*.

When a VALUE clause is used in a level 88 entry, you can specify no more than 127 ranges of values for the related condition name. A range of values is either a single literal or two literals related by the THROUGH (or THRU) keyword.

Additional rules applying to the VALUE clause in condition names are described under the headings "Restrictions on the Use of the VALUE Clause" and "Literals in the VALUE Clause", earlier in this chapter.

The condition name entries for a particular conditional variable must follow the entry describing the item with which the condition name is associated (that is, the conditional variable). Each condition name in your program must have a separate level 88 entry associated with it. A condition name cannot be associated with any data description entry containing a level number 66, another condition name, or a group item with descriptions including JUSTIFIED, SYNCHRONIZED, or USAGE (other than DISPLAY).

Example

```
01 CONDVAR          PIC 9(5)       USAGE DISPLAY.
     88 COND1  VALUE 10 THRU 25,  100 THRU 250.

  01 ALPHAVAR        PIC A.
    88 ALPHACOND    VALUE "A" , "M" THROUGH "Z".
```

8

PROCEDURE DIVISION

The PROCEDURE DIVISION is the division that specifies the operations to be carried out by the program. It is an optional part of a COBOL program and may contain declarative as well as nondeclarative procedures.

Generally, statements in the PROCEDURE DIVISION are executed by the compiler in the order in which you enter them. However, you can alter this sequential flow by using one of the following statements: IF, PERFORM, GO TO, or `EVALUATE`.

Also, through the use of the DECLARATIVES keyword coupled with the END DECLARATIVES keywords, you can specify procedures to be executed only under special circumstances.

PROCEDURE DIVISION Header

The PROCEDURE DIVISION header has the following format:

PROCEDURE DIVISION [USING {*data-name-1*} ...]

The PROCEDURE DIVISION header begins in Area A of the program. Note that the header must be terminated by a period followed by a space.

USING Clause

The USING clause in the PROCEDURE DIVISION header is required only if the program containing it is to be called by another COBOL program through the CALL statement. The CALL statement itself includes a USING clause. That is, the USING clause in a PROCEDURE DIVISION header identifies the program in which it appears as a subprogram that references data common to the program that calls it.

The data names in the USING clause must follow the rules listed below.

1. Each data item named in the USING phrase of a PROCEDURE DIVISION header must be described as 01 or 77. In ANSI COBOL'85 a data item must not have a REDEFINES clause in its description. HP COBOL II allows this as an HP extension to the ANSI COBOL standard.
2. Data items are processed according to their descriptions in the called program, and not according to their descriptions in the calling program. Note that although this implies that common data may have different usages, the data must, as a general rule, have the same usage. Results may be undefined if usages are mixed. This is because data sharing is done by passing an address of the data item (when passing by reference), with no conversion from one data type to another.
3. The descriptions of data common to both programs must define an equal number of character positions.
4. Data is passed from one program to another according to the position of its name in the USING phrase, and not by its name. Thus, data in the calling program may be known to the calling program by a completely different name. The same name can appear in the USING phrase of the CALL statement, but each name in the USING phrase of a PROCEDURE DIVISION header must be unique with respect to other names in that phrase.
5. For the limit on the number of data names listed in the USING phrase, refer to "MPE XL System Dependencies" in Appendix H for more information.

For a more general overview of passing data to and from two COBOL programs, refer to Chapter 11, "Interprogram Communication."

PROCEDURE DIVISION Format

The body of the PROCEDURE DIVISION has two general formats:

Format 1

```
[ PROCEDURE DIVISION   [ USING { data-name-1 } . . . ] .
{ paragraph-name.
      [ sentence ] . . . } . . . ]
```

Format 2

```
[ PROCEDURE DIVISION   [ USING { data-name-1 } . . . ] .

[DECLARATIVES.

{ section-name SECTION  [segment-number].
        USE statement
[paragraph-name .
        [ sentence ] . . . ] . . . } . . .

 END DECLARATIVES . ]

 { section-name  SECTION [segment-number].
 [ paragraph-name .
 [ sentence ] . . .] . . . } . . . ]
```

LG200026_073

The first format of the PROCEDURE DIVISION body can be used when you wish to use no section names in your program. In such a case, only paragraph names are used to define procedures. This is generally not the best way to write a COBOL program, since it does not allow for USE procedures or segmentation of the object program. It may be beneficial, however, if you are writing a very short, simple program.

The second format of the PROCEDURE DIVISION body is used when you wish to allow for segmentation of your program or define declarative procedures.

In the second format, section names may be used to define procedures, with paragraph names being used as subsections. If any part of the PROCEDURE DIVISION is written using a section name, the entire PROCEDURE DIVISION must be written using section names. Therefore, if a section name is used, either the entire PROCEDURE DIVISION is a single section or the PROCEDURE DIVISION consists of several sections.

PROCEDURE DIVISION Syntax Rules

This section discusses syntax rules for the following areas:

- Declarative Sections
- Procedures
- Sections and Section Headers
- Segmentation

Declarative Sections

Declarative sections are optional. If used, they may appear in COBOL subprograms as well as main programs.

When you define declarative sections, they must be the first sections within the PROCEDURE DIVISION, be preceded by the keyword DECLARATIVES beginning in area A and followed by a period and a space, and be on a line by itself. To indicate where declarative sections end and the remainder of the PROCEDURE DIVISION begins, you must use the keywords END DECLARATIVES beginning in area A and followed by a period and a space, and be on a line by itself. USE procedures consist of a section name followed by a space and the keyword SECTION, followed by an optional segment number, a period and a space, a declarative sentence, and one or more optional paragraphs.

A declarative sentence is one that contains a USE statement. The USE statements themselves are not executed. They simply define the conditions calling for the execution of the USE procedures. The declarative procedures are the optional paragraphs following the declarative sentence.

A single USE procedure is terminated in a source program by either a new section name, which indicates the beginning of another declarative statement, or by the keywords END DECLARATIVES, which indicate the end of the list of declarative sections.

As the preceding paragraph implies, you must define a new section for each USE statement entered in the source program.

Declarative procedures must not reference nondeclarative procedures, although you may use a PERFORM statement in the nondeclarative portion of a program to refer to procedures associated with a USE statement.

Below is an example of the declaratives portion of a COBOL program:

```
PROCEDURE DIVISION.
DECLARATIVES.
IN-FILE-ERR SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON IN-FILE.
REPORT-ERR-PARA.
    DISPLAY "ERROR IN IN-FILE. ".
    DISPLAY "FILE-STATUS ITEM IS  " FILE-STAT.
    DISPLAY "WHAT ACTION?".
    DISPLAY "ENTER C OR A FOR CONTINUE OR ABORT".
    ACCEPT DECISION.
    IF DECISION IS EQUAL TO "A" MOVE "ON" TO STOP-IT.
FILE-LABEL SECTION.
    USE AFTER STANDARD BEGINNING FILE LABEL
    PROCEDURE ON OUT-FILE.
WRITE-LABEL-PARA.
    MOVE "HP 3000" TO LABEL-REC.
END DECLARATIVES.
```

In the above example, if an error occurs during execution of an OPEN, CLOSE, READ, WRITE, REWRITE, EXCLUSIVE, or UN-EXCLUSIVE statement referencing `IN-FILE`, and no AT END phrase is used in the statement, the USE procedure, `IN-FILE-ERR`, is executed.

When `OUTFILE` is opened, the second USE procedure called `FILE-LABEL`, is executed. This procedure creates a user label, and as an implicit part of its operation, writes the label on the file.

For more information on USE procedures, refer to "USE Statement" in Chapter 9.

Procedures

A procedure consists either of one or more paragraphs or sections. A procedure name is a word you choose that refers to a paragraph or section in the source program. A procedure name consists of a section name or a paragraph name. A paragraph name may be qualified.

The physical end of the PROCEDURE DIVISION is that physical position in the source program after which no further procedures appear.

Sections and Section Headers

A section consists of a section header followed by zero or more successive paragraphs. A paragraph consists of a paragraph name followed by a period, a space, and zero or more successive sentences. (Paragraphs, sentences, statements, and so forth are described in Chapter 2.)

In the PROCEDURE DIVISION, a section header consists of a section name followed by the word SECTION, an optional segment number, then a period. For example:

```
BEGIN-INITIALIZATION SECTION 05.
```

or:

```
END-INITIALIZATION SECTION.
```

If no section header is specified, the entire PROCEDURE DIVISION constitutes one section. Section names in both main programs and subprograms must be unique.

(For more information on section and paragraph names, refer to "MPE XL System Dependencies" in Appendix H.)

The segment number appearing in a section header is used to segment the PROCEDURE DIVISION. Thus, all sections with the same segment number constitute a program segment. If no segment number is specified, COBOL assumes it to be 0. Although sections with the same segment numbers are a part of the same segment, they need not be physically contiguous in the source program.

Segmentation

Segmentation is an obsolete feature of the 1985 ANSI COBOL standard.

Segmentation has no effect on the physical layout of the object program.

Segment Numbers

Sections in the DECLARATIVES portion of a PROCEDURE DIVISION must have segment numbers less than 50.

The term *initial state* refers to the original setting of GO TO statements before they are modified at run time by the ALTER statement Refer to the "ALTER Statement" in Chapter 9.

A segment with a segment number from 0 to 49 is in its initial state only when it is first used in a given run-unit. Upon subsequent entries into such a segment, its state is the same as when it was exited from a previous usage.

There are three exceptions where a segment with a segment number from 50 to 99 is always in its initial state whenever control is transferred to that section. The first exception concerns the appearance of a SORT, MERGE, or PERFORM statement, or any statement that implicitly calls a USE procedure, in a section whose segment number is greater than 49. When one of these statements implicitly transfers control to a procedure outside of the segment in which it appears, the segment is reentered in its last used state following the execution of the procedure.

The second exception is when a subprogram is called from a section whose segment number is greater than 49. In this case, if the EXIT PROGRAM or GOBACK statement is executed in the called program, the calling program is reentered at the statement following the CALL statement. If this statement is within the same segment as the CALL statement, the segment is in its last used state when it is reentered.

The third exception is when a PERFORM statement is executed. If the sections or paragraphs named in the PERFORM statement have segment numbers greater than 49, then the segment of which they are a part is in its initial state the first time it is executed. It remains in its last used state for all subsequent executions. Of course, following the completion of the PERFORM, the associated segment is again in its initial state.

Since segment numbers greater than 49 are always (with the noted exceptions) in their initial states when used, the compiler must initialize each section when control is passed to it, thus lengthening execution time. Modifying a GO TO statement in such a section from outside by using an ALTER statement in another section is impossible, since all GO TO statements in that section are set to their initial state once control is passed to that section.

(For more information on segmentation and internal naming conventions, refer to "MPE XL System Dependencies" in Appendix H.)

PROCEDURE DIVISION Statements and Sentences

There are three types of statements and sentences in the PROCEDURE DIVISION:

- Conditional statements and sentences.
- Compiler directing statements and sentences.
- Imperative statements and sentences.

Conditional Statements and Sentences

A conditional statement specifies that a condition is to be tested, and depending upon the truth value of the condition, determines the action of the object program.

HP COBOL II contains the following conditional statements:

- `EVALUATE`, IF, SEARCH, or RETURN statement.
- READ statement specifying the AT END, `NOT AT END`, INVALID KEY, `NOT INVALID KEY` phrase.
- WRITE statement specifying the INVALID KEY, `NOT INVALID KEY`, END-OF-PAGE or `NOT AT END-OF-PAGE` phrase.
- START, REWRITE, or DELETE statement specifying the INVALID KEY, or `NOT INVALID KEY` phrase.
- Arithmetic statements (ADD, COMPUTE, DIVIDE, MULTIPLY, or SUBTRACT) specifying the ON SIZE ERROR or `NOT ON SIZE ERROR` phrase.
- STRING or UNSTRING statements specifying the ON OVERFLOW or `NOT ON OVERFLOW` phrase.
- CALL statement specifying the ON OVERFLOW, ON EXCEPTION, or `NOT ON EXCEPTION` phrase.
- ACCEPT statement specifying the ON INPUT ERROR, or `NOT ON INPUT ERROR` phrase.

A conditional sentence is a conditional statement, optionally preceded by an imperative statement, terminated by a period followed by a space.

Compiler Directing Statements and Sentences

A compiler directing statement consists of a compiler directing verb (either COPY, USE or `REPLACE`) followed by the verb's operands. It causes the compiler to take a specific action during compilation.

A compiler directing sentence is a single compiler directing statement terminated by a period followed by a space.

Imperative Statements and Sentences

An imperative statement either begins with an imperative verb and specifies an unconditional action to be taken, or it is a conditional statement that is delimited by its explicit scope terminator (delimited scope statement). Scope terminators are described later in this chapter.

An imperative statement may consist of a sequence of one or more imperative statements.

Note that when the phrase *imperative-statement* appears in a format, it refers to that sequence of consecutive imperative statements that must be ended in one of the following ways:

- By a period.
- By an ELSE phrase associated with a previous IF statement.
- By a WHEN phrase associated with a previous SEARCH statement.
- By the verb's explicit scope terminator.

An imperative sentence is an imperative statement terminated by a period followed by a space. Verbs used in forming imperative statements are shown in Table 8-1 below.

Table 8-1. Imperative Verbs

ACCEPT[1]	EXCLUSIVE	RELEASE
ADD[2]	EXAMINE	REWRITE[3]
ALTER	EXIT	SET
CALL[4]	GO TO	SORT
CANCEL	GOBACK	START[3]
CLOSE	INITIALIZE	STOP
COMPUTE[2]	INSPECT	STRING[5]
CONTINUE	MERGE	SUBTRACT[2]
DELETE[3]	MOVE	TERMINATE
DISPLAY	MULTIPLY[2]	UN-EXCLUSIVE
DIVIDE[2]	OPEN	UNSTRING[5]
ENTER	PERFORM	WRITE[6]
EVALUATE	READ[7]	

1 Without the optional ON INPUT ERROR and NOT ON INPUT ERROR phrase.

2 Without the optional ON SIZE ERROR and NOT ON SIZE ERROR phrases.

3 Without the optional INVALID KEY and NOT INVALID KEY phrases.

4 Without the optional ON OVERFLOW, ON EXCEPTION, and NOT ON EXCEPTION phrases.

5 Without the optional ON OVERFLOW and NOT ON OVERFLOW phrases.

6 Without the optional INVALID KEY, NOT INVALID KEY, END-OF-PAGE, and NOT AT END-OF-PAGE phrases.

7 Without the optional AT END, NOT AT END, INVALID KEY, and NOT INVALID KEY phrases.

Categories of Statements

HP COBOL II statements fall into 11 categories. These categories, and the verbs used in them, are listed in Table 8-2.

Table 8-2. Categories of Statements

Category	Verbs
Arithmetic	ADD COMPUTE DIVIDE MULTIPLY SUBTRACT
Compiler Directing	COPY `REPLACE` USE
Conditional	ACCEPT (ON INPUT ERROR or `NOT ON INPUT ERROR`) ADD (SIZE ERROR or `NOT ON SIZE ERROR`) CALL (ON OVERFLOW, ON EXCEPTION, `NOT ON EXCEPTION`) COMPUTE (SIZE ERROR or `NOT ON SIZE ERROR`) DELETE (INVALID KEY or `NOT INVALID KEY`) DIVIDE (SIZE ERROR or `NOT ON SIZE ERROR`) `EVALUATE` IF MULTIPLY (SIZE ERROR or `NOT ON SIZE ERROR`) READ (AT END, `NOT AT END` INVALID KEY, or `NOT INVALID KEY`) RETURN (AT END or `NOT AT END`) REWRITE (INVALID KEY or `NOT INVALID KEY`) SEARCH START (INVALID KEY or `NOT INVALID KEY`) STRING (ON OVERFLOW or `NOT ON OVERFLOW`) SUBTRACT (SIZE ERROR or `NOT ON SIZE ERROR`) UNSTRING (ON OVERFLOW or `NOT ON OVERFLOW`) WRITE (INVALID KEY, `NOT INVALID KEY`, END-OF-PAGE, or `NOT AT END-OF-PAGE`)
Data Movement	ACCEPT (DATE, DAY, DAY-OF-WEEK, or TIME) EXAMINE (REPLACING) `INITIALIZE` INSPECT (REPLACING) `(CONVERTING)` MOVE STRING UNSTRING

Table 8-2. Categories of Statements (continued)

Category	Verbs
Ending	STOP STOP RUN GOBACK (in main program)
Input-Output	ACCEPT (identifier) CLOSE DELETE DISPLAY EXCLUSIVE OPEN READ REWRITE SEEK START STOP (literal) UN-EXCLUSIVE WRITE
Interprogram Communication	CALL CANCEL ENTRY EXIT PROGRAM GOBACK
Ordering	MERGE RELEASE RETURN SORT
Procedure Branching	ALTER CALL EXIT GO TO PERFORM
Table Handling	SEARCH SET
No Operation	`CONTINUE`

Scope Terminators

Scope terminators mark the end of the PROCEDURE DIVISION statements that contain them. There are two types of scope terminators: explicit and implicit.

The explicit scope terminators are:

```
END-ACCEPT          END-IF              END-SEARCH
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-EVALUATE
```

Examples

In the following example, the READ and IF statements have explicit scope terminators.

```
READ IN FILE
    AT END
       MOVE 'YES' TO EOF-SW
       IF IN-COUNT = 0
             DISPLAY "EMPTY FILE"
       END-IF
END-READ
```

The implicit scope terminators are:

- At the end of a sentence: the separator period, which terminates the scope of all previous statements not yet terminated.
- Within any statement containing another statement: the next phrase of the containing statement following the contained statement terminates the scope of any unterminated statement. ELSE, WHEN, and NOT AT END are examples of such phrases.

In the next example, the IF statement in line 2 is terminated by the ELSE clause on line 6. The IF statement on line 1 is terminated by the period (.) on line 7.

```
1 IF HOURS > 40
2    IF PAYCODE = NONEXEMPT
3       PERFORM OVERTIME
4    ELSE
5       PERFORM NORMAL-PAY
6 ELSE
7    PERFORM NORMAL-PAY.
```

Arithmetic Expressions

Arithmetic expressions are used in the COMPUTE statement and in relation conditions. They enable you to use exponentiation, as well as the addition, subtraction, multiplication, division, and negation operations that can be performed using arithmetic statements.

Arithmetic expressions allow you to combine arithmetic operations without the restrictions on "composites of operands" and receiving data items that exist for arithmetic statements. (See "COMPUTE Statement" in Chapter 9 for rules concerning calculation of intermediate results.)

For machine specific limitations on the maximum number of digits in arithmetic expressions, refer to the *HP COBOL II/XL Programmer's Guide*.

The number of decimal places used in evaluating an arithmetic expression is determined by the maximum number of decimal places within the expression and within the operand of a COMPUTE statement intended to receive the result.

An arithmetic expression can be any of the following:

- An identifier of a numeric elementary item or COBOL function.
- A numeric literal.
- Identifiers and literals as described above, 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.

Arithmetic Operators

There are five binary and two unary arithmetic operators. Each is represented by a specific character or characters. When an operator is used, it must be preceded and followed by a space.

The binary operators are listed below:

+	symbolizes addition.
-	symbolizes subtraction.
*	symbolizes multiplication.
/	symbolizes division.
**	symbolizes exponentiation.

The following are unary operators:

+	is equivalent to multiplying by +1.
-	is equivalent to multiplying by -1.

An arithmetic expression may only begin with a left parenthesis, a plus or minus sign, or an identifier or numeric literal. It may only end with an identifier or numeric literal, or with a right parenthesis.

Also, there must be a one-to-one correspondence between left and right parentheses, and each left parenthesis must be to the left of its corresponding right parenthesis.

Hierarchy of Operations

When parentheses are not used or they do not entirely enclose an arithmetic expression, the order in which the various operands are applied in evaluating the expression is determined in the following manner:

1. Any unary operator (+ or-) is executed first.
2. Following the execution of a unary operator, any exponent specified in the expression is executed.
3. Next, if multiplication or division is specified, the multiplication or division is executed. If consecutive multiplications and/or divisions are specified, each operation is performed in turn, starting from the left and continuing until the rightmost multiplication or division has been performed.
4. Following the execution of multiplication or division, any addition or subtraction specified in the expression is performed next. As with multiplication and division, if any consecutive combination of these operators is used, evaluation begins with the leftmost and terminates with the execution of the rightmost operator in the consecutive list.

In general, when the sequence of execution of an arithmetic expression is not specified by parentheses, the order of execution of consecutive operations of the same hierarchical level is from left to right.

For example, the following arithmetic expression:

```
-5 + 3 ** 2 * 4 + 7 - 21
```

Is evaluated as follows:

```
-5    is evaluated, resulting in -5.
3 ** 2  is evaluated, resulting in  9.
9 * 4   is evaluated, resulting in 36.
-5 + 36 is evaluated, resulting in 31.
31 + 7  is evaluated, resulting in 38.
38 - 21 is evaluated, resulting in 17.
```

Use of Parentheses

Parentheses may be used in arithmetic expressions to specify the order in which elements are to be evaluated. Always use parentheses in pairs.

Expressions within parentheses are evaluated first. Within nested parentheses, evaluation begins with the innermost set of parentheses, and continues outward until the expression contained in the outermost set is evaluated.

Use of parentheses allows you to eliminate ambiguities in logic where consecutive operations of the same hierarchical level appear, or to modify the normal hierarchical sequence of execution in an arithmetic expression.

To illustrate the use of parentheses, the following example uses the previous arithmetic expression. The following two expressions are equivalent:

```
-5 + 3 ** 2 * 4 + 7 - 21

(((-5 + ((3 ** 2) * 4)) + 7) - 21)
```

Both expressions result in 17. The following two expressions are also equivalent:

```
-5 + 3 ** (2 * 4) + 7 - 21

-5 + 3 ** 8 + 7 - 21
```

Both expressions result in 6542.

Valid Combinations in Arithmetic Expressions

The ways in which operators, variables, and parentheses may be combined in an arithmetic expression are summarized in Table 8-3 below. The word "Variable" means a numeric literal or an identifier of a numeric elementary item.

Table 8-3. Valid Combinations of Symbols in Arithmetic Expressions

First Symbol	Second Symbol				
	Variable	**+ - * / ****	**Unary + or -**	**(**	**)**
Variable	Not valid.	Permissible	Not valid.	Not valid.	Permissible.
+ - * / **	Permissible.	Not valid.	Permissible.	Permissible.	Not valid.
Unary + or -	Permissible.	Not valid.	Not valid.	Permissible.	Not valid.
(	Permissible.	Not valid.	Permissible.	Permissible.	Not valid.
)	Not valid.	Permissible.	Not valid.	Not valid.	Permissible.

Exponentiation

ANSI COBOL'85 defines the following special cases of exponentiation:

- If a value less than or equal to zero is raised to a power of zero, the size error condition results.
- If no real number exists as the result of the evaluation, the size error condition results.

Conditional Expressions

Conditional expressions identify conditions to be tested to select one of alternate paths of control. This selection is determined by the truth value of a condition. Conditional expressions are used in the IF, SEARCH, EVALUATE, and PERFORM statements.

There are two categories of conditions associated with conditional expressions:

- Simple conditions.
- Complex conditions.

You can enclose either category within any number of paired parentheses without changing its category. This section describes both simple conditions and complex conditions.

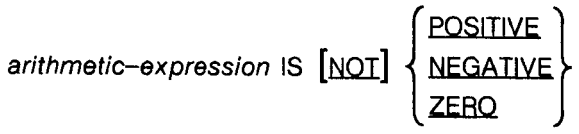
Simple Conditions

The six simple conditions are:

- Sign condition.
- Class condition.
- Switch-status condition.
- Relation condition.
- Condition name condition.
- Intrinsic relation conditions. Intrinsic relation conditions are an HP extension to the ANSI COBOL standard.

Sign Condition

The sign condition tests whether or not the algebraic value of an arithmetic expression is less than, greater than, or equal to zero. The sign condition has the following format:



LG200026_075

Parameters

arithmetic-expression	any valid arithmetic expression, as described on the preceding pages. It must contain at least one reference to a variable.
`NOT`	coupled with one of the next keywords, algebraically tests *arithmetic-expression*. If NOT POSITIVE is specified, return a value of "true" if *arithmetic-expression* is negative or equal to zero; return a value of "false" otherwise. If NOT NEGATIVE is specified, return a value of "true" if *arithmetic-expression* is equal to zero or positive; return a value of "false" otherwise. If NOT ZERO is specified, return a value of "true" if *arithmetic-expression* is positive or negative; return a value of "false" if *arithmetic-expression* is equal to zero.
`POSITIVE`, `NEGATIVE`, and `ZERO`	each used without the NOT keyword, algebraically tests *arithmetic-expression*. If POSITIVE is specified, return a value of "true" if *arithmetic-expression* is greater than zero; return a value of "false" otherwise. If NEGATIVE is specified, return a value of "true" if *arithmetic-expression* is less than zero; return a value of "false" otherwise. If ZERO is specified, return a value of "true" if *arithmetic-expression* is equal to zero; return a value of "false" otherwise.

Example

Assume that the variable `A` identifies the numeric value -5.

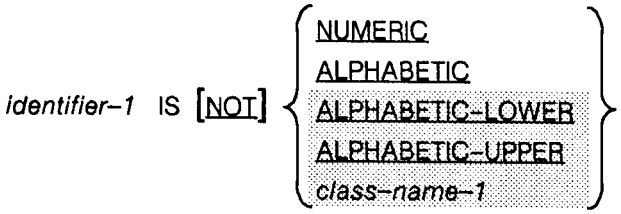
```
IF A IS ZERO NEXT SENTENCE
    ELSE DIVIDE A INTO SUMS.
```

In this example, because `A` is not zero, the statement `DIVIDE A INTO SUMS` is executed. If `A` were zero, the sentence immediately following the condition sentence would be executed.

Class Condition

The class condition determines whether an operand consists entirely of numbers and an operational sign, or letters, or a user-defined class.

Syntax



LG200026_076a

Parameters

identifier-1 names the operand to be tested. It must be described implicitly or explicitly as USAGE IS DISPLAY. Other restrictions apply if the keyword NUMERIC is used. HP COBOL II allows `PACKED-DECIMAL` items to be tested for NUMERIC as an HP extension to the ANSI COBOL standard. If *identifier-1* is a function-identifier, it must reference an alphanumeric function.

`NOT` coupled with one of the next keywords negates the condition.

`ALPHABETIC` means a value of "true" is returned if the operand consists entirely of characters selected from the letters a through z, A through Z, and a space. Otherwise a value of "false" is returned.

`NUMERIC` means a value of "true" is returned if the operand consists entirely of numerals selected from the set 0 through 9 and a single operational sign. Otherwise a value of "false" is returned.

`ALPHABETIC-LOWER` means a value of "true" is returned if the operand consists entirely of the lowercase letters a through z and space. Otherwise a value of "false" is returned.

`ALPHABETIC-UPPER` means a value of "true" is returned if the operand consists entirely of the uppercase letters A throuth Z and space. Otherwise a value of "false" is returned.

`class-name-1` means a value of "true" is returned if the operand consists entirely of the characters listed in the definition of class-name-1 in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

Description

You cannot use a NUMERIC test if the operand has a data description defining it as alphabetic, or as a group item composed of elementary items whose data descriptions indicate the presence of an operational sign or signs.

If the data description of the operand does not indicate the presence of an operational sign, the operand is considered numeric only if it consists of numerals, and has no operational sign.

If the data description of the operand does indicate an operational sign, the operand is considered numeric only if it consists of numerals from the set 0 through 9, and a single valid operational sign.

Valid operational signs are determined by the presence or absence of the SIGN IS SEPARATE clause in the data description of the operand.

If the SIGN IS SEPARATE clause is present, the valid operational signs are the standard data format characters, + and -.

If the SIGN IS SEPARATE clause is not present, the valid operational signs in standard data format are shown in Table 7-6 under the heading, USAGE IS DISPLAY.

The ALPHABETIC, `ALPHABETIC-LOWER, ALPHABETIC-UPPER, and class name` tests cannot be used with an operand whose data description describes it as numeric.

To illustrate the class condition, the following example uses an operand which, in standard data format, is 35798D.

Data description of operand:

```
01 FIRST-NUMBER PIC S9(6)  SIGN IS TRAILING.
```

Condition test:

```
FIRST-NUMBER IS NUMERIC
```

In this case, the test returns a value of “true”, since D is a valid operational sign. D has the value +4, thus making the numeral 35798D equivalent to +357984.

Switch-Status Condition

A switch-status condition determines the on or off status of a defined switch. The function name and the on or off value associated with the condition must be specified in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. The Switch-Status condition has the following format:

condition-name-1

Parameter

condition-name-1 — the name associated with the function name in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

The result of the test is true if the switch is set to the specified position corresponding to *condition-name-1*.

Example

```
ENVIRONMENT DIVISION
SPECIAL-NAMES.
    SW0, OFF STATUS IS NOADD, ON STATUS IS ADDONE.
        ⋮
PROCEDURE DIVISION.
PRINT-ROUTINE.
     IF NOADD THEN PERFORM OTHER-ACTION.
        ⋮
```

In the above example, if the status of switch SW0 is "off", then a routine named `OTHER-ACTION` is performed. If the status of switch SW0 is "on", then `OTHER-ACTION` is not performed, and control passes to the next executable statement.

Relation Conditions

There are two types of relation conditions in HP COBOL II. One is ANSI standard; the other is used for checking condition codes after intrinsic calls. The intrinsic relation condition is described following the description of ANSI standard relation conditions.

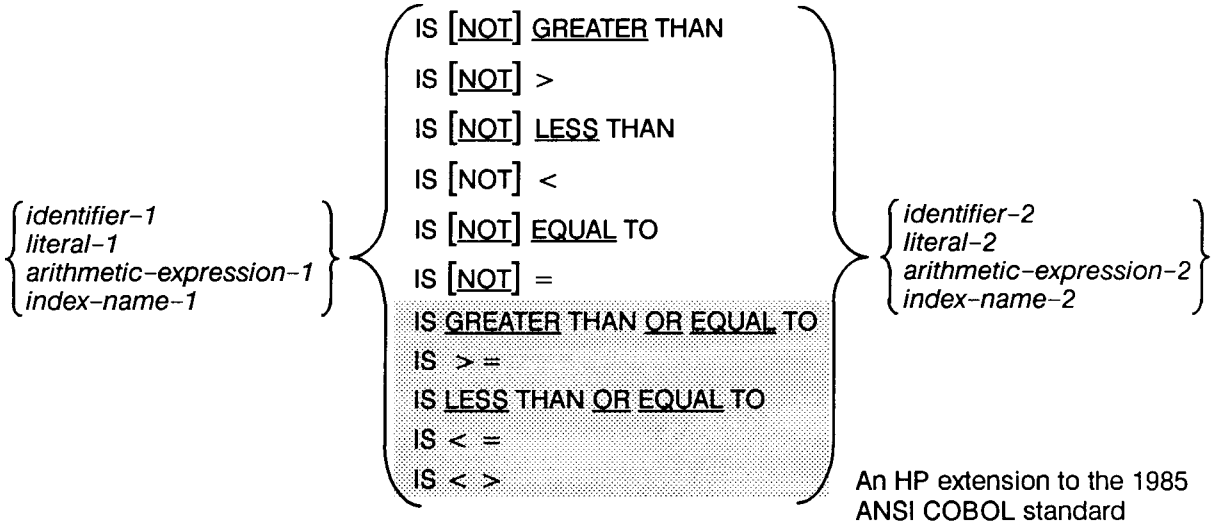
ANSI Standard Relation Conditions

A relation condition compares two operands, each of which may be a data item referenced by an identifier, a literal, or the value resulting from an arithmetic statement.

If a specified relation exists between the two operands, the relation condition value is "TRUE".

You may compare two numeric operands, regardless of their respective usages; however, if you want to compare two operands and one of them is not numeric, then both must have the same usage. Note that since group items are treated as alphanumeric, nonnumeric comparison rules apply.

A relation condition must contain at least one reference to a variable. Relation conditions have the following format:



LG200026_077a

Note The required relational characters '>' , '<' , '=' , '<=' , '>=' , and '<>' are not underlined to avoid confusion with other symbols such as '≥' (greater than or equal to).

Parameters

identifier-1 or *literal-1* or *arithmetic-expression-1* or *index-name-1*	the subject of the condition.
identifier-2 or *literal-2* or *arithmetic-expression-2* or *index-name-2*	the object of the condition.
[NOT] GREATER THAN	equivalent to [NOT] >
[NOT] LESS THAN	equivalent to [NOT] <
[NOT] EQUAL TO	equivalent to [NOT] = or < > (< > is an HP extension to the ANSI COBOL standard.)

NOT	coupled with the next keyword or relation character has the following meaning:
	NOT GREATER or NOT > means less than or equal;
	NOT LESS or NOT < means greater than or equal;
	NOT EQUAL or NOT = means greater than or less than.
	`GREATER THAN OR EQUAL TO is equivalent to >=.`
	`LESS THAN OR EQUAL TO is equivalent to <=.`

Comparison of Numeric Operands.

For operands belonging to the numeric class, a comparison is made with respect to the algebraic values of the operands. The number of digits in an operand is not significant. Also, no distinction is made between a signed or unsigned value of zero.

Comparison of numeric operands is not affected by their usages. Unsigned numeric operands are considered to be positive when they are used as operands in a comparison.

Comparisons Using Index Names and Index Data Items.

Relation tests may be made using any of the index names and index data items described below.

- Two index names. The result is the same as if the corresponding occurrence numbers were compared.
- An index name and a data item (other than an index data item) or literal. The occurrence number corresponding to the value of the index name is compared to the data item or literal.
- An index data item and an index name or another index data item. The actual values are compared without conversion.

An index data item should only be compared with another index data item or an index name. Comparison of an index data item with any other data item or a literal gives an undefined result.

Comparison of Nonnumeric Operands.

For nonnumeric operands, or one nonnumeric operand and one numeric operand, a comparison is made with respect to the specified collating sequence (refer to "OBJECT-COMPUTER Paragraph" in Chapter 6).

If one of the operands is numeric, it must be an integer data item or an integer literal. It must also have the same usage as the nonnumeric operand.

If the nonnumeric operand is an elementary data item or a literal, the numeric operand is treated as though it were moved to an elementary alphanumeric data item of the same size as the numeric data item. The contents of this alphanumeric data item are then compared to the nonnumeric operand.

If the nonnumeric operand is a group item, the numeric operand is treated as though it were moved to a group item of the same size as the numeric data item. The contents of this group item are then compared to the nonnumeric operand. Remember, a group item is always classified as alphanumeric.

Note In the previous paragraphs, "the same size as the numeric data item" means the size of the numeric data item in standard data format. If the P character of the PICTURE clause is included in the description for numeric operand, it must not be included in determining the size of the operand.

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

When operands are of unequal size, comparison proceeds as though the shorter operand is extended on the right by sufficient spaces to make the operands of equal size.

When operands are of equal size (or have been adjusted as described in the preceding paragraph), the comparison proceeds on a character-by-character basis, starting with each leftmost character and continuing until either the last character of each operand has been compared, or a pair of unmatched characters is found.

The operands are considered equal if each pair of characters match, from the leftmost to the rightmost.

The first time a pair of characters is found to be unequal (that is, do not match), their positions in the program collating sequence, and the character having the numerically larger index in the collating sequence, is considered to be greater than the other character.

Example

```
01 SUBJECT  PIC X(06) VALUE  'FLAXEN'.
01 OBJECT   PIC X(07) VALUE  'FLATTER'.
```

The relative condition is:

```
SUBJECT IS EQUAL TO OBJECT
```

The comparison takes place as follows:

```
F matches F; therefore, proceed.
L matches L; therefore, proceed.
A matches A; therefore, proceed.
X does not match T; therefore find the indices of each
   in the ASCII collating sequence;

Index of X: 88
Index of T: 84
```

`X` is greater than `T`; thus, `FLAXEN` is greater than `FLATTER`, and the relation condition above returns a "false" value.

Condition Name Conditions

In a condition name condition, a conditional variable is tested to determine whether or not its value is equal to one of the values associated with *condition-name*. Condition names have the following format:

condition-name-1

Parameter

condition-name-1	an identifier described under an 88 level data description entry in the DATA DIVISION.

If *condition-name* is associated with a range of values, then the conditional variable is tested to determine whether or not its value falls in this range, including the end values.

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

The result of such a comparison is true if one of the values of *condition-name* equals the value of its associated conditional variable.

Example

```
DATA DIVISION.
01 CON-VAR         PICTURE 999.
   88 CON-NAME1 VALUES ARE 001 THRU 100.
        ⋮
PROCEDURE DIVISION.
        ⋮
    IF CON-NAME1 THEN PERFORM UNDER-VALUE
     ELSE NEXT SENTENCE.
        ⋮
```

In the above example, the test is performed to see if `CON-VAR` has a value of 1, 100, or any number between 1 and 100. If it does, a procedure named `UNDER-VALUE` is performed. Otherwise, the next sentence is executed.

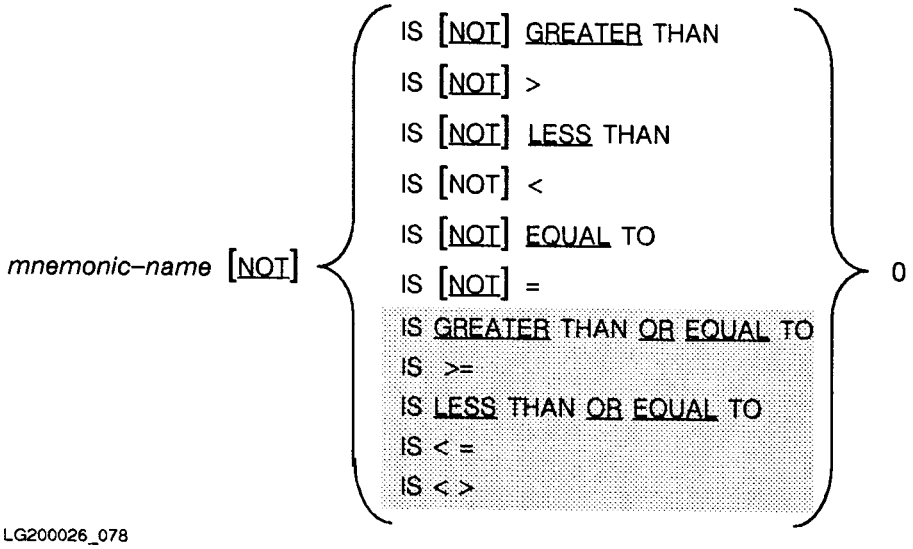
Intrinsic Relation Conditions

Intrinsic relation conditions are an HP extension to the ANSI COBOL standard.

Intrinsic relation conditions are used only to test the condition codes returned by HP operating system intrinsics after they have been called with the CALL INTRINSIC statement.

Syntax

Intrinsic relation conditions have the following format:



Note The required relational characters '>' , '<' , and '=' are not underlined to avoid confusion with other symbols.

Parameter

mnemonic-name — a user-defined name that represents the CONDITION-CODE function. It must be defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

Description

If *mnemonic-name* is zero, execution of the intrinsic was successful. If *mnemonic-name* is not zero, then an error probably occurred. Specific meanings are numerous, since they vary from intrinsic to intrinsic. Refer to the *MPE Intrinsics Reference Manual* for your system for meanings of the values returned. *Mnemonic-name* should be tested immediately after the intrinsic call, since the value may be altered by the execution of subsequent instructions.

Note When using CALLs to operating system intrinsics, the condition code returned is not saved by COBOL and as such is only available until an instruction is executed that changes the condition code. The condition code can be successfully tested with the following examples that illustrate possible correct and incorrect methods.

Examples

The following are correct and incorrect examples of intrinsic relation conditions.

Correct Example

The generated code for the example below causes control to branch to the following test only if the test result is FALSE and to the next sentence if the test result is TRUE. This method does not allow any intermediate operations, since such operations may cause the state of the condition code to change, thereby causing incorrect program logic flow.

```
IF CC = 0 DISPLAY " CC = "
ELSE
IF CC > 0 DISPLAY " CC > "
ELSE
IF CC < 0 DISPLAY " CC < ".
```

Incorrect Examples

The generated code for the incorrect case below causes incorrect condition code status branching for subsequent tests when the test result of the prior test is TRUE, causing execution of the DISPLAY statement.

```
IF CC = 0 DISPLAY " CC = ".
IF CC > 0 DISPLAY " CC > ".
IF CC < 0 DISPLAY " CC < ".
```

The generated code for the incorrect case below causes incorrect condition code status branching for subsequent tests when the OR condition test is required, as the operations needed to test the condition causes changes to the condition code.

```
IF CC > 0 OR FLAG-TRUE
DISPLAY "CC >  OR FLAG-TRUE"
ELSE
IF CC = 0 DISPLAY " CC = "
ELSE
IF CC <  0 DISPLAY " CC < ".
```

Complex Conditions

A complex condition is formed by using the logical operators AND and OR to combine simple, combined, and/or complex conditions, or by negating these conditions with the logical negation operator, NOT.

The truth value of a complex condition, regardless of the use of parentheses, results from the interaction of all the stated logical operators on the individual truth values of simple conditions. It can also be the result of the intermediate truth values of conditions logically connected or negated.

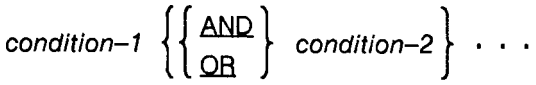
The meanings of the three logical operators are listed below.

Logical Operator	Meaning
`AND`	Logical conjunction; the truth value is "true" if both of the conjoined conditions are true and "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 is true and "false" if both included conditions are false.
`NOT`	Logical negation or reversal of truth value; the truth value is "true" if the condition is false and "false" if the condition is true.

When a logical operator is used, it must be preceded and followed by a space.

Combined Conditions

You can form a combined condition by connecting conditions with one of the logical operators, AND or OR. Combined conditions have the following format:



LG200026_081

Parameter

condition-1 where *condition-1* is one of five possible types of conditions:

- A simple condition.
- A negated simple condition.
- A combined condition.
- A negated combined condition (combined condition enclosed by parentheses and preceded by the NOT logical operator).
- Combinations of any of the four types listed above, as specified in Table 8-4.

Although you do not need parentheses when you form a combined condition using AND or OR, you may use parentheses to clarify and to affect the final result of a combined condition.

Table 8-4 indicates the ways in which conditions and logical operators can be combined and parenthesized.

There must be a one-to-one correspondence between left and right parentheses. Any left parenthesis must be to the left of its corresponding right parenthesis.

As an illustration of the use of the table, note that the pair OR NOT is acceptable, whereas NOT OR is not acceptable.

Table 8-4.
Valid Combinations of Conditions, Logical Operators, and Parentheses

Given the following element:	Location in conditional expression		In a left-to-right sequence of elements:	
	First	Last	Element, when not first, may be immediately preceded by only:	Element, when not last, may be immediately followed by only:
simple-condition	Yes	Yes	OR, NOT, AND, (	OR, AND,)
OR or 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,)

Negated Simple Conditions

The NOT operator can be used to negate simple conditions.

A negated simple condition has a value of "true" only if the value of the simple condition is "false". Conversely, a negated simple condition has a value of "false" only if the simple condition itself has a value of "true".

Negated simple conditions have the following format:

NOT *condition-1*

Parameter

condition-1 where *condition-1* is one of five possible types of conditions:

- A simple condition.
- A negated simple condition.
- A combined condition.
- A negated combined condition (combined condition enclosed by parentheses and preceded by the NOT logical operator).
- Combinations of any of the four types listed above, as specified in Table 8-4.

Example

```
IF CON-NAME1 THEN PERFORM UNDER-VALUE
         ELSE NEXT SENTENCE.
```

is equivalent to:

```
IF NOT CON-NAME1 THEN NEXT SENTENCE
         ELSE PERFORM UNDER-VALUE.
```

Condition Evaluation Rules

You may use parentheses to specify the order in which individual conditions of complex conditions are to be evaluated when you wish to modify the precedence rules (as listed below) for evaluating such conditions.

Conditions within parentheses are evaluated first. Within nested parentheses, the condition bounded by the innermost set is evaluated first, followed by the condition within the next innermost set. The process continues until the condition within the outermost set of parentheses is evaluated.

When parentheses are not used, or when they completely contain a condition, the following rules (in the order listed) are used to determine the truth value:

1. The order of precedence of logical operators is NOT, AND, OR. The order of precedence establishes hierarchical levels of conditions at the same precedence level.
2. Conditions in the same hierarchical level are evaluated from left to right. Evaluation of that level terminates as soon as a truth value for the level is determined, regardless of whether all the constituent conditions within that level have been evaluated.
3. Values are established for arithmetic expressions if and when it is necessary to evaluate them.
4. Negated conditions are evaluated if and when it is necessary to evaluate them.

Application of the above rules is illustrated in Figure 8-1 through Figure 8-4 on the following pages.

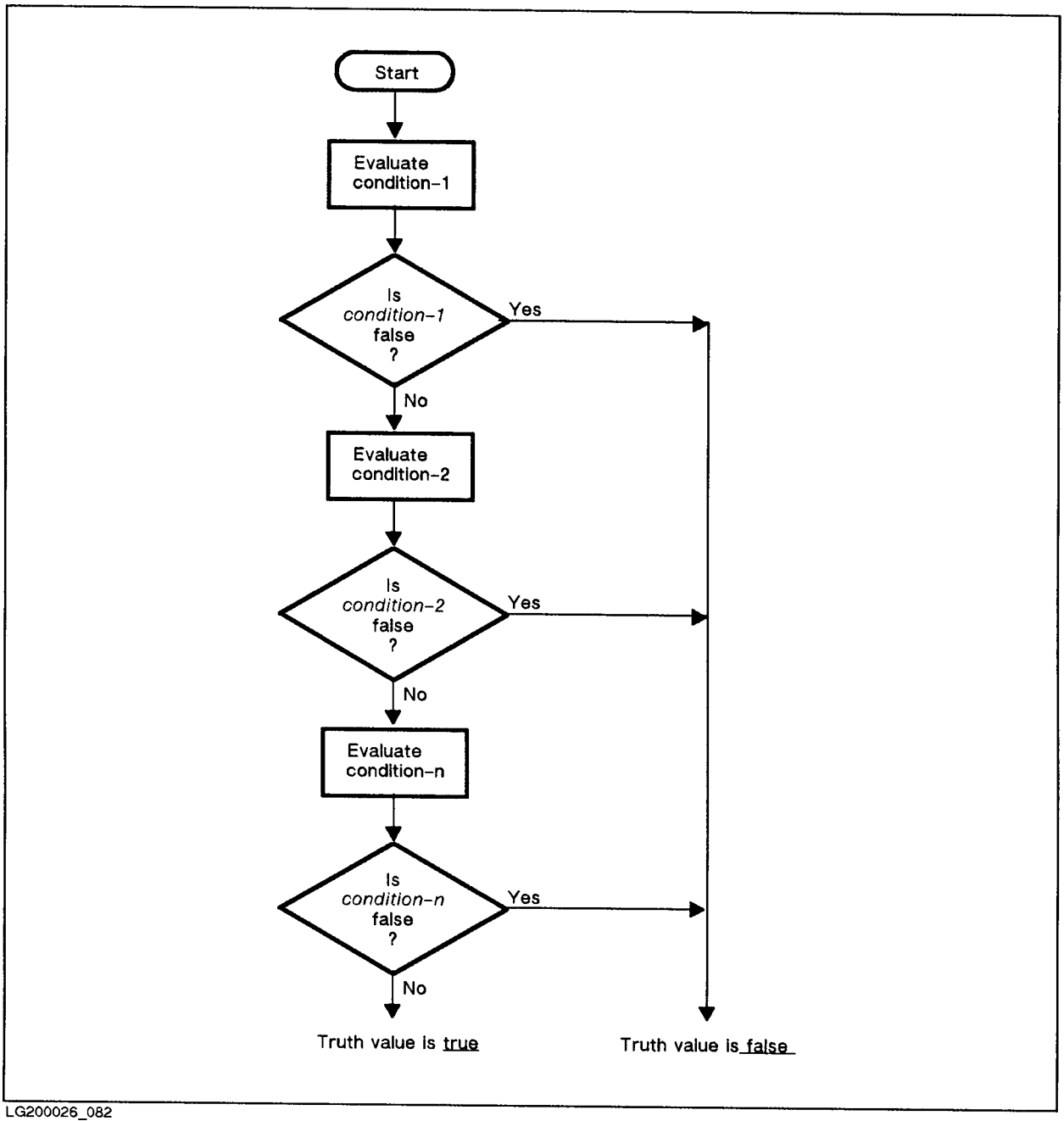
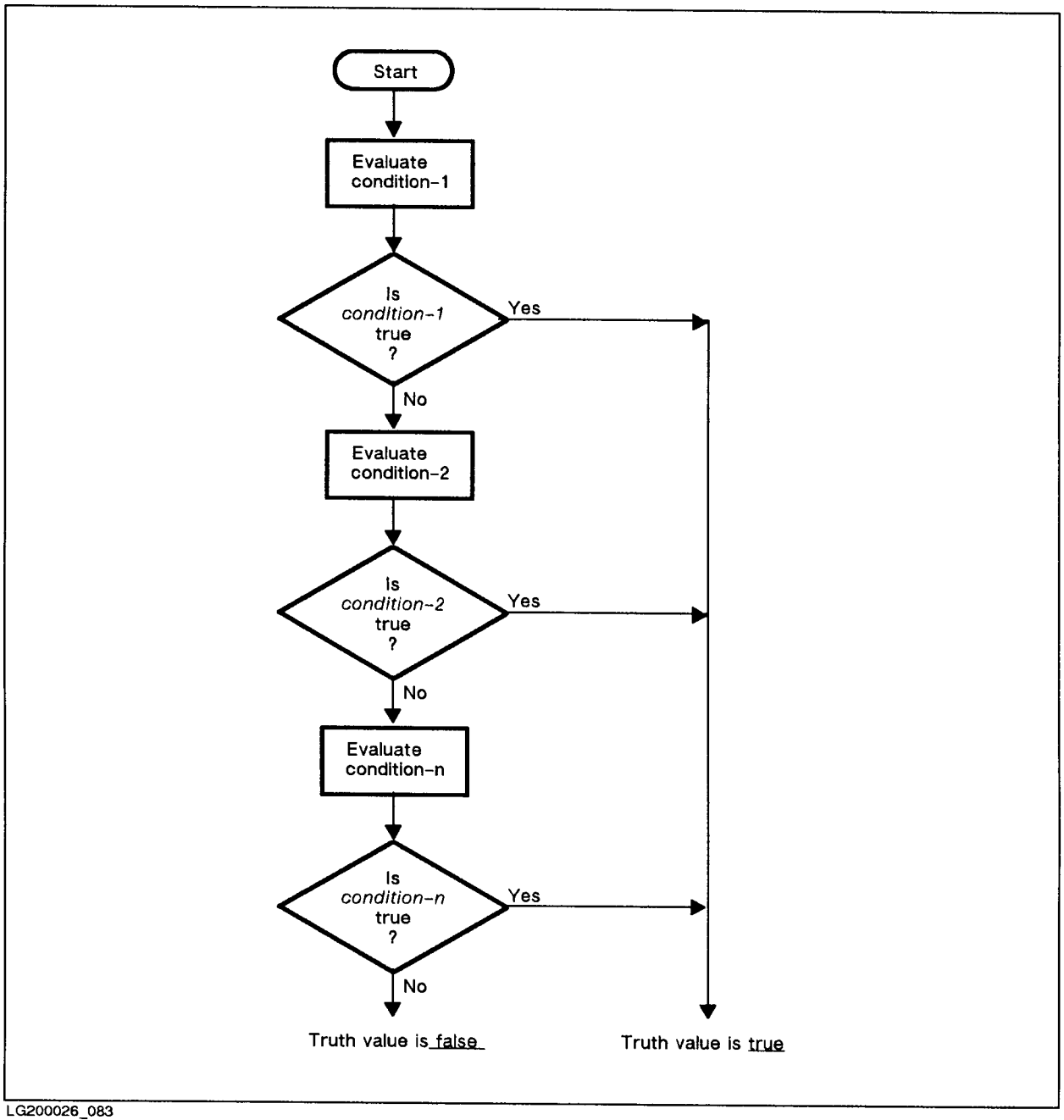
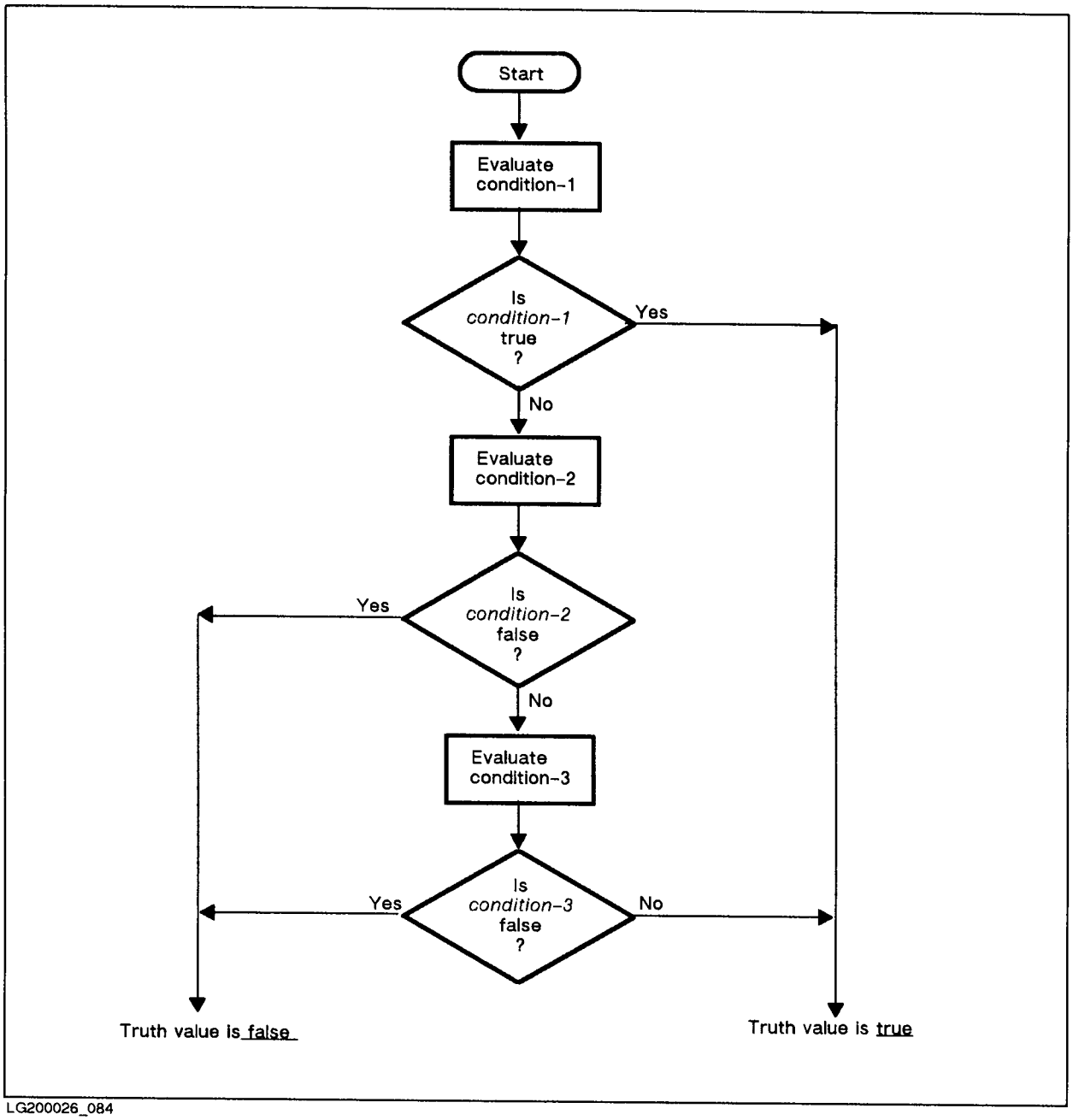


Figure 8-1.
Evaluation of the hierarchical level condition-1 and condition-2 and ... condition-n



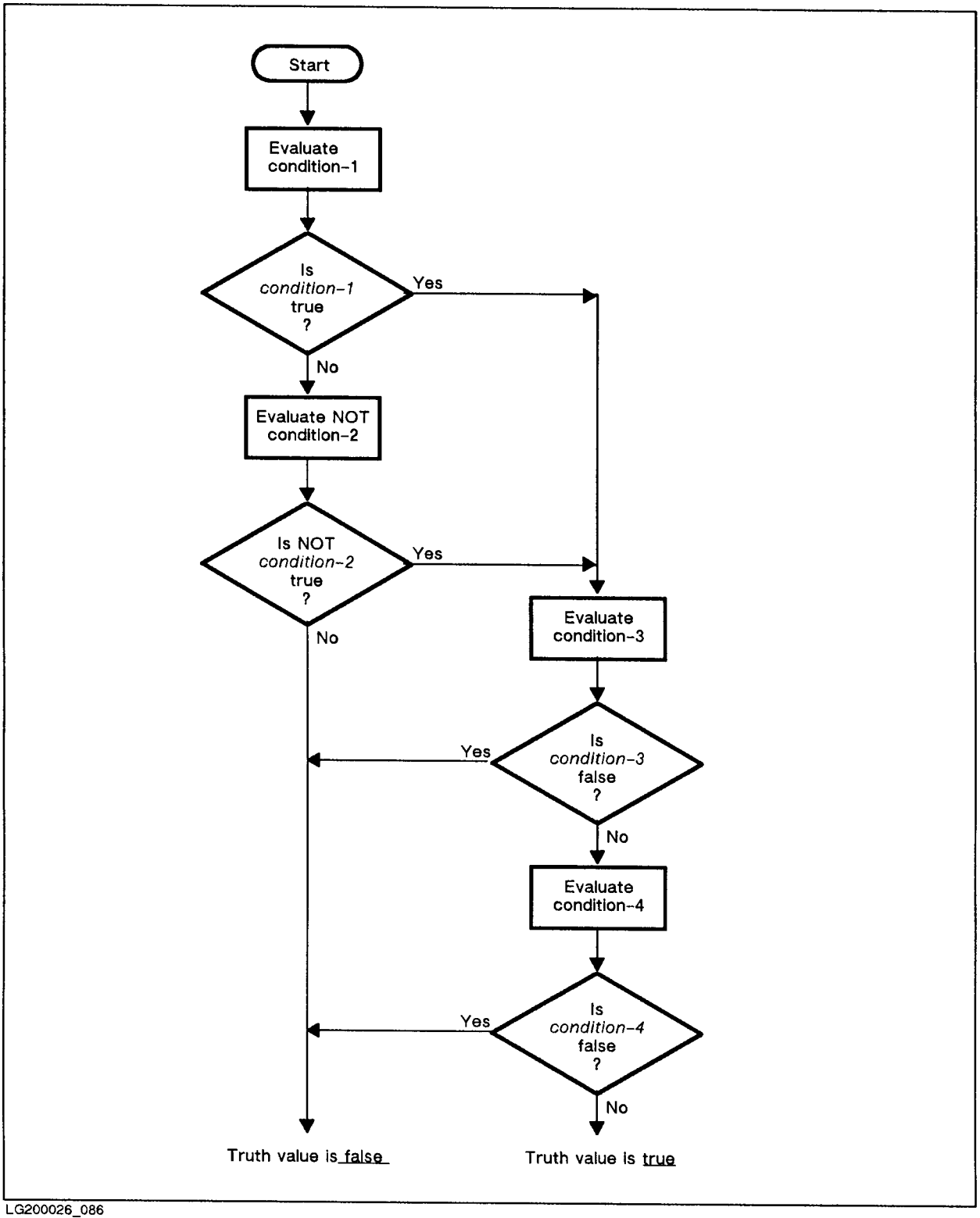
LG200026_083

Figure 8-2. Evaluation of the hierarchical level condition-1 or condition-2 or ... condition-n



LG200026_084

Figure 8-3. Evaluation of condition-1 or condition-2 and condition-3



LG200026_086

Figure 8-4. Evaluation of (condition-1 or not condition-2) and condition-3 and condition-4

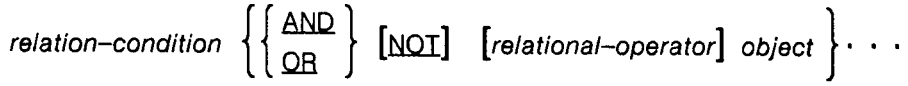
Abbreviated Combined Relation Conditions

If you combine simple or negated simple relation conditions with logical connectives (AND and OR) in a consecutive sequence in such a way that:

- No parentheses are used in the consecutive sequence, and
- A succeeding relation condition contains the same subject as the preceding relation condition, or
- A succeeding relation condition contains the same subject and the same relational operator,

you can abbreviate any relation condition, except the first, within the consecutive sequence.

There are two ways by which you can abbreviate such relation conditions. The first is by omitting the subject of the relation condition; the second is by omitting the subject and the relational operator of the relation condition. Abbreviated combined relation conditions have the following format:



LG200026_087

The effect of using such abbreviations is that the last preceding stated subject is inserted in place of the omitted subject, and the last stated relational operator is inserted in place of the relational operator.

In order to ensure that an abbreviated relation condition is valid, insert the omitted subject and relational operator. If, after insertion, the combined relation condition is valid according to the rules in Table 8-4 above, the abbreviated relation condition is valid.

The end of an abbreviated relation condition is signified by the first occurrence of a complete simple condition within a complex condition.

The word NOT can be used in two different ways: as part of a relational operator or as the logical negation operator. The rules for its usage in an abbreviated combined relation condition are as follows:

- If the word immediately following NOT is one of the following: GREATER, LESS, EQUAL, or one of the equivalent symbols (>, <, =), then NOT participates as part of the relational condition.
- If the word immediately following NOT is not one of those listed in the above paragraph, it is considered to be the negation operator. Thus, it negates only the first occurrence of the abbreviated relation condition.

Examples

```
A > B AND NOT < C OR D
```

is equivalent to:

```
((A > B) AND (A NOT < C))
OR (A NOT < D)
```

```
A NOT EQUAL B OR C
```

is equivalent to:

```
(A NOT EQUAL B) OR (A NOT EQUAL C)
```

```
NOT A = B OR C
```

is equivalent to:

```
(NOT (A = B)) OR (A = C)
```

```
NOT (A GREATER B OR < C)
```

is equivalent to:

```
NOT ((A GREATER B) OR (A < C))
```

```
NOT (A NOT > B AND C AND NOT D)
```

is equivalent to:

```
NOT ((((A NOT > B) AND ( A NOT > C)) AND (NOT (A NOT > D))))
```

Common Phrases

The NOT, ROUNDED, SIZE ERROR, and CORRESPONDING phrases are common phrases used in several PROCEDURE DIVISION statements. In order to avoid describing each of these phrases each time they appear in a particular statement, they are described just once in the following paragraphs.

In the description that follows, the term *resultant identifier* means the identifier associated with a result of an arithmetic operation.

NOT Phrases

The NOT phrases are a feature of the 1985 ANSI COBOL standard.

You can use NOT phrases with the statements that have conditionally executed exception phrases. The imperative statements in the NOT phrases execute when the exception does not occur. Table 8-5 lists the new NOT phrases and their associated verbs.

Table 8-5. NOT Phrases and Associated Verbs

Phrase	Statement
NOT AT END-OF-PAGE	WRITE
NOT AT END	READ RETURN
NOT INVALID KEY	DELETE READ REWRITE START WRITE
NOT ON EXCEPTION	CALL
NOT ON OVERFLOW	STRING UNSTRING
NOT ON SIZE ERROR	ADD COMPUTE DIVIDE MULTIPLY SUBTRACT
NOT ON INPUT ERROR	ACCEPT

ROUNDED Phrase

The ROUNDED phrase consists entirely of the keyword, ROUNDED.

In an arithmetic operation, if, after decimal point alignment, there are more decimal places in the fraction of the result than is specified for the resultant identifier, truncation is performed on the result. The number of digits truncated is dependent upon the number of decimal places specified for the fractional part of the resultant identifier.

If you want to round the result before truncation occurs, you can use the ROUNDED option.

If the ROUNDED phrase is specified in an arithmetic operation, the absolute value of the resultant identifier is increased by one whenever the most significant digit of the excess portion of the result is greater than or equal to 5. The excess portion is then truncated.

When the low-order integer positions in a resultant identifier are represented by the P character in the PICTURE clause of that resultant identifier, rounding occurs relative to the rightmost integer position for which storage is allocated.

SIZE ERROR Phrase

The SIZE ERROR phrase has the format

`[ON SIZE ERROR` *imperative-statement*`]`

where *imperative-statement* is one or more imperative statements.

If, after decimal point alignment, the number of digits in a result exceeds the number of digits specified for the associated resultant identifier, a SIZE ERROR condition exists.

The *imperative-statement* is executed if a SIZE ERROR condition occurs.

The SIZE ERROR condition applies only to the final result of most arithmetic operations; it applies to intermediate results, however, when the MULTIPLY, DIVIDE, and COMPUTE statements are used.

Note that division by 0 (zero) and violation of the rules for exponentiation always forces a SIZE ERROR condition.

If the ROUNDED phrase is specified in an arithmetic operation, rounding is done before a SIZE ERROR check is performed.

Note When a SIZE ERROR condition occurs and the SIZE ERROR phrase is not specified, the values of any resultant identifiers affected are undefined.

If other resultant identifiers are involved in a particular arithmetic operation for which a SIZE ERROR condition occurs, their values are unaffected. Only the resultant identifiers for which the SIZE ERROR occurs have undefined values.

Example 1

If the following arithmetic operation forces a SIZE ERROR condition for `B`, but not for `C`, only `B` has an undefined value:

```
ADD A TO B, C
```

When the SIZE ERROR phrase is specified for an arithmetic operation and a SIZE ERROR condition exists for the values of one or more of the resultant identifiers involved, their values remain as they were before the operation was executed.

Values of other resultant identifiers involved in the operation are unaffected by size errors. Therefore, their values are changed according to the arithmetic operation specified.

The SIZE ERROR phrase includes an *imperative-statement* following the words SIZE ERROR. This statement is executed following the occurrence of a size error in an arithmetic statement for which the SIZE ERROR phrase is specified.

Example 2

```
WORKING-STORAGE SECTION.

01 SIZE-ERR.
   02 NOTIFY    PIC X(10) VALUE 'SIZE ERROR'.
   02 PARAMETERS.
      03 PARM-1    PIC Z(18)  VALUE 0.
      03 PARM-2    PIC Z(18)  VALUE 0.
PROCEDURE DIVISION.
       .
       .
    ADD A B TO C D ROUNDED
       ON SIZE ERROR PERFORM NOTIFICATION.
       .
       .
NOTIFICATION.
    MOVE C TO PARM-1.
    MOVE D TO PARM-2.
    WRITE SIZE-ERR AFTER ADVANCING 1 LINE.
       .
       .
```

If an ADD or SUBTRACT statement uses the CORRESPONDING phrase as well as the SIZE ERROR phrase and an operation produces a size error condition, the imperative statement in the SIZE ERROR phrase is not executed until all individual additions or subtractions are completed.

CORRESPONDING Phrase

The CORRESPONDING phrase consists entirely of the word CORRESPONDING, or of the equivalent abbreviation, CORR. The purpose of the CORRESPONDING phrase is to allow you to add, subtract, or move a data item subordinate to a group item to a data item subordinate to some other group item.

Two data items are said to correspond if three conditions are met. For purposes of description, assume that D1 and D2 are group items.

A data item from D1 is said to correspond to a data item from D2 if:

1. Both of the data items have the same name, the name is not FILLER, and both have the same qualifiers up to, but not including D1 and D2.
2. When the CORRESPONDING phrase is being used in a MOVE statement, at least one of the data items is an elementary data item; when the CORRESPONDING phrase is used in an ADD or SUBTRACT statement, both data items are elementary data items.
3. The descriptions of D1 and D2 do not contain a 66, 77, or 88 level number and do not contain a USAGE IS INDEX clause.

Any data item that is a candidate for use in a CORRESPONDING phrase is ignored if it contains a REDEFINES, RENAMES, OCCURS, or USAGE IS INDEX clause, even if it meets the conditions above. Furthermore, any data items subordinate to such a data item are also ignored.

These restrictions do not apply to D1 and D2, except as noted in condition 3 above.

Example

```
01 FIRST-DATA.
   02 ENTRY-1.
      03 ENTRY-1A    PIC 9(5)V99.
      03 ENTRY-1B    PIC 9(3)V99.
   02 ENTRY-2        PIC X(30).

01 SECOND-DATA.
   02 ENTRY-1.
      03 ENTRY-1A    PIC 99V99.
      03 ENTRY-1B    PIC 999.
   02 FINISH         PIC X(20).
```

ENTRY-1A of FIRST-DATA corresponds to ENTRY-1A of SECOND-DATA, and ENTRY-1B of FIRST-DATA corresponds to ENTRY-1B of SECOND-DATA.

ENTRY-1 of FIRST-DATA does not correspond to ENTRY-1 of SECOND-DATA because of the second condition of correspondence.

The ADD statement below uses the CORRESPONDING phrase to add ENTRY-1A of FIRST-DATA to ENTRY-1A of SECOND-DATA, and ENTRY-1B of FIRST-DATA to ENTRY-1B of SECOND-DATA. The results are stored in ENTRY-1A and ENTRY-1B of SECOND-DATA.

```
ADD CORRESPONDING FIRST-DATA TO SECOND-DATA.
```

Note There is a limit of approximately 500 matching pairs allowed in a single MOVE CORRESPONDING statement. Multiple MOVEs are necessary to exceed this limit. Compiler errors 390 and 457 are given for this condition.

Common Features of Arithmetic Statements

The five arithmetic statements, ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE have the following features in common:

1. The data descriptions of operands in an arithmetic statement need not be the same.

   If operands are of mixed types, the compiler generates any data conversion routines necessary to format the data. Note that this does increase the size of the code space.

   If the operands are already defined as COMPUTATIONAL SYNCHRONIZED, the compiler does not have to generate conversion routines. This reduces the object program size and its execution time. Therefore, to maximize efficiency of arithmetic operations, remember to define the operands as being COMPUTATIONAL SYNCHRONIZED or with a usage of COMPUTATIONAL-3.

   For more information, see the *HP COBOL II/XL Programmer's Guide*.

2. The maximum size of each operand is 18 digits. The composite of operands must not contain more than 18 decimal digits.

   The composite of operands is the hypothetical data item resulting from the superimposition of specified operands in an arithmetic statement after the operands have been aligned on their decimal points.

   For example, in format 1 of the ADD statement, the composite of operands is determined by using all of the operands in a given statement.

   Therefore, if `A` = 1234.567, `B` = 1.2359, and `C` = 10340.77, the composite of operands of the statement `ADD A, B TO C` is 10340.2359.

   This number was arrived at by selecting the operand with the greatest number of digits to the right of the decimal point (in this case, 1.2359), and then the operand with the greatest number of digits to the left of the decimal point (which is 10340.77).

   These two operands were then superimposed, with the larger number to the left or right of the decimal point masking the smaller.

3. Arithmetic statements can have multiple results. For example, the following ADD statement gives the multiple results, `A + B`, `A + C`, and `A + D`:

   ```
   ADD A TO B, C, D.
   ```

   Such statements behave as though they had been written in the following way:

   a. A statement was first written that performs the specified arithmetic operation, and stores the results in a temporary location.

   b. A sequence of statements was then written that transfers or combines the value in the temporary location with each of the single data items specified as a result in the original arithmetic statement. This hypothetical sequence of statements was written to perform the transferring or combining of the temporary value in the same left-to-right sequence as the multiple results are listed.

Example

The following example illustrates how a temporary location is used in an ADD statement:

```
ADD A, B, C TO C, D, E
```

The above ADD statement is equivalent to the following ADD statements, where TEMP is a temporary location that stores the intermediate result:

```
ADD A,B,C GIVING TEMP
          ADD TEMP TO C
          ADD TEMP TO D
          ADD TEMP TO E
```

Overlapping Operands and Incompatible Data

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

Furthermore, except for a class condition, when the contents of a data item are referenced in the PROCEDURE DIVISION and the contents of that data item are not compatible with the class, the sign, or the range of values specified by its PICTURE clause, the result of such a reference is undefined.

Variable-Length Receiving Items

In ANSI COBOL'85, when a receiving item is a variable-length data item and contains the object of the DEPENDING ON phrase, the maximum length of the item is used.

In ANSI COBOL'74, the length is computed using the object of the DEPENDING ON phrase.

Input-Output Error Handling Procedures

Input-output error handling procedures are controlled by the following programming options in the sequence shown.

- First, the FILE STATUS item (if declared) is set for the associated file.
- Second, the INVALID KEY, AT END, or AT EOP phrases on selected I-O statements are executed.
- Third, if no INVALID KEY, AT END, or AT EOP is executed, a USE statement and its associated procedure is executed.

Ten input-output statements allow exception condition processing with a USE procedure, a FILE STATUS item, or the INVALID KEY, AT END, or AT EOP phrases. Table 8-6 lists these statements and the kinds of exception condition processing each one can use.

Table 8-6. Input-Output Statements and Exception Condition Options

Statement	USE Procedure	FILE STATUS Item	INVALID KEY AT END AT EOP
CLOSE	Yes	Yes	No
DELETE	Yes	Yes	Yes
EXCLUSIVE	Yes	Yes	No
OPEN	Yes	Yes	No
READ	Yes	Yes	Yes
RETURN	No	No	Yes
REWRITE	Yes	Yes	Yes
START	Yes	Yes	Yes
UN-EXCLUSIVE	Yes	Yes	No
WRITE	Yes	Yes	Yes

If an exception condition occurs and:

- Neither a FILE STATUS item, USE statement procedure, INVALID KEY or AT END phrase is specified, the program is aborted along with a file system tombstone display.
- An INVALID KEY or AT END phrase is specified, with no USE statement specification, and an INVALID KEY or AT END condition occurs, the program continues executing in-line code. If a USE procedure was specified, it is executed.
- There are any errors in DISPLAY, ACCEPT (without ON INPUT ERROR), SORT, MERGE and RELEASE, the program aborts.

These precedence rules are also defined in Figure 8-5, which includes tests for the ANSI COBOL'85 clauses NOT INVALID KEY and NOT AT END.

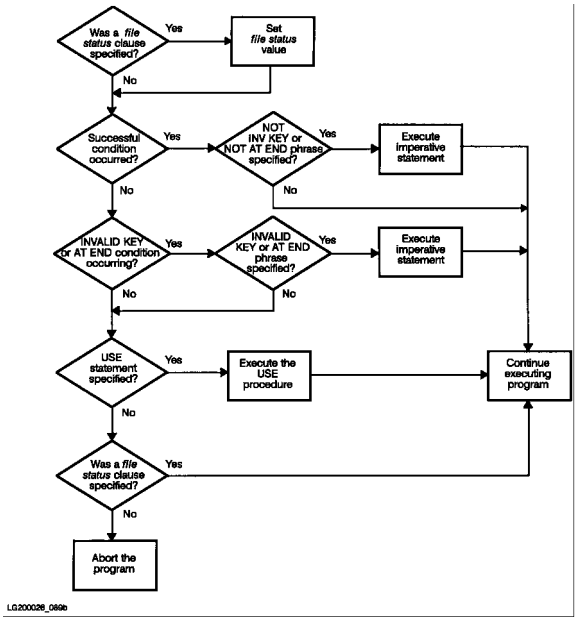


Figure 8-5. Input-Output Error Handling

9

PROCEDURE DIVISION Statements

All statements that can be used in the PROCEDURE DIVISION are described in alphabetical order in this chapter.

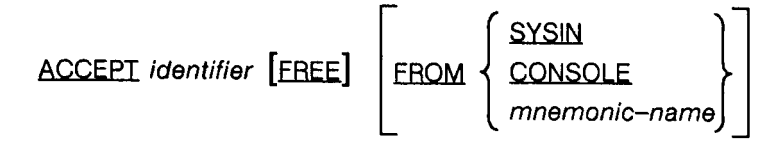
ACCEPT Statement

The ACCEPT statement can be used for low volume input from a specified device.

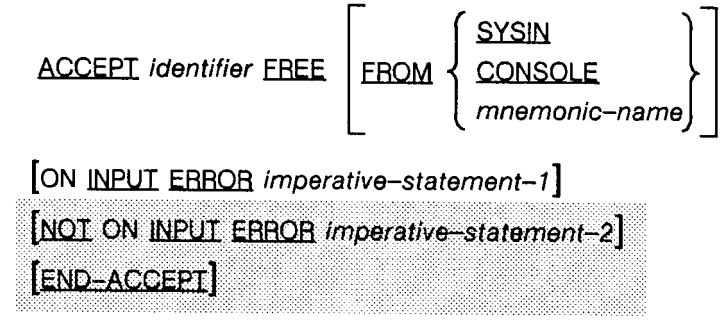
Syntax

ACCEPT has three general formats, as shown below:

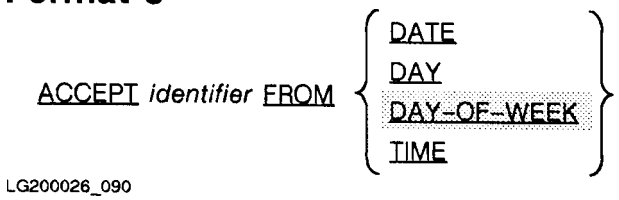
Format 1



Format 2



Format 3



LG200026_090

Parameters

identifier — a valid data name; it receives the data entered by the execution of the ACCEPT statement.

`SYSIN` — in a batch job, the input stream file; in a session, this name indicates the terminal used to initiate execution of your program. There is no indication of a pending user response; thus, you should use the DISPLAY statement immediately before the ACCEPT statement to indicate that the ACCEPT statement is awaiting input. The SYSIN device for jobs (that is, in batch mode), is the stream file. For sessions, this is your terminal.

When the FROM option is not specified, the compiler assumes the SYSIN device.

Note Special care must be taken when the SYSIN device is used for the ACCEPT statement, and the program is running in batch mode. In this case, the ACCEPT statement simply reads the next record of the job stream. Without careful planning, this record could be a data record, or an MPE command.

`CONSOLE` — the operator's console. When an ACCEPT statement specifying the FROM mnemonic name for CONSOLE option or FROM CONSOLE is executed, the field specified by *identifier* must not exceed 31 characters and the following actions result:

1. A system-generated message is automatically displayed at the console, followed by the message, AWAITING REPLY.
2. Object program execution is suspended.
3. When the computer operator enters the input data requested, this data is moved to the field specified by the identifier. Data positioning and/or conversion is performed subject to whether the FREE phrase specification was included.

Note I-O errors that occur during execution of ACCEPT do not produce a "tombstone" since the I-O is not done by the file system.

---

mnemonic-name — a name assigned in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. It must be a name for either SYSIN or CONSOLE, and has the same effect as the device name to which it is equated.

imperative-statement — one or more imperative statements. The INPUT ERROR phrase in which it appears can only be used if the FREE phrase is used.

`DATE` — composed of the year of the century, month of year, and day of month, in that order. Thus, for example, February 16, 1985 is transmitted as 850216. COBOL moves this data as an unsigned elementary numeric integer data item six digits in length.

DAY	composed of the year of the century and day of the year, in that order. For example, February 16, 1985 is accessed as 85047. COBOL moves this data as an unsigned elementary numeric integer data item five digits in length.
DAY-OF-WEEK	composed of a single data element whose content represents the day of the week. The value, 1 represents Monday, 2 represents Tuesday, ... , and 7 represents Sunday. COBOL moves this data as an unsigned elementary numeric data item one digit long.
TIME	the time of day, taken from a 24 hour clock, in hours, minutes, seconds and tenths of a second. The minimum value of time is 00000000, and the maximum is 23595990. COBOL moves this data as an unsigned elementary numeric integer data item eight digits long.

ACCEPT Statement - Formats 1 and 2

When formats 1 and 2 are used, data is accepted from an input spool file (if your program is running in batch mode), the terminal from which your program is executed (if it is running in session mode), or from the operator's console (if the CONSOLE option is used). This data is then used to replace the contents of the data item named by *identifier*.

FREE and INPUT ERROR Phrases

The FREE and INPUT ERROR phrases are HP extensions to the 1985 ANSI COBOL standard.

The FREE phrase allows you to use free-field format to enter data.

The INPUT ERROR phrases may also be used if the FREE phrase has been specified. They may not, however, be specified if the FREE phrase is not. This is the distinction between formats 1 and 2 of the ACCEPT statement.

Free-field format uses the pound sign (#) to indicate the end of data. The ampersand (&), if used as the last nonblank character in a record, indicates a continuation of data from one record or line to another. An ampersand takes precedence over the pound sign.

If the ACCEPT statement is issued against a terminal (operator's console or otherwise), the pound sign is not required to terminate data. The pound sign need only be used to indicate the end of data on a terminal when the last nonblank character of data to be read is an ampersand. Otherwise, simply pressing the RETURN key on the terminal indicates the end of data.

If you want to enter a pound sign as part of your data, you must use two consecutive pound signs, in which case, your program takes a single pound sign as a data character. Thus, for example, if you enter the characters, ABC##&, a single pound sign is treated as part of the data, and the ampersand is assumed to indicate a continuation of the data to the next line.

In free-field format, alphanumeric data is left justified (or right justified if JUSTIFIED [RIGHT] is specified in the PICTURE clause for the receiving data item), with blank fill for any unused character positions. Numeric data is aligned on the decimal point, with zero fill for unused character positions.

If the identifier named in the ACCEPT statement names a numeric or numeric-edited data item, the input must be a numeric value, with an optional leading separate sign. Any necessary conversion takes place automatically as in elementary moves (see the "MOVE Statement").

In any case other than numeric or numeric-edited data, input is assumed to be alphanumeric. No conversion takes place, but justification and space filling is performed as described above.

If you use the FREE phrase, you can also specify the ON INPUT ERROR and `NOT ON INPUT ERROR` phrases. These phrases allow you to handle the following three input error conditions:

- An illegal digit or illegal sign in a numeric item, or too many digits. The input data will not fit without left or right truncation.
- A physical I-O error or an end-of-file error.
- An input string that is too long for the receiving field.

When such an error condition occurs and the ON INPUT ERROR phrase is specified, control is passed to the imperative statement of that phrase. If none of these conditions occurs, the ON INPUT ERROR phrase is ignored and control is transferred to the end of the ACCEPT statement, or to the imperative statement specified in the NOT ON INPUT ERROR if it is used.

Note The maximum input record length for the ACCEPT statement with the FREE phrase is 256 characters. Use of the ampersand (&) continuation character, as the last nonblank character in the data record line input, allows the record length to be continued to the defined length of the identifier, which is then only limited by the available user stack space to contain the identifier.

Example

```
DATA DIVISION.
01 IN-DATA             PICTURE X(19) VALUE SPACES.
     ⋮
PROCEDURE DIVISION.
     ⋮
   ACCEPT IN-DATA FREE;
        ON INPUT ERROR DISPLAY "DATA TOO LONG".
   DISPLAY "'", IN-DATA, "'".
     ⋮
```

The following is user input to the above program:

```
DOUBLE&
TROUBLE &
BUBBLE GUM
```

The result of the above user input would be:

```
DATA TOO LONG
'DOUBLETROUBLE BUBBL'
```

And if the following were user input to the above program:

```
ADD & GET ## OF SUM#
```

The result would be:

```
'ADD & GET # OF SUM␣'
```

In the first response above, the message `DATA TOO LONG` was returned because the user response exceeded 19 characters. Note that the data stored did not include the five characters, `E GUM`. If the ON INPUT ERROR had not been specified, there would have been no indication that the data had been truncated. In the second example, the user response was 18 characters so the compiler adds a trailing character blank.

ACCEPT Statement Without the FREE Phrase

If a format 1 ACCEPT statement is used without the FREE phrase and the receiving data item requires fewer characters than the hardware imposed maximum, when the input data is transferred and it is the same length as the receiving data item, no problems arise.

If a hardware device is not capable of transferring data of the same size as the receiving data item, two cases must be considered.

First, if the size of the receiving data item exceeds the size of the transmitted data, the transmitted data is stored in the leftmost characters of the receiving data item. Additional data is then requested. The next group of data elements transmitted (if any) is aligned to the right of the rightmost character already occupying positions in the receiving data item. This process continues until either the receiving data item is full or the RETURN key is depressed (in session mode).

Refer to "MPE XL System Dependencies" in Appendix H for more information on the ACCEPT statement and the receiving data item.

Note An ACCEPT operation prematurely terminated by a :EOD or :EOJ (in job mode) causes a read error condition and abort of the program.

You can use the linefeed key to continue the transmission of characters from your screen after you have reached the right margin. This allows you to enter up to 256 characters per line before you press the RETURN key. In most cases, this avoids the necessity of sending only part of the characters required to fill the receiving data item at a given time.

In the second case, if the size of the transferred data exceeds the size of the receiving data item, or of the portion of the receiving data item not yet occupied, only the leftmost characters of the transferred data are stored in the area available in the receiving data item. The remaining characters are ignored.

Programming Considerations

The ACCEPT statement does not signal that it is waiting for a response. Therefore, a DISPLAY statement should usually precede an ACCEPT statement. This DISPLAY statement serves the dual purposes of warning you that a response is required to continue the program, and to indicate what the expected response might be.

The maximum number of characters that can be read by an ACCEPT statement is 256; however, certain hardware constraints apply to the ACCEPT statement. For example, when the SYSIN device is a card reader, the maximum number of characters that can be transferred is 80. The maximum number of characters that can be transferred from a terminal depends on the width of the device's carriage; you must terminate responses from these devices.

The ACCEPT statement issues multiple requests for data until sufficient data is read. If the identifier specifies 60 characters and the SYSIN device is a card reader, the last 20 characters on the card(s) are ignored.

When numeric data is to be input through the ACCEPT statement, you must resolve the problems of decimal point alignment and negative input values as well as leading and trailing zero fill. The following conventions should be observed:

- Identifier must be defined as X-type data or as a group item.
- The number of characters input should always be equal to the length defined for the identifier.
- If the period character is entered as a decimal point along with the significant data, the program must strip out the period before the numeric data can be used in arithmetic operations. This technique simplifies the task of data entry (and is, therefore, less error prone) at the cost of programming overhead.

Examples

The following coding is a typical example of an ACCEPT statement. Notice the use of the DISPLAY statement before the ACCEPT statement.

```
DISPLAY "IS THIS END-OF-MONTH? REPLY YES OR NO".
ACCEPT E-O-M-FLAG.
```

The following example presents one technique for removing a period entered as a part of a numeric field that must be used for subsequent arithmetic operations. The following data description coding appears in the WORKING-STORAGE SECTION:

```
01     INPUT-AMOUNT.
       02 FIELD-1          PIC XX.
       02 FILLER           PIC X.
       02 FIELD-2          PIC XX.
01     HOLD-AMOUNT.
       02 FIELD-A          PIC 99.
       02 FIELD-B          PIC 99.
01     CALC-AMT REDEFINES HOLD-AMOUNT.
       02 CALC-AMOUNT      PIC 99V99.
```

The following coding appears in the PROCEDURE DIVISION:

```
GET-AMOUNT.
    DISPLAY "ENTER AMOUNT. FORMAT EQUALS 99.99.".
    DISPLAY "SUPPLY LEADING ZERO IF REQUIRED.".
    ACCEPT INPUT-AMOUNT.
ERROR-CHECK.
    MOVE FIELD-1 TO FIELD-A.
    MOVE FIELD-2 TO FIELD-B.
    IF CALC-AMOUNT IS NOT NUMERIC GO TO BAD-AMOUNT.
    ADD CALC-AMOUNT TO FACTOR-X.
        .
        .
        .
BAD AMOUNT.
    DISPLAY "AMOUNT ENTERED INCORRECTLY. TRY AGAIN.".
    GO TO GET-AMOUNT.
```

The following two examples present possible techniques for handling the input of negative values.

Example 1: Minus sign conversion method

This example illustrates a method whereby the person entering the data precedes the quantity with a minus sign, and the program checks for the character and converts the value to an internal negative value.

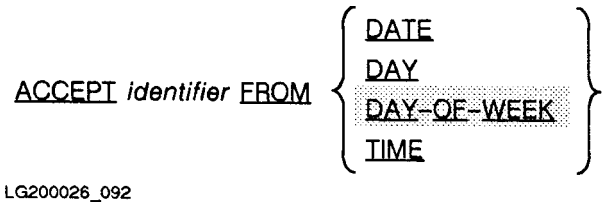
```
ID DIVISION.
PROGRAM-ID. JUNK-1.

ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  IN-DATA.
    02  SIGN-BYTE   PIC X.
    02  DATA-BYTES  PIC 9(6).
01  DATA-VAL        PIC S9(6).

PROCEDURE DIVISION.
01-ENTER-DATA.
    DISPLAY "ENTER SIGN FOLLOWED BY SIX-DIGIT NUMBER".
    ACCEPT IN-DATA.

02-VALIDATE DATA.
    IF DATA-BYTES NOT NUMERIC THEN
        DISPLAY "ILLEGAL DIGITS IN INPUT--PLEASE RE-ENTER"
        GO TO 01-ENTER-DATA
    ELSE
        MOVE DATA-BYTES TO DATA-VAL
        IF SIGN-BYTE = "-" THEN
            COMPUTE DATA-VAL = - DATA-BYTES
        ELSE IF SIGN-BYTE NOT EQUAL TO "+" THEN
            DISPLAY "ILLEGAL SIGN IN INPUT--PLEASE RE-ENTER"
            GO TO 01-ENTER-DATA.

03-DISPLAY-RESULTS.
    DISPLAY "DATA-VAL = ", DATA-VAL.
    STOP RUN.
```

Example 2: Use of SIGN IS LEADING SEPARATE phrase

This example illustrates a method whereby the person entering the data precedes the quantity with a plus or minus sign.

```
01 IN-DATA-2       PIC S9(6)  SIGN IS LEADING SEPARATE.
   ACCEPT IN-DATA-2.
   MOVE IN-DATA-2 TO DATA-VAL.
```

ACCEPT Statement - Format 3



Format 3 is used to transmit the date, day, day of the week, or time from the internal software clock of the system to the identifier named in the ACCEPT statement. The hardware clock is not used for these items.

Example

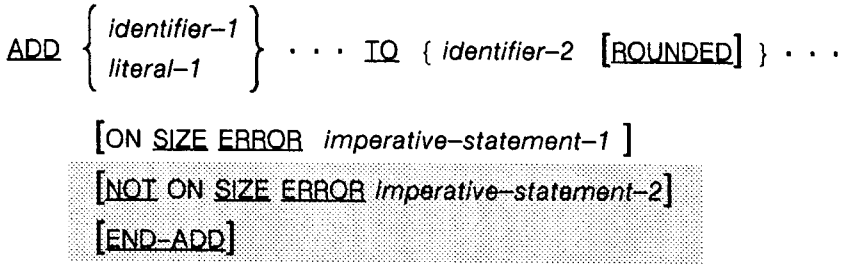
```
     ENVIRONMENT DIVISION.
     CONFIGURATION SECTION.
     SPECIAL-NAMES.
          FROM-TERMINAL IS SYSIN.
     DATA DIVISION.
     WORKING-STORAGE SECTION.
     01  NUMBER-IN            PIC 999V99.
     01  DATE-IN.
         02 YR                PIC X(2).
         02 MO                PIC X(2).
         02 DY                PIC X(2).
     01  DATE-OUT.
         02 MONTH-OUT         PIC X(2).
         02 FILLER            PIC X      VALUE '/'.
         02 DAY-OUT           PIC X(2).
         02 FILLER            PIC X      VALUE '/'.
         02 YEAR-OUT          PIC X(2).
            :
   PROCEDURE DIVISION.
            :
         ACCEPT DATE-IN FROM DATE.
         MOVE DY TO DAY-OUT.
         MOVE MO TO MONTH-OUT.
         MOVE YR TO YEAR-OUT.
         WRITE DATE-OUT AFTER ADVANCING 1 LINES.
            :
         ACCEPT NUMBER-IN.
         IF NUMBER-IN IS LESS THAN 125.50 THEN PERFORM BILL-LOW.
```

ADD Statement

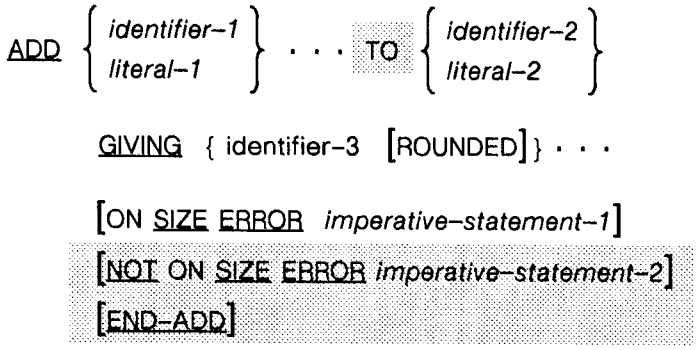
The ADD statement computes the sum of two or more operands and stores the result.

Syntax

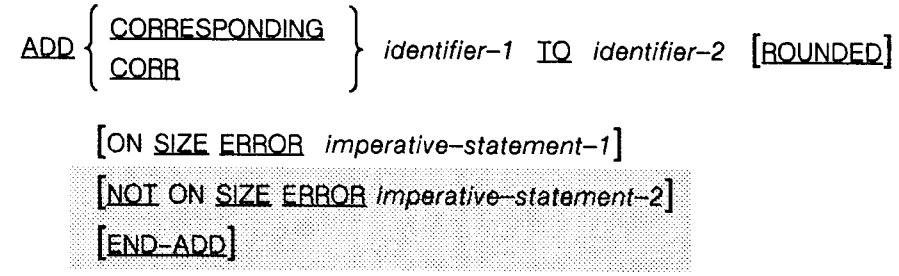
Format 1



Format 2



Format 3



LG200026_093a

Parameters

In format 1 and 2, *identifier-1*, *identifier-2*, and so forth must refer to elementary numeric items, except that in format 2, each identifier following the word GIVING may also be an edited numeric data item. Also, the word *literal* means numeric literal.

In format 3, both identifiers must refer to group items.

Description

When format 1 is used, the values of all identifiers and literals to the left of the keyword `TO` are added together and the resulting sum is added to the current contents of *identifier-2*. The results are then stored into *identifier-2*. This process of adding the resulting sum to an identifier and then storing the results into the identifier is continued until all identifiers to the right of the TO keyword have been used.

When format 2 is used, all literals and values of identifiers to the left of the GIVING keyword are added, and the result is stored into each identifier named to the right of the GIVING keyword.

When format 3 is used, data items in *identifier-1* are added to corresponding data items in *identifier-2*. The results are stored in corresponding data items of *identifier-2*. Thus, format 3 is equivalent to using format 1 for each pair of corresponding data items.

See Chapter 8 for details on the ROUNDED, SIZE ERROR, and CORRESPONDING phrases.

The composite of operands must not exceed 18 digits (see "Arithmetic Expressions" in Chapter 8). In format 1, it is calculated by using all of the operands in the statement; in format 2, it is calculated using all of the operands to the left of the GIVING phrase; in format 3, the composite of operands is calculated using pairs of corresponding data items.

During execution, the compiler always ensures that enough places are carried to avoid losing any significant digits.

For an example of format 3 usage, refer to "CORRESPONDING Phrase" in Chapter 8. For an example of format 1 usage refer to "Arithmetic Expressions" in Chapter 8.

Example

Following is an example of the ADD statement using format 2.

The operands and their assumed values are:

```
01  SUM-IT          PICTURE  9(9)V999.     Assumed value is 329^182
01  SUM-AT          PICTURE 999V9.         Assumed value is 203^9
```

The receiving data items are:

```
01   TAKE-1          PICTURE 9(9)V999.
01   TAKE-2          PICTURE 9(8)V9.
```

The example ADD statement is:

```
ADD SUM-IT, SUM-AT GIVING TAKE-1
   TAKE-2 ROUNDED
   ON SIZE ERROR PERFORM
      REPORT-IT.
```

The composite of operands is: 329^182

The results of the ADD statament are:

TAKE-1 has the value 533^082

TAKE-2 has the value 533^1 because of the ROUNDED phrase.

ALTER Statement

The ALTER statement is an obsolete feature of the 1985 ANSI COBOL standard.

The ALTER statement allows you to modify a predetermined sequence of operations.

Syntax

ALTER {*procedure-name-1* TO [PROCEED TO] *procedure-name-2*} ...

Parameters

procedure-name-1 a paragraph containing a single sentence consisting of a GO TO without the DEPENDING phrase.

procedure-name-2 a paragraph or section in the PROCEDURE DIVISION.

Execution of an ALTER statement modifies the GO TO statement in the specified paragraphs so that subsequent executions of the modified GO TO statements cause transfer of control to the section or paragraph named by *procedure-name-2*.

For example, the paragraph:

```
GO-PARA.
   GO TO CHECK-SECTION.
```

is altered to be equivalent to the paragraph:

```
GO-PARA.
   GO TO FINISH-UP.
```

by the ALTER statement:

```
ALTER GO-PARA TO PROCEED TO FINISH-UP.
```

Segmentation Considerations

The ALTER statement must not refer to a GO TO statement that appears in a section whose segment number is greater than 49 unless the ALTER statement is in the same segment.

Refer to "MPE XL System Dependencies" in Appendix H for more information.

CALL Statement

In ANSI COBOL, the CALL statement can be used to transfer control from one object program to another within the same run-unit. HP COBOL II adds the ability to invoke operating system intrinsics from within a given object program. For more information on the CALL statement refer to Chapter 11, “Interprogram Communication”.

CANCEL Statement

The CANCEL statement restores a program to its initial state and closes all files currently in open mode. For more information on the CANCEL statement refer to Chapter 11, “Interprogram Communication.”

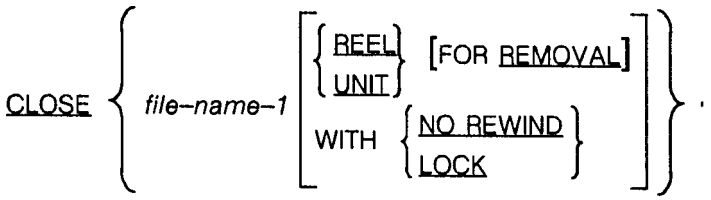
CLOSE Statement

The CLOSE statement terminates the processing of sequential, random, relative, and indexed files. It can only be executed for an open file.

Syntax

The CLOSE statement has two formats, depending upon whether you want to close a sequential file or one of the other three types of files.

Sequential Files - Format 1



LG200026_095

Description

Rules that apply to a CLOSE statement for any type of file are described below. For information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

A CLOSE statement can only be issued for a file that is open, and has not yet been closed.

If a CLOSE statement has been successfully executed for a file, no other statement can be executed that references the closed file, either explicitly or implicitly, unless an intervening OPEN statement for that file is executed. There is one exception to this rule. A sequential file that has been closed may be referred to in SORT and MERGE statements that use the USING or GIVING phrases. In this case, the file or files named in the USING and GIVING phrases must not be open.

Following the successful execution of the CLOSE statement (without the REEL or UNIT phrases in the case of sequential files), the record area associated with the name of the closed file is no longer available.

If a CLOSE statement is unsuccessful in its execution, the availability of the record area for the specified file is undefined.

If a CLOSE statement has not been issued for an open file when a STOP RUN statement (or a GOBACK statement in a main program) is executed, the file is automatically closed by the COBOL run-time system.

If a called program has been canceled by the CANCEL statement, all open files of that program will be closed.

If the file being closed is a new file or a temporary file, it is closed in the temporary file domain. If it is a permanent file, it remains in the permanent file domain when it is closed.

The FILE STATUS data item, if any, specified for the file named in the CLOSE statement is updated to indicate the success or failure of the closing operation. Refer to "FILE STATUS Clause" in Chapter 6 for valid status keys.

Using a format 1 CLOSE statement, as shown above, allows you to terminate the processing of files whose organization is sequential. It also provides you with the options of placing the serial access device at its physical beginning and of locking the file so that it cannot be opened again during the execution of the current run-unit.

REEL/UNIT and REMOVAL Phrases

The REEL/UNIT phrase and the REMOVAL phrase are treated as comments in format 1 of the CLOSE statement. Furthermore, if the REEL/UNIT phrase is specified in a format 1 CLOSE statement, the entire CLOSE statement is treated as a comment. Thus, the file specified in the CLOSE REEL/UNIT statement remains open.

Each of the remaining optional phrases are described below.

If no optional phrases are used, that is, if the format 1 CLOSE statement consists entirely of the statement,

```
CLOSE file-name-1
```

then the system's closing operations are executed, no matter what kind of operations (input, input-output, output or extend) the file was opened for. If the file resides on a magnetic tape, the reel is rewound when the file is closed.

NO REWIND Phrase

The NO REWIND phrase applies only to labeled magnetic tape files.

Used without either a REEL or UNIT phrase, the NO REWIND phrase alters the execution of the system's standard closing procedure. The tape device, instead of being rewound when the file is closed, remains in its current position.

This phrase should be used in the closing of a file only if another file residing near the end of the same tape is to be opened later in the program. Upon completion of the program, the tape is rewound by the operating system.

If the file resides on a device that allows no rewinding, such as a line printer, the NO REWIND phrase is ignored when specified for that file in a CLOSE statement; it has no effect on the file.

WITH LOCK Phrase

The WITH LOCK phrase can be used in the CLOSE statement to ensure that the file being closed cannot be opened again during the execution of the current run-unit.

This locking is accomplished by the program, following the successful closing of the file.

Random, Relative and Indexed Files - Format 2

The second format of the CLOSE statement is:

CLOSE {*file-name-1* [WITH LOCK] } ...

This form of CLOSE closes the files named by *file-name-1*, and so forth, and optionally locks the files so that they cannot be opened again during execution of the current run-unit.

The files named by *file-name-1*, and so forth need not all have the same organization or access.

When a CLOSE statement without the LOCK phrase is issued for a relative, random, or indexed file, the MPE file system closing procedures are used to close the file or files specified, no matter how the files are used (that is, input, input-output, or output).

Additionally, if the LOCK phrase is used with a relative, random access, or indexed file, the compiler ensures that the file cannot be opened again during execution of the current run-unit.

Example

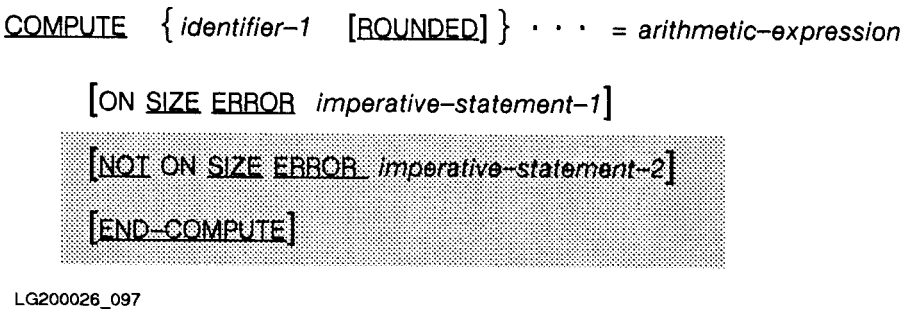
```
        ⋮
ENVIRONMENT DIVISION.
        ⋮
INPUT-OUTPUT SECTION.

   FILE-CONTROL.
   SELECT INDEXER
      ASSIGN TO "FILE-INDX, DA, A, DISC"
         ORGANIZATION IS INDEXED
         ACCESS MODE IS DYNAMIC
         RECORD KEY IS INDX-FOR-FL.

   SELECT RNDM-FL
      ASSIGN TO "RANDOM"
         ACCESS MODE IS RANDOM
         PROCESSING MODE IS SEQUENTIAL
         ACTUAL KEY IS DATA-5.
          ⋮
PROCEDURE DIVISION.
        ⋮
CLOSE INDEXER WITH LOCK, RNDM-FL WITH LOCK.
        ⋮
```

In the above CLOSE statement, the files named `INDEXER` and `RNDM-FL` are closed and locked so that they may not be opened again during execution of the run-unit.

COMPUTE Statement

The COMPUTE statement evaluates an arithmetic expression (Refer to Chapter 8, under "Arithmetic Expressions"), and assigns the result to one or more data items.

Syntax



LG200026_097

Parameters

identifier-1 refers to either an elementary numeric, or an elementary numeric-edited data item.

arithmetic-expression any valid COBOL arithmetic expression.

The ROUNDED and SIZE ERROR and `NOT ON SIZE ERROR` phrases are described in Chapter 8, as are multiple results and other information pertaining to arithmetic statements.

The COMPUTE statement allows you to combine arithmetic operations without the restrictions on composites of operands or receiving data items imposed by the arithmetic statements ADD, SUBTRACT, MULTIPLY, and DIVIDE.

When the COMPUTE statement executes, the arithmetic expression is evaluated, and all of the identifiers to the left of the equal sign are assigned the value of the result. Rounding is done where specified and necessary. For example,

```
COMPUTE DAILY-RTE-1, DAILY-RTE-2 = (INT - RTE / 360) * DAYS
    ON SIZE ERROR PERFORM RATE-ERROR-RTNE.
```

In the above statement, a daily interest rate is calculated, and the results are stored in the two data items, `DAILY-RTE-1` and `DAILY-RTE-2`. If a size error occurs, no data is stored in the two receiving data items and the error handler, `RATE-ERROR-RTNE`, is performed.

Calculation of Intermediate Results

The following description presents the conceptual compiler algorithms for determining the size and number of decimal places reserved for intermediate results. This information is provided since the manipulations performed on the intermediate results are not always obvious. These algorithms apply to all arithmetic and compute statements.

The following abbreviations are used:

d	number of decimal places carried for an intermediate result.
dmax	maximum number of decimal places defined for any operand in a particular statement including the result.
op1	first operand in a generated arithmetic statement.
op2	second operand in a generated arithmetic statement.
d1,d2	number of decimal places defined for *op1* or *op2*, respectively.
ir	intermediate result field obtained from the execution of a generated arithmetic statement or operation. *Ir1, ir2*, etc., represent successive intermediate results. Successive intermediate results may have the same location.

Most arithmetic statements generate intermediate results except for simple cases, (for example, single pair of operands) where the result can be stored without decimal point alignment or conversion.

The compiler treats the statement as a succession of operations. For example, consider the following statement:

```
COMPUTE Y = A + B * C - D / E + F ** G
```

The above COMPUTE statement is replaced by the following:

```
MULTIPLY C      BY B            Yielding ir1
ADD ir1         TO A            Yielding ir2
DIVIDE D        BY E            Yielding ir3
SUBTRACT ir3    FROM ir2        Yielding ir4
RAISE F         TO THE POWER G  Yielding ir5
ADD ir4         TO ir5          Yielding ir6
STORE ir6       TO Y
```

The compiler determines the maximum value that the *ir* can contain by performing the statement in which the *ir* occurs.

- If an operand in this statement is a data name, the value for the data name is equal to the numerical value of the PICTURE for the data name (for example, PICTURE 9V99 has the value 9.99).
- If an operand is a literal, the literal's actual value is used, except in case of DIVIDE.
- If an operand is an intermediate result, the value determined for the intermediate result in a previous arithmetic operation is used.
- If the operation is division:
  - If *op2* is a data name, the value used for *op2* is the minimum nonzero value of the digit in the PICTURE for the data name (for example, PICTURE 9V99 has the value 0.01).
  - If *op2* is an intermediate result, the intermediate result is treated as though it had a PICTURE, and the minimum nonzero value of the digits in this PICTURE is used.

When the maximum value exceeds the machine specific limit, a warning (#050) is generated and the maximum size is set at that limit. For limitations on arithmetic expressions refer to "MPE XL System Dependencies" in Appendix H for more information.

The number of decimal places contained in an *ir* is calculated as:

Operation	Decimal Places
+ or -	*d1* or *d2*, whichever is greater
*	*d1* + *d2*
/	*d1-d2* or *dmax*, whichever is greater
**	*dmax*

Note When any operand is an IEEE floating point (from the result of a COBOL function), the resulting intermediate data item is also IEEE floating point. The intermediate floating point data items always have 15 digits of precision.

CONTINUE Statement

The CONTINUE statement indicates that no executable statement is present. It is a no-operation statement and has no effect on the execution of the program.

Syntax

CONTINUE

Description

CONTINUE can be used anywhere a conditional statement or an imperative statement is used.

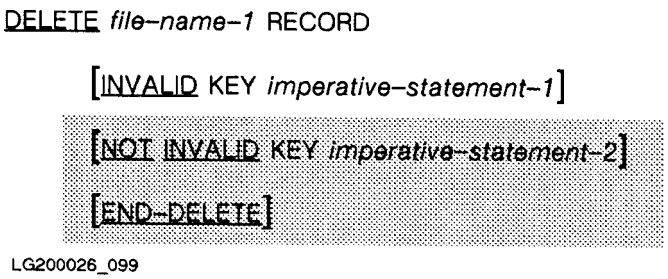
Example

```
IF A < B THEN
      IF A < C THEN
         CONTINUE
      ELSE
         MOVE ZERO TO A
      END-IF
      ADD B TO C.
SUBTRACT C FROM D.
```

DELETE Statement

The DELETE statement logically removes a record from a relative or indexed file.

Syntax



Parameters

file-name-1	the name of the file from which the record is to be deleted. The file must be an indexed or relative file opened in input-output mode.
imperative-statement-1 and *imperative-statement-2*	one or more imperative statements.

Description

If the specified file is being used in sequential access mode, the INVALID KEY or `NOT INVALID KEY` phrase must not be specified. However, if the file is being used in any other access mode and there is no applicable USE procedure, these clauses must be specified. For information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

After a successful execution of a DELETE statement, the selected record has been logically removed from the file, and can no longer be accessed.

Note "Logically removed" means that the record has been marked for deletion, and not physically removed. The contents of the record area associated with the specified file are unaffected. The file position indicator is also unaffected.

The execution of the DELETE statement causes the value of the FILE STATUS data item, if specified for the file, to be updated. Refer to "FILE STATUS Clause" in Chapter 6 for valid status keys.

Selection of the record to be deleted is accomplished in one of two ways, depending upon what access mode is specified for the file.

For files being used in sequential access mode, the record to be deleted is the last record read by a successfully executed READ statement. The READ statement must be the last input-output operation performed on the file prior to execution of the DELETE statement.

For files in dynamic or random access mode, the record removed is that record identified by the contents of the RELATIVE KEY or RECORD KEY data item associated with the specified file. If the file does not contain the record specified by the key, an INVALID KEY condition exists.

If the INVALID KEY phrase has been specified, and the execution of a DELETE statement causes an INVALID KEY condition, the imperative statements specified in the INVALID KEY phrase are executed. Any USE procedure specified for the file is ignored.

When no INVALID KEY condition exists, control is transferred to the end of the DELETE statement `or to the imperative statement specified in the NOT INVALID KEY phrase, if specified.` For more information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

For indexed files, if the primary record key has duplicates specified for it, you should use the DELETE statement only when the file is open in sequential access mode. This is because a DELETE statement for such files open in dynamic or random access mode deletes the first record written to the file that has the same primary key value as the value placed in the RECORD KEY data item. This first occurrence of the duplicate value may not be the record you want to delete.

Example

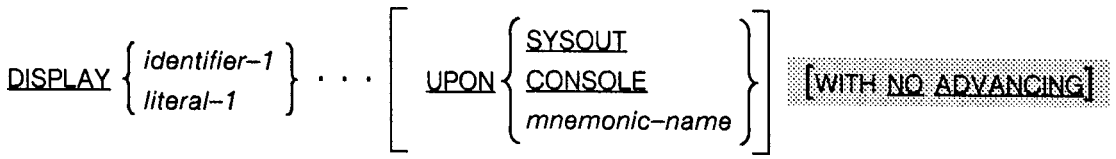
The following example shows the DELETE statement:

```
         ⋮
  ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT REL-FILE
      ASSIGN TO "DATAFL1"
        ORGANIZATION IS RELATIVE
        ACCESS MODE IS SEQUENTIAL
        FILE STATUS IS CHK-REL-FILE.
  SELECT INDX-FILE
      ASSIGN TO "DATAFL2"
        ORGANIZATION IS INDEXED
        ACCESS MODE IS RANDOM
        RECORD KEY IS KEY-DATA
        FILE STATUS IS CHK-INDX-FILE.
  DATA DIVISION.
  FILE SECTION.
  FD REL-FILE.
  01 REL-DATA
     02  DATA-1                    PIC 99.
  FD INDX-FILE.
  01 INDX-DATA.
     02 KEY-DATA                   PIC X(5).


  PROCEDURE DIVISION.
         ⋮
      OPEN I-O REL-FILE  INDX-FILE.
      READ REL-FILE RECORD  AT END STOP RUN.
      IF DATA-1 IS EQUAL TO 0 THEN DELETE REL-FILE RECORD.
         ⋮
      MOVE "MEYER" TO KEY-DATA.
      DELETE INDX-FILE RECORD
          INVALID KEY PERFORM CHECK-OUT.
         ⋮
  CHECK-OUT.
      DISPLAY "VALUE OF CHK-INDX-FILE IS"  CHK-INDX-FILE.
      DISPLAY "WHAT ACTION TO BE TAKEN?".
      ACCEPT ACTION-ITEM.
```

DISPLAY Statement

The DISPLAY statement can be used to transfer low volume data to the operator's console, a terminal, or the line printer. If more than one name is specified, each data item is listed in the order specified in the DISPLAY statement.

Syntax



LG200026_100

Parameters

identifier-1 and *literal-1*	identifiers of data items, unsigned numeric integer literals, the special registers (TALLY, TIME-OF-DAY, CURRENT-DATE, WHEN-COMPILED, LINAGE-COUNTER, and DEBUG-ITEM) and any figurative constant except ALL.
`SYSOUT`	in batch mode, the line printer. In session mode, it is the terminal from which the COBOL program was initiated. This is the default if the UPON phrase is not used.
`CONSOLE`	the operator's console.
mnemonic-name	the name specified by you, and defined under the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION as either SYSOUT or CONSOLE.

Description

Note I-O errors that occur during execution of the DISPLAY statement do not produce a "tombstone" since the I-O is not done by the file system.

If an item is described as USAGE COMPUTATIONAL, `BINARY`, COMPUTATIONAL-3, or `PACKED-DECIMAL`, the compiler translates it into a USAGE DISPLAY item for purposes of displaying it.

If TIME-OF-DAY is used as an identifier, the time is displayed in edited form. That is, in the form, HH:MM:SS where HH is the hour taken from a 24 hour clock, MM is the number of minutes after the hour, and SS is the number of seconds after the minute.

If a figurative constant is specified as an operand, only one occurrence of the constant is displayed. This is true even when the figurative constant ALL is specified.

When a DISPLAY statement contains more than one operand, the size of the data to be transmitted is the sum of the sizes of all the operands. The values of the operands are transferred in the sequence in which the operands are listed.

Length of Data Being Displayed

As with the ACCEPT statement, hardware record sizes determine the display of the data specified in the DISPLAY statement. The following methods are used, depending upon whether the size of the sending item is equal to, shorter than, or longer than the hardware device designated to receive the data:

- If the sending item is the same length, no problem arises and the data is transmitted.
- If the sending item is shorter than the device, the transferred data is displayed beginning with the leftmost position of the device, continuing to the right until all data characters have been displayed.
- If the sending item is longer than can be displayed on one line of the device, the first line of the device is filled with as many characters as possible, then the next line is filled, and so forth until the entire sending item has been displayed. The order in which the sending data is displayed is the same as the order in which it is transmitted.

The WITH NO ADVANCING Phrase

The WITH NO ADVANCING phrase of the DISPLAY statement provides interaction with a hardware device having vertical positioning. If the WITH NO ADVANCING phrase is specified, the positioning of the hardware device is not reset to the next line or changed in any other way following the display of the last operand. If the hardware device can be set to a specific character position, it remains set at the character position immediately following the last character of the last operand displayed. If the hardware device cannot be set to a specific character position, only the vertical position, if applicable, is affected. This may cause overprinting if the hardware supports overprinting. If you redirect STDLIST to a file, the file must use carriage control (CCTL) or WITH NO ADVANCING has no effect.

If the WITH NO ADVANCING phrase is not specified, the positioning of the hardware device is reset to the leftmost position of the next line of the device after the last operand has been transferred to the hardware device.

If vertical positioning is not applicable on the hardware device, the operating system ignores the vertical positioning that is specified or implied.

Example 1

Following is an example DISPLAY statement:

```
WORKING-STORAGE SECTION.
01  BEGIN-MSG      PIC X(21) VALUE "␣PROGRAM BEGINNING␣␣".
        ⋮
PROCEDURE DIVISION.
        ⋮
    DISPLAY CURRENT-DATE, BEGIN-MSG, TIME-OF-DAY UPON SYSOUT.
```

If the date is Tuesday, July 30, 1991 at exactly 10:45 a.m. and the above DISPLAY statement is executed, the following message is displayed on the terminal where the program was run:

```
07/30/91␣PROGRAM BEGINNING␣␣10:45:00
```

Example 2

The following DISPLAY statement illustrates the WITH NO ADVANCING phrase:

```
DISPLAY "ENTER CLASS CODE" WITH NO ADVANCING.
ACCEPT CLASS-CODE.
```

When the above statement is executed, the cursor is left on the same line as `ENTER CLASS CODE` on the screen:

```
ENTER CLASS CODE
               ↑
               Position of cursor following the display.
```

DIVIDE Statement

The DIVIDE statement divides one numeric data item into one or more others, assigns the result to a data item, and optionally assigns a remainder to another data item.

Syntax

There are five formats of the DIVIDE statement:

Format 1

```
DIVIDE { identifier-1 } INTO { identifier-2 [ROUNDED] } . . .
       { literal-1    }
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-DIVIDE]
```

Format 2

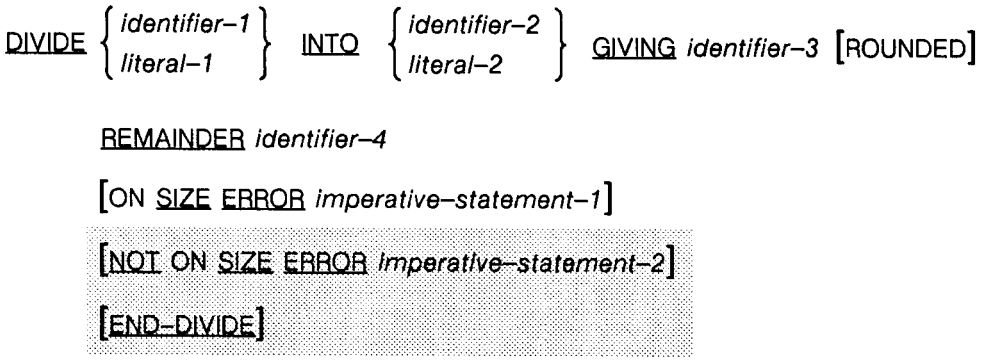
```
DIVIDE { identifier-1 } INTO { identifier-2 }
       { literal-1    }      { literal-2    }
    GIVING { identifier-3 [ROUNDED] } . . .
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-DIVIDE]
```

Format 3

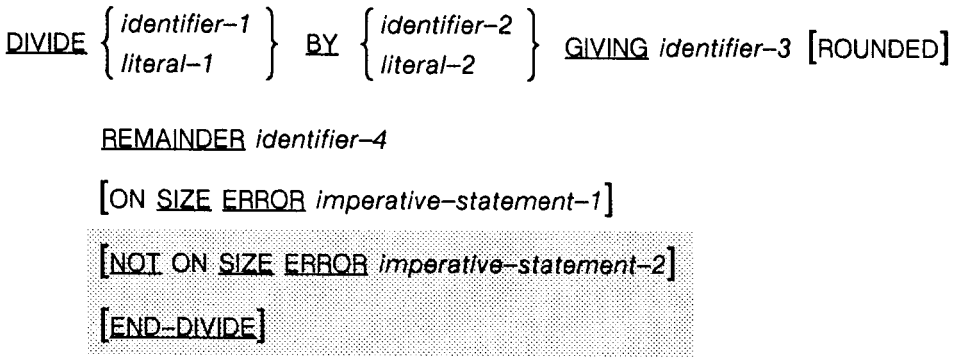
```
DIVIDE { identifier-1 } BY { identifier-2 }
       { literal-1    }    { literal-2    }
    GIVING { identifier-3 [ROUNDED] } . . .
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-DIVIDE]
```

LG200026_101

Format 4



Format 5



LG200026_102

Parameters

identifier-1, *identifier-2*, and so forth	names of elementary numeric items, except that those associated with a GIVING or REMAINDER phrase may be elementary numeric-edited items.
literal-1, *literal-2*, and so forth	numeric literals.

Description

The ROUNDED and SIZE ERROR phrases are described under the heading, "Common Phrases", in Chapter 8.

The composite of operands for the DIVIDE statement is determined using all of the receiving data items of a particular statement except the data item associated with the REMAINDER phrase. This composite must not exceed 18 digits. Refer to Chapter 8, under "Arithmetic Expressions" for details on how to determine the composite of operands.

When format 1 of the DIVIDE statement is used, each identifier following the INTO keyword is divided, in turn, by the identifier or literal to the left of the INTO keyword. Each result is rounded if specified and necessary, and is then stored in the data item referenced by the identifier that acted as the dividend in that particular division.

When format 2 is used, the literal or data item specified by the identifier between the keywords INTO and GIVING is divided by the literal or data item specified by *identifier-1*, and the result is stored in each identifier listed in the GIVING phrase.

When the third format of the DIVIDE statement is used, the data item specified by *identifier-1* or *literal-1* is divided by *literal-2* or the contents of *identifier-2*. The result is then stored in each identifier following the GIVING phrase, with rounding being used where specified and needed.

Formats 4 and 5 can be used to obtain a remainder from a division operation. In COBOL, the remainder is defined as the difference between the product of the quotient and the divisor and the dividend.

For example, in format 4 of the DIVIDE statement:

```
DIVIDE A INTO B GIVING C REMAINDER D
       ↑      ↑        ↑
    divisor dividend  quotient
```

The remainder D has the value determined by multiplying C times A and subtracting this product from B. Thus, if A=7 and B=16, then C=2 and D=2 because 16-7*2=2.

If *identifier-3* (the quotient) is defined as numeric-edited, the quotient used to calculate the remainder is an internal, intermediate field containing the unedited quotient.

Also, if the ROUNDED phrase is specified, the quotient used to calculate the remainder is kept in an intermediate field and is truncated rather than rounded.

Appropriate decimal alignment and truncation are performed on the remainder as needed.

When the SIZE ERROR phrase is specified for a format 4 or 5 DIVIDE statement, and a size error condition occurs for the quotient, the contents of data items referenced by *identifier-3* and *identifier-4* are unchanged. However, if the size error condition occurs for the remainder and not the quotient, only the remainder is unchanged. *Identifier-3* still contains the new quotient.

Example

```
FILE SECTION.
FD PAY-FILE.
01 PAY-INFO.
   02 EMP-NAME          PIC X(30).
   02 EMP-NUM           PIC X(9).
   02 PAY               PIC 999V99.
   02 HOURS             PIC 99.
WORKING-STORAGE SECTION.
77 RATE            PIC 99         VALUE ZERO.
77 CHECK           PIC V99        VALUE ZERO.
     ⋮
PROCEDURE DIVISION.
MAIN-100.
     DIVIDE PAY BY HOURS GIVING RATE REMAINDER CHECK
        ON SIZE ERROR PERFORM SIZE-ERR.
           ⋮
SIZE-ERR.
     IF RATE = 0 THEN
        DISPLAY "SIZE ERROR IN RATE USING " PAY, HOURS
     ELSE
        DISPLAY "SIZE ERROR IN CHECK".
          ⋮
```

The DIVIDE statement above uses format 5. If a size error occurs, the SIZE-ERR routine is performed, and a check is made to determine whether the size error occurred because of RATE or CHECK.

ENTER Statement

The ENTER statement is an obsolete feature of the 1985 ANSI COBOL standard.

In ANSI COBOL'74, this statement provides a means of allowing the use of more than one language in the same program. It is, however, not allowed in HP COBOL II. Thus, if specified in your program, it is treated as a comment. The format is listed below for information only.

The format of this statement is shown below:

ENTER *language-name* [*routine-name*].

ENTRY Statement

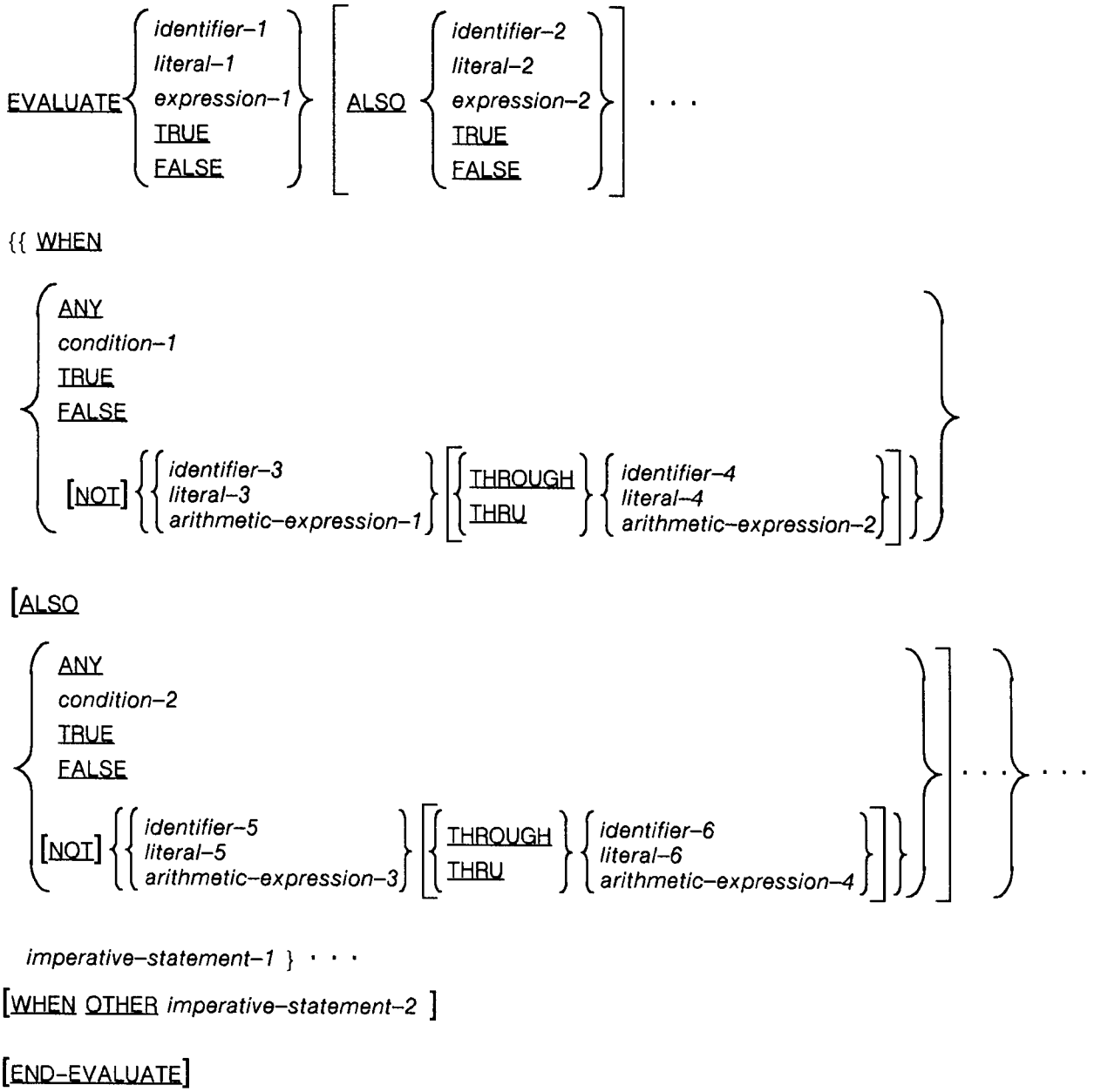
The ENTRY statement is an HP extension to the ANSI COBOL standard.

The ENTRY statement establishes a secondary entry point in an HP COBOL II subprogram. For more information on the ENTRY statement refer to Chapter 11, "Interprogram Communication".

EVALUATE Statement

The EVALUATE statement adds a multi-condition case construct to COBOL. This statement causes a set of subjects to be evaluated and compared with a set of objects. The results of these evaluations determine the subsequent sequence of code execution.

Syntax



LG200026_104

Subjects and Objects

The operands or the words TRUE and FALSE that appear before the first WHEN phrase of the EVALUATE statement are referred to individually as *subjects*. Collectively, they are referred to as *the set of subjects*. The operands or the words TRUE, FALSE, and ANY that appear in a WHEN phrase of an EVALUATE statement are individually called *objects*. Collectively, they are called *the set of objects*.

The words THROUGH and THRU are equivalent. Two operands connected by a THROUGH phrase must be of the same class. The two connected operands constitute a single object.

The number of objects within each set of objects must be equal to the number of subjects.

Correspondence Between Subjects and Objects

A subject-object pair consists of a subject and object having the same ordinal position within each set. Each pair must conform to the following rules:

- Identifiers, literals, and arithmetic expressions must be valid operands for a comparison between the subject and object.
- Conditions or the words TRUE or FALSE appearing as an object must correspond to a conditional expression or the words TRUE or FALSE.
- The word ANY may correspond to a selection subject of any type.

Evaluation of Subjects and Objects

Execution of the EVALUATE statement operates as if each subject and object were evaluated and assigned a numeric or nonnumeric value, a range of numeric or nonnumeric values, or a truth value (TRUE or FALSE). These values are determined as follows:

- Any subject or object specified by an identifier, without either the NOT or the THROUGH phrases, is assigned the value and class of the data item referenced by the identifier.
- Any subject or object specified by a literal, without either the NOT or the THROUGH phrases, is assigned the value and class of the specified literal. When an object is assigned the figurative constant ZERO, it is assigned the class of the corresponding subject.
- Any subject or object in which an expression is specified as an arithmetic expression without either the NOT or the THROUGH phrases, is assigned a numeric value according to the rules for evaluating an arithmetic expression. (Refer to Chapter 8, under "Arithmetic Expressions".)
- A subject or object specified by a conditional expression is assigned a truth value (TRUE or FALSE) according to the rules for evaluating conditional expressions. (Refer to Chapter 8, "Conditional Expressions.")
- A subject or object specified by the words TRUE or FALSE is assigned the appropriate truth value.
- No further evaluation is done for an object specified by the word ANY.
- If the THROUGH phrase is specified for an object, without the NOT phrase, the range of values includes all values of the subject that are greater than or equal to the first operand and less than or equal to the second operand.

- If the NOT phrase is specified for an object, the values assigned to that item are all values that are not equal to the value, or included in the range of values, that would have been assigned to the item without the NOT phrase.

Refer to Chapter 8, "Relation Conditions," for more information on NOT phrases.

Comparison Operation of EVALUATE

The execution of the EVALUATE statement proceeds as if the values assigned to the subjects and objects were compared, to determine if any WHEN phrase satisfies the set of subjects. This comparison proceeds as follows:

1. A subject-object pair comparison is satisfied if the following conditions are true:
   a. If the items being compared are assigned numeric, nonnumeric, or a range of numeric or nonnumeric values, the comparison is satisfied if the value, or one of the range of values, assigned to the object is equal to the value assigned to the subject.
   b. If the items being compared are assigned truth values, the comparison is satisfied if the items are assigned the identical truth values.
   c. If the object being compared is specified by the word ANY, the comparison is always satisfied, regardless of the value of the subject.
2. If the above comparison is satisfied for every object within the set of objects being compared, the first WHEN phrase for which each subject-object pair comparison is satisfied is selected as the one that satisfies the set of subjects.
3. If the above comparison is not satisfied for one or more objects within the set of objects being compared, that set of objects does not satisfy the set of subjects.
4. This procedure is repeated for subsequent sets of objects in the order of their appearance in the source program. The comparison operation continues until either a WHEN phrase that satisfies the set of subjects is selected or until all sets of objects are exhausted.

Execution of EVALUATE

After the comparison operation is completed, execution of the EVALUATE statement proceeds as follows:

1. If a WHEN phrase is selected, execution continues with the first imperative statement following the selected WHEN phrase.

   If a WHEN phrase is followed by other WHEN phrases with no intervening imperative statement, the WHEN conditions are ORed together. In other words, if any of the WHEN phrases is selected, the *first* imperative statement that follows is executed, even if that imperative statement is part of a *following* WHEN phrase. See the following section for an example.

   Use the CONTINUE statement to indicate no operation on a WHEN clause. See the following section for examples.
2. If no WHEN phrase is selected and a WHEN OTHER phrase is specified, execution continues with the imperative statement following the WHEN OTHER phrase.
3. The execution of the EVALUATE statement is terminated when execution reaches the end of the imperative statement of the selected WHEN phrase, or when no WHEN phrase is selected and no WHEN OTHER phrase is specified.

Examples

The following example shows an EVALUATE statement with two data items (HOURS-WORKED and EXEMPT) as subjects:

```
EVALUATE HOURS-WORKED ALSO EXEMPT
   WHEN             0 ALSO ANY PERFORM NO-PAY
   WHEN         NOT 0 ALSO "Y" PERFORM SALARIED
   WHEN     1 THRU 40 ALSO "N" PERFORM HOURLY-PAY
   WHEN NOT 1 THRU 40 ALSO "N" PERFORM OVERTIME-PAY
   WHEN OTHER                  DISPLAY HOURS-WORKED
                               DISPLAY EXEMPT
                               MOVE 0 TO HOURS-WORKED
END-EVALUATE.
```

The following shows a relation condition (GRADE > 3.0) and a data item (COLLEGE-CODE) as the subjects of an EVALUATE:

```
EVALUATE GRADE > 3.0 ALSO COLLEGE-CODE
   WHEN TRUE ALSO "01"   PERFORM DEANS-LIST-AGGIES
   WHEN TRUE ALSO "02"   PERFORM DEANS-LIST-S-AND-H
   WHEN TRUE ALSO "03"   PERFORM DEANS-LIST-ENG
   WHEN TRUE ALSO ANY    PERFORM MISC-LIST
END-EVALUATE.
```

The following shows two equivalent EVALUATE statements that illustrate that subjects and objects must be of the same type. The first EVALUATE statement shows the truth value, TRUE, as the subject and several condition name conditions as objects. The second shows the data item INPUT-FLAG as the subject and nonnumeric literals as objects. Notice also that if INPUT-FLAG is "C", the EVALUATE statement executes the CONTINUE statement, which simply continues execution at the statement following the EVALUATE statement:

```
WORKING-STORAGE SECTION.
01  INPUT-FLAG   PIC X VALUE SPACE.
    88  INPUT-YES      VALUE "Y".
    88  INPUT-NO       VALUE "N".
    88  INPUT-QUIT     VALUE "Q".
    88  INPUT-CONTINUE VALUE "C".
          ⋮
EVALUATE TRUE
   WHEN INPUT-CONTINUE CONTINUE
   WHEN INPUT-YES      MOVE PROD-NO TO OUTPUT-REC
   WHEN INPUT-NO       MOVE SPACES  TO OUTPUT-REC
   WHEN INPUT-QUIT     PERFORM TERMINATION-ROUTINE
   WHEN OTHER          PERFORM GET-INPUT
END-EVALUATE.
```

```
EVALUATE INPUT-FLAG
   WHEN "Y"   MOVE PROD-NO TO OUTPUT-REC
   WHEN "N"   MOVE SPACES  TO OUTPUT-REC
   WHEN "Q"   PERFORM TERMINATION-ROUTINE
   WHEN "C"   CONTINUE
   WHEN OTHER PERFORM GET-INPUT
END-EVALUATE.
```

The following example shows two WHEN phrases without an intervening imperative statement. If either the first or the second WHEN phrase is selected, that is, if NUMBER-OF-THINGS is either 1 or 2, the DISPLAY statement after WHEN 2 is executed:

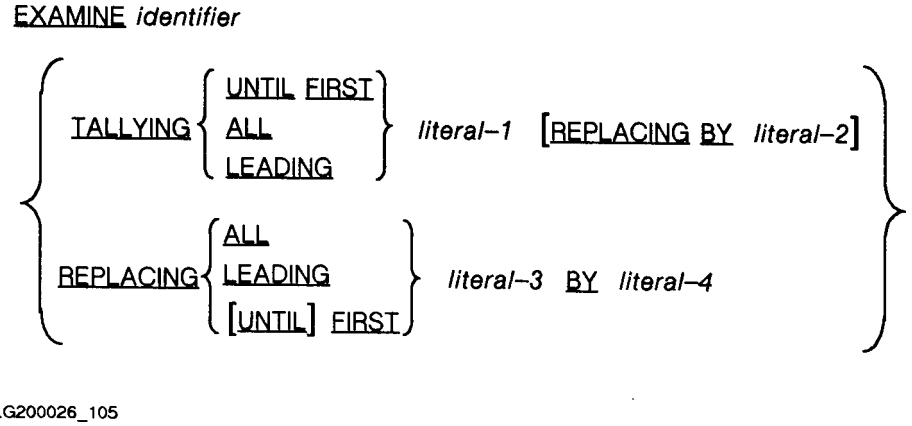
```
EVALUATE NUMBER-OF-THINGS
   WHEN 1
   WHEN 2  DISPLAY "The value is 1 or 2"
   WHEN 3  STOP RUN
   WHEN OTHER DISPLAY "Input again."
END-EVALUATE.
```

EXAMINE Statement

The EXAMINE statement is an HP extension to the ANSI COBOL standard. It has been replaced by the INSPECT statement, covered later in this chapter. Although HP COBOL II includes the EXAMINE statement for compatibility with COBOL'68, it is advisable that you use the INSPECT statement for coding new programs.

The EXAMINE statement replaces or counts the number of occurrences of a given character in a data item.

Syntax



LG200026_105

Parameters

identifier names a data item whose usage is DISPLAY. It is this data item that is examined.

literal-1, literal-2, and so forth each a single character whose data type is the same as identifier. Any or all of these literals may be any figurative constant except ALL.

Description

When the EXAMINE statement is executed, it acts differently depending upon whether *identifier* names a numeric or a nonnumeric data item.

If *identifier* is a nonnumeric data item, examination begins with the leftmost character, and proceeds to the right. Each character is examined in turn.

If *identifier* is a numeric data item, the data item may contain a sign, and examination proceeds on a digit by digit basis. This examination starts with the leftmost digit and proceeds to the right. If a sign is included in the data item being examined, it is ignored regardless of its physical location.

TALLYING Phrase

When the TALLYING phrase is used in an EXAMINE statement, a count is placed in the special HP COBOL II register named TALLY. This count is an integer and represents a value that is dependent upon the keywords following the word TALLYING.

If TALLYING UNTIL FIRST is specified, the integer in the TALLY register after execution of an EXAMINE statement is the number of occurrences of characters in *identifier* before the first occurrence of *literal-1*.

If TALLYING ALL is specified, every occurrence of *literal-1* is counted and the result of this counting is placed in the TALLY register.

If TALLYING LEADING is specified, only those occurrences of *literal-1* that precede any other characters in the data item named by *identifier* are counted. For example, if the first character of *identifier* is not *literal-1*, the EXAMINE statement ceases execution immediately.

If the REPLACING phrase is used in conjunction with the TALLYING phrase, then, depending upon which keywords are used with the TALLYING phrase, those occurrences of *literal-1* that participate in the tallying are replaced by *literal-2*. For example, if the EXAMINE statement:

```
EXAMINE ABMASK TALLYING ALL A REPLACING BY B.
```

is executed and ABMASK contains the value `ABBBBABBABAAAB` before execution, when execution of the EXAMINE statement is complete, the value in the TALLY register is 6 and `ABMASK` contains the value `BBBBBBBBBBBBB`.

REPLACING Phrase

The REPLACING phrase acts in the same manner as the REPLACING verb in the TALLYING phrase. However, since no tallying takes place, the TALLY register remains unchanged. The rules of the REPLACING phrase are stated below:

- If REPLACING ALL is specified, all occurrences of *literal-3* in *identifier* are replaced by *literal-4*.
- If REPLACING LEADING is specified, each occurrence of *literal-3* is replaced by *literal-4* until the first occurrence of a character other than *literal-3* or the rightmost character of the data item is examined.
- If REPLACING UNTIL FIRST is specified, every character of the data item represented by *identifier* is replaced by *literal-4* until *literal-3* is encountered in the data item. If *literal-3* does not appear in the data item, the entire data item is filled with *literal-4*.
- If REPLACING FIRST is specified, only the first occurrence of *literal-3* is replaced by *literal-4*. If *literal-3* does not appear in the data item represented by *identifier*, the data item is unchanged after execution of the EXAMINE statement.

EXCLUSIVE Statement

The EXCLUSIVE statement is an HP extension to the ANSI COBOL standard.

The EXCLUSIVE statement provides you with a method for locking a file that has been opened for shared access.

Note Use of EXCLUSIVE within a program causes any OPEN of the associated file to enable the dynamic locking facility.

This "locking" does not stop anyone from accessing the file. Locking and unlocking files must be done on a cooperative basis. That is, if all users who intend to access a shared file agree to attempt to lock the file before accessing its records, then no problems arise. However, since this form of "locking" only sets a flag on the file, if other users do not check to see if the flag is set (by attempting to lock it themselves), then they can do anything with the file that other file security mechanisms allow.

A locked file remains locked until an UN-EXCLUSIVE statement is issued for that file.

Syntax

EXCLUSIVE *file-name* [CONDITIONALLY]

Parameters

file-name the name of the file you want to lock. It must be opened before the EXCLUSIVE statement is executed. Also, the file may have a USE procedure associated with it in case an error occurs during execution of the EXCLUSIVE statement. If an error does occur, the USE procedure is executed.

Description

If used without the CONDITIONALLY option, the EXCLUSIVE statement continues to try to lock the file until it succeeds. If the file is already locked (for example, by another user), this means your program will pause until the lock succeeds.

To prevent the above from occurring, you can use the CONDITIONALLY option. This option attempts to lock the file and, if unsuccessful, returns immediately to your COBOL program.

The FILE STATUS data item, if any, associated with the file named in the EXCLUSIVE statement is updated to indicate whether or not the attempt to lock the file was successful. If the lock was successful, the STATUS-KEYS are set to "00". If the file is in use by another process and the lock condition is FALSE, or file options do not specify dynamic locking, or the calling process does not have multiple RIN capability, STATUS-KEY-1 is set to "9" and STATUS-KEY-2 contains the binary error code. For more information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

Programs that are to access an indexed file concurrently, within an environment that includes modification of the file, must include EXCLUSIVE/UN-EXCLUSIVE statements to maintain data integrity.

Refer to "MPE XL System Dependencies" in Appendix H for more information.

Example

The following example shows the EXCLUSIVE statement:

```
ENVIRONMENT DIVISION.
FILE-CONTROL.
     SELECT CUSTFILE ASSIGN TO "CUSTDATA" FILE STATUS IS CHECKER.
          ⋮
PROCEDURE DIVISION.
          ⋮
    OPEN I-O CUSTFILE.
    EXCLUSIVE CUSTFILE CONDITIONALLY.
    IF CHECKER IS EQUAL TO "00" PERFORM CUSTOMER-UPDATE
         ELSE PERFORM FIND-WHY.
```

EXIT Statement

The EXIT statement provides a common end point for a series of procedures.

Syntax

paragraph-name.

EXIT.

paragraph/section-name.

Paragraph-name and *paragraph/section-name* are not a part of the EXIT statement. They are shown to clarify the fact that:

- EXIT must appear in a sentence by itself.
- EXIT must be the only sentence in a paragraph.

An EXIT statement serves only to enable you to terminate a procedure and has no other effect on the compilation or execution of the program.

Example

```
PROCEDURE DIVISION.
        .
        .
    PERFORM FIX-IT THRU OUT.
        .
        .
    PERFORM EXCESS THRU OUT.
        .
        .
FIX-IT.
        IF CHARS IS ALPHABETIC THEN GO TO OUT.
        .
        .
EXCESS.
        IF OVER-AMT IS EQUAL TO 0 THEN GO TO OUT.
        .
        .
OUT.
        EXIT.
NEXT-PAR.
        .
        .
```

In the above illustration, both of the IF statements are the first lines of procedures executed by PERFORM statements. If the condition in either of the IF statements returns a "true" value, the statement branches to the `OUT` paragraph, the EXIT statement is executed, and control passes to the statement following the PERFORM statement that called the procedure.

EXIT PROGRAM Statement

The EXIT PROGRAM statement marks the logical end of a program. For more information on the EXIT PROGRAM statement refer to Chapter 11, "Interprogram Communication".

GOBACK Statement

The GOBACK statement is an HP extension to the ANSI COBOL standard.

The GOBACK statement marks the logical end of a program. For more information on the GOBACK statement refer to Chapter 11, "Interprogram Communication".

GO TO Statement

The GO TO statement transfers control from one part of the PROCEDURE DIVISION to another.

The optionality of *procedure-name-1* of the GO TO statement is an obsolete feature of the 1985 ANSI COBOL standard. See the description below for more information.

Syntax

The GO TO statement has two formats:

GO TO [*procedure-name-1*]

GO TO {*procedure-name-1*} ... DEPENDING ON *identifier-1*

Parameters

procedure-name-1	and its subsequent occurrences are names of procedures within the PROCEDURE DIVISION of your program.
identifier-1	the name of a numeric elementary data item that has no positions to the right of the decimal point.

Description

The first format of the GO TO statement transfers control to the procedure named by *procedure-name-1* or, if no procedure is named, to the procedure specified in a previously executed ALTER statement. An ALTER statement must be issued for this type of GO TO statement before it is executed if no procedure is named in it. This also implies that a GO TO statement without a procedure name specification must make up the only sentence in a paragraph. Refer to the ALTER statement description in this chapter for other restrictions.

Both the ALTER statement and the optionality of *procedure-name-1* in the GO TO statement are obsolete features of the 1985 ANSI COBOL standard.

If the first format of the GO TO statement does specify a procedure name and if it appears in a sequence of imperative statements within a sentence, it must be the last statement in that sentence.

In the second format of the GO TO statement, *identifier* must be the name of a numeric elementary item described with no positions to the right of the decimal point. It is used to determine which procedure is to be executed. If the contents of *identifier* is an integer in the range one to `n`, where `n` is the number of procedure names appearing in the GO TO statement, then control passes to the procedure in the position corresponding to the value of *identifier*. Otherwise, no transfer occurs and control passes to the next statement following the GO TO statement.

Examples

In the program below, the GO TO statement in the GO-PARA paragraph is equivalent to GO TO WHICH because of the ALTER statement preceding it. This form of the GO TO statement is an obsolete feature of the 1985 ANSI COBOL standard.

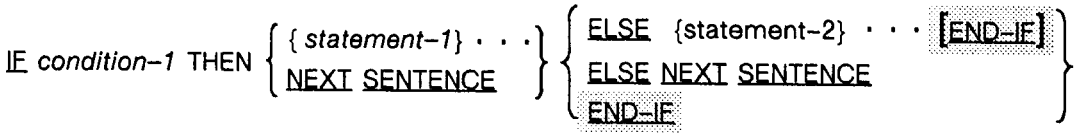
The second GO TO statement branches to UNDER, OVER, or EXACT, depending upon whether SELECTOR has a value of 1, 2, or 3 respectively. If SELECTOR has any other value, the DISPLAY statement is executed.

```
WORKING-STORAGE SECTION.
01 SELECTOR            PIC 9(3).
      :
PROCEDURE DIVISION.
         :
     ALTER GO-PARA TO PROCEED TO WHICH.
         :
GO-PARA.  GO TO.
AFTER-GO-PARA.
         :
     GO TO UNDER, OVER, EXACT DEPENDING ON SELECTOR.
     DISPLAY "SELECTOR OUT OF RANGE - VALUE IS ", SELECTOR.
         :
UNDER.
  :
OVER.
  :
EXACT.
  :
WHICH.
  :
```

IF Statement

The IF statement evaluates a condition and, depending upon the truth value of the condition, determines the subsequent action of the program.

Syntax



LG200026_109

Parameters

statement-1 and *statement-2*	each are imperative or conditional statements, optionally followed by a conditional statement.
condition-1	any valid COBOL condition as described under "Conditional Expressions" in Chapter 8.

Description

You may omit the ELSE NEXT SENTENCE phrase if it immediately precedes the period used to terminate the sentence.

If the END-IF phrase is specified, the NEXT SENTENCE phrase must not be specified.

The scope of the IF statement may be terminated by any of the following:

- An END-IF phrase at the same level of nesting.
- A separator period, which terminates IF statements at all levels of nesting.
- If nested, by an ELSE phrase associated with an IF statement at a higher level of nesting.

When an IF statement is executed, the following transfers of control occur:

- If the truth value of the condition is "true" and *statement-1* is specified, then if *statement-1* is a procedure branching or conditional statement, control is explicitly transferred according to the rules for that statement. If *statement-1* does not contain such a statement, then *statement-1* is executed and control passes to the end of the IF statement.

  If the truth value of the condition is "true" and the NEXT SENTENCE phrase is used instead of *statement-1*, control immediately passes to the next executable sentence.

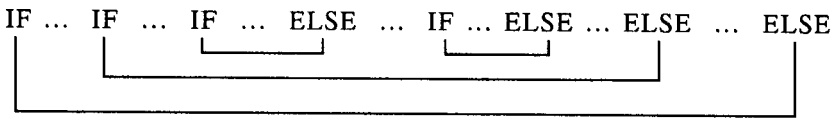
- If the truth value of the condition is "false" and if *statement-2* is specified, then if *statement-2* is a procedure branching or conditional statement, control is explicitly transferred according to the rules for that statement. If *statement-2* is not such a statement, then *statement-2* is executed and control passes to the end of the IF statements.

- If the truth value of the condition is "false" and *statement-2* is not specified, then the ELSE NEXT SENTENCE phrase, if specified, causes transfer of control to the next executable sentence. If the condition is false and the ELSE phrase is not specified then *statement-1* is ignored and control passes to the end of the IF statement.

Statement-1 or *statement-2* may be (or may contain) an IF statement, according to their description in the previous paragraphs. This is called a *nested IF statement*.

IF statements within IF statements may be considered as paired IF, ELSE, and `END-IF` combinations, proceeding from left to right. Thus, any ELSE or `END-IF` encountered is considered to apply to the immediately preceding IF that has not been already paired with an ELSE or `END-IF` respectively.

To clarify, the IF/ELSE pairing is shown in the following illustration:



LG200026_110

Example

```
BEGIN SECTION.
DATA-IN.
  READ REC-FILE RECORD INTO DATA-REC.
     ⋮
  IF DATA-REC IS NOT ALPHABETIC
     THEN
          IF DATA-REC IS NOT NUMERIC
             PERFORM ILLEGAL-CHARACTER
          ELSE NEXT SENTENCE
  ELSE PERFORM ALPHA-TYPE.
     ⋮
```

The IF statements above check that the data read into DATA-REC is either all alphabetic or all numeric.

The first IF statement consists of the IF/ELSE pair:

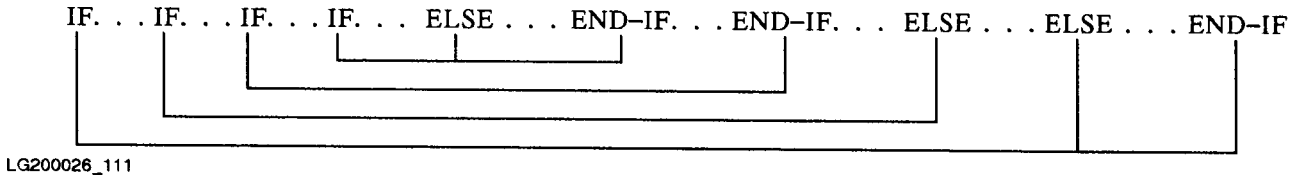
```
IF DATA-REC IS NOT ALPHABETIC...ELSE PERFORM ALPHA-TYPE.
```

The second IF/ELSE pair is:

```
IF DATA-REC IS NOT NUMERIC...ELSE NEXT SENTENCE.
```

Thus, if DATA-REC has any nonnumeric character or characters not from the English alphabet, the procedure ILLEGAL-CHARACTER is performed.

To clarify, the IF/ELSE and END-IF pairing is shown in the following illustration.

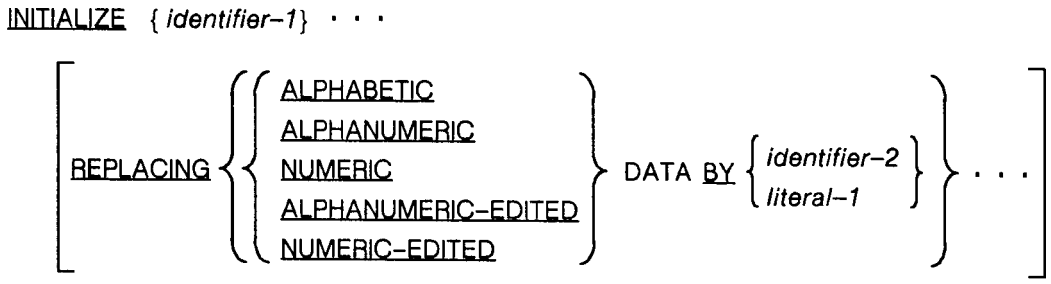


LG200026_111

INITIALIZE Statement

The INITIALIZE statement sets selected types of data fields to predefined values. For example, INITIALIZE can set numeric data to zeros or alphanumeric data to spaces.

Syntax



LG200026_112

Parameters

literal-1 and *identifier-2* — represent the sending area.

identifier-1 — represents the receiving area.

Description

The description of the data item referenced by *identifier-1* or any items subordinate to *identifier-1* may not contain the DEPENDING phrase of the OCCURS clause.

The data description entry for the data item referenced by *identifier-1* must not contain a RENAMES clause.

Each category stated in the REPLACING phrase must be a permissible category as a receiving operand in a MOVE statement, where the corresponding data item referenced by *identifier-2* or *literal-1* is used as the sending operand. (See "MOVE Statement", later in this chapter.)

The same category cannot be repeated in a REPLACING phrase.

An index data item may not be used as an operand in an INITIALIZE statement.

Initializing Data Fields

Following are rules for initializing data fields:

- The keyword following the word REPLACING corresponds to a category of data as defined under "PICTURE Clause" in Chapter 7 of this manual.
- INITIALIZE is executed as if a series of moves had been written. The receiving item of each MOVE is always an elementary item even if *identifier-1* refers to a group item.

  When the REPLACING phrase is specified:

  - If *identifier-1* references a group item a move is executed from *identifier-2* or *literal-1* to each elementary item of *identifier-1* that belongs to the category specified by the REPLACING phrase.
  - If *identifier-1* references an elementary item, a move is executed from *identifier-2* or *literal-1* to *identifier-1*, if it belongs to the category specified by the REPLACING phrase.

    The only exceptions are those fields specified in the first two rules below.
- Index data items and elementary FILLER data items are not affected by the execution of INITIALIZE.
- Any item that is subordinate to a receiving area identifier and contains the REDEFINES clause, or any item that is subordinate to such an item, is not initialized. However, a receiving area identifier may have a REDEFINES clause or be subordinate to a data item with a REDEFINES clause.
- When the statement is written without the REPLACING phrase, data items of the categories alphabetic, alphanumeric, and alphanumeric-edited are set to spaces. Data items of the categories numeric and numeric-edited are set to zeros. In this case, the operation is as if each affected data item is the receiving area in an elementary MOVE statement with the indicated source literal (that is, spaces or zeros).
- In all cases, the content of the data item referenced by *identifier-1* is set to the indicated value in the order of appearance of *identifier-1* (left to right) in the INITIALIZE statement. Where *identifier-1* references a group item, affected elementary items are initialized in the sequence of their definition within the group.
- If *identifier-1* occupies the same storage area as *identifier-2*, the result of the execution of this statement is undefined, even if both identifiers are defined by the same data description entry.

Example

```
WORKING-STORAGE SECTION.
01 A.
   05 B     PIC 999.
   05 C REDEFINES B.
      10 D  PIC X.
      10 E  PIC XX.
         ⋮
PROCEDURE DIVISION.
         ⋮
     INITIALIZE A.
     INITIALIZE C.
     INITIALIZE C  A.
```

When the INITIALIZE statements in the example above are executed, the data items are initialized as follows:

1. In the first INITIALIZE, `B` is set to zeroes, while `C`, `D` and `E` are ignored.
2. In the second INITIALIZE, `D` and `E` are set to blanks.
3. In the third INITIALIZE, `D` and `E` are set to blanks. `B` is set to zeroes. The net effect is that `D` and `E` are set to zeroes.

INSPECT Statement

The INSPECT statement can be used to perform one of three actions:

- It can count the number of occurrences of a given character or character substring within a data item.
- It can replace a given character or characters within a specified data item with another character or set of characters.
- It can perform both of the functions described above in a single operation.

By using the LEADING, BEFORE, and AFTER phrases, you can use INSPECT to replace only certain occurrences of characters within a data item. Also, by using CHARACTERS, you can tally and replace every character (or subset of characters when used in conjunction with LEADING, BEFORE, and AFTER) in a data item.

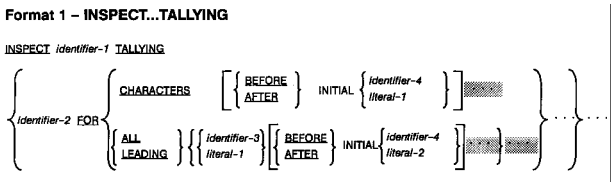
Syntax

The INSPECT statement has four formats as shown below.

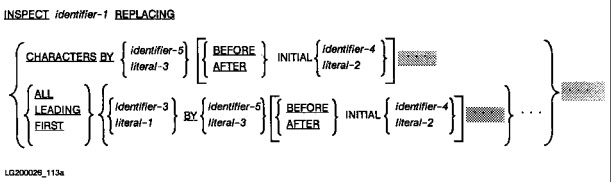
Note

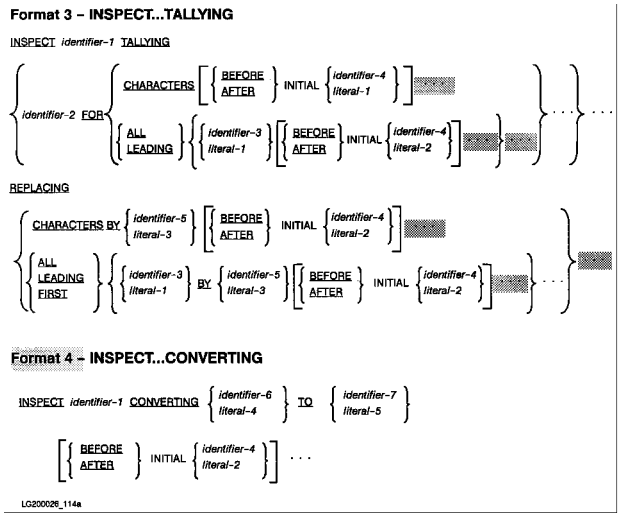
```
Format 4 (INSPECT CONVERTING) is a way to specify a translation
table converting one set of characters into another. Format 4 is
equivalent to format 2, but provides a concise way of achieving
the same results.
```

Format 1 – INSPECT...TALLYING



Format 2 – INSPECT...REPLACING





Parameters

identifier-1 — a variable representing either a group item or any category of elementary item described implicitly or explicitly as USAGE IS DISPLAY. This is the data item to be inspected.

identifier-2 — names an elementary numeric data item. It is used to contain the results of tallying occurrences of a character or characters in the data item represented by *identifier-1*. *Identifier-2* is not initialized by the INSPECT statement; therefore, if you want it initialized, you must do so programmatically before the INSPECT statement is executed.

identifier-3 through *identifier-n* — each must reference either an elementary alphabetic, alphanumeric, or numeric data item described implicitly or explicitly as USAGE IS DISPLAY.

literal-1 through *literal-5* — are each nonnumeric literals. Each may be any figurative constant except ALL.

If, in formats 1 and 3, *literal-1* is a figurative constant, that implicitly refers to a single character constant.

In formats 2 and 3, the size of the data referenced by *literal-3* or *identifier-5* must be equal to the size of the data item referenced by *literal-1* or *identifier-3*.

If *literal-3* is a figurative constant, its size is implicitly equal to the size of *literal-1*, or the size of the data item referenced by *identifier-3*.

If *literal-1* is a figurative constant, the data referenced by *literal-3* or *identifier-5* must be a single character.

When the CHARACTERS phrase is used, *literal-3* (or *literal-2*), or the size of the data item referenced by *identifier-5* (or *identifier-4*) must be one character in length.

No more than one BEFORE phrase and one AFTER phrase can be specified for any one ALL, LEADING, CHARACTERS, FIRST, or `CONVERTING` phrase.

The size of *literal-5* or the data item referenced by *identifier-7* must be equal to the size of *literal-4* or the data item referenced by *identifier-6*. When a figurative constant is used as *literal-5*, the size of the figurative constant is equal to the size of *literal-4* or the size of the data item referenced by *identifier-6*.

The same character must not appear more than once either in *literal-4* or in the data item referenced by *identifier-6*.

Description

When inspection takes place, the data items referenced by *identifier-1* and *identifiers-3* through *5* are all considered to be character strings, regardless of whether they are alphanumeric, alphanumeric-edited, numeric-edited, unsigned or signed numeric.

For any data item except signed numeric or alphanumeric, inspection is accomplished by inspecting the items as though they have been redefined as alphanumeric, and the INSPECT statement written to reference the redefined data item.

For signed numeric data items, inspection is accomplished by treating the data item as if it had been moved to an unsigned numeric data item, and then inspecting it as described in the preceding paragraph.

`CONVERTING Phrase`

A format 4 INSPECT statement is interpreted and executed as though a format 2 INSPECT statement specifying the same identifier-1 has been written. This is done with a series of ALL phrases, one for each character of literal 4. The result is as if each of these ALL phrases were referenced as literal-1, a single character of literal-4 and referenced, as literal-3, the corresponding single character of literal-5. Correspondence between the characters of literal-4 and the characters of literal-5 is by ordinal position within the data item.

If identifier-4, identifier-6, or identifier-7 occupy the same storage area as identifier-1, the result of the execution of this statement is undefined. This is true even if they are defined by the same data description entry. Refer to "Overlapping Operands and Incompatible Data" in Chapter 8 for more information.

Example

The following two INSPECT statements are equivalent:

```
INSPECT D-ITEM CONVERTING "ABCD" TO "XYZX" AFTER QUOTE BEFORE "#".

INSPECT D-ITEM REPLACING
      ALL "A" BY "X" AFTER QUOTE BEFORE "#"
      ALL "B" BY "Y" AFTER QUOTE BEFORE "#"
      ALL "C" BY "Z" AFTER QUOTE BEFORE "#"
      ALL "D" BY "X" AFTER QUOTE BEFORE "#".
```

The results of these statements are:

```
Initial value of D-ITEM:   AC"AEBDFBCD#AB"D
Final value of D-ITEM:     AC"XEYXFYZX#AB"D
```

How the Comparison Operation Occurs

To facilitate the following description of the comparison operation, the various groups of identifiers and literals are renamed. The names used and the identifiers or literals they represent are:

searchchars	represents *literal-1* or the contents of *identifier-3* and *5%*.
initchars	represents *literal-2* or the contents of *identifier-4*.
replacechars	represents *literal-3* or the contents of *identifier-5*.
data	represents the contents of *identifier-1*.

When inspection takes place, the elements of *data* are compared to *searchchars*. For each properly matched occurrence of *searchchars*:

- In formats 1 and 3, tallying occurs using *identifier-2* to contain the results.
- In formats 2 and 3, each set of properly matched characters in *data* is replaced by *replacechars*.

When the INSPECT statement is used in its simplest form, that is, without the LEADING, BEFORE, and AFTER phrases, the inspection occurs as follows:

The first set of *searchchars* is compared with an equal number of characters in *data*, starting with the leftmost character of *data*.

If no match occurs, the second set of *searchchars* is compared with an equal number of characters in *data*, again starting with the leftmost character of *data*.

This process continues for each set of *searchchars* until either a match occurs or all sets of *searchchars* are exhausted.

If all sets of *searchchars* are exhausted and no matches have occurred, comparison begins again using the first set of *searchchars*, but starting this time with the character immediately to the right of the leftmost character of *data*.

Again, comparison proceeds as described above until either a match occurs or all sets of *searchchars* are used.

If all sets are used and no matches have occurred, comparison begins again, starting with the character of *data* to the right of the leftmost character of *data*.

This continuing cycle of shifting one character to the right in the characters of *data* and using all of the sets of *searchchars* is terminated when, if no matches have occurred, the rightmost character of *data* has been used in a comparison with the last set of *searchchars*.

If a match does occur for some set of *searchchars* and TALLYING is specified, *identifier-2* is incremented by one. If REPLACING is specified, the matched characters of *data* are replaced by the *replacechars* that correspond to the set of *searchchars* being compared when the match occurred.

When a match occurs for a particular set of *searchchars*, the characters of *data* that matched are not compared with any following *searchchars*. Comparison begins again using the first set of *searchchars*, but starting with the character immediately to the right of the leftmost character of *data* that matched.

Inspection continues in the manner described above until the rightmost character of *data* has been used as the first character in a comparison with the last set of *searchchars*, or has been matched.

If TALLYING and REPLACING are both specified in the INSPECT statement, two completely separate comparisons, as described above, take place. The first comparison is used for TALLYING, and the second is used for REPLACING.

Example

The following INSPECT statement illustrates this form of inspection:

```
INSPECT WORD TALLYING COUNTER FOR ALL "X"
      REPLACING ALL "EE" BY "OX", "9" BY "E".
```

`WORD` contains the alphanumeric data, `YEEXE9XY`, and `COUNTER` is the variable used to hold the tally. Since INSPECT does not initialize `COUNTER`, assume it is initialized to zero before the INSPECT statement.

Following are the step-by-step comparisons that take place when this INSPECT statement executes:

1. Initially, `WORD=YEEXE9XY` and `COUNTER=0`. Begin the comparison starting at the first character of `YEEXE9XY`:

   ```
   YEEXE9XY
   ↑
   =X?
   ```

2. No match occurred. Begin the comparison again, starting at the second character:

   ```
   YEEXE9XY
    ↑
    =X?
   ```

3. No match occurred. Begin the comparison again, starting at the third character:

   ```
   YEEXE9XY
     ↑
     =X?
   ```

4. No match occurred. Begin the comparison again, starting at the fourth character:

```
YEEXE9XY
   ↑
   =X?
```

5. A match occurred. Increment COUNTER by 1, and begin the comparison again, starting with the fifth character of YEEXE9XY:

```
YEEXE9XY
    ↑
    =X?
```

6. No match occurred. Begin the comparison again, starting with the sixth character:

```
YEEXE9XY
     ↑
     =X?
```

7. No match occurred. Begin the comparison again, starting with the seventh character:

```
YEEXE9XY
      ↑
      =X?
```

8. A match occurred. Increment COUNTER by 1, and begin the comparison again, starting with the eighth character of YEEXE9XY:

```
YEEXE9XY
       ↑
       =X?
```

9. No match occurred. The last character of YEEXE9XY has been used as the first element in a comparison. The tallying cycle is complete. Begin the replacing cycle starting with the first characters of YEEXE9XY:

```
YEEXE9XY → YEEXE9XY
↑↑         ↑
=EE?       =9?
```

10. No match occurred. Begin the comparison again, starting with the second character:

```
YEEXE9XY
 ↑↑
 =EE?
```

11. A match occurred. Change EE to OX, and begin the comparison again, starting with the fourth character of YOXXE9XY:

```
YOXXE9XY → YOXXE9XY
   ↑↑         ↑
   =EE?       =9?
```

12. No match occurred. Begin the comparison again, starting with the fifth character:

```
YOXXE9XY → YOXXE9XY
    ↑↑         ↑
    =EE?       =9?
```

13. No match occurred. Begin the comparison again, starting with the sixth character:

```
YOXXE9XY → YOXXE9XY
     ↑↑          ↑
     =EE?        =9?
```

14. A match occurred. Replace `9` by `E` in `YOXXE9XY`, and begin the comparison again, at the seventh character of `YOXXEEXY`:

```
YOXXEEXY → YOXXEEXY
      ↑↑          ↑
      =EE?        =9?
```

15. No match occurred. Begin the comparison again, starting with the eighth character:

```
YOXXEEXY → YOXXEEXY
       ↑↑          ↑
       =EE?        =9?
```

16. No match occurred. The last character of `YOXXEEXY` has been used as the first character in a comparison. The comparison cycle for REPLACING is complete. This ends execution of the INSPECT statement.

The result of this INSPECT statement is summarized by the fact that `WORD` now contains the character string, `YOXXEEXY`, and `COUNTER` now contains the integer 2.

BEFORE and AFTER Phrases

No more than one BEFORE phrase and one AFTER phrase can be specified for any one ALL, LEADING, CHARACTERS, FIRST, or `CONVERTING` phrase.

The comparison operation described on the preceding pages is affected by the BEFORE and AFTER phrases in the following way:

If the BEFORE phrase is used, the associated *searchchars* are used only in those comparison cycles that make comparisons of characters of *data* to the left of the first occurrence of the associated *initchars*.

If *initchars* does not appear in *data*, the BEFORE phrase has no effect upon the comparison operation.

If the AFTER phrase is used, the associated *searchchars* are used only in those comparison cycles that make comparisons of characters of *data* to the right of the first occurrence of the associated *initchars*.

If *initchars* does not appear in *data*, the associated *searchchars* are never used in the comparison cycle. This is equivalent to not using the clause in which the AFTER phrase appears.

Multiple occurrences of the BEFORE/AFTER phrase allow the TALLYING/REPLACING operation to be initiated after the beginning of the inspection of data begins and/or is terminated before the end of the inspection of data ends.

LEADING Phrase

If the LEADING phrase is used in an INSPECT statement, it causes *identifier-2* (the variable used to hold the tally) to be incremented by one for each contiguous matching of searchchars with a character of data, provided that the matching begins with the leftmost character of the characters that make up data.

For replacing, the LEADING phrase has the effect of replacing each contiguous occurrence of matched characters to be replaced by *replacechars*, provided that the matching begins with the leftmost character of *data*.

If the first character (or characters) of *data* is not the same as *searchchars*, the clause in which the LEADING phrase appears has no effect upon the data, or the variable used to hold the tally.

ALL Phrase

When used in tallying, the ALL phrase causes the contents of *identifier-2* to be incremented by one for each occurrence of *searchchars* within *data*. When used in replacing, the ALL phrase causes each set of characters in *data* matched with the *searchchars* to be replaced.

CHARACTERS Phrase

When the CHARACTERS phrase is used in tallying, the contents of *identifier-2* are incremented by 1 for each character in the set of characters used in the comparison cycle. This does not necessarily imply that all characters are tallied, since the BEFORE and AFTER phrases can limit comparison to only part of *data*.

When the CHARACTERS phrase is used in replacement, each character in the set of characters used in the comparison cycle is replaced by *replacechars*, regardless of what the character in *data* is. For example, this phrase can be used to initialize a data item by not using the BEFORE or AFTER phrases to limit the part of *data* to be acted upon.

```
Multiple occurrences of the REPLACING CHARACTERS phrase are allowed.
```

FIRST Phrase

When the FIRST phrase is used in replacement, the leftmost occurrence in *data* of *searchchars* is replaced by the associated *replacechars*.

Examples

Assuming that the variable `THISONE` has the data `"WARNING"` in it, the INSPECT statement,

```
INSPECT THISONE REPLACING ALL "N" by "P" BEFORE INITIAL "I".
```

results in `THISONE` having the data `WARPING` in it.

If `REC` has the data `"JIMGIRAFFEEGAVEGUMDROPS"` and `ACUM` is zero initially, the following `INSPECT` statement results in `ACUM` being 2, and `REC` containing `JIMRIRAFFEERAVERUMDROPS`.

```
INSPECT REC TALLYING ACUM FOR ALL "F", REPLACING ALL "G" BY "R".
```

If `PETE` has the data `"CBVFEET"` before execution of the following INSPECT statement, then following execution, `PETE` contains `CBVFOOO`.

```
INSPECT PETE REPLACING LEADING "BV" BY "AT",
      CHARACTERS BY "O" AFTER INITIAL "F".
```

In the following example, COUNT-*n* is assumed to be zero immediately prior to execution of the statement. Table 9-1 shows the result of executing the two successive INSPECT statements.

```
INSPECT ITEM TALLYING
     COUNT-0 FOR ALL "AB" BEFORE "BC"
     COUNT-1 FOR LEADING "B" AFTER "D"
     COUNT-2 FOR CHARACTERS AFTER "A" BEFORE "C".

INSPECT ITEM REPLACING
     ALL "AB" BY "XY" BEFORE "BC"
     LEADING "B" BY "W" AFTER "D"
     FIRST "E" BY "V" AFTER "D"
     CHARACTERS BY "Z" AFTER "A" BEFORE "C".
```

Table 9-1. Results of INSPECT Statement Execution

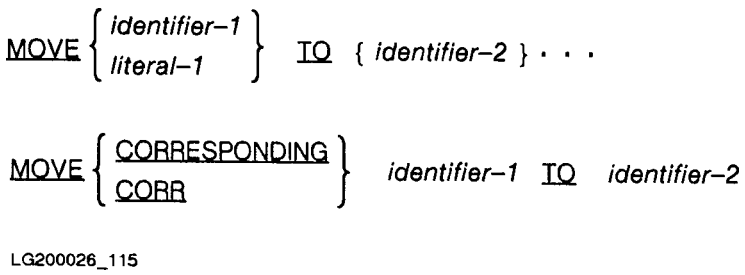
Initial Value of Item	COUNT-0	COUNT-1	COUNT-2	Final Value of Item
BBEABDABABBCABEE	3	0	2	BBEXYZXYXYZCABVE
ADDDDC	0	0	4	AZZZZC
ADDDDA	0	0	5	AZZZZZ
CDDDDC	0	0	0	CDDDDC
BDBBBDB	0	3	0	BDWWWDB

MOVE Statement

The MOVE statement transfers data to one or more data areas in accordance with the rules of editing.

Syntax

The MOVE statement has two general formats:



Parameters

identifier-1 and *literal-1*	the sending areas. The special registers, TALLY, TIME-OF-DAY, CURRENT-DATE, and WHEN-COMPILED may be used as sending items.
identifier-2	and its subsequent occurrences, are the receiving areas.
CORR	an abbreviation for CORRESPONDING. An index data item cannot be used as an operand of a MOVE statement.

Description

If you use format 2, both identifiers must be group items. Selected items are moved from within *identifier-1* to selected items within *identifier-2*. The results are the same as if you referred to each pair of corresponding identifiers in separate MOVE statements. The rules governing correspondence are presented in Chapter 8 under the heading, "CORRESPONDING Phrase".

If you use format 1, the data designated by *literal-1*, or by *identifier-1* is moved, in turn, to *identifier-2*. Any subscripting or indexing associated with identifiers to the right of the keyword TO is evaluated immediately before the data is moved to the respective data items.

Any subscripting, indexing, reference modification, or function associated with *identifier-1* is evaluated only once, immediately before the data is moved to the first of the receiving operands.

For example, the result of the following statement:

```
MOVE A(B) TO B, C(B)
```

Is equivalent to the following three statements:

```
MOVE A(B) TO temp
MOVE temp TO B
MOVE temp TO C(B)
```

Where temp is an intermediate result item used internally by the compiler. Note that the move of A(B) to B affects the element of C to which A(B) is moved. That is, if B is initially one and A(B) is 9, then after 9 is moved to B, A(1) is moved to C(9). It is not moved to C(1).

Rules For Moving Data

All data is moved according to the rules for moving elementary data items to elementary data items. This is called an elementary move. Valid and invalid moves are determined by the categories of the sending and receiving data items. Refer to "PICTURE Clause" in Chapter 7 for a description of the various categories.

Any move that is not an elementary move is treated exactly as if it were a move from one alphanumeric elementary data item to another, except that there is no conversion from one form of internal representation to another. In such a move, the receiving data item is filled without respect to the individual elementary or group items contained in either the sending or receiving area, except when the sending data item contains a table whose OCCURS clause uses the DEPENDING ON clause. In this case, only the area specified by the DEPENDING ON clause is filled or moved.

```
When a receiving item is a variable length data item and contains the object
of the DEPENDING ON phrase, the maximum length of the item is used.
```

If the move is from a group to an elementary item, justification takes place if specified in the receiving item.

Rules For Elementary Moves

The following rules apply to an elementary move between data items belonging to one of the five categories of data:

- All numeric literals and the figurative constant ZERO belong to the numeric category; all nonnumeric literals, and all figurative constants except SPACE and ZERO belong to the alphanumeric category; SPACE belongs to the alphabetic category.
- An alphanumeric-edited or alphabetic data item cannot be moved to a numeric or numeric-edited data item.
- A numeric or numeric-edited data item cannot be moved to an alphabetic data item.
- A noninteger numeric literal or noninteger numeric data item cannot be moved to an alphanumeric or alphanumeric-edited data item.
- All other elementary moves are valid and are performed according to the rules listed below.

Any necessary conversion from one internal representation to another takes place during valid elementary moves, as does any editing specified for, or de-editing implied by, the receiving data item.

Alphanumeric or Alphanumeric-Edited Receiving Item

When an alphanumeric-edited or alphanumeric item is a receiving data item, alignment and any necessary space filling takes place as defined under "Data Alignment" in Chapter 4. If the size of the sending item is larger than the receiving item, the excess characters are truncated on the right after the receiving data item is filled.

If the sending item is a signed numeric item, the sign is not moved, regardless of whether the sign is separate or not. If the sign is separate, however, the sending item is considered to be one character shorter than its actual size. If the sending operand is numeric-edited, no de-editing takes place. If the usage of the sending operand is different from that of the receiving operand, the sending operand is converted to the internal representation of the receiving operand. If the sending operand is numeric and contains the PICTURE symbol 'P', all digit positions specified with this symbol are considered to have the value zero and are counted in the size of the sending operand.

Numeric or Numeric-Edited Receiving Item

When a numeric or numeric-edited item is the receiving item, alignment by decimal point and any necessary zero filling is performed as defined under "Data Alignment" in Chapter 4. The exception to this rule is when zeros are replaced because of editing requirements of the receiving data item.

For signed numeric receiving items, the sign of the sending item is placed in the receiving item. An unsigned numeric sending item causes a positive sign to be generated for the receiving item. Also, any conversion of the representation of the sign, such as from a zoned overpunched sign to a separate sign, is done as necessary.

For an unsigned numeric receiving item, the absolute value of the sending item is moved and no operational sign is generated for the receiving item.

For an alphanumeric sending item, data is moved as if the sending item were described as an unsigned numeric integer. The ANSI limit for the length of a numeric item is 18 digits; however, HP COBOL II extends the limit to the length of an intermediate result, as defined in the COMPUTE statement.

```
When the sending operand is numeric-edited, de-editing is implied to establish
the operand's unedited numeric value, which may be signed; then the unedited
numeric value is moved to the receiving field.  This means that blanks are
converted to zeros and insertion characters and floating characters are
stripped.  Any sign characters are translated into the proper internal form
of the sign described by the USAGE clause.
```

Alphabetic Receiving Item

When a receiving field is described as alphabetic, justification and any necessary space filling is performed as specified under "Data Alignment" in Chapter 4. If the size of the sending data item is larger than the receiving data item, the excess characters are truncated to the right after the receiving item is filled.

Table 9-2 summarizes the rules presented above.

Table 9-2. Permissible Moves

Source Field \ Receiving Field	Group	Alphabetic	Alphanumeric	External Decimal (DISPLAY)	Binary (COMP)	Numeric Edited	Alphanumeric Edited	Packed Decimal (COMP-3)
Group	Y	Y	Y	Y	Y^1	Y^1	Y^1	Y^1
Alphabetic	Y	Y	Y	Δ	Δ	Δ	Y	Δ
Alphanumeric	Y	Y	Y	Y^4	Y^4	Y^4	Y	Y^4
External Decimal (DISPLAY)	Y^1	Δ	Y^2	Y	Y	Y	Y^2	Y
Binary (COMP)	Y^1	Δ	Y^2	Y	Y	Y	Y^2	Y
Numeric Edited	Y	Δ	Y	Y^6	Y^6	Y^6	Y	Y^6
Alphanumeric Edited	Y	Y	Y	Δ	Δ	Δ	Y	Δ
Zero; (numeric or alphanumeric)	Y	Δ	Y	Y^3	Y^3	Y^3	Y	Y^3
Spaces	Y	Y	Y	Δ	Δ	Δ	Y	Δ
High-Value, Low-Value, Quotes	Y	Δ	Y	Δ	Δ	Δ	Y	Δ
All Literal	Y	Y	Y	Y^5	Y^5	Y^5	Y	Y^5
Numeric Literal	Y^2	Δ	Y^2	Y	Y	Y	Y^2	Y
Nonnumeric Literal	Y	Y	Y	Y^5	Y^5	Y^5	Y	Y^5
Packed Decimal (COMP-3)	Y^1	Δ	Y^2	Y	Y	Y	Y^2	Y

Y = permissible; Δ = prohibited
Y^1 = move without conversion
Y^2 = permissible only if the decimal point is to the right of the least significant digit.
Y^3 = a numeric move
Y^4 = the move is treated as an External Decimal (integer) field. (The characters must be numeric.)
Y^5 = the literal must consist only of numeric characters and is treated as an External Decimal (integer) field.
Y^6 = de-edited move

LG200026_210b

Example

```
FILE SECTION.
FD FILE-IN.
01 FILE-REC.
  02 EMP-FIELD.
     03 NAME     PIC X(20).
     03 AGE      PIC 99.
     03 EMP-NO   PIC X(9).
  02 LOCALE      PIC X(35).
     ⋮
WORKING-STORAGE SECTION.
01 FIELD.
   02 SUB-F1   PIC BBXX        VALUE SPACES.
   02 SUB-F2   PIC XX/XX/XX    VALUE SPACES.
01 NUM-IN      PIC S9(3)V99    VALUE -12099.
01 CARD-NUM    PIC S9(3)V99    SIGN IS TRAILING VALUE ZERO.
01 NUM-JUNK    PIC S9(5)       VALUE -12345

01 INFO-OUT.
   02 EMP-FIELD.
      03 NAME    PIC X(20)BBB        VALUE SPACES.
      03 AGE     PIC XXBBB           VALUE SPACES.
      03 EMP-NO  PIC XXXBXXBXXXXBBB  VALUE SPACES.
   02 EXEMPTIONS PIC 99              VALUE ZERO.
```

Given the fields described above, the MOVE statement:

```
MOVE NUM-IN TO FIELD
```

gives the result:

```
1209R␣␣␣␣␣␣␣␣
```

A group move is done with no conversion.

The statement, `MOVE NUM-JUNK TO SUB-F2`, gives the result:

```
12/34/5␣
```

The space to the right was supplied in order to fill the field, and no operational sign was moved.

Assuming that the current contents of FILE-REC are in order:

```
NAME                    JASON␣PENNY␣␣␣␣␣␣␣␣␣␣
AGE                     AGE
EMP-NO                  585241215
LOCALE                  WASHINGTON␣DISTRICT␣OF␣COLUMBIA␣␣␣␣
```

and the current contents of INFO-OUT are all spaces for NAME, AGE, and EMP-NO, and zeros for EXEMPTIONS, then the statement MOVE CORRESPONDING FILE-REC TO INFO-OUT gives the following results in INFO-OUT,

```
NAME                    JASON␣PENNY␣␣␣␣␣␣␣␣␣␣␣␣␣
AGE                     39␣␣␣
EMP-NO                  585␣24␣1215␣␣␣
EXEMPTIONS              00
```

Finally, the MOVE statement:

```
MOVE NUM-IN TO CARD-NUM
```

results in the contents of CARD-NUM being 1209R, since R is the zoned overpunch character for 9 in a negative number.

The following example contains a de-edited MOVE statement.

Given:

```
01  NUM-ITEM   PIC S9(5)V99.
01  EDITED-ITEM  PIC  $ZZZ,ZZZ.99-.

MOVE -23.00 TO EDITED-ITEM
```

The following is a valid de-edited MOVE:

```
MOVE  EDITED-ITEM  TO NUM-ITEM.
```

The results of the example above are the same as in the following example:

```
MOVE -23.00 TO NUM-ITEM
```

MULTIPLY Statement

The MULTIPLY statement multiplies a number by one or more other numbers and stores the result in one or more locations.

Syntax

The MULTIPLY statement has the following two formats:

Format 1

```
MULTIPLY { identifier-1 } BY { identifier-2 [ROUNDED] } ...
         { literal-1    }

[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-MULTIPLY]
```

Format 2

```
MULTIPLY { identifier-1 } BY { identifier-2 }
         { literal-1    }    { literal-2    }

GIVING { identifier-3 [ROUNDED] } ...
[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-MULTIPLY]
```

LG200026_116

Parameters

identifier-1, *identifier-2*, and so forth	numeric elementary items, except that in format 2, each identifier following the GIVING keyword may be numeric-edited elementary items.
literal-1 and *literal-2*	any numeric literal.

Description

The composite of operands (that is, the hypothetical data item resulting from the superimposition of all receiving data items aligned on their decimal points) in any given MULTIPLY statement must not contain more than 18 digits.

The ROUNDED, SIZE ERROR, and `NOT ON SIZE ERROR` phrases, as well as multiple results and overlapping operands, are described in Chapter 8.

When you use format 1 of the MULTIPLY statement, *literal-1* or the contents of *identifier-1* is multiplied, in turn, by the identifiers following the BY keyword. The result of each multiplication is stored in each of the identifiers following the BY keyword immediately after that particular product is determined.

When format 2 is used, the value of *identifier-1* or *literal-1* is multiplied by the value of *identifier-2* or *literal-2*, and the resulting product is stored in each identifier following the GIVING keyword.

Examples

```
MULTIPLY 2 BY ROOT, SQ-ROOT, ROOT-SQUARED.
```

If `ROOT` has the value 2, `SQ-ROOT` the value square root of 2 (1.41), and `ROOT-SQUARED` the value 4, then the above MULTIPLY statement results in `ROOT` having the value 4, `SQ-ROOT` having twice the value the square root of 2 (2.82), and `ROOT-SQUARED` having the value 8.

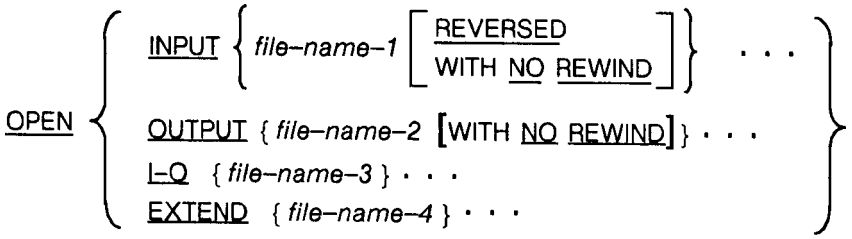
Assuming `ROOT` to be 2, the following MULTIPLY statement assigns the value 4 to `ROOT-SQUARED`:

```
MULTIPLY ROOT BY ROOT GIVING ROOT-SQUARED.
```

OPEN Statement

The OPEN statement opens a specified file or files. It also performs checking and writing of labels, and other input or output operations.

Syntax



Where *file-name-1* through *file-name-4* are the files to be opened.

Description

The NO REWIND phrase can only be used for sequential files. It has no meaning for indexed, random, or relative files and must not be used for such files. When using ANSI74 entry point, EXTEND can only be used for sequential files.

The REVERSED phrase is not implemented in HP COBOL II. If used, it is treated as a comment. The REVERSED phrase is an obsolete feature of the 1985 ANSI COBOL standard.

You can use a single OPEN statement to open several files. The files to be opened need not have the same organization or access. However, each file must have a description equivalent to the description used for it when it was created.

Prior to the successful execution of an OPEN statement for a file, the file must not be referenced implicitly or explicitly by the execution of a statement.

When a file has been successfully opened, its associated record area is made available to the program. However, execution of an OPEN statement does not obtain or release any data record of the opened file. The various input-output statements must be used to do this.

The INPUT, OUTPUT, I-O, or EXTEND phrases must only be used once in an OPEN statement for any given file. You may open a file with any of the phrases in the same program. However, the file must be closed each time using a CLOSE statement without the LOCK phrase (without the REEL/UNIT phrase in the case of sequential files) before another OPEN statement can be issued for that file.

The INPUT phrase opens a file for input operations. If the OPTIONAL phrase is specified in the SELECT clause of a file and the file is not present, the first READ statement for the file causes an AT END condition.

The OUTPUT phrase creates a file if it does not already exist and opens it for output operations. When the output file is opened, it contains no data records. The file created is a job or session temporary file using the formal file designator specified in the SELECT clause. For information about how INDEX files are created, see Appendix H, "MPE XL System Dependencies."

The I-O phrase permits the opening of a file for both input and output operations. As an HP extension to ANSI COBOL'74, HP COBOL II creates the file if it does not already exist. Under ANSI COBOL'85, the file is not created unless the OPTIONAL keyword is specified in the SELECT clause. This also applies to the EXTEND phrase.

When files are opened with the INPUT or I-O phrase, the OPEN statement (without the EXTEND or NO REWIND phrases in the case of sequential files) sets the file position indicator to the first record currently existing within the file. (Indexed files use the prime record key to determine the first record to be accessed.) If no records exist in the file, the file position indicator is set in such a manner that the next executed format 1 READ statement for the file results in an AT END condition.

```
For a relative or indexed file in the dynamic access mode, 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 at the time the READ NEXT statement executes. This is
an incompatible feature with ANSI COBOL'74.
```

In ANSI COBOL'74, the above sequence of events causes the READ NEXT statement to access the record in the file that was first when the OPEN statement executed, not necessarily the record that is first when the READ NEXT executes. If one of the WRITE statements inserts a record with a key or relative record number lower than any other record in the file, that record would not be read by the READ NEXT. This is the difference between ANSI COBOL'74 and ANSI COBOL'85. For compatibility with ANSI COBOL'74, use the ANSI74 entry point.

If a CLOSE statement has not been issued for an open file when a STOP RUN statement (or a GOBACK statement in a main program) is executed, the file is automatically closed by the COBOL run-time system.

Label Records

The LABEL RECORDS clause of the file description entry for a file indicates whether label records are present in the file.

The following rules apply when label records are present:

1. When the INPUT phrase is used in the OPEN statement, standard labels are checked in accordance with the conventions for input label checking. Any user labels specified for the file are processed according to the procedure specified by a format 2 USE statement.
2. When the OUTPUT phrase is used in the OPEN statement, standard labels are written in accordance with the conventions for output label writing. Any user labels specified for the file are written according to the procedure specified by a format 2 USE statement.
3. When the I-O phrase is used in an OPEN statement, standard labels are checked in accordance with the conventions for input-output label checking. New standard labels are

written in accordance with input-output label writing. Any user labels specified for the file are processed according to the procedure specified by a format 2 USE statement.

When label records are specified, but are not present, and the file was opened using the INPUT phrase, an input-output routine error results.

When label records are present, but not specified, and the file was opened using the INPUT phrase, the label records are ignored.

EXTEND, REVERSE, and NO REWIND Phrases

The REVERSE and NO REWIND phrases apply to sequential files. When using the ANSI74 entry point, EXTEND can only be used for sequential files.

The REVERSE and NO REWIND phrases are not recognized by the HP COBOL II compiler, and are treated as comments if specified.

The EXTEND phrase, when specified in an OPEN statement, positions the file immediately following the last logical record of the file. Subsequent WRITE statements for the file add records at the end of the file as though the file had been opened with the OUTPUT phrase.

If you specify the EXTEND phrase for multiple file reels, a compilation error diagnostic appears. This is done to conform to the 1974 ANSI COBOL standard. The object program on an HP computer system allows the EXTEND operation to execute, even for multiple file reels. However, any files following the referenced file are written over and are made inaccessible.

When the EXTEND phrase is specified and the LABEL RECORDS clause of the file description for the file indicates the existence of label records, the execution of the OPEN statement includes the following steps:

1. The beginning file labels are processed only if the file resides on a single reel or unit. Any user labels specified for the file are processed according to the procedure specified by a format 2 USE statement.
2. Processing then proceeds as though the file had been opened with the OUTPUT phrase.

Permissible Statements

The following tables indicate the statements permitted to be executed for a file of a given organization opened in a given open mode.

Table 9-3. Sequential Organization

Statement	Open Mode			
	Input	Output	Input-Output	Extend
Read	X		X	
Write		X	X	X
Rewrite			X	

WRITE with Input-Output mode is an HP extension to the ANSI COBOL standard.

Table 9-4. Relative and Indexed Organization

File Access Method	Statement	Open Mode		
		Input	Output	Input-Output
Sequential	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
	DELETE			X
	SEEK (Rel. only)			
Random	READ	X		X
	WRITE		X	X
	REWRITE			X
	START			
	DELETE			X
	SEEK (Rel. only)	X		X
Dynamic	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X
	SEEK (Rel. only)	X		X

Note For ANSI, EXTEND can be used with the WRITE statement only if Access Mode Sequential.

The SEEK statement is an HP extension to the ANSI COBOL standard.

Table 9-5. Random Organization

Statement	Open Mode		
	Input	Output	Input-Output
Seek	X		X
Read	X		X
Rewrite			X
Write		X	X

FILE STATUS Data Item

If the file named in the OPEN statement has a FILE STATUS data item associated with it, the FILE STATUS data item is updated following the execution of the OPEN statement to indicate whether or not the attempt to open the file was successful. Refer to "FILE STATUS Clause" in Chapter 6 for valid combinations of status keys 1 and 2. For more information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

PERFORM Statement

The PERFORM statement transfers control explicitly to one or more procedures, and implicitly returns control to the statement after the current PERFORM statement when execution of the specified procedure or procedures is complete. The PERFORM statement is also used to control execution of one or more imperative statements that are within the scope of that PERFORM statement.

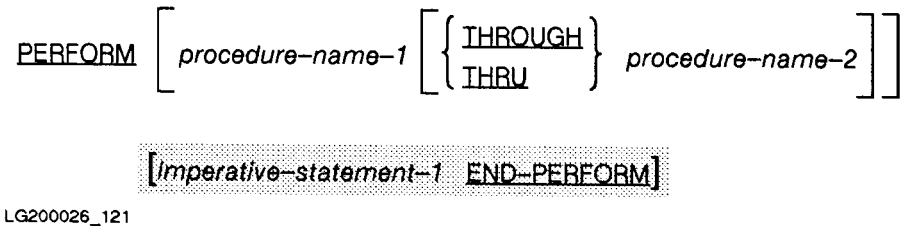
These parameters are used by the four general formats of the PERFORM statement.

Parameters

procedure-name-1 and *procedure-name-2*	names of procedures within the PROCEDURE DIVISION of the program in which the PERFORM statement appears. If one of the procedures is a declarative procedure, then both must be declarative.
identifier-1 through *identifier-7*	numeric elementary items described in the DATA DIVISION.
literal-1 through *literal-4*	numeric literals.
condition-1 and *condition-2*	any valid COBOL condition.
`THRU`	equivalent to THROUGH.
`END-PERFORM`	terminates the in-line PERFORM statement.

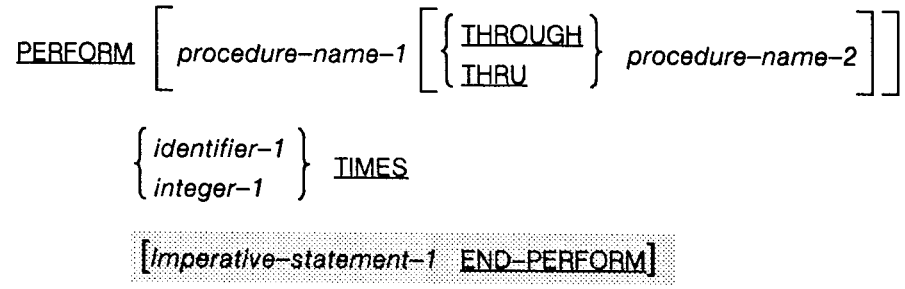
Syntax

Format 1



Format 1 of the PERFORM statement is the basic PERFORM statement. The specified set of statements is executed once as described on the preceding pages. Control then passes to the next executable statement following the PERFORM statement.

Format 2



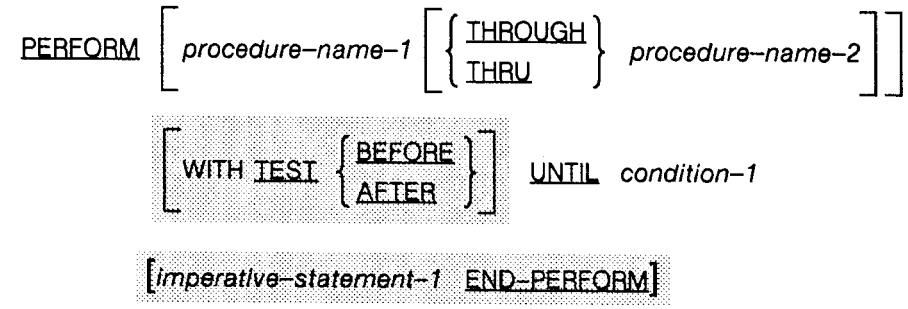
LG200026_122

A format 2 PERFORM statement allows you to perform the specified set of statements the number of times specified by *integer-1* or the numeric integer named by *identifier-1*. Following the execution of the statements, control is passed to the next executable statement following the PERFORM statement.

If *identifier-1* is a negative integer or zero (0) when the PERFORM statement is executed, control immediately passes to the next executable statement following the PERFORM statement. Thus, a negative integer or zero value for *identifier-1* is equivalent to not having the PERFORM statement in the code at the time of execution.

If *identifier-1* is referenced during execution of the PERFORM statement, it does not change the number of times the specified set of statements is executed, as indicated by the initial value of *identifier-1*.

Format 3



A format 3 PERFORM statement uses *condition-1* to control the number of times the specified set of statements is performed. The statements are performed as long as *condition-1* is false.

When *condition-1* is true, control passes to the next executable statement after the PERFORM.

Note Be sure that, within the specified set of statements, *condition-1* eventually has a value of true.

```
When TEST BEFORE is specified:
The condition is checked before the specified set of statements is
performed (the set of statements will be performed 0 or more times).

When TEST AFTER is specified:
The condition is checked after the specified set of statements has been
performed (the statements are always performed at least once).
```

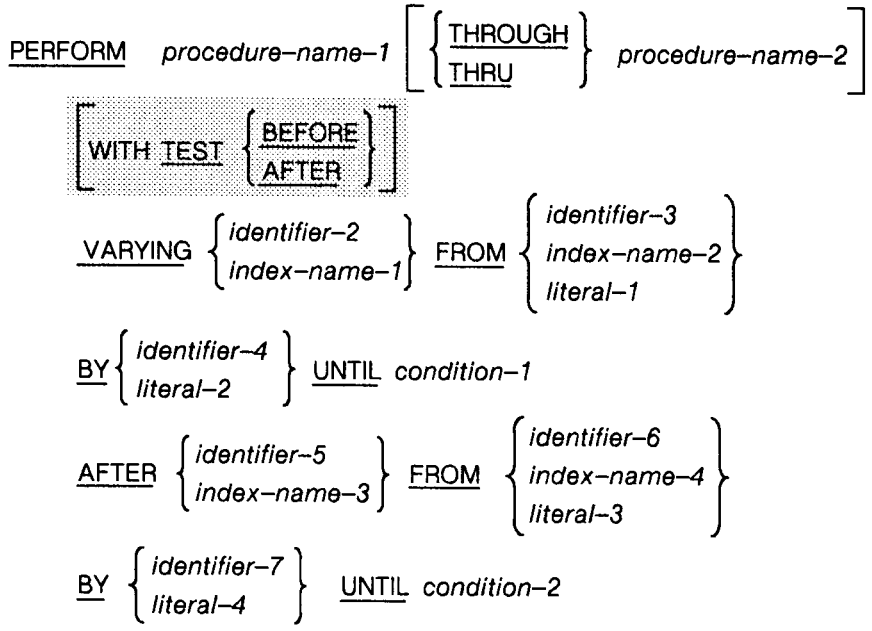
If neither TEST BEFORE nor TEST AFTER is specified, the TEST BEFORE phrase is assumed.

Format 4 PERFORM Statement (PERFORM ... VARYING)

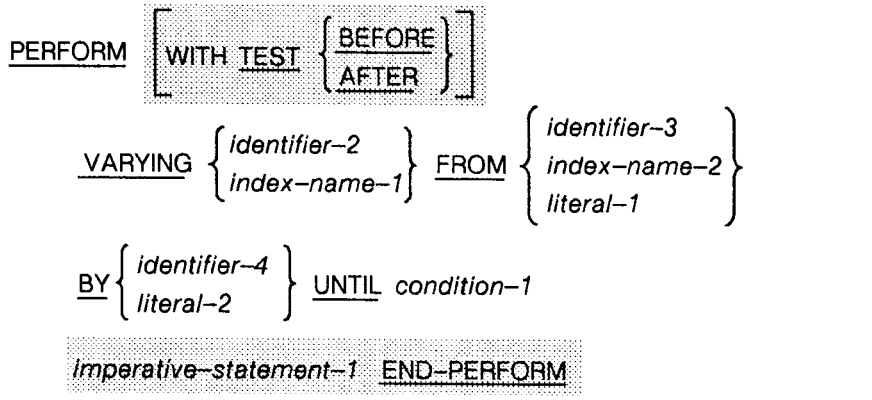
A PERFORM ... VARYING statement is used to augment the values referenced by one or more identifiers or index names in an orderly fashion during the execution of a PERFORM statement.

For clarity, two versions of format 4 are presented below: the first is for an out-of-line PERFORM and the second is for an in-line PERFORM.

Format 4a. (Out-of-line PERFORM . . . VARYING)



Format 4b. (In-line PERFORM . . . VARYING)



LG200026_124a

- Up to six AFTER phrases can be specified in format 4. However, if *procedure-name-1* is omitted, the AFTER phrase cannot be specified. Any literal used in the BY phrase, and data items referenced by *identifier-4*, *identifier-7*, and *identifier-10* must not have a value of zero. Also, if an index name is used in the VARYING or AFTER phrase, then the following is true:
  - If an identifier or a literal is specified in the associated FROM phrase, the data item referenced by the identifier or the value of the literal must be a positive integer.
  - If an identifier is specified in the associated BY phrase, it must name an integer data item. If a literal is specified, it must be a positive integer.

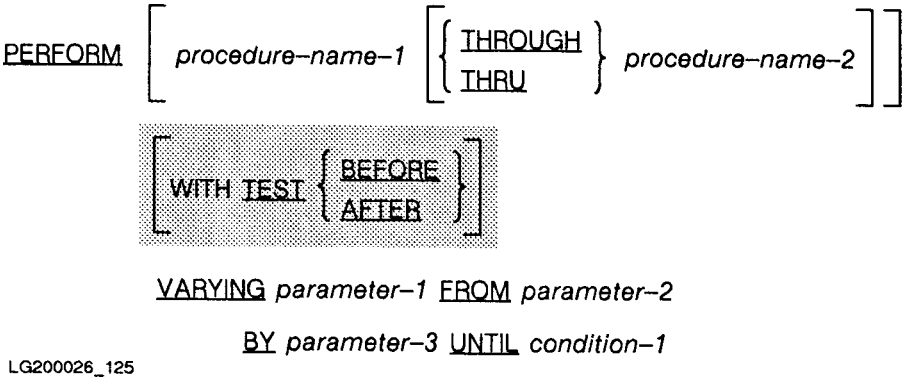
  If an index name is specified in the FROM phrase, the following is true:

  - If an identifier is used in the associated VARYING or AFTER phrase, it must name an integer data item.
  - If an identifier or literal is used in the associated BY phrase, the literal or the data item referenced by the identifier must be an integer.

  When an index name appears in a VARYING or an AFTER phrase, it is initialized and subsequently augmented according to the rules of the SET statement. When an index name appears in a FROM phrase, any identifier appearing in an associated VARYING or AFTER phrase is initialized according to the rules of the SET statement. It is subsequently augmented using the SET statement rules as described below.

Variation of a Single Identifier

Variation of a single identifier is accomplished by using a format 4 PERFORM statement of the following form:



Out-of-Line PERFORM

When a PERFORM statement is executed, control is transferred to the first statement of the procedure named *procedure-name-1*. This transfer occurs only once for each execution of a PERFORM statement.

An implicit transfer of control to the next executable statement following the PERFORM statement is established as follows:

- If *procedure-name-1* is a paragraph name and *procedure-name-2* is not specified, then the return occurs after the last statement of *procedure-name-1*.
- If *procedure-name-1* is a section name and *procedure-name-2* is is not specified, then the return occurs after the last statement of the last paragraph of the section named by *procedure-name-1*.
- If *procedure-name-1* and *procedure-name-2* are specified, and *procedure-name-2* is a paragraph name, the return occurs after the last statement of *procedure-name-2*.
- If *procedure-name-2* is specified, and is the name of a section, the return occurs after the last statement of the last paragraph in the section named by *procedure-name-2*.

No relationship is necessary between *procedure-name-1* and *procedure-name-2*, except that a consecutive sequence of operations is to be executed beginning at *procedure-name-1*, and ending with the execution of *procedure-name-2*.

If a procedure within the range of a format 2 PERFORM statement is contained in a section (or is a section) whose section number is greater than 49, the section in which it is contained is in its initial state only the first time the section is entered. Each time the section is subsequently entered as a result of the PERFORM statement, it is in its last used state. After the procedure has been executed the specified number of times, it is entered in its initial state the next time the procedure is referenced. The term *initial state* refers to the original setting of GO TO statements before they are modified at run time by the ALTER statement (refer to "ALTER Statement" in Chapter 9).

The following is an example of an out-of-line PERFORM statement:

```
PERFORM READ-INPUT WITH TEST AFTER
                   UNTIL EOF-FLAG="YES"
```

The above example performs `READ-INPUT` at least once.

In-Line PERFORM

If an in-line PERFORM statement is specified, an execution of the PERFORM statement is completed after the last statement contained within it has been executed.

When an in-line PERFORM statement is executed, control is transferred to the first statement of *imperative-statement-1*. This transfer occurs only once for each execution of a PERFORM statement.

Following is an example of an in-line PERFORM statement:

```
PERFORM
    MOVE A TO B
    ADD 1 TO B
END-PERFORM.


PERFORM VARYING I FROM 1 BY 1 UNTIL I > 4
    WRITE P-FILE  FROM HEADING(I)
END-PERFORM.
```

General Rules of PERFORM

The following rules apply to all four formats of the PERFORM statement.

- When *procedure-name-1* is specified, the PERFORM statement is referred to as an **out-of-line** PERFORM statement. When procedure-name-1 is omitted, the PERFORM statement is referred to as an in-line PERFORM statement.
- The statements contained within the range of *procedure-name-1* (through *procedure-name-2*, if specified) for an out-of-line PERFORM statement, or those statements contained within an in-line PERFORM statement itself, are referred to as the *specified set of statements*.
- If procedure-name-1 is omitted, imperative-statement-1 and the END-PERFORM phrase must be specified. If procedure-name-1 is specified, imperative-statement-1 and the END-PERFORM phrase must not be specified.

Range of the PERFORM Statement

The range of the PERFORM statement includes all statements that are executed as a result of executing the PERFORM statement through an implicit transfer of control to the end of the PERFORM statement.

The range includes all statements that are executed as the result of a transfer of control by CALL, EXIT, GO TO and PERFORM statements within the range of the PERFORM statement. The range also includes all statements in declarative procedures that are executed as a result of executing statements in the range of the PERFORM statement. It is not necessary that the statements in the range of a PERFORM statement appear in consecutive order in the source program.

Statements executed as the result of a transfer of control caused by executing an EXIT PROGRAM statement are not considered to be a part of the range of the PERFORM statement when the following is true:

- The EXIT PROGRAM statement is specified in the same program in which the PERFORM statement is specified.
- The EXIT PROGRAM statement is within the range of the PERFORM statement.

A PERFORM statement that appears in an independent program segment can have within its range, in addition to any declarative segment executed within that range, only one of the following:

- Sections or paragraphs wholly contained in one or more nonindependent segments.
- Sections or paragraphs wholly contained in the same independent segment as that PERFORM statement.

A PERFORM statement that appears in a nonindependent program segment can have within its range, in addition to the declarative sections executed within that range, only one of the following:

- Sections or paragraphs wholly contained in one or more nonindependent segments.
- Sections or paragraphs wholly contained in a single independent segment.

Nested PERFORM Statements

If a procedure or sequence of procedures contains another PERFORM statement, the procedure(s) associated with the nested PERFORM statement must be totally included in, or totally excluded from, the procedures referenced by the first PERFORM statement.

PERFORM Constructs

Figure 9-1 gives three illustrations of valid PERFORM constructs.

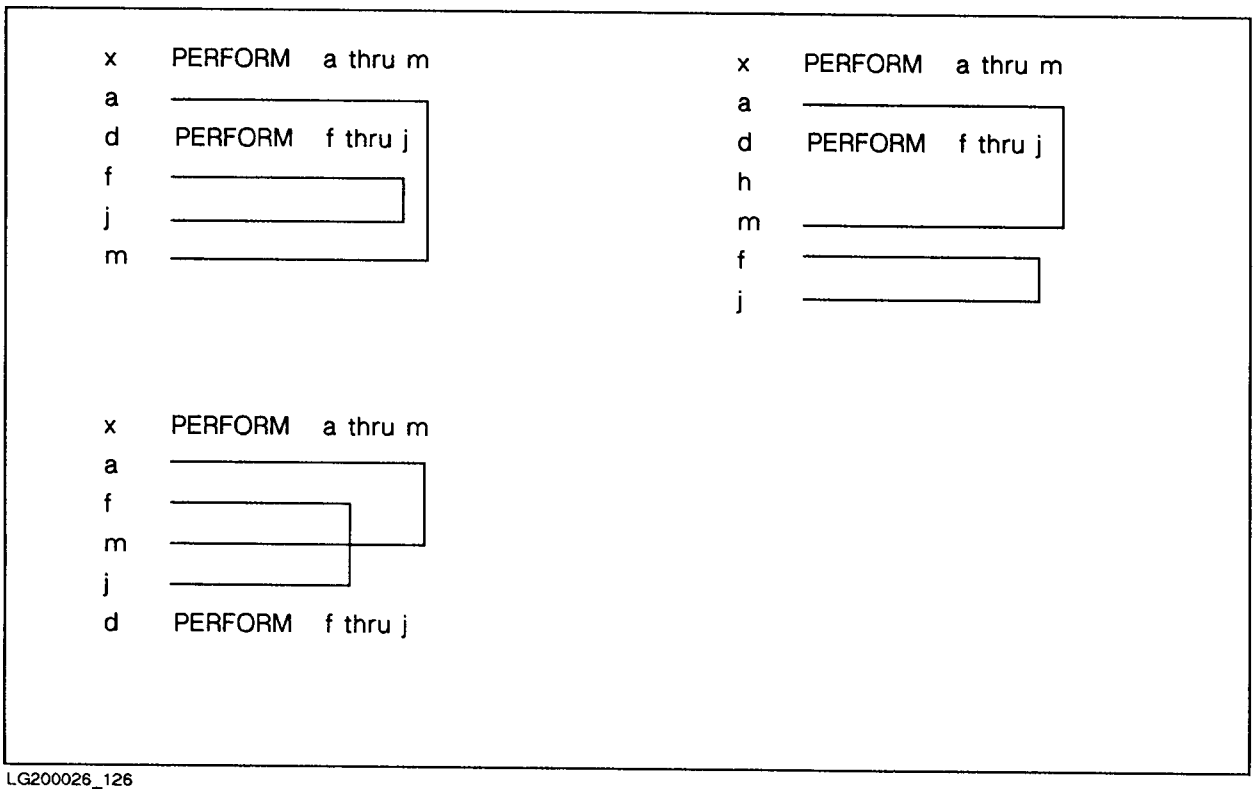


Figure 9-1. Valid PERFORM Constructs

Note Refer to "MPE XL System Dependencies" in Appendix H for more information on the PERFORM statement.

When the TEST BEFORE phrase is specified or implied:

Parameter-1 is set to the value of parameter-2 when the PERFORM statement is initially executed. If condition-1 is true, control is transferred to the next executable statement following the PERFORM statement. Otherwise, the specified set of statements is executed once.

The value of parameter-1 is then augmented by the increment or decrement value specified by parameter-3, and condition-1 is evaluated. If condition-1 is false, the cycle of executing the specified set of statements, augmenting parameter-1, and evaluating condition-1 is repeated. This cycle continues until condition-1 is true. At this time, control is passed to the first executable statement after the PERFORM statement.

The flowchart in Figure 9-2 illustrates variation of a single identifier when TEST BEFORE is specified.

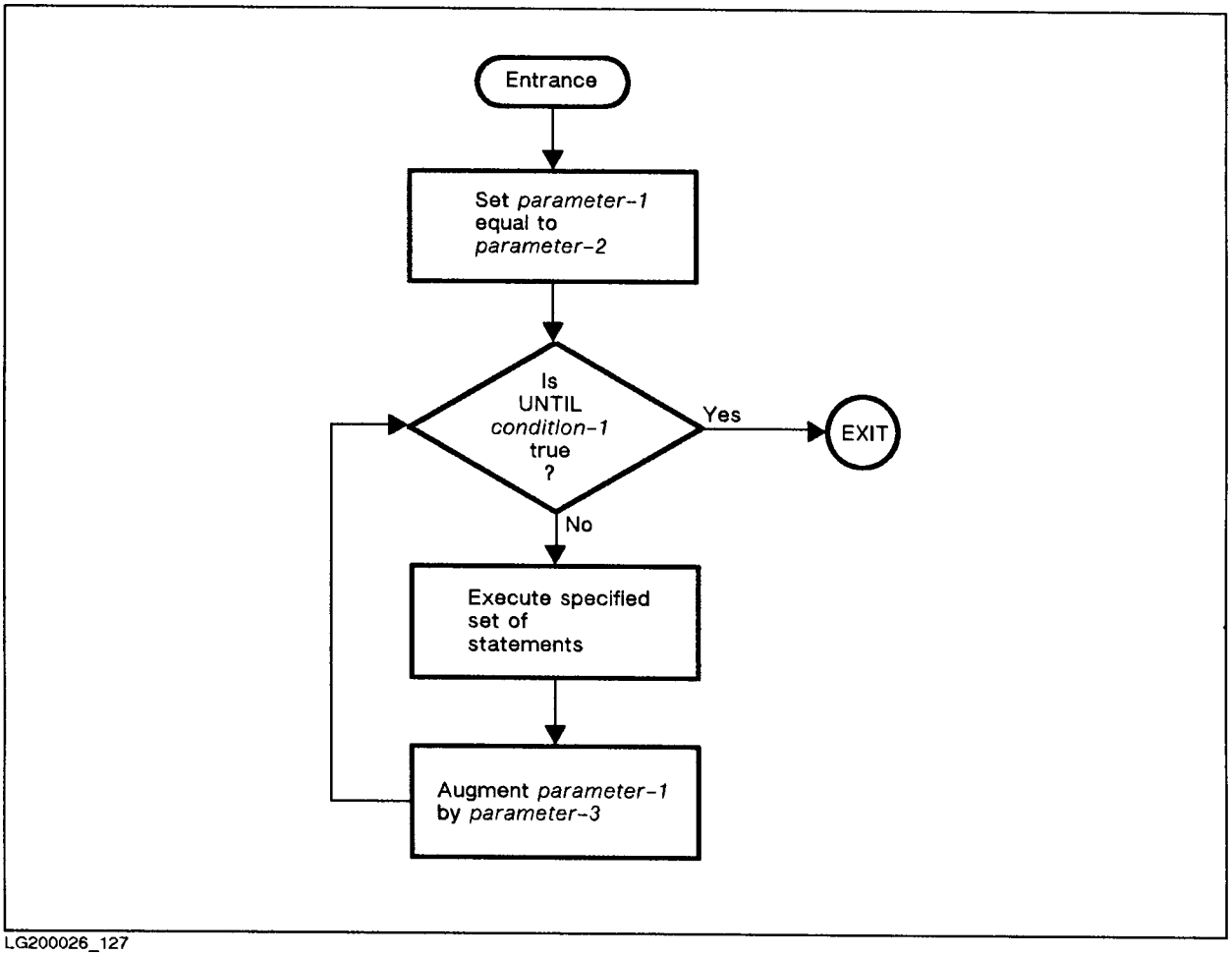
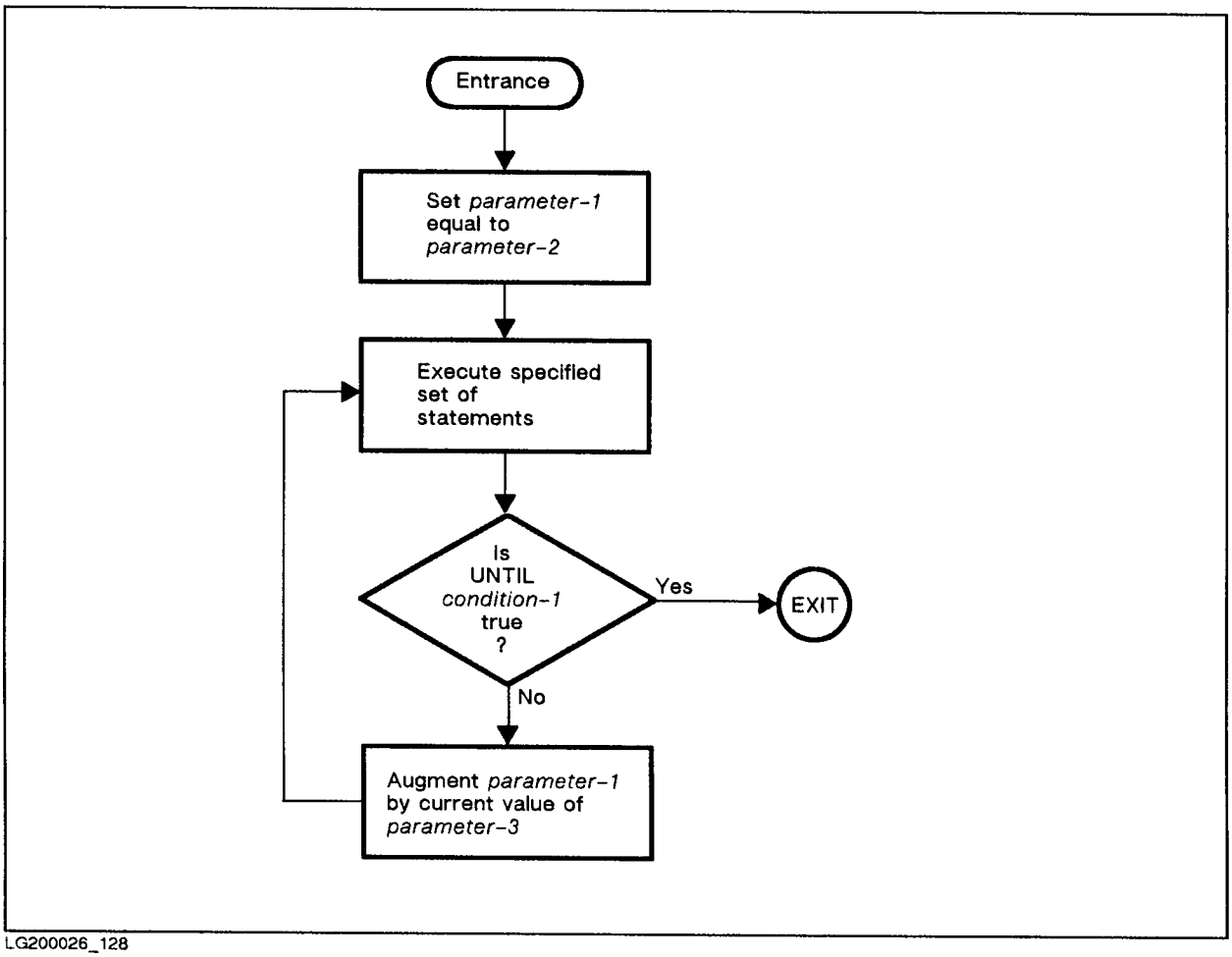


Figure 9-2. Variation of a Single Identifier with TEST BEFORE

When the TEST AFTER phrase is specified:

Parameter-1 is set to the current value of parameter-2 when the PERFORM statement is executed. Then, the specified set of statements is executed once and condition-1 is evaluated. If condition-1 is false, parameter-1 is augmented by the specified increment or decrement value of parameter-3 and the specified set of statements is executed again. The cycle continues until condition-1 is evaluated and found to be true. Control is then transferred to the end of the PERFORM statement.

The flowchart in Figure 9-3 illustrates variation of a single identifier when TEST AFTER is specified.

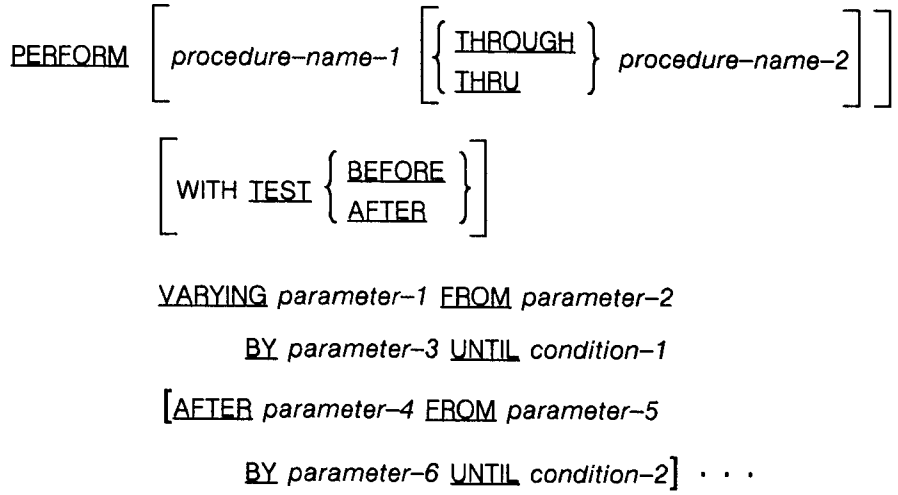


LG200026_128

Figure 9-3. Variation of a Single Identifier with TEST AFTER

Variation of Two or More Identifiers

Variation of two or more identifiers is accomplished using format 4 of the PERFORM statement in the following form:



LG200026_129

Note The format indicates that variation of two or more identifiers only applies to the out-of-line PERFORM; AFTER phrases are not included in the in-line PERFORM.

There are three cases of the PERFORM ... VARYING statement with two or more identifiers. The first case conforms to ANSI COBOL'74, and the other two cases conform to ANSI COBOL'85.

Note The latter two forms of the PERFORM ... VARYING statement are incompatible with ANSI COBOL'74. For compatibility with ANSI COBOL'74, use the ANSI74 entry point.

In this case, *parameter-1* and *parameter-4* are set to the current values of *parameter-2* and *parameter-5* respectively. Following this, *condition-1* is tested and if true, causes the PERFORM statement to cease execution. Control is transferred to the next executable statement following the PERFORM statement. If *condition-1* is false, *condition-2* is tested with the same consequences as *condition-1* if the result is true.

If *condition-2* is false, the specified procedures are executed once, *parameter-4* is augmented by *parameter-6* and *condition-2* is tested. This cycle continues until *condition-2* is true. Then *parameter-4* is set to the value of *parameter-5, parameter-1* is augmented by *parameter-3* and *condition-1* is evaluated again. If *condition-1* is true, the statement is complete. If it is false, the cycle using *parameter-4* and *condition-2* is repeated.

These two cycles are repeated until, after the cycle using *parameter-4* and *condition-2* is complete, *condition-1* is true.

During execution of the procedures, any change in the values of *parameter-1*, *parameter-2*, or *parameter-4* is taken into consideration, and affects the operation of the PERFORM statement.

At the end of this type of PERFORM statement, *parameter-4* has the current value of *parameter-5*, and *parameter-1* differs in value from its last used setting by the value of *parameter-3*. This is always true except when *condition-1* is true initially. In this case, *parameter-1* has the value of *parameter-2*.

The flowchart in Figure 9-4 illustrates PERFORM ... VARYING with two conditions.

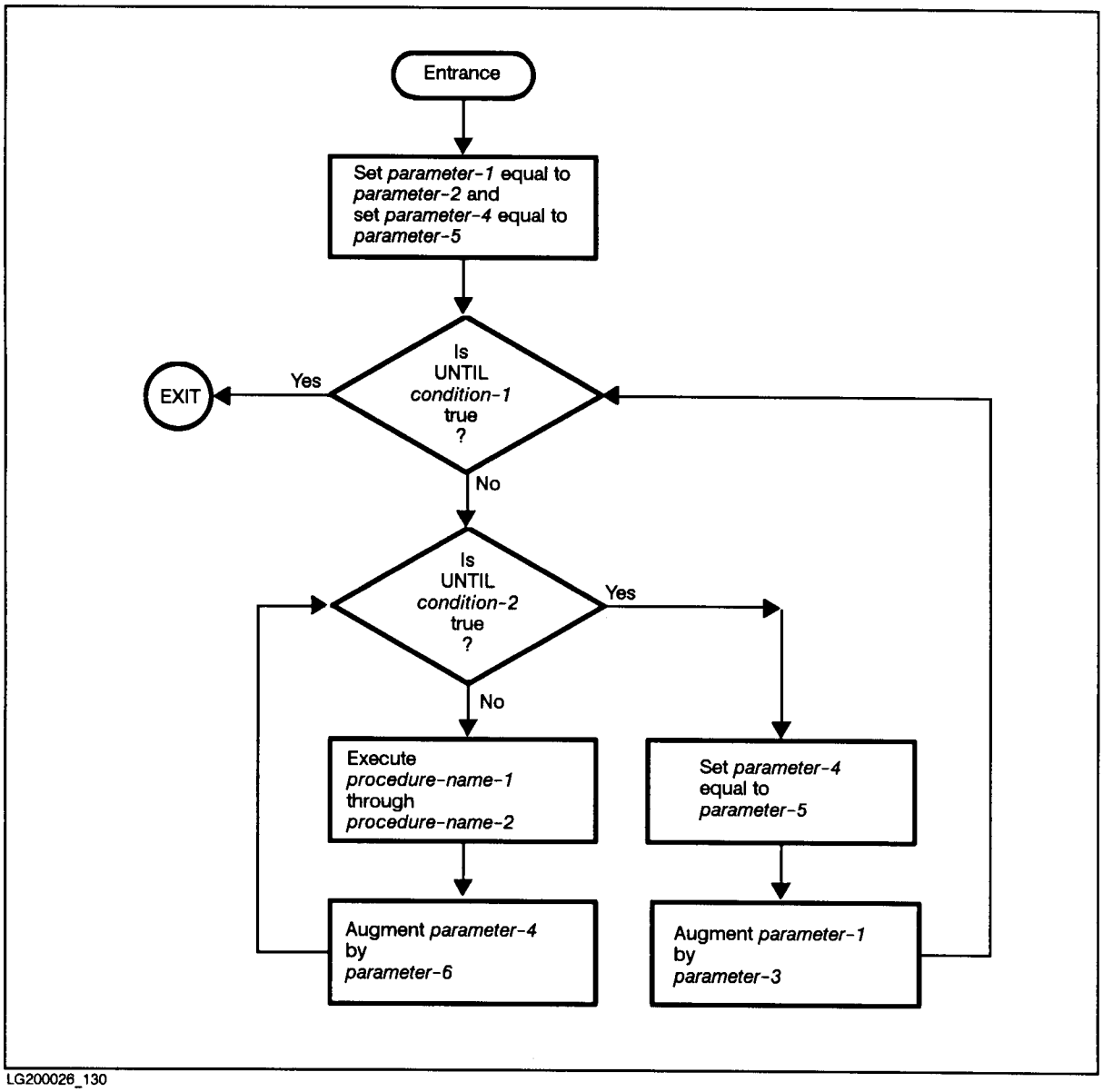


Figure 9-4. Variation of Two Conditions (ANSI COBOL'74)

When TEST BEFORE is specified:

Parameter-1 and parameter-4 are set to parameter-2 and parameter-5. Afterward, condition-1 is evaluated. If the condition is true, control is transferred to the end of the PERFORM statement; if false, condition-2 is evaluated.

If condition-2 is false, the specified set of statements is executed once, then parameter-4 is augmented by parameter-6, and condition-2 is evaluated again. This cycle of evaluation and augmentation continues until the condition is true.

When condition-2 is true, parameter-1 is augmented by parameter-3, parameter-4 is set to parameter-5, and condition-1 is reevaluated. The PERFORM statement is completed if condition-1 is true; if not, the cycle continues until condition-1 is true.

Figure 9-5 illustrates the PERFORM ... VARYING statement with the TEST BEFORE phrase having two conditions.

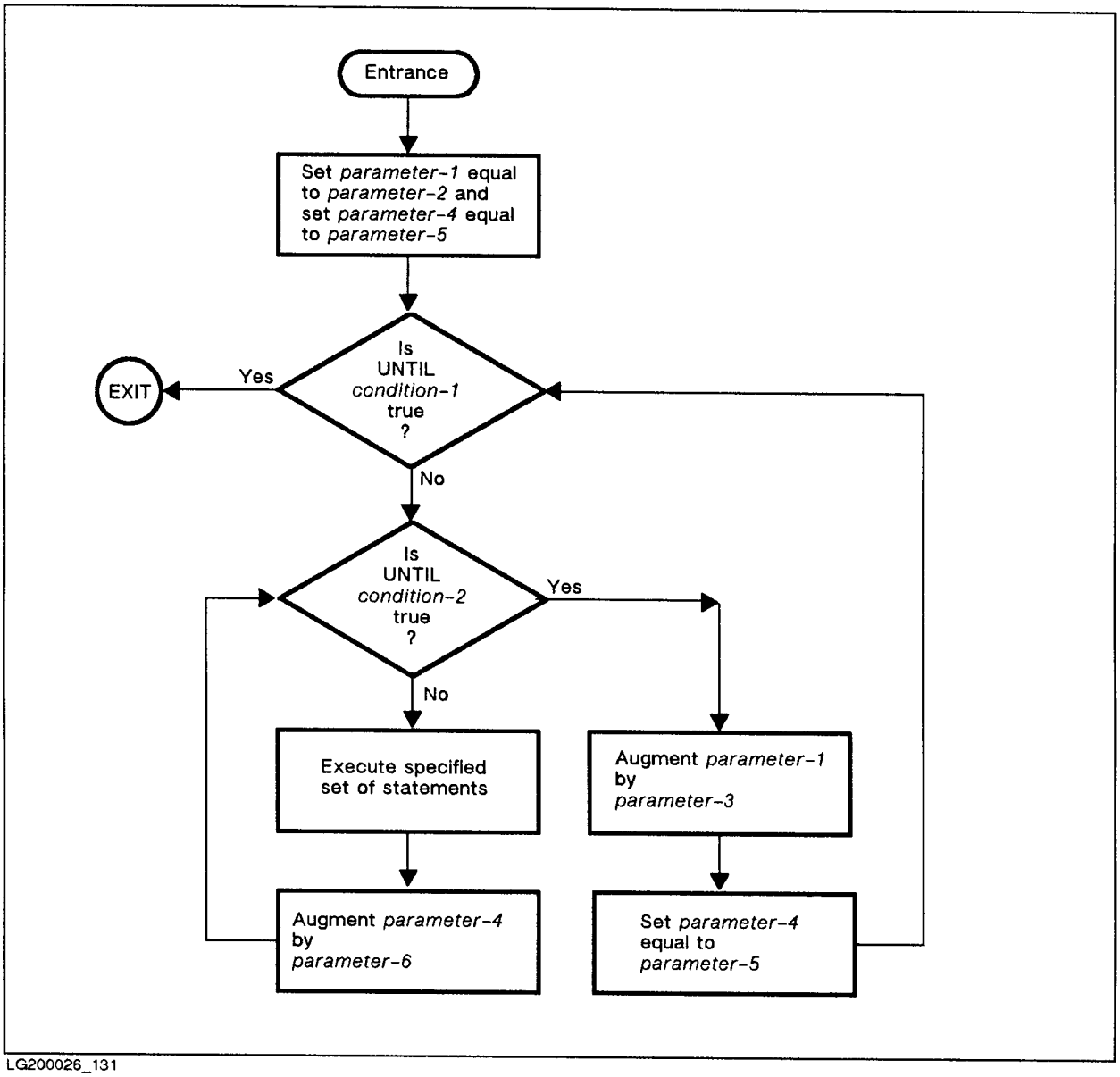


Figure 9-5. Variation of Two Conditions with TEST BEFORE (ANSI COBOL'85)

When TEST AFTER is specified:

Parameter-1 and parameter-4 are set to parameter-2 and parameter-5, and the specified set of statements is executed. Condition-2 is then evaluated; if false, parameter-4 is augmented by parameter-6, and the specified set of statements is again executed. The cycle continues until condition-2 is again evaluated and found to be true, at which time condition-1 is evaluated.

If condition-1 is false, parameter-1 is augmented by parameter-3. Also, parameter-4 is set to parameter-5, and the specified set of statements is again executed. This cycle continues until condition-1 is again evaluated and found to be true, at which time control is transferred to the end of the PERFORM statement.

After the completion of the PERFORM statement, each data item varied by an AFTER or VARYING phrase contains the same value it contained at the end of the most recent execution of the specified set of statements.

Figure 9-6 illustrates the PERFORM ... VARYING statement with the TEST AFTER phrase having two conditions.

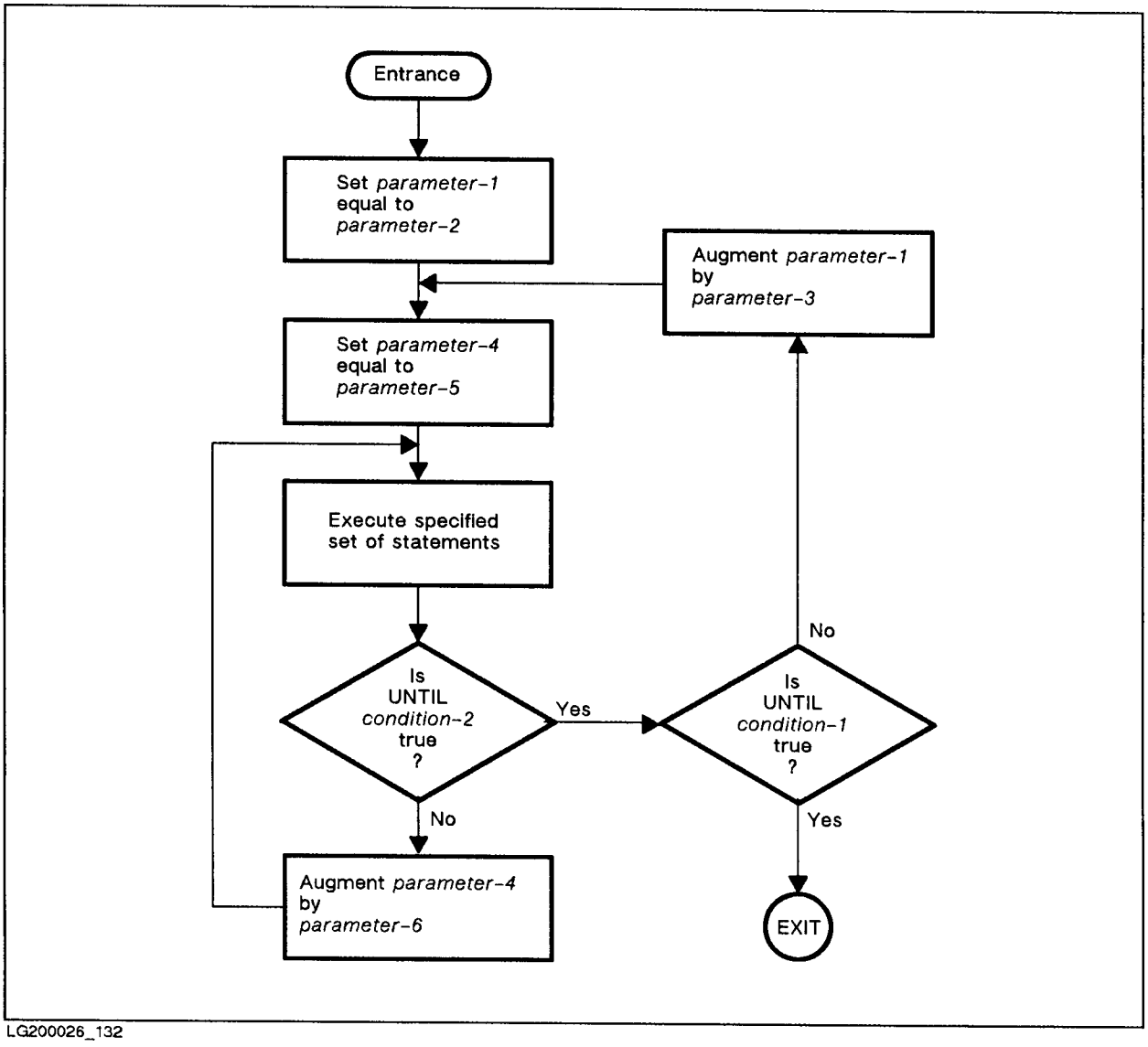


Figure 9-6. Variation of Two Conditions with TEST AFTER (ANSI COBOL'85)

Variation of More than Two Identifiers

When data items referenced by two identifiers are varied, the data item referenced by *identifier-5* goes through a complete cycle (FROM, BY, UNTIL) each time the content of the data item referenced by *identifier-2* is varied.

When the contents of three or more data items referenced by identifiers are varied, the mechanism is the same as for two identifiers. The only exception is that the data item being varied by each AFTER phrase goes through a complete cycle each time the data item being varied by the preceding AFTER phrase is augmented. Thus the last AFTER phrase varies fastest.

Incompatibility Between ANSI COBOL'74 and ANSI COBOL'85

The order of steps in a multi-conditional PERFORM ... VARYING statement has been changed. This change creates an incompatibility when there is a dependence between *identifier-2* and *identifier-5*. The following example illustrates this difference:

```
PERFORM PARA3 VARYING X FROM 1 BY 1 UNTIL X IS GREATER THAN 3
    AFTER Y FROM X BY 1 UNTIL Y IS GREATER THAN 3.
```

Under ANSI COBOL'74, PARA3 is executed eight times with the following values:

```
X:    1   1   1   2   2   2   3   3
Y:    1   2   3   1   2   3   2   3
```

Under ANSI COBOL'85, PARA3 is executed six times with the following values:

```
X:    1   1   1   2   2   3
Y:    1   2   3   2   3   3
```

One would expect the above example to perform the same as the example below:

```
    PERFORM PARA2 VARYING X FROM 1 BY 1 UNTIL X IS GREATER THAN 3.

PARA2.
    PERFORM PARA3 VARYING Y FROM X BY 1 UNTIL Y IS GREATER THAN 3.
```

Under ANSI COBOL'74, PARA3 is executed eight times, whereas under ANSI COBOL'85, PARA3 is executed six times, as shown above.

Examples

The following program fragment shows several examples of PERFORM statements.

```
01  DISKIN-RECORD.
    03  DI-DATA         OCCURS 4 TIMES.
        05  DI-NAME      PIC  X(20).
        05  DI-ADDRESS   PIC  X(20).
        05  DI-CTY-ST    PIC  X(20).
        05  DI-ZIP       PIC  9(05).
        05  DI-AMOUNT    PIC  9(03)V99 OCCURS 3 TIMES.
01  WS-FILE-CTR.
    03  WS-AMOUNT        PIC  9(06)V99 COMP VALUE 0.
    03  SUB-1            PIC  9(01).
    03  SUB-2            PIC  9(01).
    03  WS-DISKIN-CTRL   PIC  9(01)  VALUE 0.
        88  WS-DISKIN-TO-OPEN        VALUE 0.
        88  WS-DISKIN-OPEN           VALUE 1.
        88  WS-DISKIN-EOF            VALUE 2.
PROCEDURE DIVISION.
200-PROGRAM-START.

** EXAMPLE OF PERFORM USING THRU OPTION ******************
    PERFORM 300-GET-DISKIN-RTN THRU 300-GET-DISKIN-EXIT.

** EXAMPLE OF PERFORM USING UNTIL OPTION *****************
     PERFORM 250-PROCESS-RECORD UNTIL WS-DISKIN-EOF.
     STOP RUN.
250-PROCESS-RECORD.

** EXAMPLE OF PERFORM USING VARYING & UNTIL OPTIONS ******
    PERFORM 260-UNBLOCK-RECORD
       VARYING SUB-1 FROM 1 BY 1
           UNTIL SUB-1 GREATER 5.
260-UNBLOCK-RECORD.
    MOVE DI-NAME (SUB-1)    TO P-NAME.
    MOVE DI-ADDRESS (SUB-1) TO P-ADDRESS.
    MOVE DI-CTY-ST (SUB-1)  TO P-CTY-ST.
    MOVE ZERO               TO SUB-2.
```

```
** EXAMPLE OF PERFORM USING # OF TIMES OPTION ************
     PERFORM 270-ACCUMULATE-AMOUNTS 3 TIMES.
 270-ACCUMULATE-AMOUNTS.
     ADD 1                      TO SUB-2.
     ADD DI-AMOUNT (SUB-1,SUB-2)TO WS-AMOUNT.
 300-GET-DISKIN-RTN.
    IF  WS-DISKIN-TO-OPEN
        OPEN INPUT DISKIN-FILE
        MOVE 1 TO WS-DISKIN-CTRL.
    READ DISKIN-FILE
        AT END
        CLOSE DISKIN-FILE
        MOVE 2 TO WS-DISKIN-CTRL.
 300-GET-DISKIN-EXIT.
     EXIT.
```

READ Statement

The READ statement makes a record of a file available to your program.

Syntax

READ has three formats depending on the type of organization of the file from which a record is made available.

Format 1 - Sequential, Relative, Random, and Indexed Files

```
READ file-name-1 [NEXT] RECORD [INTO identifier-1 ]
     [AT END imperative-statement-1]
     [NOT AT END imperative-statement-2]
     [END-READ]
```

Format 2 - Relative and Random Files

```
READ file-name-1 RECORD [INTO identifier-1 ]
     [INVALID KEY imperative-statement-1]
     [NOT INVALID KEY imperative-statement-2]
     [END-READ]
```

Format 3 - Indexed FIles

```
READ file-name-1 RECORD [INTO identifier-1 ]
     [KEY IS data-name-1 ]
     [INVALID KEY imperative-statement-1]
     [NOT INVALID KEY imperative-statement-2]
     [END-READ]
```

LG200026_133

Parameters

file-name-1	name of the file to be read.
identifier-1	data item described in the WORKING-STORAGE or FILE SECTION.
imperative-statement-1	one or more imperative statements, executed when an INVALID KEY or AT END condition occurs.
`imperative-statement-2`	`one or more imperative statements, executed when a NOT INVALID KEY or NOT AT END condition occurs.`
data-name-1	name of a data item, as specified by either the RECORD KEY or ALTERNATE RECORD KEY clause, of the associated file. It may be qualified.

Description

The following rules apply to any of the formats of the READ statement. Each format is described separately following these common rules.

When a READ statement is executed for a file, the file must be open in the INPUT or I-O mode.

The execution of a READ statement causes the FILE-STATUS data item (if specified) of the file being read to be updated. Refer to Chapter 6, under "FILE STATUS Clause", for valid status keys. For more information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

When the logical records of a file are described with more than one record description, these records automatically share the same storage area. This is equivalent to an implicit redefinition of the area. Those character positions of the current data record not filled in by the READ statement contain data items that are undefined at completion of the execution of the READ statement.

If the INTO phrase is specified, the input file must not contain logical records whose sizes vary according to their record descriptions. Also, the storage area associated with *identifier-1*, and the record area associated with the file being read must be distinct from one another. That is, these areas must not be the same storage areas.

When the INTO phrase is used, the record being read is placed into the input record area of the file, and the data item contained in the input record area is copied into the storage area of *identifier-1* according to the rules of the MOVE statement (without the CORRESPONDING phrase). This implied MOVE does not occur if the execution of the READ statement is unsuccessful.

Any subscripting or indexing associated with *identifier-1* is evaluated after the input record has been read, and immediately before it is moved into the storage area of *identifier-1*.

If no exception exists, control is transferred to the end of the READ statement, or to *imperative-statement-2*, if specified.

READ Statement - Format 1

A format 1 READ statement may be used for sequential, relative, random, or indexed files. The READ statement must be used for relative or indexed files whose access mode is sequential (see the SELECT clause). It must also be used, including the NEXT phrase, when the access mode for a relative or indexed file is dynamic and records of the file are being accessed sequentially and to read records sequentially from a random access file.

Note The NEXT phrase is optional for files whose access mode is sequential.

For a sequential, relative, or indexed file being accessed sequentially, the record to be made available by a format 1 READ statement is determined as follows:

- The record pointed to by the file position indicator is made available provided that the file position indicator was positioned by a SEEK, START, or OPEN statement, and the record is still accessible through the path indicated by the file position indicator. If the record is no longer available, the file position indicator is updated to point to the next existing record (within the established key of reference for indexed files) and that record is then made available.

Note A record may not be available because it was never written (for random access files), it was deleted (for relative files), or because an alternate record key has been changed (for indexed files).

- If the file position indicator was positioned by the execution of a previous READ statement, the file position indicator is updated to point to the next existing record (with the established key of reference for indexed files) and that record is made available.

A format 1 READ statement for a random access file, or for a relative or indexed file whose access mode is dynamic, uses the NEXT phrase to position the file position indicator to the next logical record of the file. The record made available is then determined as for files opened in sequential access mode. If no next logical record exists for the file and a READ statement attempts to execute for that file, the READ statement is unsuccessful and an AT END condition occurs. The steps taken by the program in such a situation are essentially the same as for a sequential file.

That is,

1. The value of any FILE STATUS data item specified for the file is changed to indicate an AT END condition.
2. If an AT END phrase is specified in the READ statement, control is transferred to the imperative statement following the AT END keywords and any USE procedure specified for the file is ignored.

If no AT END phrase is specified in the READ statement, a USE procedure must be specified either implicitly or explicitly for the file, and that procedure is executed.

If no USE procedure is defined for the file, the AT END phrase must be used in the READ statement.

Following the unsuccessful execution of any READ statement, the contents of the associated record area and the position of the file position indicator are undefined. For indexed files, the key of reference is also undefined.

When the AT END condition exists, a format 1 READ statement for the file must not be executed without first executing one of the following:

- A successful CLOSE statement followed by a successful OPEN statement for that file;
- A successful START for that file (if it is relative or indexed);
- A successful format 2 READ statement for a relative or random access file, or a format 3 READ statement for an indexed file.

In the case of unlabeled magnetic tapes, the AT END condition indicates an EOF mark was read. Subsequent READ statements cause reading of the next file on the tape. Since this may cause reading off the end of the tape, the exact number of files on the tape must be known.

If the end of a labeled magnetic tape reel is found during the execution of a READ statement, and the logical end of the file is not encountered, a reel swap is performed, and the first record of the new reel is made available. Execution of a format 1 READ statement may be unsuccessful for one of three reasons:

- The position of the file position indicator is undefined;
- No next logical record exists in the file (that is, the end-of-file has been encountered).
- The OPTIONAL phrase was used in the SELECT statement for the file, and the file was not present at the time the file was opened.

If the RELATIVE KEY phrase is specified in the SELECT clause for a relative file, successful execution of a format 1 READ statement updates the contents of the RELATIVE KEY data item so that it contains the relative record number of the record made available.

For a random access file, the ACTUAL KEY data item is updated to contain the relative record number of the record just read.

For an indexed file being sequentially accessed, records having the same duplicate value in an alternate record key being used for the key of reference are made available in the same order in which they are released by WRITE statements, or by execution of REWRITE statements that create duplicate values.

READ Statement - Format 2

Format 2 of the READ statement can be used for relative files whose access mode is random, or whose access mode is dynamic when records are to be retrieved randomly. It can also be used for random access files.

When a format 2 READ statement is executed for a relative file, the current record pointer is set to the record whose relative record number is contained in the data item named in the RELATIVE KEY phrase for the file (see the SELECT clause). This record is then made available.

If the file does not contain such a record, an INVALID KEY condition exists, and the READ statement is unsuccessful.

When an INVALID KEY condition exists, two actions are performed. First, the data item specified by the FILE STATUS clause, if used, is updated to reflect the condition.

Next, if there is a USE procedure for the file, and an INVALID KEY condition exists but the INVALID KEY phrase was not specified in the READ statement, the procedure named in the USE statement is executed.

If there is no USE procedure specified for the file, the INVALID KEY phrase must be used in the READ statement; when an INVALID KEY condition exists, control is passed to *imperative-statement-1* in the INVALID KEY phrase.

When a format 2 READ statement is executed for a random access file, it is equivalent to executing a SEEK statement for the file, followed by the READ statement.

The contents of the data name specified in the ACTUAL KEY clause of the SELECT statement are used to set the file position indicator to the record to be read. The record is then made available unless an INVALID KEY condition exists.

An INVALID KEY condition exists for a random access file if the contents of the ACTUAL KEY data item do not point to a record within the file. When this occurs, the imperative statement in the INVALID KEY phrase is executed.

READ Statement - Format 3

Format 3 of the READ statement is used for indexed files whose access mode is random, or is dynamic and records are to be retrieved randomly.

If the KEY phrase is specified, *data-name-1*, as specified by either the RECORD KEY or ALTERNATE RECORD KEY clause, is established as the key of reference for this retrieval. Also, if the access mode of the file is dynamic, this key of reference is used for retrievals by any subsequent executions of format 1 READ statements for the file until a different key of reference is established for the file.

If the KEY phrase is not specified, the prime record key, as specified by the RECORD KEY clause of the SELECT statement, is established as the key of reference for retrieval. It acts in the same manner as an alternate key when format 2 READ statements are subsequently issued.

Execution of a format 3 READ statement causes the value of the key of reference to be compared with the value contained in the corresponding data item of the stored records in the file.

The records used in the comparison are selected according to the ascending values of their keys, and not by the order in which the records were written (called "chronological order").

The comparison continues either until the first record having the same value is found, or until no such value is found.

If a value is found that matches the key of reference, the file position indicator is positioned to the record containing the matched value and that record is made available to your program.

If the comparison fails, an INVALID KEY condition exists and execution of the READ statement is unsuccessful.

When an INVALID KEY condition occurs and no INVALID KEY phrase is specified, a USE procedure must be specified, and is executed.

When an INVALID KEY condition occurs and the INVALID KEY phrase is specified, control is transferred to the imperative statement appearing in the INVALID KEY phrase; any USE procedure that was specified for the file is ignored.

RELEASE Statement

The RELEASE statement can be used in an input procedure of a SORT statement to transfer records from your program to the initial phase of the sort operation. For more information on the RELEASE statement refer to Chapter 12, "SORT/MERGE Operations".

RETURN Statement

The RETURN statement can be used in an output procedure of a SORT or MERGE statement. It cannot be used in any other type of procedure. For more information on the RETURN statement refer to Chapter 12, "SORT/MERGE Operations".

REWRITE Statement

The REWRITE statement logically replaces an existing record in a sequential, relative, random, or indexed mass storage file. The file position indicator is unaffected by the execution of a REWRITE statement. Variable length records are not allowed when using the REWRITE statement.

Syntax

There are two formats of the REWRITE statement:

Format 1 – Sequential Files

```
REWRITE record-name-1 [FROM identifier-1]
[END-REWRITE]
```

Format 2 – Relative, Random, and Indexed Files

```
REWRITE record-name-1 [FROM identifier-1]
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
    [END-REWRITE]
```


LG200026_134a

Parameters

record-name-1	name of a logical record in the FILE SECTION of the DATA DIVISION. *Record-name-1* can be qualified. It cannot refer to the same storage area as that referred to by *identifier-1*.
identifier-1	name of a data item in the program or a function-identifier. When *identifier-1* is not a function, it may be described in any section of the DATA DIVISION, but must not refer to the same storage area as *record-name-1*. If *identifier-1* is a function, it must be an alphanumeric function.
imperative-statement-1	one or more imperative statements.

Description

The file associated with *record-name-1* must be a mass storage file, and must be opened in I-O mode when the REWRITE statement is executed.

The number of character positions in the record referenced by *record-name-1* must be equal to the number of character positions in the record being replaced.

For sequential files, and for relative or indexed files open in sequential access mode (see the SELECT statement), the last input-output statement executed for the associated file prior to the execution of the REWRITE statement must have been a READ statement. The record replaced by the REWRITE statement is the record accessed by the READ.

For indexed files, this is accomplished by using the primary key. Thus, the value contained in the primary record key data item of the record to be replaced must be equal to the value of the primary record key of the last record read from the file. If an indexed file has the DUPLICATES phrase specified for its primary record key, the REWRITE statement should be used only when the indexed file is in sequential access mode. This is because a REWRITE statement issued for such a file whose access mode is dynamic or random will only rewrite the first record having the duplicate primary key.

If the primary record key data item of the record to be replaced is not equal to the value of the primary record key of the last record read from the file, an INVALID KEY condition exists. In such a case, the REWRITE operation fails and the record that was to be replaced is unaffected. Also, if the INVALID KEY phrase is specified, control is passed to *imperative-statement-1* of that phrase, whether a USE procedure s specified for the file or not. If a USE procedure is not specified for the file, the INVALID KEY phrase must be specified. However, if a USE procedure is specified and an INVALID KEY phrase is not, the USE procedure is executed when an INVALID KEY condition exists for the file.

If a relative file is open in sequential access mode, the INVALID KEY phrase must not be used.

If a random file is open, or a relative file is open in random or dynamic access mode, the record to be logically replaced is specified by the contents of the RELATIVE KEY or ACTUAL KEY data item associated with the file. If the file does not contain the record specified by the key, an INVALID KEY condition exists. Thus, the operation does not succeed and the data in the record area is unaffected. Also, if no USE procedure has been defined for the file, the INVALID KEY phrase must be specified, and when an INVALID KEY condition exists, control is transferred to *imperative-statement-1* of that phrase. If a USE procedure has been defined and the INVALID KEY phrase is not specified, the USE procedure is executed when an INVALID KEY condition exists. However, if both a USE procedure and an INVALID KEY phrase are specified, the USE procedure is ignored and control is transferred to *imperative-statement-1* of the INVALID KEY phrase.

For indexed files open in dynamic or random access mode, the record to be replaced is specified by the primary record key data item. An INVALID KEY condition exists for this type of REWRITE if the value of the primary record key in the record to be rewritten does not equal that of any record stored in the file. An INVALID KEY condition also exists for this type of REWRITE when the value contained in an alternate record key data item equals the value of an alternate record key of another record and the DUPLICATES clause has not been specified for that key in the SELECT statement for that file. The action taken for the occurrence of an INVALID KEY condition when an indexed file is open in random or dynamic access is the same as for when the indexed file is open for sequential access.

For more information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

When an indexed file is the object of a REWRITE statement, the contents of alternate record key data items of the record being rewritten may differ from those of the record being replaced. These alternate keys are used during the execution of the REWRITE statement in such a way that subsequent access of the record may be made based upon any of the specified record keys.

The logical record written by a successful REWRITE statement is no longer available in the record area (memory) unless the associated file (whether indexed, relative or sequential) is named in the SAME RECORD AREA clause.

If the file is named in the SAME RECORD AREA clause, the written logical record is available to your program as a record of other files named in the clause, as well as to the file associated with the record to be replaced (that is, it remains in memory).

When a REWRITE statement completes execution, whether successfully or not, the value of the FILE STATUS data item, if any, associated with the file being accessed in the REWRITE statement is updated. If no exception exists, control is transferred to the end of the REWRITE statement, `or to the imperative statement of the NOT INVALID KEY phrase, if specified.` Refer to Chapter 6, "FILE STATUS Clause", for valid combinations of status keys 1 and 2.

FROM Phrase

If the FROM phrase is used in a REWRITE statement, execution of the statement is equivalent to the execution of the following MOVE statement, followed by the execution of the same REWRITE statement without the FROM phrase:

`MOVE` *identifier-1* `TO` *record-name-1*

The contents of the record area prior to the execution of the implicit MOVE have no effect upon the execution of the REWRITE statement. The REWRITE statement executed following the implicit MOVE follows the rules and restrictions listed above.

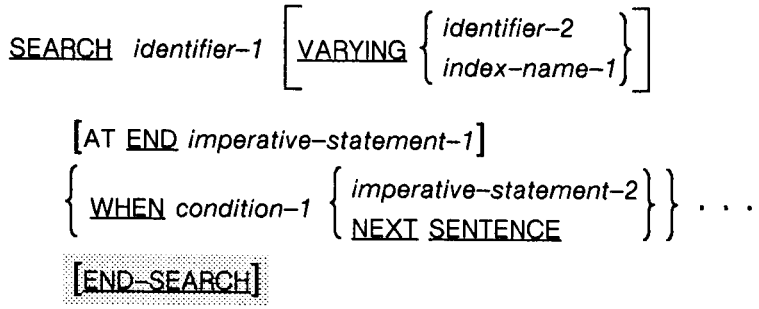
SEARCH Statement

The SEARCH statement searches a table for an element that satisfies some condition, and sets the table's index name to the value of the occurrence number of the element that was found.

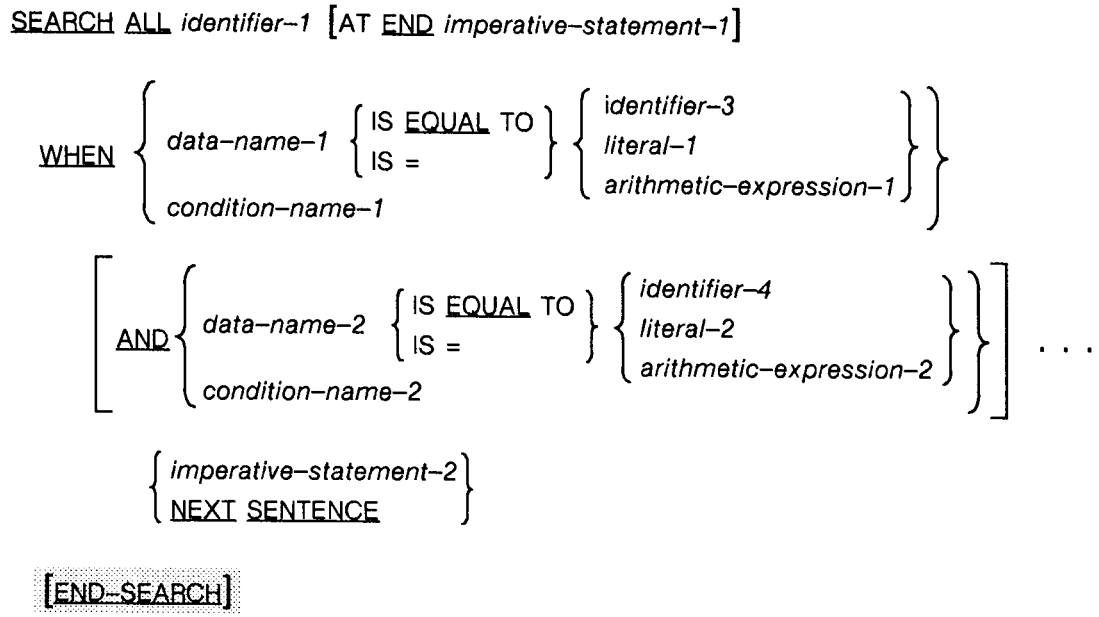
Syntax

The two formats of the SEARCH statement are shown below:

Format 1



Format 2



LG200026_136

Parameters

identifier-1	name of a table. It must not be subscripted or indexed, and must also contain an OCCURS and an INDEXED BY clause in its description. If used in format 2, it must also contain a KEY IS phrase in its OCCURS clause.
identifier-2	if used, must be described either as USAGE IS INDEX, or as a numeric elementary item with no positions to the right of the assumed decimal point.
condition-1 and *condition-2*	may be any condition as described under the heading, "Conditional Expressions", in Chapter 8.
condition-name-1 and *condition-name-2*	condition names whose descriptions (level 88) list only a single value. The data name associated with a condition name must appear in the KEY IS clause of *identifier-1*.
data-name-1 and *data-name-2*	bmay each be qualified, and each must be indexed by the first index name associated with *identifier-1*, as well as any other indices or literals, as required. Each must also be referenced in the KEY IS clause of *identifier-1*.
identifier-3 and *identifier-4*	must not be referenced in the KEY IS clause of *identifier-1* or be indexed by the first index name associated with *identifier-1*.
arithmetic-expression-1 and *arithmetic-expression-2*	can be any arithmetic expression as described under the heading, "Arithmetic Expressions", in Chapter 8. However, any identifiers appearing in any arithmetic expression in a SEARCH statement are subject to the same restrictions as *identifier-3* and *identifier-4*.
imperative-statement-1 and *imperative-statement-2*	each one or more imperative statements.

Description

If *identifier-1* is a data item subordinate to a data item containing an OCCURS clause (that is, a two or three dimensional table), an index name must be associated with each dimension of the table represented by *identifier-1*. This is done through the INDEXED BY phrase of the OCCURS clause for *identifier-1*. Only the setting of the index name associated with *identifier-1* (and the data item named by *identifier-2* or *index-name-1*, if used) is modified by the execution of the SEARCH statement.

To search an entire two or three dimensional table, you must execute a SEARCH statement several times. You must also use SET statements to adjust index names to appropriate settings.

SEARCH Statement - Format 1

When you use a format 1 SEARCH statement, a serial search is performed, starting with the current index setting of the index name associated with *identifier-1*.

If the index name associated with *identifier-1* contains a value corresponding to an occurrence number greater than the highest possible occurrence number of *identifier-1*, the SEARCH statement is terminated immediately. If an AT END phrase is specified, the *imperative-statement-1* within the phrase is executed. If an AT END phrase is not specified, control passes to the next executable statement following the SEARCH statement.

If the index name associated with *identifier-1* contains a value corresponding to an occurrence number within the range of *identifier-1*, the SEARCH statement evaluates each condition in the order that it is written. If none of the conditions is satisfied, the index name for *identifier-1* is incremented to reference the next occurrence. The process of evaluating each condition is then repeated using the new index name settings, provided the value of the index name is within the permissible range of occurrences for *identifier-1*. If the new setting is not within range, the search terminates in the manner described in the preceding paragraph.

If one of the conditions is satisfied when it is evaluated, the search terminates and the *imperative-statement-1* associated with that condition is executed. The index name retains the value at which it was set when the condition was satisfied.

After execution of *imperative-statement-1*, if it is not a GO TO statement, control passes to the next executable sentence.

VARYING Phrase

If you use the VARYING phrase, *index-name-1* may or may not appear in the INDEXED BY phrase of *identifier-1*. If it does, *index-name-1* is the index name used in the search. If it does not, or if you specified *identifier-2* instead, the first (or only) index name given in the INDEXED BY phrase of *identifier-1* is used for the search.

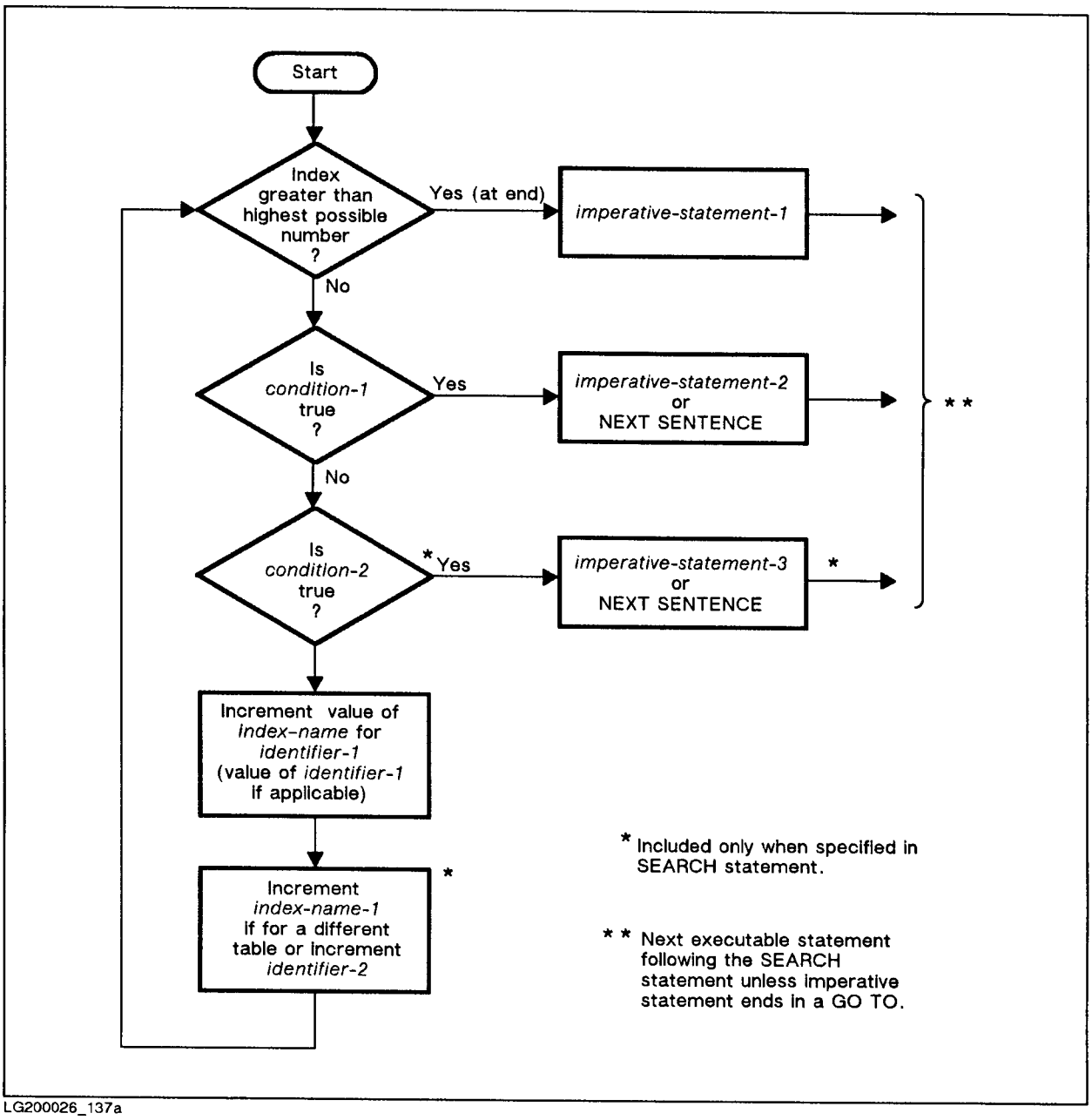
If you specify VARYING *index-name-1*, and *index-name-1* is associated with a different table (that is, does not appear in the INDEXED BY phrase of *identifier-1*), the occurrence number represented by *index-name-1* is incremented by the same amount as, and at the same time as, the occurrence number represented by the index name associated with *identifier-1*.

If you specify VARYING *identifier-2*, and *identifier-2* is an index data item, the data item represented by *identifier-2* is incremented in the same way as for *index-name-1* in the preceding paragraph.

If *identifier-2* is not an index data item, the data item referenced by *identifier-2* is incremented by one (1) at the same time that the index referenced by the index name associated with *identifier-1* is incremented.

If you do not specify 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 *identifier-1*. Any other index name for *identifier-1* remains unchanged.

The flowchart in Figure 9-7 shows the execution of a format 1 SEARCH statement specifying two WHEN phrases.



LG200026_137a

Figure 9-7. Execution of Format 1 SEARCH Statement

SEARCH Statement - Format 2

If you use a format 2 SEARCH statement, a binary search is done.

The index name used in the search is the first (or only) index name appearing in the INDEXED BY phrase of *identifier-1*. Any other index names for *identifier-1* remain unchanged.

The results of a format 2 SEARCH statement are predictable only under three conditions:

1. The data in the table is ordered in the same manner as described in the ASCENDING (or DESCENDING) KEY clause associated with the description of *identifier-1*.
2. The contents of the key or keys referenced in the WHEN clause are sufficient to identify a unique table element.
3. All entries in the table contain valid data. In other words, the table is filled properly depending on how it is defined. You can use the OCCURS DEPENDING ON clause to control the size of the table and make sure all elements contain valid data. Alternatively, if the table uses the ASCENDING KEY clause, any unused alphanumeric entries in the table should be at the end of the table and filled with HIGH-VALUES. If the table uses the DESCENDING KEY clause, any unused alphanumeric entries in the table should be at the end of the table and filled with LOW-VALUES.

When a data name in the KEY clause of *identifier-1* is referenced, or when a condition name associated with a data name in the KEY clause of *identifier-1* is referenced, all preceding data names in the KEY clause of *identifier-1* or their associated condition names must also be referenced. The maximum number of reference keys is 12.

When the search begins, the initial setting of the index name associated with *identifier-1* is ignored and its setting is varied during the search in such a way as to make the search as fast as possible. Of course, it is not set to a value outside of the possible range of occurrence numbers for the table.

If there is no possible valid setting for the index name associated with *identifier-1* that satisfies all the conditions specified in the WHEN clause, control passes to the imperative statement specified in the AT END phrase if it has been specified. If the AT END phrase is not specified, and the conditions cannot all be satisfied, control passes to the next executable sentence. In either case, the final setting of the index is not predictable.

If all conditions in the WHEN phrase can be satisfied, the index indicates the occurrence that allows the conditions to be satisfied, and control passes to *imperative-statement-2*.

After execution of *imperative-statement-2*, providing that *imperative-statement-2* does not contain a GO TO statement, control passes to the next executable sentence.

The example below uses both formats of the SEARCH statement. In the format 1 SEARCH statement, a search is performed to find out if a carton is part of the inventory. If it is part of the inventory, its first dimensions (there are five possible) as well as its part number are displayed.

The second SEARCH statement is a format 2 SEARCH. It is used to find a container of a given volume whose height and width both equal 5.

Example

```
WORKING-STORAGE SECTION.
  01 PARTS-TABLE.
     02  PARTS-INFO OCCURS 10 TIMES INDEXED BY PT-INDX.
         03  PART-NAME          PIC X(20).
         03  PART-NUMBER        PIC X(10).
         03  MEASURES OCCURS 5 TIMES
             ASCENDING KEY IS VOLUME, HEIGHT, WIDTH
             INDEXED BY IND-T2.
             04  HEIGHT     PIC 999V999.
             04  WIDTH      PIC 999V999.
             04  LGTH       PIC 999V999.
             04  VOLUME     PIC 999V999.
  77 NEXT-NUM               PIC 99 VALUE 1.
  01 CONTAINER-INFO.
     02  PT-NAME            PIC X(20).
     02  FILLER             PIC X(5) VALUE SPACES.
     02  PT-NO              PIC X(10).
     02  FILLER             PIC X(3) VALUE SPACES.
     02  DIMENSIONS.
         04 H               PIC 999V999.
         04 W               PIC 999V999.
         04 L               PIC 999V999.
         04 V               PIC 999V999.
            .
            .
            .
PROCEDURE DIVISION.
SEARCH-PARTS-INFO.
    SET IND-T2 PT-INDX TO 1.
    SEARCH PARTS-INFO
      AT END
        PERFORM UNFOUND-RTN
      WHEN PART-NAME (PT-INDX) = 'CARTON'
        PERFORM FOUND-IT.
            .
            .
            .
FOUND-IT.
    MOVE MEASURES (PT-INDX, IND-T2) TO DIMENSIONS.
    MOVE PART-NAME (PT-INDX) TO PT-NAME
    MOVE PART-NUMBER (PT-INDX) TO PT-NO
    DISPLAY HEADER.
    DISPLAY CONTAINER-INFO.
    DISPLAY SPACES.
         .
         .
         .
```

```
   CONTAINER-SELECTION.
      DISPLAY "THIS SECTION SEARCHES FOR A CONTAINER".
      DISPLAY "OF A SPECIFIED VOLUME WHOSE HEIGHT AND".
      DISPLAY "WIDTH BOTH ARE FIVE.  PLEASE SPECIFY THE VOLUME".
      DISPLAY "REQUIRED (IN AN EVEN NUMBER OF CUBIC FEET.)".
      ACCEPT NEEDED-VOLUME.
   SEARCH-PARTS-TABLE.
      MOVE 1 TO NEXT-NUM.
      SET PT-INDX IND-T2 TO 1.
      SEARCH ALL MEASURES
         AT END
            PERFORM NO-SUCH-CONTAINER.
         WHEN VOLUME (PT-INDX, IND-T2) = NEEDED-VOLUME
            AND WIDTH (PT-INDX, IND-T2) = 5
            AND HEIGHT (PT-INDX, IND-T2) = 5
            PERFORM FOUND-IT.
               .
               .
               .
   NO-SUCH-CONTAINER.
      IF NEXT-NUM < 10
        ADD 1 TO NEXT-NUM
        SET PT-INDX TO NEXT-NUM
        SET IND-T2 TO 1
        PERFORM SEARCH-PARTS-TABLE
      ELSE
        DISPLAY "NO CONTAINER MEETS THESE REQUIREMENTS".
```

SEEK Statement

The SEEK statement is an HP extension to the ANSI COBOL standard. It is provided for COBOL'68 compatibility.

The SEEK statement initiates access to a relative file whose access mode is dynamic or random, and to a random access file, prior to execution of a format 2 READ statement.

Syntax

SEEK *file-name* `RECORD`

Description

The SEEK statement is valid only for input files.

The file specified in a SEEK statement must be opened prior to the first SEEK statement.

The SEEK statement causes a transfer of the physical record containing the logical record to be read in a subsequent READ statement from storage into main memory.

Using the SEEK statement before a format 2 READ statement may improve the performance of your program. However, because the SEEK function is implicit in the READ statement, its use is not mandatory in this case.

Two SEEK statements may logically follow each other. However, it is poor programming practice to do this because of the time expended on input-output operations for data that will not be accessed.

For relative files, the SEEK statement uses the value of the data item named in the RELATIVE KEY clause of the file SELECT statement to find the desired record. Thus, before a SEEK statement is executed for a relative file, the relative record number of the desired record must be moved to the RELATIVE KEY data item.

Similarly, for random access files, the SEEK statement uses the value of the data item named in the ACTUAL KEY clause of the file's SELECT statement to find the desired record.

Thus, you must move the ACTUAL KEY value for the desired record to the data item named by the ACTUAL KEY clause before executing a SEEK statement for the file.

In either case, if the value moved to the data item named in a RELATIVE KEY or ACTUAL KEY clause is invalid, the SEEK statement is ignored. If a subsequent format 2 READ statement is executed for the file, an INVALID KEY condition exists, and the appropriate action is taken.

SET Statement

The SET statement establishes reference points for table handling operations by setting index names associated with table elements. This statement can also be used to alter the status of external switches, and to alter the value of conditional variables.

Syntax

The formats of the SET statement are shown below:

Format 1

```
SET { index-name-1  } ... TO { index-name-2  }
    { identifier-1  }        { identifier-2  }
                             { integer-1     }
```

Format 2

```
SET { index-name-3 } ... { UP BY   } { identifier-3 }
                         { DOWN BY } { integer-2    }
```

Format 3

```
SET { { mnemonic-name-1 } ... TO { ON  } } ...
                                 { OFF }
```

Format 4

```
SET { condition-name-1 } ... TO TRUE
```

LG200026_138a

Parameters

identifier-1, *identifier-2*	must each name either index data items or elementary items described as integers.
identifier-3	must be described as an elementary numeric integer.
integer-1 and *integer-2*	may be signed, with the restriction that *integer-1* must be positive.
index-name-1, *index-name-2*, and so forth	each related to a given table.
mnemonic-name-1	must be associated with a software switch, whose status can be altered. This name is associated with a software switch in the SPECIAL-NAMES paragraph in the ENVIRONMENT DIVISION.
condition-name-1	must be associated with a conditional variable.

Note The value of the index associated with an index name may be undefined following the execution of a SEARCH or PERFORM statement in which it is used. Also, when a sending and a receiving item share part of their storage areas, the result of execution of a SET statement using them is undefined.

SET Statement - Format 1

If you use *index-name-2*, the value of the associated index before execution of the SET statement must correspond to an occurrence number of an element in the table whose index is being set.

Index-name-1, if used, is set to a value causing it to refer to the table element whose occurrence number corresponds to the occurrence number of the table element referenced by *index-name-2*, *identifier-2*, or *integer-1*.

The value to which *index-name-1* is set must correspond to an occurrence number of an element in the table to which *index-name-1* is associated.

If *identifier-2* is an index data item, or if *index-name-2* is related to the same table as *index-name-1*, no conversion takes place.

If *identifier-1* is an index data item, it may be set equal to the contents of either *index-name-2* or *identifier-2* if *identifier-2* is an index data item. No conversion takes place. *Identifier-1* may not be set equal to *integer-1* in this case.

If *identifier-1* is not an index data item, only *index-name-2* may be used. The value to which *identifier-1* is set is, in this case, an occurrence number that corresponds to the value of *index-name-2*.

Any indexing or subscripting associated with *identifier-1* is evaluated immediately before the value of the indexed or subscripted data item is changed.

Any remaining identifiers or index names are set in the same way, with the same restrictions as *identifier-1* or *index-name-1*. If *index-name-2* or *identifier-2* has been specified, its associated value is used as it was at the beginning of the execution of the SET statement.

Table 9-6 shows the validity of various operand combinations in the SET statement.

Table 9-6.
Validity of Different Combinations of Operands in the SET Statement

Sending Item	Receiving Item		
	Integer Data Item	**Index Name**	**Index Data Item**
Integer Literal	Not allowed	Valid	Not allowed
Integer Data Item	Not allowed	Valid	Not allowed
Index Name	Valid	Valid[1]	Valid[2]
Index Data Item	Not allowed	Valid[2]	Valid[2]

1 No conversion takes place if row sizes are equal.

2 No conversion takes place.

SET Statement - Format 2

When format 2 is used, the value of the index associated with an index name to be set must correspond, before and after execution of the SET statement, to an occurrence number of an element in the associated table.

The value of *index-name-3* is incremented (if UP BY is used) or decremented (if DOWN BY is used) by the value of *integer-2* or the integer named by *identifier-3*.

If other index names are specified, each is set up or down, as specified, just as the first was. The value of *identifier-3* is unchanged and is used as it was at the beginning of the execution of the SET statement.

SET Statement - Format 3

In format 3, the status of each external software switch associated with mnemonic-name-1 is modified such that the truth value resultant from evaluation of a condition name associated with that switch reflects an “on status” if the ON phrase is specified, or an “off status” if the OFF phrase is specified.

SET Statement - Format 4

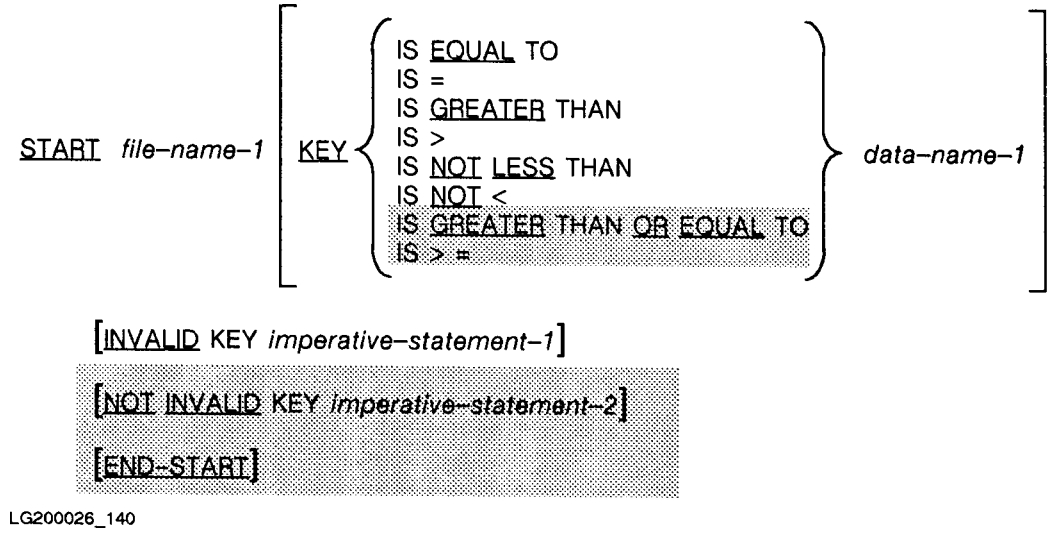
In format 4, the literal in the VALUE clause associated with condition-name-1 is placed in the conditional variable according to the rules of the VALUE clause. If more than one literal is specified in the VALUE clause, the conditional variable is set to the value of the first literal that appears in the VALUE clause.

If multiple condition names are specified, the results are the same as if a separate SET statement had been written for each condition-name-1 in the same order as specified in the SET statement.

START Statement

The START statement provides a basis for logical positioning within a relative or indexed file, in sequential or dynamic access mode, for subsequent retrieval of records.

Syntax



Note The required relational characters '>' , '<' , '>=' , and '=' are not underlined to avoid confusion with other symbols such as '≥' (greater than or equal to).

Parameters

file-name-1 — the name of a relative or indexed file. The file must be open in INPUT or I-O mode when the START statement is executed.

data-name-1 — for a relative file, must be the data item named in the RELATIVE KEY phrase of the SELECT statement for the file. For an indexed file, *data-name-1* may reference a data item specified as a record key associated with the named index file, or it may reference any alphanumeric data item subordinate to the data name of a data item specified as a record key of the named file, provided that its leftmost character position corresponds to the leftmost character position of that record key data item. *Data-name-1* may be qualified.

imperative-statement-1 and *imperative-statement-2* — one or more imperative statements. The INVALID KEY phrase must be used in the START statement if no applicable USE procedure is specified for the file.

Description

When the START statement executes, a comparison is made between a key associated with *file-name-1*, and a data item. If the KEY phrase is unused, the relational operator, "IS EQUAL TO" is assumed. The data item used in the comparison depends upon whether the file named in the START statement is a relative or an indexed file.

If the file is a relative file, the comparison uses the data item referenced in the RELATIVE KEY clause of the file's SELECT statement. This data item is always used, whether the KEY phrase is specified or not. So, for example, the following statements cause the file position indicator to be positioned at the fifth record of `REL-FILE` if `REL-KEY` is the name specified in the RELATIVE KEY phrase of the SELECT statement for the relative file `REL-FILE`:

```
MOVE 5 TO REL-KEY.
START REL-FILE.
```

If the file named in the START statement is an indexed file, the data item used in the comparison depends upon whether the KEY phrase is used.

If the KEY phrase is not used for an indexed file, the primary key, that is, the data item named in the RECORD KEY clause of the file, is used.

If the KEY phrase is used, the comparison uses the data item referenced by *data-name-1*. This data item must be either a primary or alternate key for the file, or must be a data item whose first character is the first character of one of the keys for the file.

If the key associated with a record of an indexed file differs from the size of the data item used in the comparison, the comparison proceeds as though the longer of the two were truncated on the right so that its length is equal to the length of the shorter. A nonnumeric comparison is then performed following all the rules for such comparisons. The PROGRAM COLLATING SEQUENCE, however, is not used for the comparison, even if it was specified. The ASCII collating sequence is always used on an HP computer system.

When comparison takes place for either type of file, the file position indicator is positioned to the first logical record currently existing in the file whose key satisfies the comparison.

If the comparison is not satisfied by any record of the file, an INVALID KEY condition exists, the execution of the START statement is unsuccessful, and the *imperative-statement* of the INVALID KEY phrase (if specified) is executed. If the INVALID KEY phrase is not specified, then a USE procedure must be specified and that procedure is executed. In such a case, the position of the file position indicator is undefined.

The execution of a START statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated. Refer to Chapter 6, under "FILE STATUS Clause", for valid combinations of status keys 1 and 2. For more information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

Upon completion of a successful START statement for an indexed file, a key of reference is established and used in subsequent format 1 READ statements.

If the KEY phrase is not specified in an indexed file START statement, the primary key is established as the key of reference. If the KEY phrase is specified, and *data-name-1* is any record key (primary or alternate) for the file, that record key becomes the key of reference.

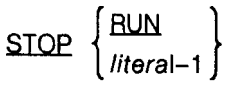
If the KEY phrase is specified, and *data-name-1* is not a record key of the file, then the first character of the data item contained in *data-name-1* is the same as the first character of some key for the file, and that key becomes the key of reference.

If a START statement for an indexed file is unsuccessful, then the key of reference and the file position indicator are undefined.

STOP Statement

The STOP statement provides a means of temporarily suspending execution of your object program, as well as a means of stopping it completely.

Syntax



LG200026_141

Parameters

`RUN` if specified, causes the entire run-unit to cease execution when it is encountered, regardless of whether the STOP RUN statement is in a subprogram or a main program. Control is then returned to the operating system.

literal-1 may be numeric, nonnumeric, or any figurative constant except ALL. If the literal is numeric, it must be an integer. Use of a literal in a STOP statement temporarily suspends the object program.

The literal variation of the STOP statement is an obsolete feature of the 1985 ANSI COBOL standard.

Description

If the STOP RUN statement is used in a consecutive sequence of imperative statements within a sentence, it must appear as the last statement in this sequence.

If STOP literal is used, the object program suspends and the literal is displayed at the operator's console. A system generated message is then displayed, followed by the message TYPE GO TO RESUME.

When the operator responds with "GO", program execution resumes with the next executable statement following the STOP literal statement. For example:

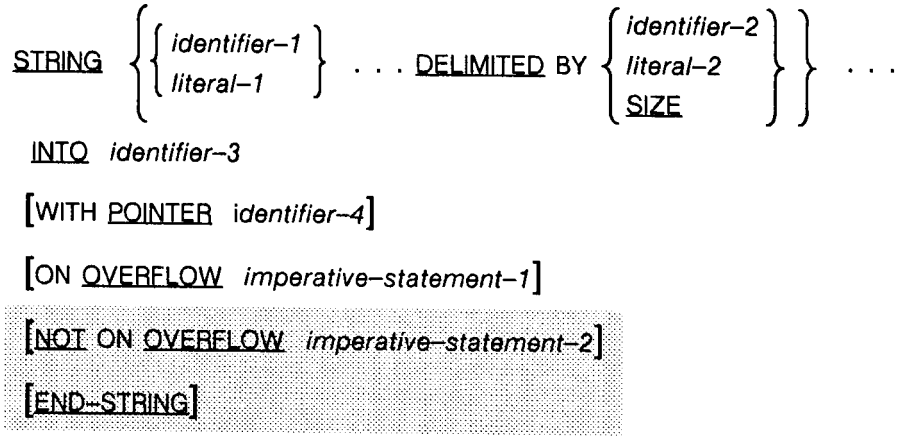
```
STOP "MOUNT TAPE COBTEST"
OPEN INPUT COBTEST.
```

The STOP statement above is to instruct the operator to mount a magnetic tape to be used as an input file for the program. After the operator mounts the tape and makes appropriate console responses, the program resumes execution when the word "GO" is typed in.

STRING Statement

The STRING statement concatenates the partial or complete contents of two or more data items into a single data item.

Syntax



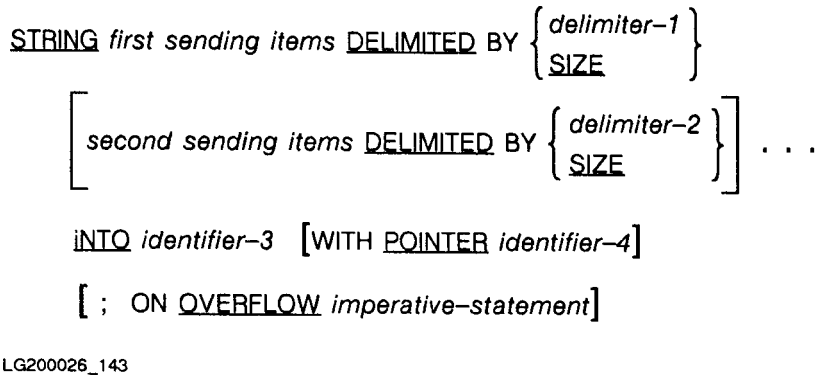
LG200026_142

Parameters

literal-1 and *literal-2*	any figurative constant or any nonnumeric literal except ALL; none of them may be numeric literals.
identifier-1 and *identifier-2*	described implicitly or explicitly as USAGE DISPLAY. If any of these identifiers represent an elementary numeric data item, it must be described as an integer without the 'P' symbol in its PICTURE character string.
identifier-3	must represent an alphanumeric data item without editing symbols or JUSTIFIED clause, and must have a USAGE IS DISPLAY (implied or explicit) in its description. It may not be reference modified.
identifier-4	must represent an elementary numeric integer data item of sufficient size to contain a value equal to the size, plus 1, of the area referenced by *identifier-3*. The symbol P must not be used in the PICTURE of *identifier-4*.
imperative-statement-1 and *imperative-statement-2*	one or more imperative statements.

Description

All references to *identifier-1*, *identifier-2*, *identifier-3*, *literal-1*, and *literal-2* apply equally to *identifier-4*, respectively, and all recursions thereof. Thus, to aid in the description of the STRING statement, the format is rewritten as follows:



First sending items and second sending items represent the groups of literals and the data items named by the identifiers appearing between the STRING and DELIMITED keywords, or between a delimiter (or the keyword SIZE) and the next use of the DELIMITED keyword. These are the items that are juxtaposed into *identifier-3*, the receiving data item.

Delimiter-1 and *delimiter-2* indicate the character or characters delimiting the characters moved from first sending items and second sending items, respectively. If the SIZE phrase is used, the complete group of sending items is moved.

If a figurative constant is used as a delimiter, it stands for single character numeric literal, whose USAGE is DISPLAY. If a figurative constant is used for a literal in a group of sending items, it refers to an implicit one character data item whose USAGE is DISPLAY.

Execution of the STRING Statement

When the STRING statement executes, the characters in the first sending items are transferred to the contents of *identifier-3* in accordance with the rules of alphanumeric to alphanumeric moves, except that no space filling takes place.

If the DELIMITED phrase is specified using *delimiter-1*, the contents of the first sending items are moved to the contents of *identifier-3* in the sequence specified in the STRING statement, starting with the leftmost character and continuing until all character positions of *identifier-3* have been filled, or until the character or characters that make up *delimiter-1* are encountered. The characters of *delimiter-1* are not transferred.

If *delimiter-1* is found in the first sending items before *identifier-3* is filled, the second sending items are processed in the same way as the first and transferring ceases in the same way, using *delimiter-2* rather than *delimiter-1*.

If the DELIMITED phrase contains the word SIZE rather than *delimiter-1* or *delimiter-2*, then either all characters in the sending items are transferred to *identifier-3*, or as many characters as possible are transferred before the end of the data area reserved for *identifier-3* has been reached.

The POINTER phrase is available for you to define the starting position of *identifier-3* to which data is to be moved. For example, if the phrase WITH POINTER COUNT is used and the value of COUNT is 10, the first character transferred from the sending items is placed in the tenth character position (from the left) of *identifier-3*. Not using the POINTER phrase is equivalent to specifying WITH POINTER 1.

After a character is moved into the data item referenced by *identifier-3*, the pointer value is incremented by one. Thus, the value of *identifier-4* at the end of a STRING statement is equal to its initial value, plus the number of characters transferred.

If the value of *identifier-4* is less than one, or exceeds the number of character positions in the data item referenced by *identifier-3*, and execution of the STRING statement is not complete, an overflow condition occurs. At this point, regardless of whether or not any data has already been moved to the data item referenced by *identifier-3*, no more data is moved.

Furthermore, if the ON OVERFLOW statement is specified in the STRING statement, the imperative statement in the phrase is executed.

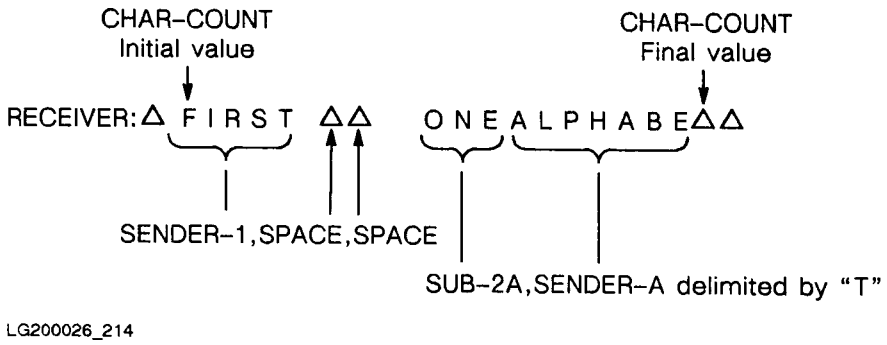
If the ON OVERFLOW phrase is not specified when an overflow condition is encountered, control is transferred to the end of the STRING statement, `or to the end of the NOT ON OVERFLOW phrase, if specified.`

At the end of execution of the STRING statement, only the portion of the data item referenced by *identifier-3* that was referenced during the execution of the STRING statement is changed. All other portions of the data item contain the data that was present before this execution of the STRING statement.

Examples

```
WORKING-STORAGE SECTION.
01 RECEIVER          PIC X(20)  VALUE SPACES.
01 SENDER-1          PIC X(5)   VALUE "FIRST".
01 SENDER-2.
   02  SUB-2A        PIC A(3)   VALUE "ONE".
   02  SUB-2B        PIC 99V99  VALUE ZERO.
01 SENDER-A          PIC X(15)  VALUE "ALPHABETICALLY".
77 CHAR-COUNT        PIC 99     VALUE 1.
77 LIMITER           PIC X      VALUE "T".
     .
     .
     .
PROCEDURE DIVISION.
     .
     .
     .
     ADD 1 TO CHAR-COUNT.
     STRING SENDER-1, SPACE, SPACE DELIMITED BY SIZE,
         SUB-2A, SENDER-A DELIMITED BY LIMITER
         INTO RECEIVER WITH POINTER CHAR-COUNT
         ON OVERFLOW DISPLAY "OVERFLOW IN RECEIVER",
         " VALUE OF COUNTER IS ", CHAR-COUNT.
              .
              .
              .
```

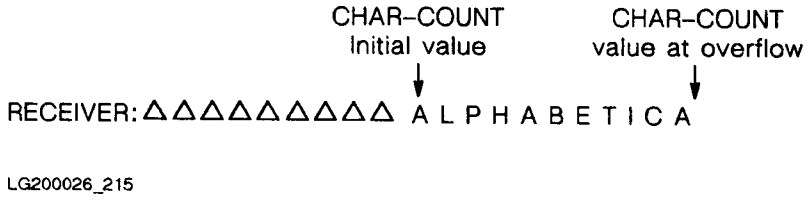
With the definitions of data names as described in the WORKING-STORAGE SECTION, and with CHAR-COUNT set to 2, the STRING statement fills RECEIVER as follows:



If the three statements below are used instead, an overflow condition is caused.

```
MOVE SPACES TO RECEIVER.
MOVE 10 TO CHAR-COUNT
STRING SENDER-A DELIMITED BY SIZE
      INTO RECEIVER WITH POINTER CHAR-COUNT
      ON OVERFLOW DISPLAY "OVERFLOW IN RECEIVER"
                  DISPLAY "VALUE OF COUNTER IS ", CHAR-COUNT.
```

The STRING statement now fills RECEIVER as follows:



When this overflow occurs, the following message is sent to the terminal from which the program was initiated:

```
OVERFLOW IN RECEIVER
VALUE OF COUNTER IS 21
```

SUBTRACT Statement

The SUBTRACT statement subtracts one or more numeric data items from one or more numeric data items and stores the result in one or more data items.

Syntax

The SUBTRACT statement has three formats:

Format 1

```
SUBTRACT { identifier-1 } ... FROM { identifier-3 [ROUNDED] } ...
         { literal-1    }

  [ON SIZE ERROR imperative-statement-1]
  [NOT ON SIZE ERROR imperative-statement-2]
  [END-SUBTRACT]
```

Format 2

```
SUBTRACT { identifier-1 } ... FROM { identifier-2 }
         { literal-1    }          { literal-2    }

  GIVING { identifier-3 [ROUNDED] } ...

  [ON SIZE ERROR imperative-statement-1]
  [NOT ON SIZE ERROR imperative-statement-2]
  [END-SUBTRACT]
```

Format 3

```
SUBTRACT { CORRESPONDING } identifier-1 FROM identifier-2 [ROUNDED]
         { CORR          }

  [ON SIZE ERROR imperative-statement-1]
  [NOT ON SIZE ERROR imperative-statement-2]
  [END-SUBTRACT]
```

LG200026_144

Parameters

identifier-1, *identifier-2*, and so forth	elementary numeric data items, or if to the right of the keyword GIVING, may be elementary numeric-edited data items. The exception is in format 3, where *identifier-1* and *identifier-2* must be group items.
literal-1, *literal-2*, and so forth	numeric literals.
CORR	abbreviation for CORRESPONDING.

Description

The compiler always ensures that enough places are carried in order to avoid losing significant digits.

The ROUNDED, SIZE ERROR, NOT ON SIZE ERROR, and CORRESPONDING phrases, as well as rules applying to multiple results and overlapping operands are described in Chapter 8.

The composite of operands, a hypothetical data item obtained by the superimposition of data items, must not exceed 18 digits. This composite is determined for each format as follows:

Format 1: Composite is determined by using all of the operands in a given statement.

Format 2: Composite is determined using all of the operands in a given statement except those following the GIVING phrase.

Format 3: Composite is determined using each pair of corresponding data items separately.

When format 1 of the SUBTRACT statement is used, all literals or values of identifiers preceding the FROM phrase are added together, and the resulting sum is subtracted from the value of each identifier specified in the FROM phrase. As each subtraction is completed, the result is stored in the identifier of the FROM phrase used as operand.

When format 2 is used, all literals or values of identifiers preceding the FROM phrase are added together, and then subtracted from *literal-2* or the value of *identifier-3*. The result of this subtraction is then stored in each identifier following the GIVING keyword.

When format 3 is used, data items in *identifier-1* are subtracted from and stored in corresponding data items of *identifier-2*.

Examples

```
SUBTRACT FIRST-YR, SECOND-YR FROM THIRD-YR.

SUBTRACT HIRE-DATE FROM AGE GIVING YEARS-OF-SERVICE.
```

In the first example above, the value of FIRST-YR is added to the value of SECOND-YR, and this sum is subtracted from, and stored in THIRD-YR.

In the second example, the value of AGE is subtracted from the value of HIRE-DATE, and the results are stored in YEARS-OF-SERVICE.

In the example below, QTY-1, QTY-2, and QTY-3 of PARTS-OUT are subtracted from QTY-1, QTY-2, and QTY-3 of CURRENT-PARTS, and the results are stored in QTY-1, QTY-2, and QTY-3 of CURRENT-PARTS.

```
FILE SECTION.
FD INVFILE.
01 PARTS-INV.
       03  PARTS-OUT.
           04  PARTS-NUM-1          PIC X(10).
           04  QUANTITY             PIC 9(6).
           04  SUB-PARTS-OUT.
               05  QTY-1            PIC 9(6).
               05  QTY-2            PIC 9(6).
               05  QTY-3            PIC 9(6).
        03  CURRENT-PARTS.
           04  PART-NUM-1           PIC X(10).
           04  QUANTITY             PIC 9(6).
           04  CURRENT-SUB-PARTS.
               05  QTY-1            PIC 9(5).
               05  QTY-2            PIC 9(5).
               05  QTY-3            PIC 9(5).
                    ⋮
PROCEDURE DIVISION.
         ⋮
     SUBTRACT CORRESPONDING SUB-PARTS-OUT FROM CURRENT-SUB-PARTS.
```

UN-EXCLUSIVE Statement

The UN-EXCLUSIVE statement is an HP extension to the ANSI COBOL standard.

The UN-EXCLUSIVE statement releases a file that has been previously locked by the EXCLUSIVE statement.

Syntax

UN-EXCLUSIVE *file-name-1*

Parameter

file-name-1 is the name of a file that has been locked using the EXCLUSIVE statement.

Description

It is not necessary to unlock a locked file before closing it. An implicit UN-EXCLUSIVE statement is performed when you close the file. However, if a user issues an unconditional EXCLUSIVE statement naming the file that you have locked, that user's program suspends execution until the file is available to be locked. Thus, you should use the UN-EXCLUSIVE statement to unlock the file as soon as the program has finished accessing it.

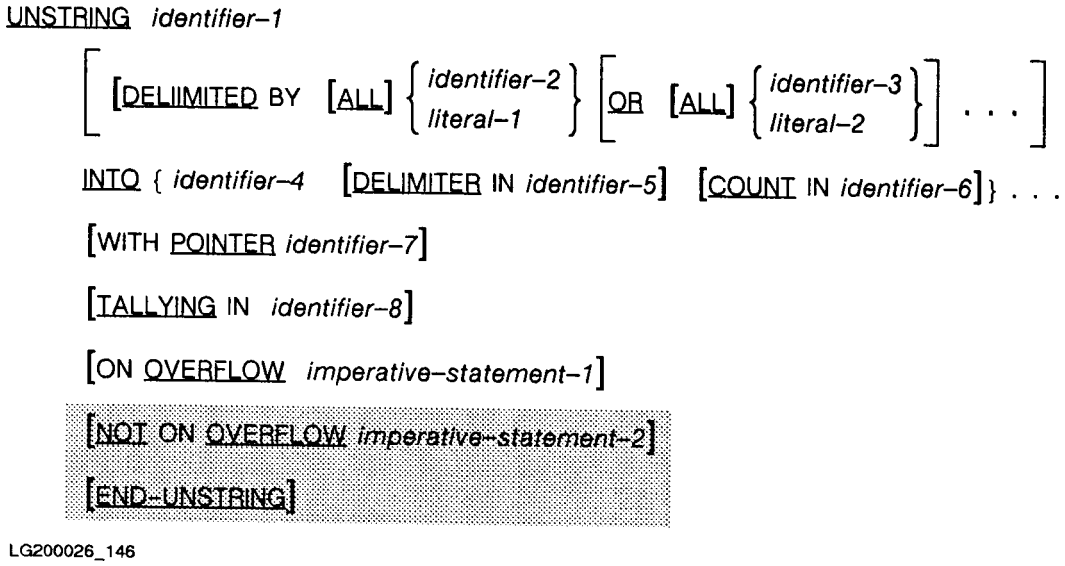
If the unlock is successful or the file is not locked, the STATUS-KEYs are set to "00". If the file options do not specify dynamic locking or the file number is invalid, STATUS-KEY-1 is set to "9" and STATUS-KEY-2 contains a binary error code.

A USE procedure may be specified for the file being unlocked. See "USE Statement" later in this chapter and "Declarative Sections" in Chapter 8 for more details on USE procedures. Also, for more information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

UNSTRING Statement

The UNSTRING statement divides data in a sending field and places the segments of the data into multiple receiving fields.

Syntax



LG200026_146

Parameters

literal-1 and *literal-2*	nonnumeric literals; may also be any figurative constants without the optional word ALL.
identifier-1, *identifier-2*, *identifier-3*, and *identifier-5*,	must be described implicitly or explicitly as alphanumeric data items. *Identifier-1* may not be reference modified.
identifier-4	may be described as alphabetic, alphanumeric, or numeric data items.
identifier-7	must be described as an elementary numeric integer data item of sufficient size to contain a value equal to 1 plus the size of the data item referenced by *identifier-1*. The symbol 'P' may not be used in the PICTURE character string of either *identifier-4* or *identifier-7*.

Note Edited receiving fields are not permitted.

identifier-6 and *identifier-8*	must be described as elementary numeric integer data items. The P symbol may not be used in their descriptions.

Description

No identifier may name a level 88 entry.

If the DELIMITED BY phrase is not specified, the DELIMITER IN and COUNT IN phrases must not be used.

Identifier-1 represents the sending item.

Identifier-4 represents the receiving data item.

Identifier-2 or its associated literal, and *identifier-3* or its associated literal, represent delimiters on the sending item. If a figurative constant is used as a delimiter, it stands for a single character nonnumeric literal.

Identifier-5 names the receiving item for the specified delimiter.

Identifier-6 is used to hold the count of the number of characters in the sending item moved to the receiving item. This value does not include the count of the delimiter character(s).

A delimiter may be any character available in the ASCII collating sequences. Also, when a delimiter contains two or more characters, all of the characters must be in contiguous positions of the sending item, and be in the order specified to be recognized as a delimiter.

When more than one delimiter is specified, each delimiter is compared to the sending item in turn. If a match occurs in one of these comparisons, examination of the sending field ceases.

Delimiters may not overlap. That is, no character or characters in the sending item can be considered part of more than one delimiter.

When the ALL keyword is used in the DELIMITED phrase, and an associated delimiter is encountered while examining the sending item, each contiguous occurrence of the delimiter, beginning at the point where the delimiter first occurs, is considered as part of that delimiter. If the DELIMITER IN phrase has been specified, the entire string of contiguous delimiters is moved to the appropriate delimiter receiver.

After a single delimiter (or a string of delimiters as described in the preceding paragraph) has been found, if the next character or set of characters is a delimiter, the current receiving item is space or zero filled, depending upon how the receiving item is described. If the DELIMITER IN phrase is specified for that particular receiving item, the delimiter is then moved to the corresponding delimiter receiver.

The data item represented by *identifier-7* contains an integer used to indicate the first character, counting from the leftmost character of the sending item, to be examined. You are responsible for setting the initial value of this item. If it is less than one or is greater than the number of characters in the sending item when the UNSTRING statement is initiated, an OVERFLOW CONDITION exists.

When an UNSTRING statement completes execution, if the POINTER phrase is specified, the value of *identifier-7* is equal to the initial value plus the number of characters examined in the data item referenced by *identifier-1*.

The data item referenced by *identifier-8* is a counter that records the number of receiving items acted upon during the execution of an UNSTRING statement. As with *identifier-7*, you must initialize the value of *identifier-8*. When the UNSTRING statement completes execution, if the TALLYING phrase has been specified, the value of *identifier-8* is the initial value of *identifier-8* plus the number of data receiving items acted upon.

Execution of the UNSTRING Statement

When the UNSTRING statement is initiated, the current receiving item is the first receiving item.

If the POINTER phrase is specified, the sending item is examined beginning with the character position indicated by the contents of the data item referenced by *identifier-7*. If this phrase is not used, examination begins with the leftmost character of the sending item.

If the DELIMITED BY phrase is specified, the examination proceeds left to right until either a delimiter is found or no delimiters are found, and the last character of the sending item is examined.

If the DELIMITED BY phrase is not specified, the number of characters in the current receiving item is used to determine how many characters of the sending item are to be examined. That is, the number of characters examined is equal to the size of the receiving data item. However, if the receiving data item is a numeric data item described with the SIGN IS SEPARATE clause, the number of characters examined is one less than the size of the receiving data item.

When examination is complete, the examined characters, excluding any delimiters encountered, are treated as an elementary alphanumeric data item, and are moved into the receiving data item according to the rules for an alphanumeric MOVE. Refer to the description of the MOVE statement, earlier in this chapter.

If the DELIMITER IN phrase is specified for the current receiving item and a delimiter was encountered, the character (or characters) making up the delimiter are treated as an elementary alphanumeric data item and is moved into the delimiter receiver according to the rules of the MOVE statement. If the examination of the sending item ceased for a reason other than the occurrence of a delimiter, the delimiter receiver is filled with spaces.

If the COUNT IN phrase is specified for the current receiving item, a value equal to the number of characters examined, excluding any delimiter characters, is moved to the count receiver according to the rules for an elementary move. This completes the initial phase of execution of the UNSTRING statement.

If all characters of the sending item (beginning from the position specified by *identifier-7* if the POINTER phrase is specified) have been examined, the UNSTRING statement is complete and control passes to the next executable statement, `or to the imperative statement of the NOT ON OVERFLOW phrase, if specified.`

If all characters have not been used and another receiving item is specified, examination of the sending item begins again. This second examination begins with the character immediately to the right of the delimiter (if any) that caused termination of the initial examination. If no delimiter was specified, meaning that examination ceased because the number of characters in the current receiving item had been examined, examination begins with the character immediately to the right of the last character transferred. The contents of the data item referenced by *identifier-7* are incremented by one for each character examined in the sending item.

A new phase of examination and transfer is executed for each receiver item specified, or until all characters in the sending item have been examined. Each new phase of the UNSTRING statement is executed in the same way.

Overflow Conditions

An overflow condition is caused by one of two situations.

The first, described under the parameters description above, is caused by an invalid value for the data item represented by *identifier-7*.

The second situation is when all receiving items have been acted upon, but there remain unexamined characters in the sending item.

When an overflow condition occurs, execution of the UNSTRING condition ceases.

If the ON OVERFLOW phrase is specified and an overflow condition occurs, the imperative statement in the ON OVERFLOW phrase is executed.

If the ON OVERFLOW phrase is not specified, control is passed to the next executable statement following the UNSTRING statement.

Subscripting or Indexing of Identifiers

Subscripting or indexing of an identifier is evaluated only once, immediately before any data is transferred as the result of the initial phase of the UNSTRING statement.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01   ID-INFO                 PIC X(35).
01   EMPLOYEE-TABLE.
     02  EMPLOYEE-STATS OCCURS 30 TIMES.
         03 NAME             PIC X(40).
         03 BIRTH-DATE       PIC X(6).
         03 HAIR-COLOR       PIC X(12).
         03 EYE-COLOR        PIC X(12).
         03 HEIGHT           PIC X(2).

01  SUBSCRIPTOR              PIC X.
01  SUBSCRIPT                PIC 99 VALUE 1.
01  INCREMENT                PIC X VALUE ";".
01  CHARS                    PIC S9(4) USAGE COMP.
01  COMPLETE-INFO            PIC S9(4) USAGE COMP.
      ⋮
    MOVE 1 TO CHARS.
    MOVE 0 TO COMPLETE-INFO.
    UNSTRING ID-INFO
        DELIMITED BY "," OR INCREMENT
        INTO NAME (SUBSCRIPT)
             BIRTH-DATE (SUBSCRIPT)
             HAIR-COLOR (SUBSCRIPT)
             EYE-COLOR (SUBSCRIPT)
             HEIGHT (SUBSCRIPT)
        DELIMITER IN SUBSCRIPTOR
        WITH POINTER CHARS
        TALLYING IN COMPLETE-INFO
        ON OVERFLOW PERFORM FIND-CAUSE.
```

If ID-INFO is in standard data format:

```
WILSON JAMES,030250,BLONDE,BLUE,59;
```

and the initial value of CHARS is 1, and of COMPLETE-INFO is 0, when the UNSTRING statement above is executed it goes through the phases described below.

- Phase 1:

```
    WILSON JAMES,
    ↑          ↑
Initial     delimiter
pointer     found
```

  Move "WILSON JAMES" into NAME(1) filling in spaces to the left of the rightmost character. Increment the value of CHARS by 13, giving 14.

- Phase 2:

```
WILSON JAMES,030250,
             ↑     ↑
   new pointer     delimiter found
```

  Move 030250 into BIRTH-DATE(1). Increment the value of CHARS by 7, giving 21.

- Phase 3:

```
WILSON JAMES,030250,BLONDE,
                     ↑     ↑
          new pointer     delimiter found
```

  Move "BLONDE" into HAIR-COLOR(1), filling in spaces to the left of the rightmost character. Increment the value of CHARS by 7, giving 28.

- Phase 4:

```
WILSON JAMES,030250,BLONDE,BLUE,
                           ↑   ↑
                 new pointer   delimiter found
```

  Move "BLUE" into EYE-COLOR(1). Increment the value of CHARS by 5, giving 33.

- Phase 5:

```
WILSON JAMES,030250,BLONDE,BLUE,59;
                                ↑
                       new pointer; first delimiter not found.

WILSON JAMES,030250,BLONDE,BLUE,59;
                                ↑ ↑
                      new pointer second delimiter found.
```

  Move "59" to HEIGHT(1), and ";" to SUBSCRIPTOR. Increment the value of CHARS by 3, giving 36.

Since all receiving items have been used, this completes the execution of the UNSTRING statement. The value of CHARS is 36 and the value of COMPLETE-INFO is 5, since five receiving items were acted upon.

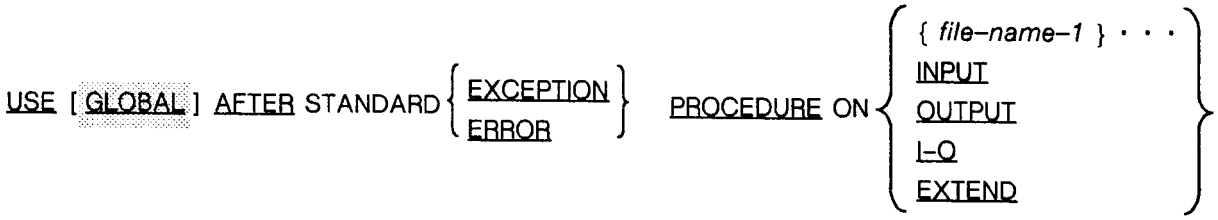
USE Statement

The USE statement specifies procedures for input-output error handling, user label processing, and debugging. These procedures are an addition to the standard procedures provided by the input-output control system.

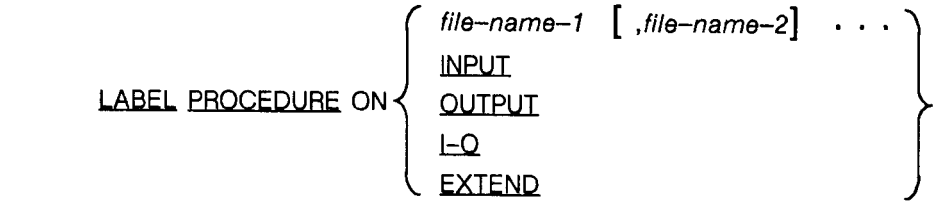
Syntax

There are three general formats of the USE statement:

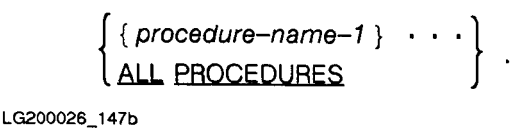
Format 1 – Error Handling Procedures



Format 2 – User Label Procedures



Format 3 – Debugging



LG200026_147b

For a description of format 3, refer to Chapter 12, "Debug Module".

Parameters

file-name-1 and *file-name-2*	the names of files to be acted upon by the appropriate procedures when an input-output error has occurred, or when a user label is to be processed. These names must not name sort-merge files.
ERROR and EXCEPTION	synonymous and may be used interchangeably.

Description

The rules below apply to both formats of the USE statement.

A USE statement, when specified, must immediately follow a section header in the declaratives section and must be followed by a period and a space. The remainder of the section must consist of zero, one, or more procedural paragraphs that define the procedures to be used. These paragraphs make up the procedures that are executed when required. The USE statement itself is never executed. It merely defines the conditions calling for the execution of the USE procedures.

Within a USE procedure, there must not be any reference to any nondeclarative procedures. Conversely, in the nondeclarative portion there must not be any reference to procedure names in the declarative portion, except that PERFORM statements may refer to a USE statement or to the procedures associated with such a USE statement.

Within a USE procedure, no statement can be executed that would cause the execution of a USE procedure that has been previously invoked, but has not yet returned control to the invoking routine.

In a USE statement, the files affected by the procedures related to the USE statement may be explicitly or implicitly specified.

They are explicitly specified by using their names in the USE statement itself.

To implicitly specify a file, the words, INPUT, OUTPUT, I-O, and EXTEND are used. For example, if a USE statement uses the word INPUT, any file opened for input, as opposed to output or input-output, is implicitly specified in that USE statement.

The same file name may appear in different USE statements. However, the appearance of a file name in a USE statement may not cause the simultaneous request for execution of more than one USE procedure.

USE Statement - Format 1

The first format of the USE statement is for error handling procedures.

The file implicitly or explicitly referenced in a format-1 USE statement need not have the same organization or access.

When a format 1 USE statement is specified for a file, and an input or output error occurs, the input-output system performs any applicable USE procedures either after completing the standard input-output error routine, or upon recognition of the INVALID KEY or AT END condition.

If a `GLOBAL` phrase is specified in the declarative statement, the USE procedure can be invoked from any program contained within the program in which the declarative statement is defined.

When an INVALID KEY or AT END condition occurs, the appropriate USE procedures are executed provided only that an AT END or INVALID KEY phrase has not been specified in the input-output statement that generated the error.

USE Statement - Format 2

This format of the USE statement is an HP extension to the ANSI COBOL standard.

The format 2 USE statement is for reading and writing user labels on a file, starting immediately after the operating system label at the beginning of a file.

There may be as many as eight user labels following an operating system label, and each label may consist of 80 characters.

When user labels are read by your program, only one location is made available for them in memory. Thus, only one may be read at a time, requiring only a single description of a label record data item per file.

Whether a file is explicitly or implicitly referenced in a USE statement, that statement does not apply and is ignored if the referenced file includes the LABEL RECORDS ARE OMITTED clause in its description.

The transfer of control to USE procedures occurs when a file is opened, as follows:

- INPUT, I-O, and EXTEND - control is passed to the appropriate USE procedure after a beginning input label check procedure is executed.
- OUTPUT - control is transferred after a beginning output label is created, but before it is written.

Using the INPUT keyword allows you to read labels, while using I-O allows you to both read and write them. The use of the OUTPUT or EXTEND keywords only allows you to write user labels.

Using a file name allows you to either read or write, or both, depending upon how the file is opened. Thus, for example, a file opened in I-O mode allows you to read and write user labels.

Within a format 2 USE procedure, no statements are explicitly written in order to read or write a user label.

For an input or input-output file, a corresponding USE procedure automatically reads the user label into the label record of the file. The USE procedure may then be written to check that the label has the form and content desired.

All format 2 USE procedures have an exit mechanism appended to them by the compiler. This exit mechanism follows the last statement of the procedure and is used to write user labels out to the appropriate file.

Therefore, a format 2 USE procedure for a file opened in OUTPUT, I-O, or EXTEND mode may use the label record data item to define the contents of a label. When the exit mechanism is reached, the label is automatically written to the file.

With a single exception, all logical paths within a declarative procedure must lead to this exit point, thus terminating the procedure. This implies that with one execution of a format 2 USE procedure, only a single user label may be read or written.

The purpose of the exception mentioned above is to allow you to read or write more than one user label (up to eight). The exception is the use of the GO TO MORE-LABELS statement within a format 2 USE procedure.

The function of this statement varies according to how the file was opened:

- Input files - Control returns to the software that reads an additional user label, and then transfers control back to the first statement of the USE procedure. The last statement in the USE procedure must be executed in order to terminate label processing.
- Output and EXTEND files - Control returns to the software that writes out the current user label, and then transfers control back to the first statement of the USE procedure so that additional user labels can be created. The last statement in the USE procedure must be executed in order to terminate label processing.
- Input-Output and EXTEND files - Control returns to the software that writes out the current label and then reads the next label. The software then transfers control back to the first statement of the USE procedure. The last statement in the USE procedure must be executed in order to write out the last user label and to terminate label processing.

WRITE Statement

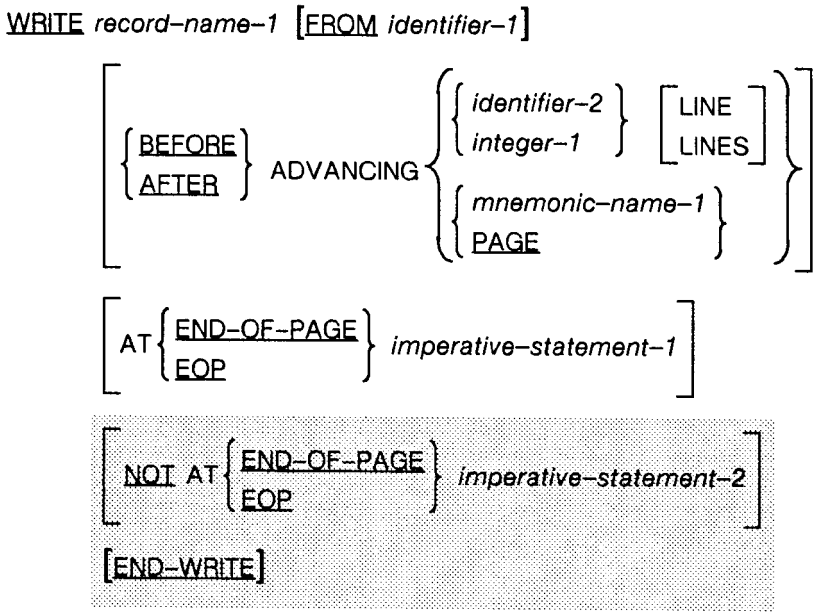
The WRITE statement releases a logical record. To use the WRITE statement for a sequential file, the file must be opened in the OUTPUT or EXTEND mode. To use the WRITE statement with an indexed, relative, or random file, the file must be opened in either OUTPUT, I-O mode, or EXTEND for access mode sequential.

For sequential files, the WRITE statement may additionally be used for vertical positioning of lines within a logical page (refer to the LINAGE clause in the File Description Entry of the DATA DIVISION in Chapter 7).

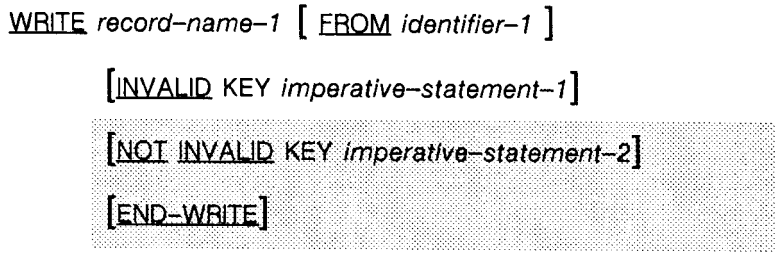
Syntax

There are two formats for the WRITE statement:

Format 1 - Sequential Files



Format 2 - Relative, Indexed, or Random-Access Files



LG200026_148

Parameters

record-name-1	the name of a logical record in the FILE SECTION of the DATA DIVISION. It may be qualified. *Record-name-1* must not reference the same storage area as *identifier-1*. Additionally for random access files, *record-name-1* must not be part of a SORT file.
identifier-1	the name of a data item described within the DATA DIVISION, or a function-identifier. If it is a function, it must be an alphanumeric function. If it is not a function, *identifier-1* and *record-name-1* must not reference the same storage area.
identifier-2	the name of an elementary integer data item. The value of the data item must be greater than or equal to zero.
integer-1	a nonnegative integer.
mnemonic-name-1	a name related to the functions, TOP, NO SPACE CONTROL, and C01 through C16. This relation is provided by the SPECIAL-NAMES clause to the CONFIGURATION SECTION of the ENVIRONMENT DIVISION.
`END-OF-PAGE and EOP`	equivalent.
imperative-statement-1 and *imperative-statement-2*,	each one or more imperative statements.

Description

The rules stated below apply to both formats of the WRITE statement.

The successful execution of a WRITE statement releases a logical record number of character positions defined by the logical description of that record in the program.

If the file's records are longer than the data being written to it, the file is padded with blanks or zeros, depending upon whether the file is an ASCII or binary file, respectively. If the file's records are shorter than the data being written to it, the data being written is truncated.

Whether execution of the WRITE statement was successful or not, the FILE STATUS data item, if any, is updated following the execution of the WRITE statement. Refer to Chapter 6, under "FILE STATUS Clause", for valid status keys. For more information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

The logical record released by the execution of the WRITE statement is no longer available in memory unless the associated file is named in a SAME RECORD AREA clause, or the execution of the WRITE statement was unsuccessful because of a boundary violation (for sequential files) or an INVALID KEY condition (for relative, indexed or random access files).

The logical record is also available to the program as a record of other files referenced in that SAME RECORD AREA clause.

FROM Phrase

The results of executing a WRITE statement using the FROM phrase is equivalent to executing the statement,

`MOVE` *identifier-1* `TO` *record-name-1*

and then executing the same WRITE statement without the FROM phrase.

Unlike the record area for *record-name-1*, the data in *identifier-1* always remains in memory and is available after execution of the WRITE statement, regardless of whether a SAME RECORD AREA clause was used for any file in which *identifier-1* names a data item.

Note that the maximum record size of a file is established at the time the file is created, and must not subsequently be changed.

WRITE Statement - Format 1

A format 1 WRITE statement, used for sequential files only, allows you to use vertical positioning of a line within a logical page.

This is done through the ADVANCING and END-OF-PAGE phrases.

Either or both of these phrases may be used in a format 1 WRITE statement. However, if the END-OF-PAGE phrase is used, the LINAGE clause must appear in the file description entry for the associated file. Also, if the LINAGE phrase is present in the file's description, and the ADVANCING phrase is used, it cannot be used in the form ADVANCING *mnemonic-name-1*.

If neither phrase is used, automatic advancing equivalent to AFTER ADVANCING 1 LINE is provided.

Whenever the execution of a given format 1 WRITE statement cannot be fully accommodated within the current page body, an automatic page overflow condition occurs. If neither the ADVANCING nor the END-OF-PAGE phrase is specified, and a page overflow condition occurs, the WRITE statement uses an implicit AFTER ADVANCING PAGE to position the data on the next logical page.

ADVANCING Phrase

When the ADVANCING phrase is used in a format 1 WRITE statement, the line to be written is presented to the page either before or after the representation of the page is advanced. Whether it is presented before or after advancing the logical page is determined by the use of the BEFORE or AFTER keyword, respectively.

The amount of advancement of the logical page is determined by *integer-1*, *identifier-2*, PAGE, and *mnemonic-name-1* as follows:

- *Integer-1* causes the representation of the logical page to be advanced the number of lines equal to the value of integer.
- *Identifier-2* causes the representation of the logical page to be advanced the number of lines equal to the current value of the data item represented by *identifier-2*.
- PAGE causes the logical page to be advanced to the next logical page. If the record to be written is associated with a file whose description includes a LINAGE clause, the repositioning is to the first line that can be written on the new logical page as specified by the LINAGE clause. If the record to be written is associated with a file whose description

does not contain a LINAGE clause, the repositioning is to the first line of the next logical page.

If PAGE is specified for a device to which it has no meaning, advancing is provided that is equivalent to ADVANCING 1 LINE.

- If *mnemonic-name-1* is specified, the file receiving the record must not contain a LINAGE clause in its description and must be a line printer device file. *Mnemonic-name-1* can be equivalent to TOP, in which case it is equivalent to specifying PAGE for a file whose description does not contain a LINAGE clause.

Mnemonic-name-1 may also be equivalent to one of C01 to C16, or NO SPACE CONTROL.

The C01 through C16 options are related to the VFU (Vertical Form Unit) holes punched into a paper tape of a line printer. For an HP line printer, C01 through C16 have the following meanings:

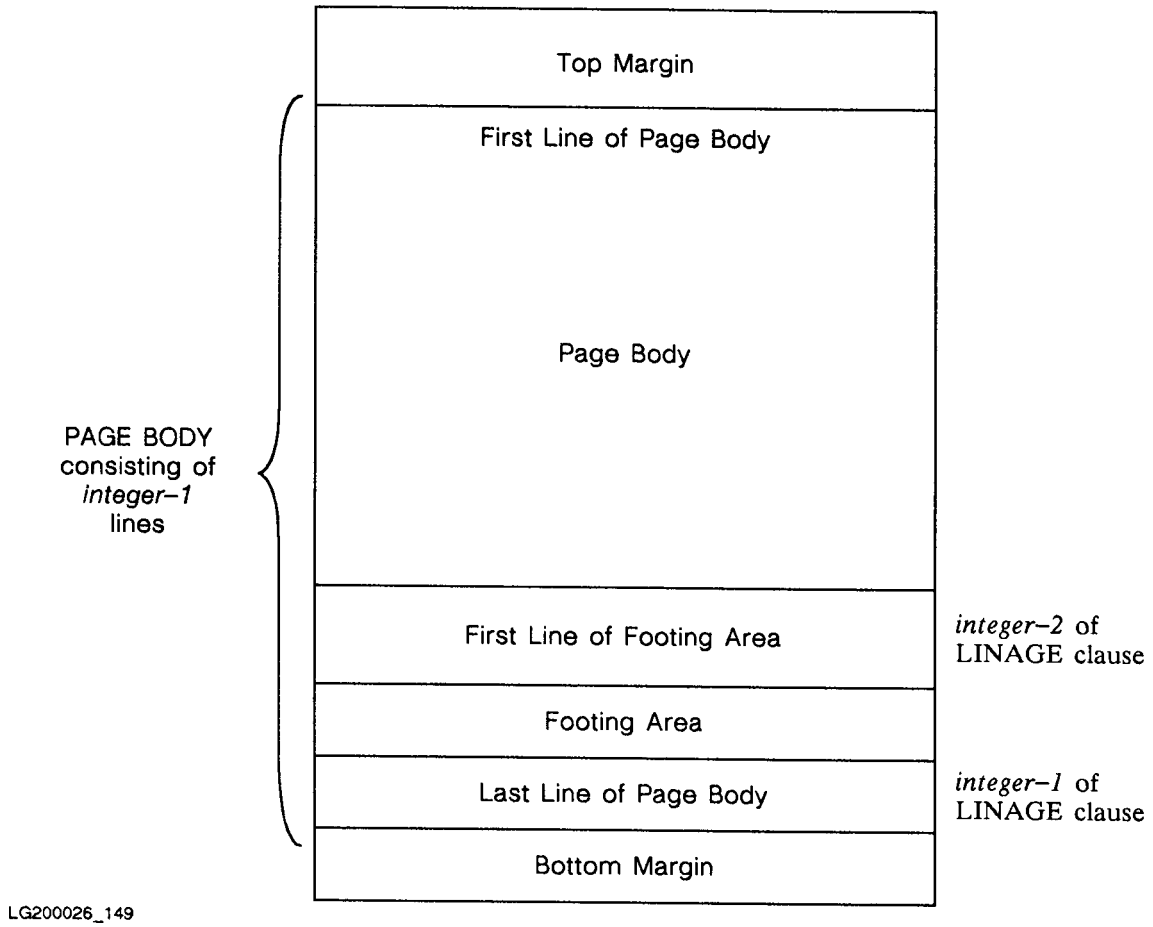
c01: Page eject (skip to top of next page).
c02: Skip to the bottom of the form.
c03: Single spacing with automatic page eject.
c04: Single space on the next odd-numbered line with automatic page eject.
c05: Triple space with automatic page eject.
c06: Space a half page with automatic page eject.
c07: Space one quarter of a page with automatic page eject.
c08: Space one sixth of a page with automatic page eject.
c09: Space to the bottom of the form.
c10: User option.
c11: User option.
c12: User option.
c13: User option.
c14: User option.
c15: User option.
c16: User option.

Note The C13 through C16 options may not be functionally operable on all printer devices; for example, printer devices that only provide 12 VFU channels.

END-OF-PAGE Phrase

The END-OF-PAGE phrase can be used only in conjunction with the LINAGE clause of a sequential file description entry.

To clarify the following description, the concept of a logical page is illustrated below:



Two conditions may occur that cause the execution of the END-OF-PAGE phrase.

The first occurs when a footing area has been defined using *integer-2* or *data-name-2* of the LINAGE Clause. In this case, when a WRITE statement using the END-OF-PAGE phrase is executed, and this execution causes printing or spacing within the footing area, an end-of-page condition occurs. This is controlled by the value of the LINAGE-COUNTER for the associated file. Thus, when LINAGE-COUNTER equals or exceeds the value of *integer-2* or the data item specified by *data-name-2*, an end-of-page condition occurs. The data is written into the footing area, and the *imperative-statement-1* of the END-OF-PAGE phrase is executed.

Note An end-of-page condition does not automatically cause the next line of data to be written on the next logical page; it is your responsibility to control this using the LINAGE-COUNTER of the file, the ADVANCING phrase, or the automatic page overflow condition.

The second condition that causes the execution of an END-OF-PAGE phrase is an automatic page overflow. An automatic page overflow occurs when the LINAGE-COUNTER for the file associated with the WRITE statement exceeds *integer-1* or the data item referenced by *data-name-1*. In this case, if an ADVANCING phrase using the BEFORE keyword is present in the WRITE statement, the record is written the specified number of lines below the end of the page body, and the device used to contain the logical pages is repositioned to the first line that can be written on the next logical page. If an ADVANCING phrase is specified (implicitly or explicitly) that uses the AFTER keyword, the device used to contain the logical pages is repositioned to the first line that can be written on the next logical page, and the record is written.

No matter whether ADVANCING BEFORE or ADVANCING AFTER is specified, when the record has been written, control is transferred to the *imperative-statement-1* of the END-OF-PAGE phrase.

If *integer-2* or the data item referenced by *data-name-2* of the LINAGE clause is not specified (thus, no footing area has been defined), or if *integer-2* or the data item referenced by *data-name-2* is equal to *integer-1* or the data item referenced by *data-name-1* of the LINAGE clause, no end-of-page condition distinct from a page overflow is detected. Thus, an end-of-page condition is, in this case, equivalent to a page overflow condition.

Bounds Overflow

When an attempt is made to write beyond the boundaries of a sequential file, an exception condition exists. The contents of the record area specified by *record-name-1* are unaffected by such a condition and the following actions take place:

- The FILE STATUS data item, if any, of the associated file is set to indicate a boundary violation.
- If a USE AFTER STANDARD EXCEPTION declarative is specified (explicitly or implicitly) for the file, the associated procedure is executed. If no USE statement is specified for the file, the program aborts, supplying a file error message.

For more information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

Multiple Reel/Unit Files

After an end-of-reel condition has been recognized for a multiple reel labeled tape, the WRITE statement performs the standard ending reel or unit procedure, requests a reel or unit swap, and then performs the standard reel or unit label procedure. The record is then written according to the specifications of the WRITE statement.

Print Files

A print file is organized like a sequential organization file and has carriage control. The carriage control option cannot be changed after the file is created.

The three ways to create a print file are:

1. Use the MPE :BUILD command with the carriage control option CCTL. For example:

   `:BUILD` *filename*`;CCTL`
2. Use the MPE :FILE command to cause COBOL to create a print file. For example:

   `:FILE` *filename*`;CCTL`
3. Cause the compiler to enable the CCTL option. The following conditions cause the compiler to enable the CCTL option:
   a. The SELECT ... ASSIGN clause includes the device name parameter CCTL.
   b. The SELECT ... ASSIGN clause specifies the actual file $STDLIST.
   c. The WRITE statements include the BEFORE/AFTER ADVANCING clause.
   d. A file equation for the actual file specifies CCTL.

You can disable CCTL by any of the following:

1. Avoiding the conditions that enable CCTL (see above).
2. Specifying NOCCTL in the file equation.
3. Using a disc file that did not have CCTL when it was built.

When CCTL is enabled, the compiler does the following:

1. Enables the file for CCTL operations when it is opened.
2. Writes the following two control records to the file, before the firs data record (even if the file is opened in EXTEND mode).
   a. Record one contains an ASCII "A" (%101) in the first byte, which sets the printer spacing mode to PRE-SPACE, the COBOL default. PRE-SPACE is equivalent to AFTER ADVANCING.
   b. Record two contains an ASCII "C" (%103) in the first byte, which sets the printer to the single-space option, without automatic page eject. This allows you to control the page size.
3. Prepends to each record a character that contains carriage control code. This character is transparent so you need not allow space for it when you define the record.

Carriage Control Codes. The carriage control codes that COBOL uses are defined within the MPE FWRITE intrinsic. They are written in the first byte of each data record or passed through the control parameter. The carriage control codes and their meanings are shown in Table 9-7.

Table 9-7. Carriage Control Codes and Their Meanings

Code	Meaning
%2*n*	Space *n* lines (no automatic page eject). The number *n* is in the range 0 through %77 (63).
%3*n*	Channel options C01 through C16. The number *n* is in the range 0 through %17 (15).
%61	Eject a page (go to the top of the next page; equivalent to TOP).
%100	Set printer spacing mode to POST-SPACE (equivalent to BEFORE ADVANCING).
%101	Set printer spacing mode to PRE-SPACE (equivalent to AFTER ADVANCING).
%103	No automatic page eject.
%320	No space control.

If a file has carriage control due to the MPE :FILE command or disc label, but has no ADVANCING clause, you must put a carriage control code in the first byte of each record yourself.

If a file has at least one ADVANCING clause, then all other WRITE statements for that file are treated as if AFTER ADVANCING 1 were specified.

WRITE Statement - Format 2

A format 2 WRITE statement can be used for random access, relative, or indexed files.

In a format 2 WRITE statement, if no applicable USE statement has been issued for the referenced file, the INVALID KEY phrase of the WRITE statement must be used.

Also, the INVALID KEY phrase must always be used when a format 2 WRITE statement is issued for a random access file.

Random Access Files

When a format 2 WRITE statement is issued for a random access file, the contents of the ACTUAL KEY data item associated with the file are used in an implicit seek to find the record into which the data specified by *record-name-1* is to be written. If the address specified by the ACTUAL KEY data item is invalid, an INVALID KEY condition exists, no data is written, the data in the record area is unaffected, and the *imperative-statement-1* in the INVALID KEY phrase is executed.

The address can be invalid for one of three reasons, as follows:

- It contains a negative value.
- It is greater than the highest possible relative record number in the file.
- The value moved to the ACTUAL KEY data item contains more than nine digits.

Relative Files

When a format 2 WRITE statement is issued for a relative file, and the file is open for output in sequential mode, the first execution of a WRITE statement for the file releases a record to that file, assigning a relative record number of 1 to it. Subsequent WRITE statements assign relative record numbers of 2,3,4, and so on as records are released to the file. If the RELATIVE KEY data item has been specified in the file control entry for the associated file, it is updated during each execution of the WRITE statement to indicate the relative record number of the record being written.

If a relative file is open in random or dynamic access mode, regardless of whether it is open for output only or input-output operations, your program must set the value of the relative key data item to specify where the record is to be placed in the file.

Provided that the relative key data item is a valid relative key, the data in the record area is released to the file.

INVALID KEY Conditions For a Relative File

An INVALID KEY condition exists for a relative file when an attempt is made to write beyond the externally defined boundaries of the file, or when the file is in dynamic or random access mode and the RELATIVE KEY data item specifies a record that already exists in the file.

When this condition occurs, execution of the WRITE statement is unsuccessful, the contents of the record area are unaffected, and the FILE STATUS data item, if any, is updated to indicate the cause of the condition.

If the INVALID KEY phrase was specified in the WRITE statement, control passes to the imperative statement appearing in the phrase. If no INVALID KEY phrase was specified, then a USE procedure must have been specified (either implicitly or explicitly), and is executed.

If both a USE procedure and an INVALID KEY phrase are specified, the USE procedure is ignored.

For more information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

Indexed Files

A format 2 WRITE statement for an indexed file uses the primary record key data item to write records to the file; therefore, the value of the primary record key must be unique within the records of the file unless the DUPLICATES phrase has been used in the RECORD KEY clause of the ENVIRONMENT DIVISION.

If alternate record keys have been specified, they must also be unique within the file unless the DUPLICATES phrase has been used in the ALTERNATE RECORD KEY clause of the ENVIRONMENT DIVISION.

If the DUPLICATES phrase is used, then after records containing duplicate keys have been written to the file, if they are later accessed sequentially, they are retrieved in the same order as they were written.

Note A WRITE statement for an indexed file open in sequential access mode must write records to the file in ascending order of prime record key values.

A WRITE statement for an indexed file in random or dynamic access mode writes a record to the file in whatever order your program specifies.

When a WRITE statement is successfully executed for an indexed file, all keys of the record are used in such a way that subsequent access of the record may be made based upon any of the specified record keys.

INVALID KEY Conditions For Indexed Files

There are three conditions under which an INVALID KEY condition can occur for an indexed file WRITE statement:

- When the file is open for output in sequential access mode, and the value of the prime record key is not greater than the value of the prime record key of a previous record.
- When the file is open for output or input-output operations in any access mode, and the value of the primary or an alternate record key for which duplicates are not allowed, equals the value of the corresponding record key of a record already existing in the file.
- When an attempt is made to write beyond the externally defined boundaries of the file.

In any case, when an INVALID KEY condition occurs, execution of the WRITE statement is unsuccessful, the record area is unaffected, and the FILE STATUS data item, if any, associated with the file is set to a value that indicates the cause of the condition.

If an INVALID KEY phrase was specified in the WRITE statement, control is transferred to the imperative statement appearing in the phrase, and any USE procedure specified for the file is ignored.

If no INVALID KEY phrase was specified, then a USE procedure must have been specified, implicitly or explicitly, and that procedure is executed.

For more information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

10

COBOL Functions

This chapter describes the built-in COBOL functions and how to call them. These functions were defined in 1989 by Addendum 1 of the ANSI COBOL '85 standard. The built-in functions provide the capability to reference a data item whose value is derived automatically at the time of reference during the execution of the program.

The following tables list and briefly describe each function:

Table 10-1. Date Functions

Function	Type	Value Returned
CURRENT-DATE	Alphanumeric	Current date and time and difference from Greenwich Mean Time.
DATE-OF-INTEGER	Integer	Standard date equivalent (YYYYMMDD) of integer date.
DAY-OF-INTEGER	Integer	Julian date equivalent (YYYYDDD) of integer date.
INTEGER-OF-DATE	Integer	Integer date equivalent of standard date (YYYYMMDDD).
INTEGER-OF-DAY	Integer	Integer date equivalent of Julian date (YYYYDDD).
WHEN-COMPILED	Alphanumeric	Date and time program was compiled.

Table 10-2. String Functions

Function	Type	Value Returned
CHAR	Alphanumeric	The character in a specified position of the program collating sequence.
LENGTH	Integer	Length, in character positions, of the parameter.
LOWER-CASE	Alphanumeric	The same parameter with all uppercase letters replaced by lowercase letters.
NUMVAL	Numeric	Numeric value of a simple numeric string.
NUMVAL-C	Numeric	Numeric value of a numeric string with optional commas and currency sign.
ORD	Integer	Ordinal position of the parameter in collating sequence.
REVERSE	Alphanumeric	Same parameter with characters in reverse order.
UPPER-CASE	Alphanumeric	Same parameter with all lowercase letters replaced by uppercase letters.

Table 10-3. General Functions

Function	Type	Value Returned
MAX	Depends on parameters.	Maximum value of all parameters.
MIN	Depends on parameters.	Minimum value of all parameters.
ORD-MAX	Integer	Ordinal position of maximum parameter.
ORD-MIN	Integer	Ordinal position of minimum parameter.

Table 10-4. Arithmetic Functions

Function	Type	Value Returned
INTEGER	Integer	The greatest integer not greater than the given numeric value.
INTEGER-PART	Integer	Integer part of the given numeric value.
LOG	Numeric	Natural logarithm of a numeric value.
LOG10	Numeric	Logarithm to base 10 of a numeric value.
MOD	Integer	Modulo of two integer parameters.
RANDOM	Numeric	Pseudo-random number.
REM	Numeric	Remainder after division.
SQRT	Numeric	Square root of a numeric value.
SUM	Integer or Numeric	Sum of parameters.

Table 10-5. Financial and Statistical Functions

Function	Type	Value Returned
ANNUITY	Numeric	Ratio of an annuity paid for a specified number of periods at a specified interest rate, to an initial investment of one.
FACTORIAL	Integer	Factorial of an integer value.
MEAN	Numeric	Arithmetic mean of parameters.
MEDIAN	Numeric	Median of parameters.
MIDRANGE	Numeric	Mean of smallest and largest parameters.
PRESENT-VALUE	Numeric	Present value of a series of future period-end amounts at a given discount rate.
RANGE	Integer or Numeric	Value of largest parameter minus value of smallest parameter.
STANDARD-DEVIATION	Numeric	Standard deviation of parameters.
VARIANCE	Numeric	Variance of parameters.

Table 10-6. Trigonometric Functions

Function	Type	Value Returned
COS	Numeric	Cosine of an angle in radians.
SIN	Numeric	Sine of an angle in radians.
TAN	Numeric	Tangent of an angle in radians.
ACOS	Numeric	Arccosine, in radians, of a numeric value.
ASIN	Numeric	Arcsine, in radians, of a numeric value.
ATAN	Numeric	Arctangent, in radians, of a numeric value.

The $CONTROL POST85 Option

You must specify $CONTROL POST85 in any program that calls a COBOL function. $CONTROL POST85 enables the COBOL functions and makes the word FUNCTION a reserved word. If you have used the word FUNCTION as an identifier, you must change it to another word before you can call any COBOL functions. Otherwise the compiler gives an error message.

ANSI85 Entry Point

You must use the ANSI85 entry point of the HP COBOL II/XL compiler to call any COBOL functions.

Function Types

Functions are treated like temporary, elementary data items. Use them wherever you would use an elementary data item, with the exceptions noted below. Functions cannot be receiving operands. Functions return alphanumeric, numeric, or integer values, as follows:

- Alphanumeric functions are of the class and category alphanumeric, and have an implicit usage of DISPLAY.
- Numeric functions are of the class and category numeric and always have an operational sign. Numeric functions can only be used in arithmetic expressions (such as in COMPUTE statements, relation conditions, or reference modification) and cannot be used where an integer operand is required, even if the function call might yield an integer value.
- Integer functions are of the class and category numeric and always have an operational sign. The definition of an integer function provides that all digits to the right of the decimal point are zero in the returned value for any possible evaluation of the function. Integer functions can only be used in arithmetic expressions. For example, the statement

  ```
  MOVE FUNCTION SIN(5) TO A
  ```

  is illegal, whereas the statement

  ```
  COMPUTE A = FUNCTION SIN(5)
  ```

  is legal.

Function Parameters

Some of the functions require one or more parameters. Function parameters can be identifiers, arithmetic expressions, or literals. See the description of each function for specific information about its parameters.

If a function requires parameters and you do not supply any, or if the parameters supplied do not comply with all restrictions for parameters to that function, the value returned by the function is undefined.

Using ALL as a Table Subscript

Some functions allow a variable number of arguments, for example MAX, MEAN, and SUM. You can pass all the elements of a table to one of these functions by specifying the table as a parameter with ALL as the table subscript. See the examples under these functions. See also "Referencing Table Items with Subscripting" in Chapter 4. In multi-dimensional tables, ALL may occur in one or more subscripts.

Precision of Numeric Functions

Some of the numeric functions convert the parameters you pass to intermediate floating point values to calculate the function result. The precision of these functions is limited to 15 significant digits. Also, fractional values may have rounding errors even if the total size of the argument is less than or equal to 15 digits. For example, using the SUM function on a table of dollars and cents values, such as PIC S9(9)V99, might not produce the correct answer. Use of the ROUNDED phrase is recommended.

See the *HP COBOL II/XL Programmer's Guide* for more information on performance and the precision of functions.

Calling COBOL Functions

To call any of the COBOL functions, simply put the function call in any statement where a data item of the function type is valid. You must use $CONTROL POST85 and the ANSI85 entry point to the HP COBOL II/XL compiler when calling any COBOL function.

Examples

To change all the characters in a string to lower case, use the LOWER-CASE function in a MOVE statement:

```
77  MY-NAME           PIC X(5) VALUE "STEVE".
      :
DISPLAY MY-NAME.
MOVE FUNCTION LOWER-CASE (MY-NAME) TO MY-NAME.
DISPLAY MY-NAME.
```

The above example displays the following:

```
STEVE
steve
```

To calculate the cosine of an angle, you can use the COS function in a COMPUTE statement:

```
77  ANGLE-RADIANS     PIC S99V9(5) VALUE 3.14159.
77  COS-OF-ANGLE      PIC S9V9(5)  VALUE ZERO.
      :
COMPUTE COS-OF-ANGLE = FUNCTION COS (ANGLE-RADIANS).
DISPLAY ANGLE-RADIANS.
DISPLAY COS-OF-ANGLE.
```

The above example displays the following:

```
+03.14159
-0.99999
```

The rest of this chapter explains each of the COBOL functions.

ACOS Function

The ACOS function returns the arccosine of the parameter. The function type is numeric.

Syntax

FUNCTION ACOS (*parameter-1*)

Parameters

parameter-1 Must be class numeric and must be between -1 and 1, inclusive.

Return Value

The value returned is the approximation of the arccosine of *parameter-1* and is between 0 and π, inclusive. The return value is a numeric value in radians.

Example

```
77  COS-OF-ANGLE       PIC S9V9(5)  VALUE +0.70711.
77  ANGLE-RADIANS      PIC S99V9(5) VALUE ZERO.
      :
COMPUTE ANGLE-RADIANS = FUNCTION ACOS (COS-OF-ANGLE).
DISPLAY COS-OF-ANGLE.
DISPLAY ANGLE-RADIANS.
```

The above example displays the following:

```
+0.70711
+00.78539
```

ANNUITY Function

The ANNUITY function (annuity immediate) returns a numeric value that is the ratio of an annuity paid at the end of each period for the number of periods specified by *parameter-2* to an initial investment of one. Interest is earned at the rate specified by *parameter-1* and is applied at the end of the period before the payment. The function type is numeric.

Syntax

FUNCTION ANNUITY (*parameter-1 parameter-2*)

Parameters

parameter-1 Must be class numeric and must be greater than or equal to zero.

parameter-2 Must be a positive integer.

Return Value

When the value of *parameter-1* is zero, the value of the function is

$$\frac{1}{\textit{parameter-2}}$$

When the value of *parameter-1* is not zero, the value of the function is

$$\frac{\textit{parameter-1}}{(1 - (1 + \textit{parameter-1}) ** (- \textit{parameter-2}))}$$

Example

```
WORKING-STORAGE SECTION.
77  NUM-RATE       PIC S9V9999 VALUE 0.08.
77  MONTHLY-RATE   PIC S9V9999 VALUE ZERO.
77  NUM-PERIODS    PIC 99 VALUE 36.
77  NUM-ANNUITY    PIC S9V9999 VALUE ZERO.
PROCEDURE DIVISION.
010-PARA.
   COMPUTE MONTHLY-RATE ROUNDED = NUM-RATE / 12
   COMPUTE NUM-ANNUITY ROUNDED =
       FUNCTION ANNUITY (MONTHLY-RATE NUM-PERIODS)
   DISPLAY MONTHLY-RATE
   DISPLAY NUM-PERIODS
   DISPLAY NUM-ANNUITY.
```

The above example displays the following:

```
+0.0067
36
+0.0314
```

ASIN Function

The ASIN function returns the arcsine of the parameter. The function type is numeric.

Syntax

FUNCTION ASIN (*parameter-1*)

Parameters

parameter-1 Must be class numeric and must be between -1 and 1, inclusive.

Return Value

The value returned is the approximation of the arcsine of *parameter-1* and is between $-\pi/2$ and $\pi/2$, inclusive. The return value is a numeric value in radians.

Example

```
77  SIN-OF-ANGLE       PIC S9V9(5)  VALUE +0.70710.
77  ANGLE-RADIANS      PIC S99V9(5) VALUE ZERO.
      :
COMPUTE ANGLE-RADIANS = FUNCTION ASIN (SIN-OF-ANGLE).
DISPLAY SIN-OF-ANGLE.
DISPLAY ANGLE-RADIANS.
```

The above example displays the following:

```
+0.70710
+00.78538
```

ATAN Function

The ATAN function returns the arctangent of the parameter. The function type is numeric.

Syntax

FUNCTION ATAN (*parameter-1*)

Parameters

parameter-1 Must be class numeric.

Return Value

The value returned is the approximation of the arctangent of *parameter-1* and is between $-\pi/2$ and $\pi/2$, inclusive. The return value is a numeric value in radians.

Example

```
77  TAN-OF-ANGLE       PIC S9(5)V9(5)  VALUE +00000.99998.
77  ANGLE-RADIANS      PIC S99V9(5) VALUE ZERO.
        ⋮
COMPUTE ANGLE-RADIANS = FUNCTION ATAN (TAN-OF-ANGLE).
DISPLAY TAN-OF-ANGLE.
DISPLAY ANGLE-RADIANS.
```

The above example displays the following:

```
+00000.99998
+00.78538
```

CHAR Function

The CHAR function returns a one-character alphanumeric value that is a character in the program collating sequence having the ordinal position equal to the value of *parameter-1*. The function type is alphanumeric.

Syntax

```
FUNCTION CHAR (parameter-1)
```

Parameters

parameter-1 Must be an integer. Must be greater than zero and less than or equal to the number of positions in the collating sequence.

Return Value

If more than one character has the same position in the program collating sequence, the character returned as the function value is that of the first literal specified for that character position in the ALPHABET clause.

If the current program collating sequence was not specified by an ALPHABET clause, the ASCII collating sequence is used. See Appendix D, "ASCII and EBCDIC Character Sets," for a listing of the ASCII character set.

Example

```
DISPLAY FUNCTION CHAR(50).
DISPLAY FUNCTION CHAR(64).
DISPLAY FUNCTION CHAR(88).
```

The above example displays the following:

```
1
?
W
```

COS Function

The COS function returns the cosine of an angle. The function type is numeric.

Syntax

FUNCTION COS (*parameter-1*)

Parameters

parameter-1 The size of an angle in radians. Must be class numeric.

Return Value

The value returned is the approximation of the cosine of *parameter-1* and is between -1 and 1, inclusive. The value returned is numeric.

Example

```
77  ANGLE-RADIANS      PIC S99V9(5) VALUE 3.14159.
77  COS-OF-ANGLE       PIC S9V9(5)  VALUE ZERO.
        ⋮
COMPUTE COS-OF-ANGLE = FUNCTION COS (ANGLE-RADIANS).
DISPLAY ANGLE-RADIANS.
DISPLAY COS-OF-ANGLE.

DIVIDE ANGLE-RADIANS BY 4 GIVING ANGLE-RADIANS.
COMPUTE COS-OF-ANGLE = FUNCTION COS (ANGLE-RADIANS).
DISPLAY ANGLE-RADIANS.
DISPLAY COS-OF-ANGLE.
```

The above example displays the following:

```
+03.14159
-0.99999
+00.78539
+0.70711
```

CURRENT-DATE Function

The CURRENT-DATE function returns the calendar date, time of day, and the difference between the local time and Universal Coordinated Time (UTC), or Greenwich Mean Time. To get the correct time differential, you need to set the environment variable TZ to your local time zone. See below for more information. The function type is alphanumeric.

This function is different from the CURRENT-DATE special register word (described in Chapter 3). One difference is that the CURRENT-DATE function provides a four-digit year.

Syntax

```
FUNCTION CURRENT-DATE
```

Return Values

This function returns a 21-character alphanumeric string with each character position defined as follows:

Character Positions	Contents
1-4	Four numeric digits of the year in the Gregorian calendar.
5-6	Two numeric digits of the month of the year, in the range 01 through 12.
7-8	Two numeric digits of the day of the month, in the range 01 through 31.
9-10	Two numeric digits of the hours past midnight, in the range 00 through 23.
11-12	Two numeric digits of the minutes past the hour, in the range 00 through 59.
13-14	Two numeric digits of the seconds past the minute, in the range 00 through 59.
15-16	Two numeric digits of the hundredths of a second past the second, in the range 00 through 99. The value 00 is returned because your system cannot provide the fractional part of a second.
17	One of the following:

Value	When Returned
-	Returned if the local time in the previous character positions is behind Greenwich Mean Time.
+	Returned if the local time indicated is the same or is ahead of Greenwich Mean Time.
0	Returned on non-MPE XL systems that do not have the facility to provide the local time differential factor.

Character Positions	Contents
18-19	Depending on the value of character position 17, one of the following:

Position 17	Contents
-	Two numeric digits in the range 00 through 12 indicating the number of hours that the reported time is behind Greenwich Mean Time.
+	Two numeric digits in the range 00 through 13 indicating the number of hours that the reported time is ahead of Greenwich Mean Time.
0	The value 00 is returned.

Character Positions	Contents
20-21	Depending on the value of character position 17, one of the following:

Position 17	Contents
-	Two numeric digits in the range 00 through 59 indicating the number of additional minutes that the reported time is behind of Greenwich Mean Time.
+	Two numeric digits in the range 00 through 59 indicating the number of additional minutes that the reported time is ahead of Greenwich Mean Time.
0	The value 00 is returned.

Setting the TZ Environment Variable

To get the correct difference between local time and Greenwich Mean Time, you must set the environment variable TZ to your local time zone. To set TZ, use the MPE XL SETVAR command. For example, the following command sets the time zone to Central Standard Time and Central Daylight Time, which would be correct for Chicago, Illinois:

```
:SETVAR TZ 'CST6CDT'
```

The following table lists some time zones. Check your local time zone to be sure you use the correct one.

Table 10-7. Time Zones and TZ Environment Variable Values

TZ Value	Time Zone	Geographic Area
HST10	Hawaiian Standard Time, Hawaiian Daylight Time.	Unites States: Hawaii.
AST10ADT	Aleutian Standard Time, Aleutian Daylight Time.	United States: Alaska (parts).
YST9YDT	Yukon Standard Time, Yukon Daylight Time.	United States: Alaska (parts).
PST8PDT	Pacific Standard Time, Pacific Daylight Time.	Canada: British Columbia. United States: California, Idaho (parts), Nevada, Oregon (parts), Washington.
MST7MDT	Mountain Standard Time, Mountain Daylight Time.	Canada: Alberta, Saskatchewan (parts). United States: Colorado, Idaho (parts), Kansas (parts), Montana, Nebraska (parts), New Mexico, North Dakota (parts), Oregon (parts), South Dakota (parts), Texas (parts), Utah, Wyoming.
MST7	Mountain Standard Time.	United States: Arizona.
CST6CDT	Central Standard Time, Central Daylight Time.	Canada: Manitoba, Ontario (parts), Saskatchewan (parts). United States: Alabama, Arkansas, Florida (parts), Illinois, Iowa, Kansas, Kentucky (parts), Louisiana, Michigan (parts), Minnesota, Mississippi, Missouri, Nebraska, North Dakota, Oklahoma, South Dakota, Tennessee (parts), Texas, Wisconsin.
EST6CDT	Eastern Standard Time, Central Daylight Time.	United States: Indiana (most).
EST5EDT	Eastern Standard Time, Eastern Daylight Time.	Canada: Ontaria (parts), Quebec (parts). United States: Connecticut, Delaware, District of Columbia, Florida, Georgia, Kentucky, Maine, Maryland, Massachusetts, Michigan, New Hampshire, New Jersey, New York, North Carolina, Ohio, Pennsylvania, Rhode Island, South Carolina, Tennessee (parts), Vermont, Virginia, West Virginia.
AST4ADT	Atlantic Standard Time, Atlantic Daylight Time.	Canada: Newfoundland (parts), Nova Scotia, Prince Edward Island, Quebec (parts).
NST3:30NDT	Newfoundland Standard Time, Newfoundland Daylight Time.	Canada: Newfoundland (parts).
WET0WETDST	Western European Time, Western European Time Daylight Savings Time.	Great Britain, Ireland.
PWT0PST	Portuguese Winter Time, Portuguese Summer Time.	
MEZ-1MESZ	Mitteleuropaeische Zeit, Mitteleuropaeische Sommerzeit.	
MET-1METDST	Middle European Time, Middle European Time Daylight Savings Time.	Belgium, Lexembourg, Netherlands, Denmark, Norway, Austria, Poland, Czechoslovakia, Sweden, Switzerland, Germany, France, Spain, Hungary, Italy, Yugoslavia.

Table 10-7. Time Zones and TZ Environment Variable Values (continued)

TZ Value	Time Zone	Geographic Area
SAST-2SADT	South Africa Standard Time, South Africa Daylight Time.	South Africa.
JST-9	Japan Standard Time.	Japan.
WST-8:00	Australian Western Standard Time.	Australia: Western Australia.
CST-9:30	Australian Central Standard Time.	Australia: Northern Territory.
CST-9:30CDT	Australian Central Standard Time, Australian Central Daylight Time.	Australia: South Australia.
EST-10	Australian Eastern Standard Time.	Australia: Queensland.
EST-10EDT	Australian Eastern Standard Time, Australian Eastern Daylight Time.	Australia: New South Wales, Tasmania, Victoria.
NZST-12NZDT	New Zealand Standard Time, New Zealand Daylight Time.	New Zealand.

If TZ is not set, CURRENT-DATE assumes Eastern Standard Time (EST5EDT).

The time differential is automatically adjusted for daylight savings time according to the values in the time and zone adjustment table (the file TZTAB.LIB.SYS).

Note

Make sure your system administrator has correctly set the hardware clock. The hardware clock must be set to Greenwich Mean Time (Universal Coordinated Time or UTC) for CURRENT-DATE to return the correct local date and time. See the *System Startup, Configuration, and Shutdown Reference Manual* for how to set the hardware clock with the CLKUTIL utility.

Example

```
01  FULL-CURRENT-DATE.
    05  C-DATE.
        10  C-YEAR       PIC 9(4).
        10  C-MONTH      PIC 99.
        10  C-DAY        PIC 99.
    05  C-TIME.
        10  C-HOUR       PIC 99.
        10  C-MINUTES    PIC 99.
        10  C-SECONDS    PIC 99.
        10  C-SEC-HUND   PIC 99.
    05  C-TIME-DIFF.
        10  C-GMT-DIR    PIC X.
        10  C-HOUR       PIC 99.
        10  C-MINUTES    PIC 99.
                 :
MOVE FUNCTION CURRENT-DATE TO FULL-CURRENT-DATE.
DISPLAY "Full date is: ", FULL-CURRENT-DATE.
DISPLAY "Year is: ",      C-YEAR.
DISPLAY "Month is: ",     C-MONTH.
DISPLAY "Day is: ",       C-DAY.
DISPLAY "Hour is: ",      C-HOUR OF C-TIME.
DISPLAY "Minute is: ",    C-MINUTES OF C-TIME.
DISPLAY "Second is: ",    C-SECONDS.
DISPLAY "Hundredths of seconds is: ", C-SEC-HUND.
DISPLAY "Difference from GMT is: ", C-GMT-DIR.
DISPLAY "Hours from GMT is: ", C-HOUR OF C-TIME-DIFF.
DISPLAY "Minutes from GMT is: ", C-MINUTES OF C-TIME-DIFF.
```

With TZ set to `PST8PDT`, the above example displays the following:

```
Full date is: 1991022017152900-0800
Year is: 1991
Month is: 02
Day is: 20
Hour is: 17
Minute is: 15
Second is: 29
Hundredths of seconds is: 00
Difference from GMT is: -
Hours from GMT is: 08
Minutes from GMT is: 00
```

DATE-OF-INTEGER Function

The DATE-OF-INTEGER function converts a date in the Gregorian calendar from integer date form to standard date form (YYYYMMDD). The function type is integer.

Syntax

`FUNCTION DATE-OF-INTEGER` (*parameter-1*)

Parameters

parameter-1 Is a positive integer that represents a number of days succeeding December 31, 1600 in the Gregorian calendar.

Return Value

The returned value represents the ISO Standard date equivalent of the integer specified in *parameter-1*.

The returned value is in the form YYYYMMDD, where YYYY represents a year in the Gregorian calendar, MM represents the month of that year, and DD represents the day of that month.

For example, the value `19910723` represents July 23, 1991.

Example

The following example shows both the INTEGER-OF-DATE and DATE-OF-INTEGER functions. First, the year-month-day form of the date is converted to an integer using INTEGER-OF-DATE. This integer represents the number of days since December 31, 1600. Then, 30 is added to this number and it is converted back to the year-month-day form.

```
01  INT-DATE     PIC 9(8) VALUE ZERO.
01  DATE-TODAY   PIC 9(8) VALUE ZERO.
01  DUE-DATE     PIC 9(8) VALUE ZERO.
      ⋮
MOVE 19910220 TO DATE-TODAY.
COMPUTE INT-DATE = FUNCTION INTEGER-OF-DATE (DATE-TODAY).
DISPLAY DATE-TODAY.
DISPLAY INT-DATE.

ADD 30 TO INT-DATE.
COMPUTE DUE-DATE = FUNCTION DATE-OF-INTEGER (INT-DATE).
DISPLAY INT-DATE.
DISPLAY DUE-DATE.
```

The above example displays the following values. The first two lines represent the date February 20, 1991 and the last two lines represent March 22, 1991, 30 days later:

```
19910220
00142484
00142514
19910322
```

DAY-OF-INTEGER Function

The DAY-OF-INTEGER function converts a date in the Gregorian calendar from integer date form to Julian date form (YYYYDDD). The function type is integer.

Syntax

```
FUNCTION DAY-OF-INTEGER (parameter-1)
```

Parameters

parameter-1 Is a positive integer that represents a number of days succeeding December 31, 1600 in the Gregorian calendar.

Return Value

The returned value represents the Julian equivalent of the integer specified in *parameter-1*.

The returned value is an integer of the form YYYYDDD, where YYYY represents a year in the Gregorian calendar and DDD represents the day of that year.

Example

The following example shows both the INTEGER-OF-DAY and DAY-OF-INTEGER functions. First the year-day form of the date is converted to an integer using INTEGER-OF-DAY. This integer represents the number of days since December 31, 1600. Then 30 is added to the integer form and it is converted back to the year-day form using DAY-OF-INTEGER.

```
01  INT-DATE       PIC 9(8) VALUE ZERO.
01  DATE-TODAY     PIC 9(7) VALUE ZERO.
01  DUE-DATE       PIC 9(7) VALUE ZERO.
      ⋮
MOVE 1991051 TO DATE-TODAY.
COMPUTE INT-DATE = FUNCTION INTEGER-OF-DAY (DATE-TODAY).
DISPLAY DATE-TODAY.
DISPLAY INT-DATE.

ADD 30 TO INT-DATE.
COMPUTE DUE-DATE = FUNCTION DAY-OF-INTEGER (INT-DATE).
DISPLAY INT-DATE.
DISPLAY DUE-DATE.
```

The above example displays the following values. The first two lines represent the date February 20, 1991. February 20 is the 51st day of 1991. The last two lines represent March 22, 1991, 30 days later. March 22 is the 81st day of 1991:

```
1991051
00142484
00142514
1991081
```

FACTORIAL Function

The FACTORIAL function returns an integer that is the factorial of *parameter-1*. The function type is integer.

Syntax

```
FUNCTION FACTORIAL (parameter-1)
```

Parameters

parameter-1 Must be an integer greater than or equal to zero. (The largest value *parameter-1* can be is 20 in order for the result to fit in 18 digits.)

Return Value

If the value of *parameter-1* is zero, the value one is returned.

If the value of *parameter-1* is positive, the factorial of *parameter-1* is returned.

Example

```
77  NUM-FACTORIAL     PIC 9(5) VALUE ZERO.
      ⋮
COMPUTE NUM-FACTORIAL = FUNCTION FACTORIAL (4).
DISPLAY NUM-FACTORIAL.
```

The above example displays the following:

```
00024
```

INTEGER Function

The INTEGER function returns the greatest integer value that is less than or equal to the argument. The function type is integer.

Syntax

FUNCTION INTEGER (*parameter-1*)

Parameters

parameter-1 Must be class numeric.

Return Value

The returned value is the greatest integer less than or equal to the value of *parameter-1*. For example, if the value of *parameter-1* is -1.5, a value of -2 is returned. If the value of *parameter-1* is +1.5, the value of +1 is returned.

Example

```
77  NUM-FRACT    PIC S99V99 VALUE 12.94.
77  NUM-INT      PIC S99V99 VALUE ZERO.
77  NUM-NEG      PIC S99V99 VALUE -12.94.
      ⋮
COMPUTE NUM-INT = FUNCTION INTEGER (NUM-FRACT).
DISPLAY NUM-FRACT.
DISPLAY NUM-INT.

COMPUTE NUM-INT = FUNCTION INTEGER (NUM-NEG).
DISPLAY NUM-NEG.
DISPLAY NUM-INT.
```

The above example displays the following:

```
+12.94
+12.00
-12.94
-13.00
```

INTEGER-OF-DATE Function

The INTEGER-OF-DATE function converts a date in the Gregorian calendar from standard date form (YYYYMMDD) to integer date form. The function type is integer.

Syntax

```
FUNCTION INTEGER-OF-DATE (parameter-1)
```

Parameters

parameter-1 Must be an integer of the form YYYYMMDD, whose value is determined as follows:

(YYYY * 10000) + (MM * 100) + DD

where YYYY represents the year in the Gregorian calendar and must be an integer greater than 1600. MM represents a month and must be a positive integer less than thirteen. DD represents a day and must be a positive integer less than 32 ; DD must be valid for the specified month and year combination.

Return Value

The returned value is an integer that is the number of days the date represented by *parameter-1* succeeds December 31, 1600 in the Gregorian calendar.

Example

The following example shows both the INTEGER-OF-DATE and DATE-OF-INTEGER functions. First the year-month-day form of the date is converted to an integer using INTEGER-OF-DATE. This integer represents the number of days since December 31, 1600. Then 30 is added to this number and it is converted back to the year-month-day form.

```
01  INT-DATE     PIC 9(8) VALUE ZERO.
01  DATE-TODAY   PIC 9(8) VALUE ZERO.
01  DUE-DATE     PIC 9(8) VALUE ZERO.
      ⋮
MOVE 19910220 TO DATE-TODAY.
COMPUTE INT-DATE = FUNCTION INTEGER-OF-DATE (DATE-TODAY).
DISPLAY DATE-TODAY.
DISPLAY INT-DATE.

ADD 30 TO INT-DATE.
COMPUTE DUE-DATE = FUNCTION DATE-OF-INTEGER (INT-DATE).
DISPLAY INT-DATE.
DISPLAY DUE-DATE.
```

The above example displays the following values. The first two lines represent the date February 20, 1991 and the last two lines represent March 22, 1991, 30 days later:

```
19910220
00142484
00142514
19910322
```

INTEGER-OF-DAY Function

The INTEGER-OF-DAY function converts a date in the Gregorian calendar from Julian date form (YYYYDDD) to integer date form. The function type is integer.

Syntax

```
FUNCTION INTEGER-OF-DAY (parameter-1)
```

Parameters

parameter-1 Must be an integer of the form YYYYDDD, whose value is obtained as follows:

(YYYY * 1000) + DDD

where YYYY represents the year in the Gregorian calendar and must be an integer greater than 1600. DDD represents the day of the year and must be a positive integer less than 367. DDD must be valid for the year specified.

Return Value

The returned value is an integer that is the number of days the date represented by *parameter-1* succeeds December 31, 1600 in the Gregorian calendar.

Example

The following example shows both the INTEGER-OF-DAY and DAY-OF-INTEGER functions. First the year-day form of the date is converted to an integer using INTEGER-OF-DAY. This integer represents the number of days since December 31, 1600. Then 30 is added to the integer form and it is converted back to the year-day form using DAY-OF-INTEGER.

```
01  INT-DATE       PIC 9(8) VALUE ZERO.
01  DATE-TODAY     PIC 9(7) VALUE ZERO.
01  DUE-DATE       PIC 9(7) VALUE ZERO.
      :
MOVE 1991051 TO DATE-TODAY.
COMPUTE INT-DATE = FUNCTION INTEGER-OF-DAY (DATE-TODAY).
DISPLAY DATE-TODAY.
DISPLAY INT-DATE.

ADD 30 TO INT-DATE.
COMPUTE DUE-DATE = FUNCTION DAY-OF-INTEGER (INT-DATE).
DISPLAY INT-DATE.
DISPLAY DUE-DATE.
```

The above example displays the following values. The first two lines represent the date February 20, 1991. February 20 is the 51st day of 1991. The last two lines represent March 22, 1991, 30 days later. March 22 is the 81st day of 1991:

```
1991051
00142484
00142514
1991081
```

INTEGER-PART Function

The INTEGER-PART function returns an integer that is the integer portion of *parameter-1* (*parameter-1* is truncated). The function type is integer.

Syntax

FUNCTION INTEGER-PART (*parameter-1*)

Parameters

parameter-1 Must be class numeric.

Return Value

One of the following, depending on the value of *parameter-1*:

parameter-1	**Return Value**
0	0
Positive	The greatest integer less than or equal to the value of *parameter-1*. For example, if the value of *parameter-1* is +1.5, the value +1 is returned.
Negative	The least integer greater than or equal to the value of *parameter-1*. For example, if the value of *parameter-1* is -1.5, the value -1 is returned.

Example

```
77  NUM-INT      PIC S99V99 VALUE ZERO.
77  NUM-FRACT    PIC S99V99 VALUE 12.94.
77  NUM-NEG      PIC S99V99 VALUE -12.94.
      ⋮
COMPUTE NUM-INT = FUNCTION INTEGER-PART (NUM-FRACT).
DISPLAY NUM-FRACT.
DISPLAY NUM-INT.

COMPUTE NUM-INT = FUNCTION INTEGER-PART (NUM-NEG).
DISPLAY NUM-NEG.
DISPLAY NUM-INT.
```

The above example displays the following:

```
+12.94
+12.00
-12.94
-12.00
```

LENGTH Function

The LENGTH function returns an integer equal to the length of the argument in character positions (bytes). To conform to ANSI standard COBOL, you can use the LENGTH function instead of the .LEN. pseudo-intrinsic (see Chapter 11, "Interprogram Communication," for details on .LEN.). The function type is integer.

Syntax

FUNCTION LENGTH (*parameter-1*)

Parameters

parameter-1 A nonnumeric literal or a data item of any class or category.

If *parameter-1* or any data item subordinate to *parameter-1* is described with the DEPENDING phrase of the OCCURS clause, the contents of the data item referenced by the data-name specified in the DEPENDING phrase are used at the time the LENGTH function is evaluated.

Return Values

If *parameter-1* is a nonnumeric literal or an elementary data item or *parameter-1* is a group data item that does not contain a variable occurrence data item, the value returned is an integer equal to the length of *parameter-1* in character positions.

If *parameter-1* is a group data item containing a variable occurrence data item, the returned value is an integer determined by evaluation of the data item specified in the DEPENDING phrase of the OCCURS clause for that variable occurrence data item. This evaluation is accomplished according to the rules in the OCCURS clause dealing with the data item as a sending data item. See the OCCURS clause for additional information.

The returned value includes implicit filler items, if any.

Example 1

```
77  CITY        PIC X(9) VALUE "CHICAGO".
77  ID-LENGTH   PIC 99 VALUE ZERO.
     ⋮
COMPUTE ID-LENGTH = FUNCTION LENGTH (CITY).
DISPLAY CITY.
DISPLAY ID-LENGTH.
```

The above example displays the following:

```
CHICAGO␣␣
09
```

Example 2

```
77  SIZER     PIC 99.
77  NUM       PIC 999.
01  TAB-REC.
    05  TAB-ELEMENT OCCURS 1 TO 10 TIMES
                    DEPENDING ON SIZER.
        10  TAB-ITEM-1  PIC X(3).
        10  TAB-ITEM-2  PIC S9(9) COMP SYNC.
PROCEDURE DIVISION.
010-PARA.
   MOVE 5 TO SIZER.
   COMPUTE NUM = FUNCTION LENGTH (TAB-ELEMENT (1)).
   DISPLAY "TAB-ELEMENT LENGTH = " NUM.
   COMPUTE NUM = FUNCTION LENGTH (TAB-REC).
   DISPLAY "TAB-REC LENGTH = " NUM.
   STOP RUN.
```

The above example displays the following:

```
TAB-ELEMENT LENGTH = 008
TAB-REC LENGTH = 040
```

The length of TAB-ELEMENT is 8 because of the implicit 1-byte filler between TAB-ITEM-1 and TAB-ITEM-2. If SYNC is removed from TAB-ITEM-2, the length of TAB-ELEMENT is 7.

LOG Function

The LOG function returns a numeric value that is the logarithm to the base e (natural log) of *parameter-1*. The function type is numeric.

Syntax

FUNCTION LOG (*parameter-1*)

Parameters

parameter-1 Must be class numeric and greater than zero.

Return Value

The value returned is the approximation of the logarithm to the base e of *parameter-1*.

Example

```
77  NUM-LOG    PIC 9(3)V9(5) VALUE ZERO.
      ⋮
COMPUTE NUM-LOG = FUNCTION LOG (10).
DISPLAY NUM-LOG.
```

The above example displays the following:

```
002.30258
```

LOG10 Function

The LOG10 function returns a numeric value that is the logarithm to the base 10 of *parameter-1*. The function type is numeric.

Syntax

FUNCTION LOG10 (*parameter-1*)

Parameters

parameter-1 Must be class numeric and greater than zero.

Return Value

The value returned is the approximation of the logarithm to the base 10 of *parameter-1*.

Example

```
77  NUM-LOG10    PIC 9(3)V9(5) VALUE ZERO.
      ⋮
COMPUTE NUM-LOG10 = FUNCTION LOG10 (50).
DISPLAY NUM-LOG10.
```

The above example displays the following:

```
001.69897
```

LOWER-CASE Function

The LOWER-CASE function returns a character string that is the same length as *parameter-1* with each uppercase letter replaced by the corresponding lowercase letter. The function type is alphanumeric.

Syntax

```
FUNCTION LOWER-CASE (parameter-1)
```

Parameters

parameter-1 Must be class alphabetic or alphanumeric and must be at least one character in length.

Return Value

The value returned is the same character string as *parameter-1*, except that each uppercase letter is replaced by the corresponding lowercase letter.

The character string returned has the same length as *parameter-1*.

Example

```
77  CITY      PIC X(7) VALUE "CHICAGO".
      ⋮
DISPLAY CITY.
DISPLAY FUNCTION LOWER-CASE (CITY).
```

The above example displays the following:

```
CHICAGO
chicago
```

MAX Function

The MAX function returns the content of the *parameter-1* that contains the maximum value. The function type depends on the parameter type, as follows:

Parameter Type	Function Type
Alphabetic	Alphanumeric
Alphanumeric	Alphanumeric
All parameters integer	Integer
Numeric (some parameters may be integer)	Numeric

Syntax

```
FUNCTION MAX ({parameter-1}...)
```

Parameters

parameter-1 If more than one *parameter-1* is specified, all parameters must be of the same class.

Return Values

The returned value is the content of the *parameter-1* having the greatest value. The greatest values are determined by the rules for simple conditions. See the section "Simple Conditions" in Chapter 8 for additional information.

If more than one *parameter-1* has the same greatest value, the content of the *parameter-1* returned is the leftmost *parameter-1* having that value.

If the type of the function is alphanumeric, the size of the returned value is the same as the size of the selected *parameter-1*.

Example

```
77  A          PIC X VALUE "A".
77  B          PIC X VALUE "Z".
77  C          PIC X VALUE "m".
77  D          PIC X VALUE "9".

77  I             PIC 9 VALUE 8.
77  J             PIC 9 VALUE 3.
77  K             PIC 9 VALUE 6.
77  L             PIC 9 VALUE 1.
77  MAX-VALUE    PIC 9 VALUE ZERO.
77  MAX-VALUE-2  PIC S999V99 VALUE 0.
01  TAB.
    05  ELEMENT   PIC S999V99
                    OCCURS 4 TIMES VALUE ZERO.
     ⋮
DISPLAY FUNCTION MAX (A B C D).

COMPUTE MAX-VALUE = FUNCTION MAX (I J K L).
DISPLAY MAX-VALUE.

MOVE 1.25 TO ELEMENT (1).
MOVE 3.50 TO ELEMENT (2).
MOVE 8.75 TO ELEMENT (3).
MOVE 0.25 TO ELEMENT (4).

COMPUTE MAX-VALUE-2 = FUNCTION MAX ( ELEMENT (ALL) ).
DISPLAY MAX-VALUE-2.
```

The above example displays the following:

```
m
8
+008.75
```

MEAN Function

The MEAN function returns a numeric value that is the arithmetic mean (average) of its parameters. The function type is numeric.

Syntax

```
FUNCTION MEAN ({parameter-1}...)
```

Parameters

parameter-1 Must be class numeric.

Return Value

The value returned is the arithmetic mean of the *parameter-1* series.

The value returned is defined as the sum of the *parameter-1* series divided by the number of occurrences referenced by *parameter-1*.

Example

```
01  TAB.
    05  ELEMENT   PIC S999V99
                     OCCURS 4 TIMES VALUE ZERO.
01  MEAN-VALUE    PIC S999V99 VALUE ZERO.
     ⋮
MOVE 1.25 TO ELEMENT (1)
MOVE 3.50 TO ELEMENT (2)
MOVE 8.75 TO ELEMENT (3)
MOVE 0.25 TO ELEMENT (4)

COMPUTE MEAN-VALUE = FUNCTION MEAN (1 7 9 23 85)
DISPLAY MEAN-VALUE
COMPUTE MEAN-VALUE = FUNCTION MEAN ( ELEMENT (ALL) )
DISPLAY MEAN-VALUE
```

The above example displays the following:

```
+025.00
+003.43
```

MEDIAN Function

The MEDIAN function returns the content of the parameter whose value is the middle value in a list formed by arranging the parameters in sorted order. The function type is numeric.

Syntax

```
FUNCTION MEDIAN ({parameter-1}...)
```

Parameters

parameter-1 Must be class numeric.

Return Value

The value returned is the content of *parameter-1* having the middle value in a list formed by arranging all the *parameter-1* values in sorted order.

If the number of occurrences referenced by *parameter-1* is odd, the returned value is such that at least half of the occurrences referenced by *parameter-1* are greater than or equal to the returned value and at least half are less than or equal. If the number of occurrencces referenced by *parameter-1* is even, the returned value is the arithmetic mean of the values referenced by the two middle occurrences.

The comparisons used to arrange the *parameter-1* values in sorted order are made according to the rules for simple conditions. See the section "Simple Conditions" in Chapter 8 for additional information.

Example

```
01  TAB.
    05  ELEMENT   PIC S999V99
                     OCCURS 4 TIMES VALUE ZERO.
01  NUM-MEDIAN    PIC S999V99 VALUE ZERO.
     ⋮
MOVE 1.25 TO ELEMENT (1).
MOVE 3.50 TO ELEMENT (2).
MOVE 8.75 TO ELEMENT (3).
MOVE 0.25 TO ELEMENT (4).

COMPUTE NUM-MEDIAN = FUNCTION MEDIAN (1, 7, 9, 23, 85).
DISPLAY NUM-MEDIAN.
COMPUTE NUM-MEDIAN = FUNCTION MEDIAN ( ELEMENT (ALL) ).
DISPLAY NUM-MEDIAN.
```

The above example displays the following:

```
+009.00
+002.37
```

MIDRANGE Function

The MIDRANGE (middle range) function returns a numeric value that is the arithmetic mean (average) of the value of the minimum parameter and the maximum parameter. The function type is numeric.

Syntax

```
FUNCTION MIDRANGE ({parameter-1}...)
```

Parameters

parameter-1 Must be class numeric.

Return Value

The returned value is the arithmetic mean of the greatest *parameter-1* value and the least *parameter-1* value. The comparisons used to determine the greatest and least values are made according to the rules for simple conditions. See the section "Simple Conditions" in Chapter 8 for additional information.

Example

```
01  TAB.
    05  ELEMENT   PIC S999V99
                     OCCURS 4 TIMES VALUE ZERO.
01  NUM-MIDRANGE  PIC S999V99 VALUE ZERO.
     ⋮
MOVE 1.25 TO ELEMENT (1)
MOVE 3.50 TO ELEMENT (2)
MOVE 8.75 TO ELEMENT (3)
MOVE 0.25 TO ELEMENT (4)

COMPUTE NUM-MIDRANGE = FUNCTION MIDRANGE (1 7 9 23 85).
DISPLAY NUM-MIDRANGE
COMPUTE NUM-MIDRANGE = FUNCTION MIDRANGE ( ELEMENT (ALL) )
DISPLAY NUM-MIDRANGE
```

The above example displays the following:

```
+043.00
+004.50
```

MIN Function

The MIN function returns the content of the *parameter-1* that contains the minimum value. The function type depends on the parameter type, as follows:

Parameter Type	Function Type
Alphabetic	Alphanumeric
Alphanumeric	Alphanumeric
All parameters integer	Integer
Numeric (some parameters may be integer)	Numeric

Syntax

```
FUNCTION MIN ({parameter-1}...)
```

Parameters

parameter-1 If more than one *parameter-1* is specified, all parameters must be of the same class.

Return Value

The returned value is the content of the *parameter-1* having the least value. The comparisons used to determine the least value are made according to the rules for simple conditions. See the section "Simple Conditions" in Chapter 8 for additional information.

If more than one *parameter-1* has the same least value, the content of the *parameter-1* returned is the leftmost *parameter-1* having that value.

If the type of the function is alphanumeric, the size of the returned value is the same as the size of the selected *parameter-1*.

Example

```
77  A         PIC X VALUE "A".
77  B         PIC X VALUE "Z".
77  C         PIC X VALUE "m".
77  D         PIC X VALUE "9".

77  I         PIC 9 VALUE 8.
77  J         PIC 9 VALUE 3.
77  K         PIC 9 VALUE 6.
77  L         PIC 9 VALUE 1.
77  MIN-VALUE PIC 9 VALUE ZERO.
01  TAB.
    05  ELEMENT   PIC S999V99
                    OCCURS 4 TIMES VALUE ZERO.
      ⋮
DISPLAY FUNCTION MIN (A, B, C, D).
COMPUTE MIN-VALUE = FUNCTION MIN (I, J, K, L).
DISPLAY MIN-VALUE.
MOVE 1.25 TO ELEMENT (1).
MOVE 3.50 TO ELEMENT (2).
MOVE 8.75 TO ELEMENT (3).
MOVE 0.25 TO ELEMENT (4).

COMPUTE MIN-VALUE = FUNCTION MIN ( ELEMENT (ALL) ).
DISPLAY MIN-VALUE.
```

The above example displays the following:

```
9
1
0
```

MOD Function

The MOD function returns an integer value that is *parameter-1* modulo *parameter-2*. The function type is integer.

Syntax

FUNCTION MOD (*parameter-1 parameter-2*)

Parameters

parameter-1 Must be an integer.

parameter-2 Must be a non-zero integer.

Return Value

The returned value is *parameter-1* modulo *parameter-2*. The returned value is defined as:

parameter-1 - (*parameter-2* * FUNCTION INTEGER(*parameter-1*/*parameter-2*))

The following table illustrates expected results for some values of *parameter-1* and *parameter-2*.

Parameter-1	*Parameter-2*	**Result**
11	5	1
-11	5	4
11	-5	-4
-11	-5	-1

Example

```
01  NUM-MOD     PIC 99 VALUE 0.
      ⋮
COMPUTE NUM-MOD = FUNCTION MOD (12 5).
DISPLAY NUM-MOD.
```

The above example displays the following:

```
02
```

NUMVAL Function

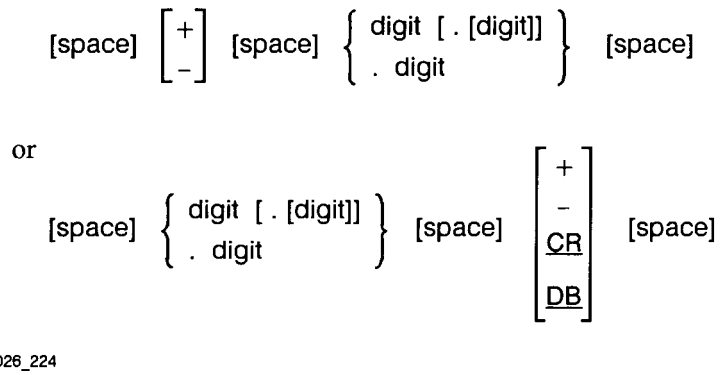
The NUMVAL function returns the numeric value represented by the character string specified by *parameter-1*. Leading and trailing spaces are ignored. The function type is numeric.

Syntax

```
FUNCTION NUMVAL (parameter-1)
```

Parameters

parameter-1 Must be a nonumeric literal or alphanumeric data item whose content has one of the following formats:



LG200026_224

where `space` is a string of zero or more spaces and `digits` is a string of one to eighteen digits. The total number of digits in *parameter-1* must not exceed 18.

If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, a comma must be used in *parameter-1* rather than a decimal point.

Return Value

The returned value is the numeric value represented by *parameter-1*.

Example

```
77  ALPHA-NUM     PIC XXX VALUE "753".
77  NUM           PIC 999 VALUE ZERO.
      ⋮
COMPUTE NUM = FUNCTION NUMVAL (ALPHA-NUM).
DISPLAY ALPHA-NUM.
DISPLAY NUM.
```

The above example displays the following:

```
753
753
```

NUMVAL-C Function

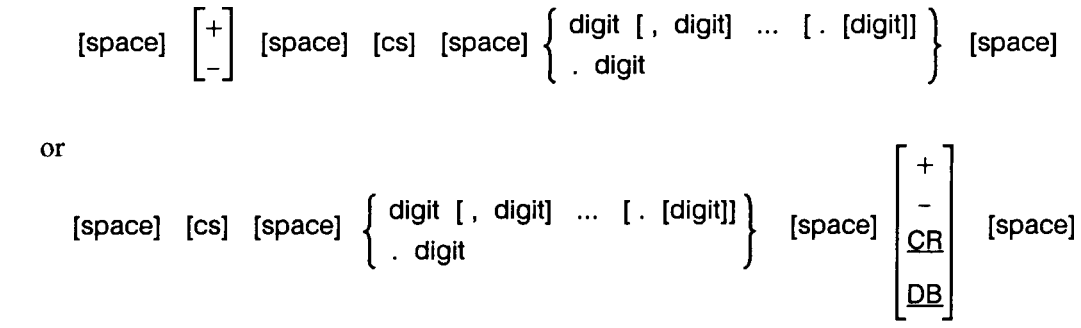
The NUMVAL-C function returns the numeric value represented by the character string specified by *parameter-1*. Any optional currency sign specified by *parameter-2* and any optional commas preceding the decimal point are ignored. The function type is numeric.

Syntax

`FUNCTION NUMVAL-C` (*parameter-1* {*parameter-2*})

Parameters

parameter-1 Must be a nonnumeric literal or alphanumeric data item whose content has one of the following formats:



LG200026_225

where `space` is a string of zero or more spaces, `cs` is the string of one or more characters specified by *parameter-2*, and `digit` is a string of one or more digits.

If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, the functions of the comma and decimal point in *parameter-1* are reversed.

The total number of digits in *parameter-1* must not exceed 18.

parameter-2 If specified, must be a nonnumeric literal or alphanumeric data item.

If *parameter-2* is not specified, the character used for cs is the currency symbol specified for the program.

Return Value

The returned value is the numeric value represented by *parameter-1*.

Example 1

```
77  ALPHA-NUM  PIC X(16) VALUE "$ 123.45 CR ".
77  NUM        PIC S9(5)V99.
PROCEDURE DIVISION.
010-PARA.
   COMPUTE NUM = FUNCTION NUMVAL-C (ALPHA-NUM).
   DISPLAY ALPHA-NUM.
   DISPLAY NUM.
   STOP RUN.
```

The above example displays the following:

```
$ 123.45 CR
-00123.45
```

Example 2

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
   DECIMAL-POINT IS COMMA.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  ALPHA-NUM  PIC X(16) VALUE SPACES.
77  NUM        PIC S9(5)V99.
PROCEDURE DIVISION.
010-PARA.
   MOVE "DM 1.150,25" TO ALPHA-NUM.
   COMPUTE NUM = FUNCTION NUMVAL-C (ALPHA-NUM "DM").
   DISPLAY ALPHA-NUM.
   DISPLAY NUM.
   STOP RUN.
```

The above example displays the following:

```
DM 1.150,25
+01150,25
```

ORD Function

The ORD function returns an integer value that is the ordinal position of *parameter-1* in the collating sequence for the program. The lowest ordinal position is 1. The function type is integer.

Syntax

```
FUNCTION ORD (parameter-1)
```

Parameters

parameter-1 Must be one character in length and must be class alphabetic or alphanumeric.

Return Value

The returned value is the ordinal position of *parameter-1* in the collating sequence for the program.

Example

```
77  NUM     PIC 999 VALUE ZERO.
     ⋮
COMPUTE NUM = FUNCTION ORD ("A").
DISPLAY NUM.
```

The above example displays the following:

```
066
```

ORD-MAX Function

The ORD-MAX function returns a value that is the ordinal number of the *parameter-1* that contains the maximum value. The function type is integer.

Syntax

```
FUNCTION ORD-MAX ({parameter-1}...)
```

Parameters

parameter-1 If more than one *parameter-1* is specified, all parameters must be of the same class.

Return Value

The returned value is the ordinal number that corresponds to the position of the *parameter-1* having the greatest value in the *parameter-1* series.

The comparisons used to determine the greatest valued argument are made according to the rules for simple conditions. See the section "Simple Conditions" in Chapter 8 for additional information.

If more than one *parameter-1* has the same greatest value, the number returned corresponds to the position of the leftmost *parameter-1* having that value.

Example

```
77  NUM     PIC 999 VALUE ZERO.
     ⋮
COMPUTE NUM = FUNCTION ORD-MAX ("M", "C", "Z", "A", "M").
DISPLAY NUM.
```

The above example displays the following. The greatest value is "Z", which is in the third position:

```
003
```

ORD-MIN Function

The ORD-MIN function returns a value that is the ordinal number of the argument that contains the minimum value. The function type is integer.

Syntax

```
FUNCTION ORD-MIN ({parameter-1}...)
```

Parameters

parameter-1 If more than one *parameter-1* is specified, all arguments must be of the same class.

Return Value

The returned value is the ordinal number that corresponds to the position of the *parameter-1* having the least value in the *parameter-1* series.

The comparisons used to determine the least valued *parameter-1* are made according to the rules for simple conditions. See the section "Simple Conditions" in Chapter 8 for additional information.

If more than one *parameter-1* has the same least value, the number returned corresponds to the positions of the leftmost *parameter-1* having that value.

Example

```
77  NUM     PIC 999 VALUE ZERO.
     ⋮
COMPUTE NUM = FUNCTION ORD-MIN ("M", "C", "Z", "A", "M").
DISPLAY NUM.
```

The above example displays the following. The least value is "A", which is in the fourth position:

```
004
```

PRESENT-VALUE Function

The PRESENT-VALUE function returns a value that approximates the present value of a series of future period-end amounts specified by *parameter-2* at a discount rate specified by *parameter-1*. The function type is numeric.

Syntax

FUNCTION PRESENT-VALUE (*parameter-1* {*parameter-2*}...)

Parameters

parameter-1 Must be of the class numeric; must be greater than -1.

parameter-2 Must be of the class numeric.

Return Value

The returned value is an approximation of the summation of a series of calculations with each term in the following form:

$$\frac{\textit{parameter-2}}{(1 + \textit{parameter-1})\ **\ \text{n}}$$

There is one term for each occurrence of *parameter-2*. The exponent n is incremented one by one for each term in the series.

Example

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  NUM-RATE       PIC S9V9999 VALUE 0.08.
01  TAB.
    05  ELEMENT  PIC S9999V99
                 OCCURS 4 TIMES VALUE ZERO.
01  NUM-P-VAL    PIC S9(6)V99.
PROCEDURE DIVISION.
010-PARA.
   MOVE 2012.54 TO ELEMENT (1)
   MOVE 2008.29 TO ELEMENT (2)
   MOVE 2015.05 TO ELEMENT (3)
   MOVE 2014.87 TO ELEMENT (4)
   COMPUTE NUM-P-VAL ROUNDED =
       FUNCTION PRESENT-VALUE ( NUM-RATE ELEMENT (ALL) )
   DISPLAY NUM-RATE
   DISPLAY NUM-P-VAL.
```

The above example displays the following:

```
+0.0800
+006665.85
```

RANDOM Function

The RANDOM function returns a numeric value that is a pseudo-random number for a rectangular distribution. The function type is numeric.

Syntax

```
FUNCTION RANDOM [(parameter-1)]
```

Parameters

parameter-1 *parameter-1* is used as the seed value to generate a sequence of pseudo-random numbers. If specified, must be in integer between zero and 999999999, inclusive.

If a subsequent reference specifies *parameter-1*, a new sequence of pseudo-random numbers is started.

If the first reference to this function in the run unit does not specify *parameter-1*, the seed value is zero. In each case, subsequent references without specifying *parameter-1* returns the next number in the current sequence.

If *parameter-1* is specified, *parameter-1* determines a sequence of pseudo-random numbers. Each time a value is returned, that 32-bit value is used in place of *parameter-1* to continue the pseudo-random sequence.

For details on the RANDOM function, refer to the *Compiler Library/XL Reference Manual*.

Return Value

The returned value is greater than or equal to zero and less than one. For a given seed value, the sequence of pseudo-random numbers is always the same.

Example

```
77  RANDOM-NUMBER    PIC V99999 VALUE ZERO.
      ⋮
COMPUTE RANDOM-NUMBER = FUNCTION RANDOM (1)
DISPLAY RANDOM-NUMBER
COMPUTE RANDOM-NUMBER = FUNCTION RANDOM
DISPLAY RANDOM-NUMBER
COMPUTE RANDOM-NUMBER = FUNCTION RANDOM
DISPLAY RANDOM-NUMBER

COMPUTE RANDOM-NUMBER = FUNCTION RANDOM (2893)
DISPLAY RANDOM-NUMBER
COMPUTE RANDOM-NUMBER = FUNCTION RANDOM
DISPLAY RANDOM-NUMBER
COMPUTE RANDOM-NUMBER = FUNCTION RANDOM
DISPLAY RANDOM-NUMBER
```

The above example displays the following:

```
.04163
.36480
.73704
.40190
.06571
.61402
```

RANGE Function

The RANGE function returns a value that is equal to the value of the maximum parameter minus the value of the minimum parameter. The function type is either integer or numeric depending on the type of the parameters, as follows:

Parameter Type	Function Type
All parameters integer	Integer
Numeric (some parameters may be integer)	Numeric

Syntax

```
FUNCTION RANGE ({parameter-1}...)
```

Parameters

parameter-1 Must be class numeric.

Return Value

The returned value is equal to the greatest value of *parameter-1* minus the least value of *parameter-1*. The comparisons used to determine the greatest and least values are made according to the rules for simple conditions. See the section "Simple Conditions" in Chapter 8 for additional information.

Example

```
01  TAB.
    05  ELEMENT   PIC S999V99
                     OCCURS 4 TIMES VALUE ZERO.
01  NUM-RANGE     PIC S999V99 VALUE ZERO.
     .
     .
     .
MOVE 1.25 TO ELEMENT (1).
MOVE 3.50 TO ELEMENT (2).
MOVE 8.75 TO ELEMENT (3).
MOVE 0.25 TO ELEMENT (4).

COMPUTE NUM-RANGE = FUNCTION RANGE (1, 7, 9, 23, 85).
DISPLAY NUM-RANGE.
COMPUTE NUM-RANGE = FUNCTION RANGE ( ELEMENT (ALL) ).
DISPLAY NUM-RANGE.
```

The above example displays the following:

```
+084.00
+008.50
```

REM Function

The REM function returns a numeric value that is the remainder of *parameter-1* divided by *parameter-2*. The function type is numeric.

Syntax

FUNCTION REM (*parameter-1 parameter-2*)

Parameters

parameter-1 Must be class numeric.

parameter-2 Must be class numeric; must not be zero.

Return Value

The returned value is the remainder of *parameter-1 / parameter-2*. The value is defined as the following expression:

parameter-1 - (*parameter-2* * FUNCTION INTEGER-PART (*parameter-1 / parameter-2*))

The following table illustrates expected results for some values of *parameter-1* and *parameter-2*.

Parameter-1	*Parameter-2*	**Result**
13	6	+1.0
13.55	-6	+1.55
-13	6	-1.0
-13.55	-6	-1.55

Example

```
01  NUM-REM     PIC 99 VALUE 0.
      ⋮
COMPUTE NUM-REM = FUNCTION REM (17 3).
DISPLAY NUM-REM.
```

The above example displays the following:

```
02
```

REVERSE Function

The REVERSE function returns a character string of exactly the same length as *parameter-1* and whose characters are exactly the same as those of *parameter-1*, except that they are in reverse order. The function type is alphanumeric.

Syntax

FUNCTION REVERSE (*parameter-1*)

Parameters

parameter-1 Must be class alphabetic or alphanumeric and must be at least one character in length.

Return Value

If *parameter-1* is a character string of length n, the returned value is a character string of length n such that for $1 \leq j \leq n$, the character in position j of the returned value is the character from position n-j+1 of *parameter-1*.

Example

```
77  CITY        PIC X(7) VALUE "CHICAGO".
     ⋮
DISPLAY CITY.
DISPLAY FUNCTION REVERSE (CITY).
```

The above example displays the following:

```
CHICAGO
OGACIHC
```

SIN Function

The SIN function returns the sine of an angle, expressed in radians. The function type is numeric.

Syntax

FUNCTION SIN (*parameter-1*)

Parameters

parameter-1 The size of an angle in radians. Must be class numeric.

Return Value

The value returned is the approximation of the sine of *parameter-1* and is between -1 and 1, inclusive. The value returned is numeric.

Example

```
77  ANGLE-RADIANS      PIC S99V9(5) VALUE 3.14159.
77  SIN-OF-ANGLE       PIC S9V9(5)  VALUE ZERO.
        ⋮
COMPUTE SIN-OF-ANGLE = FUNCTION SIN (ANGLE-RADIANS).
DISPLAY ANGLE-RADIANS.
DISPLAY SIN-OF-ANGLE.

DIVIDE ANGLE-RADIANS BY 4 GIVING ANGLE-RADIANS.
COMPUTE SIN-OF-ANGLE = FUNCTION SIN (ANGLE-RADIANS).
DISPLAY ANGLE-RADIANS.
DISPLAY SIN-OF-ANGLE.
```

The above example displays the following:

```
+03.14159
+0.00000
+00.78539
+0.70710
```

SQRT Function

The SQRT function returns a numeric value that is the square root of *parameter-1*. The function type is numeric.

Syntax

FUNCTION SQRT (*parameter-1*)

Parameters

parameter-1 Must be class numeric; the value must be zero or positive.

Return Value

The returned value is the absolute value of the approximation of the square root of *parameter-1*.

Example

```
01  A-NUMBER        PIC 99      VALUE 35.
01  NUM-SQ-ROOT     PIC 999V999 VALUE ZERO.
     ⋮
COMPUTE NUM-SQ-ROOT = FUNCTION SQRT (81).
DISPLAY NUM-SQ-ROOT.
COMPUTE NUM-SQ-ROOT = FUNCTION SQRT (A-NUMBER).
DISPLAY NUM-SQ-ROOT.
```

The above example displays the following:

```
009.00
005.916
```

STANDARD-DEVIATION Function

The STANDARD-DEVIATION function returns a numeric value that approximates the standard deviation of its arguments. The function type is numeric.

Syntax

```
FUNCTION STANDARD-DEVIATION ({parameter-1}...)
```

Parameters

parameter-1 Must be class numeric.

Return Value

The returned value is the approximation of the standard deviation of the *parameter-1* series.

The returned value is calculated as follows:

1. The difference between each *parameter-1* value and the arithmetic mean of the *parameter-1* series is calculated and squared.
2. The values obtained are added together. This quantity is divided by the number of values in the *parameter-1* series.
3. The square root of the quotient obtained is calculated. The returned value is the absolute value of this square root.

If the *parameter-1* series consists of only one value, or if the *parameter-1* series consists of all variable occurrence data items and the total number of occurrences for all of them is one, the returned value is zero.

Example

```
01  TAB.
    05  ELEMENT   PIC S999V99
                     OCCURS 4 TIMES VALUE ZERO.
01  NUM-STD-DEV   PIC S999V99 VALUE ZERO.
     ⋮
MOVE 1.25 TO ELEMENT (1).
MOVE 3.50 TO ELEMENT (2).
MOVE 8.75 TO ELEMENT (3).
MOVE 0.25 TO ELEMENT (4).

COMPUTE NUM-STD-DEV = FUNCTION STANDARD-DEVIATION (1, 7, 9, 23, 85).
DISPLAY NUM-STD-DEV.
COMPUTE NUM-STD-DEV = FUNCTION STANDARD-DEVIATION ( ELEMENT (ALL) ).
DISPLAY NUM-STD-DEV.
```

The above example displays the following:

```
+030.85
+003.28
```

SUM Function

The SUM function returns a value that is the sum of the parameters. The function type is either integer or numeric depending on the type of the parameters, as follows:

Parameter Type	Function Type
All parameters integer	Integer
Numeric (some parameters may be integer)	Numeric

Syntax

```
FUNCTION SUM ({parameter-1}...)
```

Parameters

parameter-1 Must be class numeric.

Return Value

The returned value is the sum of the parameters.

If the *parameter-1* series are all integers, the value returned is an integer.

If the *parameter-1* series are not all integers, a numeric value is returned.

Example

```
01  TAB.
    05  ELEMENT   PIC S999V99
                     OCCURS 4 TIMES VALUE ZERO.
01  TOTAL         PIC S999V99 VALUE ZERO.
     ⋮
MOVE 1.25 TO ELEMENT (1).
MOVE 3.50 TO ELEMENT (2).
MOVE 8.75 TO ELEMENT (3).
MOVE 0.25 TO ELEMENT (4).

COMPUTE TOTAL = FUNCTION SUM (1, 7, 9, 23, 85).
DISPLAY TOTAL.
COMPUTE TOTAL = FUNCTION SUM ( ELEMENT (ALL) ).
DISPLAY TOTAL.
```

The above example displays the following:

```
+125.00
+013.75
```

TAN Function

The TAN function returns the tangent of an angle, expressed in radians. The function type is numeric.

Syntax

FUNCTION TAN (*parameter-1*)

Parameters

parameter-1 The size of an angle in radians. Must be class numeric.

Return Value

The value returned is the approximation of the tangent of *parameter-1*. The value returned is numeric.

Example

```
77  ANGLE-RADIANS      PIC S99V9(5)   VALUE 3.14159.
77  TAN-OF-ANGLE       PIC S9(5)V9(5) VALUE ZERO.
        ⋮
COMPUTE TAN-OF-ANGLE = FUNCTION TAN (ANGLE-RADIANS).
DISPLAY ANGLE-RADIANS.
DISPLAY TAN-OF-ANGLE.

DIVIDE ANGLE-RADIANS BY 4 GIVING ANGLE-RADIANS.
COMPUTE TAN-OF-ANGLE = FUNCTION TAN (ANGLE-RADIANS).
DISPLAY ANGLE-RADIANS.
DISPLAY TAN-OF-ANGLE.
```

The above example displays the following:

```
+03.14159
+00000.00000
+00.78539
+00000.99998
```

UPPER-CASE Function

The UPPER-CASE function returns a character string that is the same length as *parameter-1* with each lowercase letter replaced by the corresponding uppercase letter. The function type is alphanumeric.

Syntax

```
FUNCTION UPPER-CASE (parameter-1)
```

Parameters

parameter-1 Must be class alphabetic or alphanumeric and must be at least one character in length.

Return Value

The same character string as *parameter-1* is returned, except that each lowercase letter is replaced by the corresponding uppercase letter.

The character string returned has the same length as *parameter-1*.

Example

```
77  CITY      PIC X(7) VALUE "chicago".
      ⋮
DISPLAY CITY.
MOVE FUNCTION UPPER-CASE (CITY) TO CITY.
DISPLAY CITY.
```

The above example displays the following:

```
chicago
CHICAGO
```

VARIANCE Function

The VARIANCE function returns a numeric value that approximates the variance of its parameters. The function type is numeric.

Syntax

```
FUNCTION VARIANCE ({parameter-1}...)
```

Parameters

parameter-1 Must be class numeric.

Return Value

The returned value is the approximation of the variance of the *parameter-1* series.

The returned value is defined as the square of the standard deviation of the *parameter-1* series. See the description of the STANDARD-DEVIATION function in this chapter for additional information.

If the *parameter-1* series consists of only one value or if the *parameter-1* series consists of all variable occurrence data items and the total number of occurrences for all of them is one, the returned value is zero.

Example

```
01  TAB.
    05  ELEMENT   PIC S999V99
                     OCCURS 4 TIMES VALUE ZERO.
01  NUM-VARIANCE  PIC S999V99 VALUE ZERO.
     ⋮
MOVE 1.25 TO ELEMENT (1).
MOVE 3.50 TO ELEMENT (2).
MOVE 8.75 TO ELEMENT (3).
MOVE 0.25 TO ELEMENT (4).

COMPUTE NUM-VARIANCE = FUNCTION VARIANCE (1, 7, 9, 23, 85).
DISPLAY NUM-VARIANCE.
COMPUTE NUM-VARIANCE = FUNCTION VARIANCE ( ELEMENT (ALL) ).
DISPLAY NUM-VARIANCE.
```

The above example displays the following:

```
+952.00
+010.79
```

WHEN-COMPILED Function

The WHEN-COMPILED function returns the date and time the system compiled your program and the difference between the local time when your program was compiled and Universal Coordinated Time (UTC) or Greenwich Mean Time. To get the correct time differential, you need to set the environment variable TZ to your local time zone before compiling programs that contain this function. See below for more information. The function type is alphanumeric.

This function is different from the WHEN-COMPILED special register word (described in Chapter 3). One difference is that the WHEN-COMPILED function provides a four-digit year.

Syntax

```
FUNCTION WHEN-COMPILED
```

Return Value

This function returns a 19-character alphanumeric string with each character position defined as follows:

Character Positions	Contents
1-4	Four numeric digits of the year in the Gregorian calendar.
5-6	Two numeric digits of the month of the year, in the range 01 through 12.
7-8	Two numeric digits of the day of the month, in the range 01 through 31.
9-10	Two numeric digits of the hours past midnight, in the range 00 through 23.
11-12	Two numeric digits of the minutes past the hour, in the range 00 through 59.
13-14	Two numeric digits of the seconds past the minute, in the range 00 through 59.
15-16	Two numeric digits of the hundredths of a second past the second, in the range 00 through 99. The value 00 is returned because your system cannot provide the fractional part of a second.
17	One of the following:

Value	When Returned
-	Returned if the local time of compilation in the previous character positions is behind Greenwich Mean Time.
+	Returned if the local time indicated is the same as or ahead of Greenwich Mean Time.
0	Returned on non-MPE XL systems that do not have the facility to provide the local time differential factor.

Character Positions	Contents
18-19	Depending on the value of character position 17, one of the following:

Position 17	Contents
-	Two numeric digits in the range 00 through 12 indicating the number of hours that the reported time is behind Greenwich Mean Time.
+	Two numeric digits in the range 00 through 13 indicating the number of hours that the reported time is ahead of Greenwich Mean Time.
0	The value 00 is returned.

Character Positions	Contents
20-21	Depending on the value of character position 17, one of the following:

Position 17	Contents
-	Two numeric digits in the range 00 through 59 indicating the number of additional minutes that the reported time is behind of Greenwich Mean Time.
+	Two numeric digits in the range 00 through 59 indicating the number of additional minutes that the reported time is ahead of Greenwich Mean Time.
0	The value 00 is returned.

The returned value is the date and time of compilation of the source program that contains this function. If the program is a contained program, the returned value is the compilation date and time of the separately compiled program in which it is contained.

Setting the TZ Environment Variable

To get the correct difference between local time and Greenwich Mean Time, you must set the environment variable TZ to your local time zone. To set TZ, use the MPE XL SETVAR command. For example, the following command sets the time zone to Central Standard Time and Central Daylight Time, which would be correct for Chicago, Illinois:

```
:SETVAR TZ 'CST6CDT'
```

See the function `CURRENT-DATE` earlier in this chapter for a table of time zones. Check your local time zone to be sure you use the correct one.

Example

```
01  FULL-COMPILED-DATE.
    05  C-DATE.
        10  C-YEAR       PIC 9(4).
        10  C-MONTH      PIC 99.
        10  C-DAY        PIC 99.
    05  C-TIME.
        10  C-HOUR       PIC 99.
        10  C-MINUTES    PIC 99.
        10  C-SECONDS    PIC 99.
        10  C-SEC-HUND   PIC 99.
    05  C-TIME-DIFF.
        10  C-GMT-DIR    PIC X.
        10  C-HOUR       PIC 99.
        10  C-MINUTES    PIC 99.
                 :
MOVE FUNCTION WHEN-COMPILED TO FULL-COMPILED-DATE.
DISPLAY "Full date is: ", FULL-COMPILED-DATE.
DISPLAY "Year is: ",      C-YEAR.
DISPLAY "Month is: ",     C-MONTH.
DISPLAY "Day is: ",       C-DAY.
DISPLAY "Hour is: ",      C-HOUR OF C-TIME.
DISPLAY "Minute is: ",    C-MINUTES OF C-TIME.
DISPLAY "Second is: ",    C-SECONDS.
DISPLAY "Hundredths of seconds is: ", C-SEC-HUND.
DISPLAY "Difference from GMT is: ", C-GMT-DIR.
DISPLAY "Hours from GMT is: ", C-HOUR OF C-TIME-DIFF.
DISPLAY "Minutes from GMT is: ", C-MINUTES OF C-TIME-DIFF.
```

The above example displays the following:

```
Full date is: 1991022016512900-0800
Year is: 1991
Month is: 02
Day is: 20
Hour is: 16
Minute is: 51
Second is: 29
Hundredths of seconds is: 00
Difference from GMT is: -
Hours from GMT is: 08
Minutes from GMT is: 00
```

11

Interprogram Communication

Program modules consist of separately compiled,but logically coordinated programs, which, at execution time, are subdivisions of a single process. This approach to programming lends itself to making a large problem more easily programmed and debugged, by breaking a problem into logical modules and coding each module separately.

In COBOL terminology, a program is either a source or an object program. The distinction between the two is that a *source program* is simply a syntactically correct set of COBOL statements, whereas an *object program* is the set of instructions, constants, and other data resulting from the compilation of a source program.

A *run unit* is defined as being the total machine language necessary to solve a given generated data processing problem.

One run unit may contain several object programs, some of which may not have been generated by the COBOL compiler.

A run unit, then, is a combination of one main program with, optionally, one or more subprograms. Each subprogram may itself use one or more subprograms.

The state of a program the first time it is called in a run unit is the *initial state* of a program. The initial state of a program is characterized as follows:

- The program's internal data contained in the WORKING-STORAGE SECTION is initialized. If a VALUE clause is used in the description of the data item, the data item is initialized to the defined value. If a VALUE clause is not associated with a data item, the initial value of the data item is undefined.
- Internal files associated with the program are not in open mode.
- The control mechanisms for all PERFORM statements contained in the program are set to their initial states.
- A GO TO statement referenced by an ALTER statement within the same program is set to its initial state.

In COBOL, a program can transfer control to one or more subprograms, whether or not the names of the subprograms are known at compile time. Also, it is possible for the compiler to determine the availability of that subprogram.

When a run unit contains more than one object program, there must be communication between them. Interprogram communication takes two forms: *transfer of control* from one object program to another; and *reference to common data*.

Transfer of Control

The CALL statement is the means used in COBOL programs to pass control from one object program to another, and there are no restrictions on a called program itself calling another object program. Caution should be used, however, to avoid calling a program that preceded, in the calling chain, the program currently having control. Otherwise, results of the run unit are unpredictable.

When control is passed to a called program, execution begins either at the first PROCEDURE DIVISION statement or at a secondary entry point of the PROCEDURE DIVISION. Secondary entry points are described later in this chapter under the ENTRY statement, an HP extension to ANSI COBOL'85. Program execution begins at the point of entry to the called program in the normal, line-by-line sequence, following the same conventions used for COBOL main programs. Termination takes place in a COBOL subprogram under two possible conditions.

The first condition that causes termination is when an EXIT PROGRAM or GOBACK statement is encountered. When this occurs, control reverts to the calling program, which begins execution at the line immediately following the CALL statement that originally passed control.

The second condition that causes termination in a COBOL program is when a STOP RUN statement is encountered. In this case, the entire run unit is terminated.

In summary, the EXIT PROGRAM and GOBACK statements terminate only the program in which they appear, while the STOP RUN statement terminates the entire run unit.

An exception to this is when a GOBACK statement appears in a main program. In this case, it is equivalent to issuing a STOP RUN statement.

If the called program is not a COBOL program, termination of the run unit, or the return of control to the calling program, must be performed in accordance with the rules of the language in which the called program is written.

Reference to Common Data and Files

Two or more programs can reference common data in the following situations:

- The data content of an external data record can be referenced from any program provided that the program describes that data record.
- The mechanism in which a parameter value is passed by reference from a calling program to a called program establishes a common data item. The called program, which can use a different identifier, can refer to a data item in the calling program.

Two or more programs can reference a common file in the following situation:

An external file connector can be referenced from any program that describes that file connector.

Reference to Common Data through Parameter Passing

Because a called program often accesses data that is also used by the calling program, both programs must have access to the same data items if you wish to pass data to, or return data from, the called program.

In the calling program, it does not matter which section in the DATA DIVISION is used to describe the common data.

In the called COBOL program, however, common data must be described in the LINKAGE SECTION of the DATA DIVISION under 01 or 77 level description entries. Unlike the data in the calling program, no storage is allocated for LINKAGE SECTION items when the calling program is compiled.

Communication between the called COBOL program and the common data items stored in the calling program is provided by the USING clauses in both the calling and the called program.

The USING clause in the calling program is part of the CALL statement. It lists the names of common data items described in the DATA DIVISION.

In the called COBOL program, the USING clause appears as part of the PROCEDURE DIVISION header, or of the ENTRY statement. The common data items, which must be described in the LINKAGE SECTION, are listed by this clause.

Despite how the data items are described in the calling program, they are processed according to how they are described in the LINKAGE SECTION of the called program.

The only restriction is that descriptions of common data items must define an equal number of character positions.

Common data items are related to each other in the calling and called COBOL programs by their positions in the USING clauses, not by their names.

This implies that you may use entirely different names in the called program to represent common data items of the calling program.

For example, if `EMP-INFO` is the fourth name in the USING clause of a calling program and `FOURTH-PASSED` is the fourth name in the USING clause of the called program, then any reference to `FOURTH-PASSED` is treated as if it were a reference to the corresponding data item `EMP-INFO` in the calling program.

Also, although you may not use the same name twice within the USING phrase of a called program, you may do so in the USING phrase of the calling program. This allows you to have a data item in the calling program related to more than one data name in the called program.

Reference to Common Data and Files through External Objects

This is a feature of the 1985 ANSI COBOL standard

Accessible data items usually require that certain representations of data be stored. File connectors usually require that certain information concerning files be stored. The storage associated with a data item or a file connector can be external or internal to the program in which the object is declared.

A data item or file connector is external if the storage associated with that object is associated with the run unit rather than with any particular program within the run unit. An external object may be referenced by any program in the run unit which describes the object. References to an external object from different programs using separate descriptions of the object are always to the same object. In a run unit, there is only one representative of an external object.

An object is internal if the storage associated with that object is associated only with the program that describes the object.

A data record described in the WORKING-STORAGE SECTION is given the external attribute by the presence of the `EXTERNAL` clause in its data description entry. Any data item described by a data description entry subordinate to an entry describing an external record also attains the external attribute. If a record or data item does not have the external attribute, it is part of the internal data of the program in which it is described.

A file connector is given the external attribute by the presence of the `EXTERNAL` clause in the associated file description entry. If the file connector does not have the external attribute, it is internal to the program in which the associated file name is described.

The data records described subordinate to a file description entry that does not contain the `EXTERNAL` clause or a sort-merge file description entry, as well as any data items described subordinate to the data description entries for such records, are always internal to the program describing the file name. If the `EXTERNAL` clause is included in the file description entry, the data records and the data items attain the external attribute.

A file connector is given a global attribute by the presence of the `GLOBAL` clause in the file description entry. By specifying the file connector `GLOBAL`, all the data records associated with the file are automatically given a global attribute and the record description entries do not need to have a `GLOBAL` clause. However, if a `GLOBAL` clause is specified in the record description entry and not in the file description entry, only the record name is given a global attribute. For example, the contained programs are not able to reference the file connector (no file I/O is allowed), but are able to reference the record in which a `GLOBAL` clause was specified.

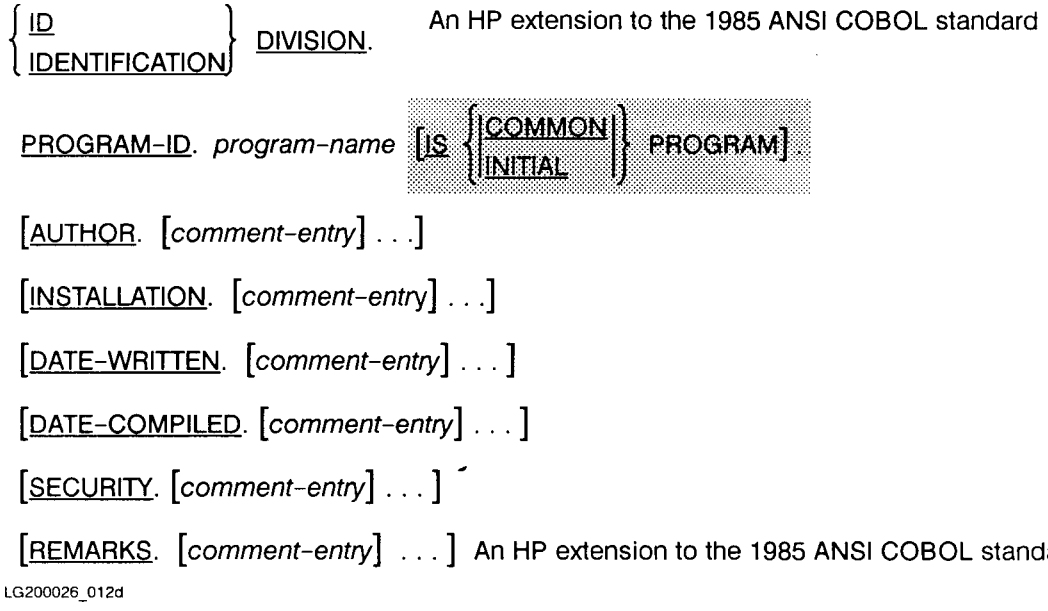
Data records, subordinate data items, and various associated control information described in the linkage of a program are always considered to be internal to the program describing that data. Special considerations apply to data described in the LINKAGE SECTION whereby an association is made between the data records described and other data items accessible to other programs.

If a data item possessing the external attribute includes a table accessed with an index, that index does not possess the external attribute.

PROGRAM-ID Paragraph

Define a subprogram using the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION.

Syntax



Description

For more information on the PROGRAM-ID paragraph, see Chapter 5, "IDENTIFICATION DIVISION."

COMMON Clause

```
The COMMON clause specifies that the program is common, a program contained
in another program.  A common program is one which, though contained within
another program, may be called by any program directly or indirectly
contained in that other program.  This clause is used in nested and
concatenated programs.  It facilitates the writing of subprograms which are
to be used by all the programs contained within a program.
```

EXTERNAL Clause

The `EXTERNAL` clause is a feature of the 1985 ANSI COBOL standard

The EXTERNAL clause specifies that a data item or a file connector is external. The corresponding data items and data group items of an external data record are available to every program in the run unit that describes that record.

Syntax

IS EXTERNAL

Description

The EXTERNAL clause can only be specified in file description entries or in record description entries in the WORKING-STORAGE SECTION.

The data-name specified as the subject of an entry with a level-number of 01 that includes the EXTERNAL clause must not be the same data-name specified for any other data description entry that includes the EXTERNAL clause.

The VALUE clause must not be used in any data description entry that includes, or is subordinate to, an entry that includes the EXTERNAL clause. The VALUE clause can be specified for condition-name entries associated with these data description entries.

The data contained in the record named by the data-name clause is external. It can be accessed and processed by any program in the run unit that describes and, optionally, redefines it according to the following rules.

- Within a run unit, if two or more programs describe the same external data record, each record-name of the record description entries must be the same. The records must define the same number of standard data format characters.

  However, a program describing an external record can contain a data description entry that includes the REDEFINES clause. The REDEFINES clause then redefines the complete external record. This complete redefinition does not need to occur identically in other programs in the run unit.

- The file connector associated with this description entry is an external file connector.

Example

The following example illustrates interprogram communication using EXTERNAL items. A main program and a subprogram share an EXTERNAL file and an EXTERNAL data item.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. EXTITEMS.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT SOUT ASSIGN TO "FILEOUTP".
 DATA DIVISION.
 FILE SECTION.
*
* THE EXTERNAL FILE SPECIFICATION
*
 FD SOUT EXTERNAL.
 01 REC-OUT.
   05 NAME PIC X(20).
   05 LOCN PIC X(10).
 WORKING-STORAGE SECTION.
*
* THE EXTERNAL DATA-ITEM SPECIFICATION
*
* NOTE THAT SINCE THE ITEM IS 'EXTERNAL' THE VALUE
* CLAUSE MAY NOT BE USED EXCEPT WITHIN A CONDITION-NAME
* ASSOCIATED WITH THE DATA ITEM.
*
 01 EXTERNAL-DATA-ITEM   EXTERNAL  PIC X(18).
   88 X-ITEM VALUE "EXTERNAL-DATA-ITEM".

 PROCEDURE DIVISION.
 START-IT.
      SET X-ITEM TO TRUE.
      OPEN OUTPUT SOUT.
      MOVE "JOHN DOE" TO NAME.
      MOVE "BLDG 48 N" TO LOCN.
      DISPLAY "DISPLAY OF EXT-DATA-ITEM FROM MAIN ** "
               EXTERNAL-DATA-ITEM.
      WRITE REC-OUT.
*
* CALL SUB1 TO WRITE A RECORD TO THE FILE. NOTE THAT
* THE EXTERNAL-DATA-ITEM VARIABLE IS NOT PASSED TO SUB1.
*
      CALL "SUB1".
      CLOSE SOUT.
```

```
*
* READ/DISPLAY THE SHARED FILE CONTENTS
*
      OPEN INPUT SOUT.
 READ-FILE-RECS.
      READ SOUT AT END
          CLOSE SOUT
          STOP RUN.
      DISPLAY REC-OUT.
      GO TO READ-FILE-RECS.


$CONTROL SUBPROGRAM
 IDENTIFICATION DIVISION.
 PROGRAM-ID. SUB1.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT SOUT ASSIGN TO "FILEOUTP".
 DATA DIVISION.
 FILE SECTION.
 FD SOUT EXTERNAL.
 01 REC-OUT.
   05 NAME PIC X(20).
   05 LOCN PIC X(10).
 WORKING-STORAGE SECTION.
 01 EXTERNAL-DATA-ITEM    EXTERNAL  PIC X(18).

 PROCEDURE DIVISION.
*
* NOTE THAT SUB1 DOES NOT REQUIRE A 'USING' CLAUSE TO
* ACCESS THE EXTERNAL DATA ITEM OR A FILE 'OPEN' TO
* ACCESS THE OUTPUT FILE.
*
 START-IT.
      MOVE "MARY JANE" TO NAME.
      MOVE "BLDG 66 W" TO LOCN.
      DISPLAY "DISPLAY OF EXT-DATA-ITEM FROM SUB1 ** "
               EXTERNAL-DATA-ITEM.
      WRITE REC-OUT.
      GOBACK.
```

When this example is compiled using the ANSI85 entry point, and run, it produces the following output:

```
DISPLAY OF EXT-DATA-ITEM FROM MAIN ** EXTERNAL-DATA-ITEM
DISPLAY OF EXT-DATA-ITEM FROM SUB1 ** EXTERNAL-DATA-ITEM
JOHN DOE            BLDG 48 N
MARY JANE           BLDG 66 W
```

GLOBAL Clause

The `GLOBAL` clause is a feature of the 1985 ANSI COBOL standard

The GLOBAL clause specifies that the data item or file connector can be referenced by the contained programs within a nested program in which the item is declared global.

Syntax

IS GLOBAL

Description

The following rules should be observed when using GLOBAL clause:

- The GLOBAL clause can only be specified in the file description entries in the FILE SECTION or in the 01 record description entries in the FILE SECTION and the WORKING-STORAGE SECTION.
- The GLOBAL clause can be specified in the record description entries which have unique names.
- If the SAME RECORD AREA clause is specified for several files, the record description entries or the file description entries for these files must not include a GLOBAL clause.
- If the GLOBAL clause is used in a record description entry that contains a REDEFINES clause, only the subject of the REDEFINES clause will be given the global attribute.
- If the GLOBAL clause is used in a record description entry that contains a table and index name, the index name is automatically given the global attribute.
- If the GLOBAL clause is used in a record description entry that contains a condition name defined with a level 88, the condition name is automatically given the global attribute.

Types of Subprograms

HP COBOL II has three kinds of subprograms:

1. Non-Dynamic.
2. Dynamic.
3. ANSISUB.

Specify which kind of subprogram you want by using one of the following:

Table 11-1. Types of Subprograms and How to Specify Them

Subprogram Type	Option or Clause
Non-Dynamic.	$CONTROL SUBPROGRAM
Dynamic.	$CONTROL DYNAMIC or the INITIAL clause of the PROGRAM-ID paragraph.
ANSISUB	$CONTROL ANSISUB

See Appendix B, "$CONTROL Options," and Appendix H, for more information on these options.

If you do not use one of these $CONTROL options and the source program contains a LINKAGE SECTION, it is compiled as a non-dynamic subprogram. A program containing no LINKAGE SECTION is considered a main program unless $CONTROL SUBPROGRAM, DYNAMIC, or ANSISUB `or PROGRAM IS INITIAL clause` is used for that program.

Non-Dynamic Subprograms

Use $CONTROL SUBPROGRAM to specify a non-dynamic subprogram. The data of a non-dynamic subprogram is declared as OWN. That is, the subprogram data retain their values between calls to the subprogram.

In other words, when you exit a called non-dynamic subprogram, its state is maintained. Thus, data items not common to the calling program or a program that called the calling program retain their values when the program in which they are used is exited.

Dynamic Subprograms

Use $CONTROL DYNAMIC or the INITIAL clause of the PROGRAM-ID paragraph to specify a dynamic subprogram. The INITIAL clause is part of the ANSI standard. Dynamic subprograms are equivalent to PROGRAM IS INITIAL and have local data storage and are *always* in their initial state when called. This implies that any files opened in dynamic subprograms should be closed before exiting. Otherwise, the file is closed for you.

ANSISUB Subprograms

Ansisub subprograms are the 1985 ANSI COBOL standard type of subprogram. Use $CONTROL ANSISUB to specify an ansisub subprogram. Ansisub subprograms are a combination of the dynamic and non-dynamic types. The data area of an ansisub subprogram retains its values between calls unless you explicitly reinitialize the data area. To reinitialize an ANSISUB subprogram, use the CANCEL statement in the calling program after the EXIT PROGRAM or GOBACK statement has been executed in the ansisub subprogram. See "MPE XL System Dependencies" in Appendix H for more information about using ansisub subprograms.

For more information on subprograms, see the *HP COBOL II/XL Programmer's Guide*.

END PROGRAM Header

The `END PROGRAM` header is a feature of the 1985 ANSI COBOL standard

The END PROGRAM header indicates the end of the named COBOL program. It must begin in Area A.

Syntax

END PROGRAM *program-name*.

Description

The following rules should be observed when using the END PROGRAM header:

- The *program-name* must be identical to the *program-name* declared in the preceding PROGRAM-ID paragraph. In a nested program, the END PROGRAM header for the contained program must precede the END PROGRAM header for the containing program.
- It must be present in every program which either contains or is contained within another program.
- If the program being terminated by the END PROGRAM is contained within another program, the next statement must be either the IDENTIFICATION DIVISION header or another END PROGRAM header to terminate the containing program.
- If the program terminated by the END PROGRAM header is not a contained program and if the next statement is a COBOL statement, it must be an IDENTIFICATION DIVISION header for the following COBOL program. These types of programs are called concatenated programs.

If the END PROGRAM header is not followed by correct program name, the compiler will issue an error message.

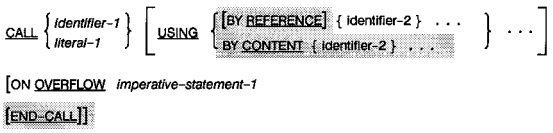
CALL Statement

In ANSI COBOL, you can use the CALL statement to transfer control from one object program to another within the same run unit. An HP extension to the 1985 ANSI COBOL standard adds the ability to invoke operating system intrinsics from within a given object program.

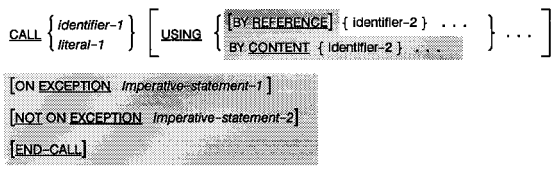
Syntax

CALL Statement Format (ANSI COBOL '85)

Format 1



Format 2

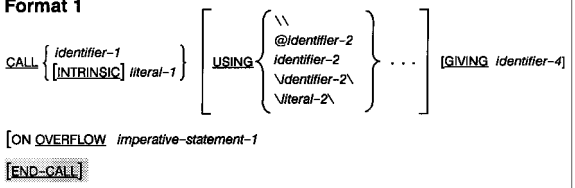


- An HP extension to the 1985 ANSI COBOL standard adds the keyword INTRINSIC to the following formats to specify that an intrinsic, not a program, is being called. Also added is the capability to pass a data item by value. These can only be used when calling non-COBOL programs.

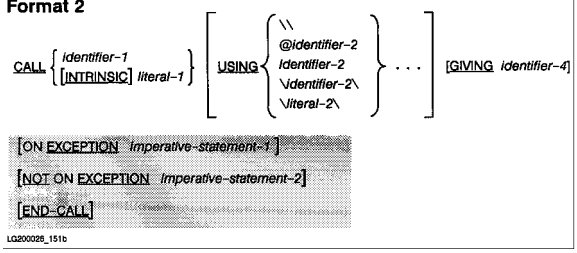
  The last format of the CALL statement is the GIVING *identifier-4* phrase. This phrase allows you to name an identifier that holds the result of a call to a typed procedure.

CALL Statement Format (An HP Extension to the 1985 ANSI COBOL Standard)

Format 1



Format 2



Parameters

identifier-1 — names an alphanumeric data item whose value is a program name. Refer to "CALL Identifier" (below) for a description of conditions that may occur when using this operation.

literal-1 — nonnumeric literal that either names an operating system intrinsic, or names a program. If an intrinsic is named by *literal-1*, the keyword, INTRINSIC, must precede it.

\\ — represents a null value passed as a parameter to an intrinsic or to a Pascal procedure that includes OPTION DEFAULT_PARMS or EXTENSIBLE. The compiler assumes a one-word parameter when the INTRINSIC option is not specified.

@ *identifier-2* — @ is ignored; this is provided for compatibility with HP COBOLII/V. *identifier-2* cannot be a function-identifier.

identifier-2, — the name of any data item in the calling program, or a file named in an FD level entry in the FILE SECTION of your program.

identifier-2 must be described in the FILE, WORKING-STORAGE, or LINKAGE SECTIONs of your program. When this value is passed to another program, it is passed by reference.

If this parameter names a file, the called program must not be a COBOL program unless the BY CONTENT phrase is used.

identifier-2 cannot be a function-identifier.

identifier-2\, and *literal-2*\, — used to indicate that the literals or data items represented by identifiers enclosed by the back slashes are to be passed by value to the called program or intrinsic. This may only be used when the called program is not a COBOL program. If an identifier is used in this way, it must represent a numeric data item of up to 18 digits.

identifier-2 cannot be a function-identifier.

identifier-4 — the name of a binary data item in the calling program. It is used in calls to non-COBOL programs and in calls to COBOL programs that return a value in the RETURN-CODE special register (see the section "GIVING Phrase When Calling COBOL Subprograms" later in this chapter for details.)

identifier-4 cannot be a function-identifier.

imperative-statement-1 — one or more imperative statements.

Description

A CALL statement may appear anywhere within a segmented program. The compiler provides all controls necessary to ensure that the proper logic flow is maintained. When a CALL statement to a subprogram appears in a section with a segment number greater than or equal to 50, and when control is returned to this segment by the EXIT PROGRAM statement in the called subprogram, this segment is in the same state as when it issued the CALL statement.

Example

The ANSI COBOL'85 standard contains several rules that govern the scope of CALL statements for nested, concatenated and mixed programs. The following example illustrates several cases. In the example, only the PROGRAM-ID, CALL, and END PROGRAM are shown for the COBOL program. For clarity, the lines are indented to illustrate the scope. However, in practice, both PROGRAM-ID and END PROGRAM must start in margin A, and a CALL statement should start in margin B.

Line	Program Structure	Scope
1	`PROGRAM-ID. A.`	*Start of program A.*
2	⋮	
3	`    PROGRAM-ID. C COMMON.`	*Start of nested program C.*
4	⋮	*Program C is contained within program A.*
5	`    END PROGRAM C.`	*End of nested program C.*
6	`END PROGRAM A.`	*End of program A.*
7		
8	`PROGRAM-ID. F.`	*Start of program F.*
9	⋮	
10	`    CALL "A".`	*Calls nested program A in line 11.*
11	`    PROGRAM-ID. A.`	*Start of nested program A.*
12	⋮	*Program A is contained within program F.*
13	`    END PROGRAM A.`	*End of nested program A.*
14		
15	`    PROGRAM-ID. B.`	*Start of nested program B.*
16	⋮	*Program B is contained within program F.*
17	`        CALL "A".`	*Calls program A in line 1.*
18	`    END PROGRAM B.`	*End of nested program B.*
19	`END PROGRAM F.`	*End of program F.*

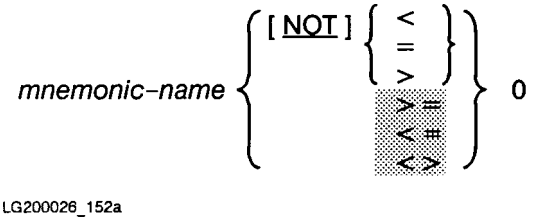
Calling Intrinsics

The INTRINSIC phrase is used to indicate that the CALL statement in which it appears is calling an operating system intrinsic rather than a subprogram. When the INTRINSIC phrase is used, *literal-1* must be used and must name an operating system intrinsic. The USING phrase specifies the various parameters to be passed to the intrinsic. When the intrinsic is a "typed procedure", the GIVING phrase specifies the parameter to be returned by the intrinsic.

As with subprograms, the parameters passed to intrinsics are specified by position. If a parameter of an intrinsic is optional, and you do not want to pass a value for that parameter, you must specify two consecutive back slashes (\\) in the position within the USING phrase of the CALL statement that corresponds to that parameter's position within the intrinsic.

Unlike subprograms, if an intrinsic expects a parameter to be passed by value rather than by reference, it is unnecessary to enclose the literal or identifier with backslashes. The intrinsic automatically assumes that the parameter is being passed by value.

The following special relation operator can be used after a call to an intrinsic to check the condition code returned by an intrinsic:



LG200026_152a

This relation condition is described under "Relation Conditions" in Chapter 8.

For information on the condition codes and the types, number, and position of parameters required to call a specific intrinsic, refer to the *MPE Intrinsics Reference Manual* and to "System Dependencies" in Appendix H.

Example

The following shows some calls to intrinsics:

```
 SPECIAL-NAMES.
   CONDITION-CODE IS C-C.
        ⋮
WORKING-STORAGE SECTION.
01 SHOWME.
    02 MPE-COMMAND     PIC X(07)  VALUE "SHOWJOB".
    02 CARRIAGE-RETURN PIC X      VALUE %15.
01 ERR       PIC S9999  USAGE IS COMP VALUE ZERO.
01 CATCHPARM PIC S9999  USAGE IS COMP VALUE ZERO.
01 REPLY     PIC X(03)  VALUE SPACES.
        ⋮
PROCEDURE DIVISION.
        ⋮
  CALL INTRINSIC "COMMAND" USING SHOWME ERR CATCHPARM.
  IF C-C   NOT = 0 THEN
        DISPLAY "COMMAND FAILED"
        STOP RUN.
  MOVE SPACES TO REPLY.
  DISPLAY "DO YOU WISH TO COMMUNICATE WITH ANYONE?"
  ACCEPT REPLY.
  IF REPLY = "YES" PERFORM BREAK-FOR-COMM.
        ⋮
BREAK-FOR-COMM.
     DISPLAY "TYPE RESUME WHEN READY TO RESUME PROGRAM".
     CALL INTRINSIC "CAUSEBREAK".
        ⋮
```

The first intrinsic call above uses the COMMAND intrinsic to issue the operating system command SHOWJOB. This allows you to see who is currently logged on to your system. The second call to an intrinsic is the programmatic equivalent to pressing the BREAK key, thus suspending your program.

The intrinsics used in the above manner could allow you to see who is using the system and to ask them, for example, to release a file from their group to allow your program to access it.

When CALL INTRINSIC is used for intrinsics in a system file:

1. The special symbols "@" and "\" are optional.
2. Data conversions for parameters passed by value are performed automatically.

Execution-Time Loading

When you use the *identifier-1* form of the CALL statement, the value of *identifier-1* determines which subprogram is called. When the program is executed, an attempt is made to load the subprogram. If the load fails, an exception condition occurs. For more information, see "System Dependencies" in Appendix H.

If the ON EXCEPTION or ON OVERFLOW phrase is specified, control is passed to the imperative statement of that phrase.

If no ON EXCEPTION or ON OVERFLOW phrase is specified and the CALL statement causes an exception condition, the entire run unit is aborted.

The *identifier-1* form of CALL may be useful for calling a procedure used optionally and infrequently by your program or for a procedure whose name is not known at load time.

Caution

Indiscriminate use of ON EXCEPTION or ON OVERFLOW may adversely affect the performance level of an active system or a system where multiple users are running the same program utilizing this capability. Do not use these phrases when using a literal to name the program to be called because the call is treated the same as a CALL identifier.

Pseudo-Intrinsics

The following pseudo-intrinsics are an HP extension to the ANSI COBOL standard.

Caution Pseudo-intrinsics are highly machine-dependent and should not be used in programs that may be run on different machines and architectures now or in the future.

.LOC. Pseudo-Intrinsic

The .LOC. pseudo-intrinsic allows you to call several useful operating system intrinsics, such as GENMESSAGE, CREATEPROCESS, MYCOMMAND, and other intrinsics that require identifiers to contain addresses. Input to this intrinsic is the name of an identifier for which an address is desired. The address of this identifier is returned in the GIVING identifier. Refer to "System Dependencies" in Appendix H for the correct size of the GIVING identifier.

.LEN. Pseudo-Intrinsic

The .LEN. pseudo-intrinsic can be used to calculate the length of items, groups or tables. .LEN. returns the length in bytes of its parameter, or the current length of a table whose length has an OCCURS DEPENDING ON clause.

Example. One important use of .LEN. is to calculate the length of records passed to intrinsics like VPLUS. For example:

```
01 V-REC.
   05  FIELD-1    PIC XX.
   05  FIELD-2    PIC X(10).
         :
77 V-REC-LEN      PIC S9(4) COMP.
         :
   CALL INTRINSIC ".LEN." USING V-REC GIVING V-REC-LEN.

   CALL INTRINSIC "VPUTBUFFER" USING COMAREA V-REC
                                              V-REC-LEN.
```

The above code sequence has the advantage that the length of the record `V-REC` is not hard coded, and additional fields can be added to `V-REC` without having to change the PROCEDURE DIVISION or a variable that contains the length. Also, with the HP COBOL II compiler doing the calculation, there are no mistakes in adding up the individual field lengths.

To conform to the 1985 ANSI COBOL standard, you can use the LENGTH COBOL function instead of .LEN. (see Chapter 10 for details).

USING Phrase (COBOL Subprograms)

The USING phrase specified in a CALL statement to another program makes data and files of the calling program available to the called program.

If a CALL statement containing the USING phrase calls a COBOL program, then the called COBOL program must contain a USING phrase in its PROCEDURE DIVISION header (or in the ENTRY statement, if a secondary entry point name is used by the CALL statement). The USING phrase of the called program must contain as many operands as does the USING phrase of the calling program. The USING option makes data items (or FD file names) defined in the calling program available to a called subprogram. However, file names cannot be passed to a COBOL subprogram unless BY CONTENT is specified. Note that the limit of data items in the USING phrase also applies to the number of 01/77 level entries, excluding redefinitions, defined in the LINKAGE SECTION of the called subprograms. Refer to "System Dependencies" in Appendix H for specifics on limitations.

The order of appearance of names of data items in the USING phrase is very important (refer to "Reference to Common Data through Parameter Passing" at the beginning of this chapter). To restate briefly, data is passed from the calling program to the called program on a positional basis, rather than by name. Therefore, the third name in a USING phrase of a calling program corresponds to the third name in the USING phrase of the called COBOL program.

```
Both the BY CONTENT and BY REFERENCE phrases cause the parameters that
follow them to be passed by content or reference until another BY CONTENT
or BY REFERENCE phrase is encountered.  If neither the BY CONTENT nor the
BY REFERENCE phrase is specified prior to the first parameter, BY REFERENCE
is assumed.
```

Note ANSI COBOL'74 parameters are always passed by reference.

Index names in the calling and called programs always refer to different indices. To pass an index value from a calling program to a called program, you must first move the index value associated with an index name to a data item that appears in the USING phrase of the calling program. The corresponding data item in the called program can then be used to set an equivalent index name to this value.

BY REFERENCE Phrase

The `BY REFERENCE` phrase is a feature of the 1985 ANSI COBOL standard.

If the BY REFERENCE phrase is either specified or implied for a parameter, the object program operates as if the corresponding data item in the called program occupies the same storage area as the data item in the calling program. The description of the data item in the called program must describe the same number of character positions as the corresponding data item in the calling program.

BY CONTENT Phrase

The `BY CONTENT` phrase is a feature of the 1985 ANSI COBOL standard.

If the BY CONTENT phrase is specified or implied for a parameter, the called program cannot change the value of this parameter as referenced in the CALL statement's USING phrase. However, the called program may change the value of the data item referenced by the corresponding data name in the called program's division header. That is, if the called program changes the value, the calling program never sees the change.

```
CALL "SUBP" USING BY CONTENT PARM1 PARM2
                  BY REFERENCE PARM3
                  BY CONTENT PARM4
```

Note Do not use the BY CONTENT phrase when calling non-COBOL subprograms that contain value parameters. BY CONTENT is not the same as BY VALUE.

USING Phrase (Non-COBOL Subprograms)

This positional correspondence extends to non-COBOL called programs. Thus, for example, if the called program is a Pascal program, then the names in the parameter list of the procedure declaration in that program are identified with those data items whose names appear in the corresponding position of the USING phrase in the calling program.

As stated above in the description of *identifier-2*, *identifier-3*, and so forth, these identifiers may name files to be passed to a called program. Furthermore, although you can enclose such a file identifier between back slashes (which are ignored), preceding it with an `@` sign results in an error.

If the file name from the FD is passed, the file number of that file is passed by value. If the subprogram is an SPL procedure, the procedure parameter corresponding to the file name must be declared as type INTEGER or LOGICAL and it must be specified as a value parameter. The file must be opened in the calling program.

To pass a data item by value, you must enclose the associated identifier in back slashes. If the value passed is a literal, the back slashes are optional. Passing a data item by value leaves the data item in the calling program unchanged following execution of the called program.

If an identifier is not passed by value (that is, is not enclosed in back slashes), it is passed as a byte pointer (that is, by reference). Thus, the data in the calling program can be altered by the called program if it is passed in this manner. In calls to COBOL programs, this is the standard method of referencing common data.

Two consecutive back slashes (\\) may be specified in a USING phrase of a CALL statement if the called program is a Pascal procedure with OPTION DEFAULT_PARMS or EXTENSIBLE. Using two consecutive back slashes indicates that a parameter is not being sent and should not be expected. Whenever an OPTION VARIABLE SPL procedure is called, an additional parameter must be added to the end of the USING parameter list. This parameter is called a "bit mask" and is used to tell the SPL procedure which parameters are being passed; each bit represents a parameter. It must be a numeric data item and represents the value derived from the bit mask. The bit mask is one or two 16-bit binary words, where a "0" represents a missing parameter, and a "1" represents an existing parameter (allows up to 32 parameters to be passed). Parameters are matched, starting from the right, in both the bit mask and the USING list, excluding the value in the USING list used for the bit mask parameter. For example,

```
CALL "SPLPROC" USING \TESTER\ \\ @RESULT \ERROR\ \%13\
```

The bit mask in this case is 0000000000001011, which is represented by the octal value \%13\, showing that the fourth, third, and first parameters are being passed, while the second parameter is not being passed.

The bit mask is generated automatically by the compiler if you specify the INTRINSIC option.

GIVING Phrase When Calling COBOL Subprograms

As an HP extension to the ANSI COBOL standard, you can return a value from a COBOL subprogram using the RETURN-CODE special register in the subprogram and the GIVING phrase in the calling program.

The data item in the GIVING phrase must be defined as PIC S9(9) BINARY. If it is not, a type checking error may occur at link time. The value returned is the value placed into the RETURN-CODE special register by the called program. If a value is not placed into RETURN-CODE by the called program, or if RETURN-CODE is a user-defined data item in the called program, the GIVING item will contain uninitialized data.

RETURN-CODE Special Register

RETURN-CODE is a predefined data name in the PROCEDURE DIVISION of a subprogram. It can be used wherever an elementary data item can be used and is predefined as PIC S9(9) BINARY.

The RETURN-CODE special register must be set before executing an EXIT PROGRAM or GOBACK statement in the called program. The value in RETURN-CODE is made available to the calling program in the GIVING phrase of the CALL statement.

If the program already contains a data item named RETURN-CODE, it takes precedence over the special register and you cannot access the special register. To access the special register, you must change your data item to something else.

Example

The following example shows a main program and a subprogram with a secondary entry point. The main program calls the subprogram twice and displays the value returned in the GIVING phrase. Notice that the special register RETURN-CODE is not defined anywhere in the DATA DIVISION of the subprogram.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. RETURN-CODE-TEST.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  BUF         PIC S9(9) BINARY VALUE 99.
01  RESULT      PIC S9(9) BINARY VALUE -99.
PROCEDURE DIVISION.
START-IT.
    CALL "SUB-MAIN-ENTRY" USING BUF GIVING RESULT.
    DISPLAY RESULT.
    CALL "SECOND-ENTRY" GIVING RESULT.
    DISPLAY RESULT.
END PROGRAM RETURN-CODE-TEST.

$CONTROL DYNAMIC
IDENTIFICATION DIVISION.
PROGRAM-ID. SUB-MAIN-ENTRY.
DATA DIVISION.
LINKAGE SECTION.
01  BUF         PIC S9(9) BINARY VALUE 99.
PROCEDURE DIVISION USING BUF.
PARA-01.
    ADD 1 TO BUF GIVING RETURN-CODE.
    EXIT PROGRAM.
PARA-02.
ENTRY "SECOND-ENTRY".
    MOVE -5000 TO RETURN-CODE.
```

When this program runs it displays the following:

```
+000000100
-000005000
```

GIVING Phrase When Calling Non-COBOL Subprograms

The GIVING phrase also allows you to call non-COBOL typed procedures.

A typed procedure (for example, a Pascal function declaration) must always return a value when it completes execution. In HP COBOL II, this is assumed to be a 2-, 4-, or 8-byte binary value of up to eighteen digits.

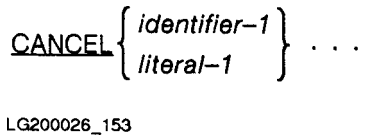
The purpose of the GIVING phrase is to provide for this returned value. Since *identifier-4* is used to hold the value returned by a typed procedure, its length must be sufficient to accommodate the value returned.

Use of the keyword INTRINSIC allows the compiler to check the procedure information within the system file and adjust the generated code to correctly prepare and clean up the stack prior to and following the execution of the CALL statement.

CANCEL Statement

The CANCEL statement restores a program to its initial state and closes all files currently in open mode.

Syntax



LG200026_153

Parameters

identifier-1	defined as an alphanumeric data item whose contents name a subprogram compiled by the HP COBOL II compiler.
literal-1	nonnumeric literal that names a COBOL subprogram compiled by the HP COBOL II compiler.

Description

When a CANCEL statement is issued for a COBOL program, the program specified in the statement must have already executed a GOBACK or EXIT PROGRAM statement.

For both dynamic and non-dynamic subprograms in HP COBOL II, the CANCEL statement has no effect. This occurs because automatic release of the memory areas accompanies the exiting of the dynamic subprogram. For non-dynamic subprograms, the data area is permanently assigned and the code segments are automatically released.

For ANSISUB subprograms, a CALL statement that is executed after a CANCEL statement naming the same program causes that program to be brought into memory in its initial state. Refer to Appendix B for details of the ANSISUB parameter.

If a CANCEL statement specifies a program that has not been called, or has been called but is presently canceled, the CANCEL statement is ignored and control passes to the next executable statement.

The CANCEL statement causes an implicit CLOSE statement to be executed for every open file associated with the program being canceled. No USE procedures are executed.

ENTRY Statement

The ENTRY statement is an HP extension to the ANSI COBOL standard.

The ENTRY statement establishes a secondary entry point in an HP COBOL II subprogram. In nested programs, this statement must begin in Area A. However, like all other COBOL statements in the PROCEDURE DIVISION, the ENTRY statement must be in a paragraph.

Syntax

ENTRY *literal-1* [USING {*data-name-1*} ...]

Parameters

literal-1 — nonnumeric literal. It must be formed according to the rules for program names, but must not be the name of the called program in which it appears. Furthermore, it must not be the name of any other entry point or program name in the run unit.

data-name-1 — as described and used in the USING phrase of the PROCEDURE DIVISION header. Refer to Chapter 8 for details.

Description

The link between the calling program and a secondary entry point of a called program is supplied by *literal-1*. That is, *literal-1* must be used in a CALL statement of the calling program and must appear in an ENTRY statement in the called program.

When the called program is invoked using such CALL and ENTRY statements, the called program is entered at the ENTRY statement that specifies *literal-1*.

When using secondary entry points the PROCEDURE DIVISION statements for each such entry point may only reference passed parameters that are declared in the USING clause for the respective entry point. Attempts to reference passed parameters declared in other entry point USING clauses will produce a run-time bounds violation.

The USING option has the same format and meaning as in the USING phrase of the PROCEDURE DIVISION header. Refer to Chapter 8 for details.

The entry name must be unique with respect to all program units (HP COBOL II main program or subroutines) compiled in a particular instance unless contained in different programs. Refer to "System Dependencies" in Appendix H for information on the resulting external name.

Example

The following example shows a main program and a subprogram. The subprogram has a secondary entry point named by the ENTRY statement.

The CALL statement in MAINPROG specifies that SUBPRO1 is to be executed, starting at the ENTRY statement rather than at the first line following the PROCEDURE DIVISION header. Also, the data areas of INV-FILE and SALES-FILE are to be used in both programs.

Following is the main program:

```
IDENTIFICATION DIVISION.
PROGRAM ID.  MAINPROG.
      ⋮
DATA DIVISION.
FILE SECTION.
FD INV-FILE.
01 INV-REC.
   02 PT-NUM          PIC X(10).
   02 PT-NAME         PIC X(30).
   02 BEGIN-QTY       PIC 9(6).
   02 PRICE-WHSL      PIC 9(3)V99.
   02 PRICE-RETAIL    PIC 9(4)V99.
FD SALES-FILE.
01 SALES-REC.
   02 SOLD-PT-NO      PIC X(10).
   02 SOLD-PART       PIC X(30).
   02 SOLD-QTY        PIC 9(6).
        ⋮
PROCEDURE DIVISION.
MAIN-PARA-001.
          ⋮
     IF SOLD-QTY IS NOT EQUAL TO ZERO
        CALL "SUBPRO1-ENTRY" USING INV-REC, SALES-REC.
          ⋮
     STOP RUN.
```

Following is the subprogram:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   SUBPRO1.
      ⋮
DATA DIVISION.
FILE SECTION.
FD PRINT-FILE.
01 P-REC              PIC X(132).
      ⋮
WORKING-STORAGE SECTION.
01 HEADER.
    ⋮
01 WRITE-SALES.
   02 FILLER        PIC X(15)     VALUE SPACES.
   02 NAME          PIC X(30)     VALUE SPACES.
   02 FILLER        PIC X(5)      VALUE SPACES.
   02 NUM-1         PIC X(10)     VALUE SPACES.
   02 FILLER        PIC X(5)      VALUE SPACES.
   02 QUANTITY      PIC Z(3)9(3)  VALUE ZERO.
   02 FILLER        PIC X(5)      VALUE SPACES.
   02 GROSS-SALES   PIC $Z(10).99 VALUE ZEROS.
   02 FILLER        PIC X(5)      VALUE SPACES.
   02 GROSS-PROFIT  PIC $9(10).99.
        ⋮
LINKAGE SECTION.
01 ORIGINAL.
   02 PT-NUM        PIC X(10).
   02 PT-NAME       PIC X(30).
   02 START-QTY     PIC 9(6).
   02 OUR-PRICE     PIC 9(3)V99.
   02 THEIR-PRICE   PIC 9(4)V99.
01 SALES.
   02 SOLD-NUM      PIC X(10).
   02 SOLD-NAME     PIC X(30).
   02 QTY-SOLD      PIC 9(6).
PROCEDURE DIVISION.
SUB-PARA-001.
      ⋮
CALC-PARA-100.
ENTRY "SUBPRO1-ENTRY" USING ORIGINAL, SALES.
     MULTIPLY QTY-SOLD BY THEIR-PRICE GIVING GROSS-SALES.
     SUBTRACT START-QTY FROM QTY-SOLD GIVING QUANTITY.
     COMPUTE GROSS-PROFIT =
        (THEIR-PRICE - OUR-PRICE) * QTY-SOLD.
     MOVE PT-NUM TO NUM-1.
     MOVE PT-NAME TO NAME.
     WRITE P-REC FROM HEADER AFTER ADVANCING 1 LINES.
     WRITE P-REC FROM WRITE-SALES AFTER ADVANCING 3 LINES.
     GOBACK.
       ⋮
```

EXIT PROGRAM Statement

The EXIT PROGRAM statement marks the logical end of a program.

Syntax

EXIT PROGRAM

Description

If you use an EXIT PROGRAM statement in a program (called or otherwise), it must appear as the only statement in a sentence, and the sentence must be the only sentence in a paragraph. Also, EXIT PROGRAM must not appear in a declarative procedure that has a `GLOBAL` phrase.

When encountered in a called program, the EXIT PROGRAM statement causes control to return to the statement of the calling program immediately following the CALL statement used to pass control to the called program.

If the EXIT PROGRAM statement is used in a main program, it is treated as an EXIT statement. Thus, in a main program, it serves only as a way of terminating a procedure.

GOBACK Statement

The GOBACK statement is an HP extension to the ANSI COBOL standard.

The GOBACK statement marks the logical end of a program.

Syntax

GOBACK

Description

The GOBACK statement must be the only statement in a sentence. If used in a series of imperative statements, it must be the last statement in the series.

If a GOBACK statement is encountered in a called program, control returns to the statement in the calling program immediately following the CALL statement that initiated the called program. Thus, in a subprogram, the GOBACK statement acts in the same way as an EXIT PROGRAM statement.

If a GOBACK statement is used in a main program, it is equivalent to a STOP RUN statement. Thus, it indicates the logical end of the run unit, and control is passed to the operating system.

Table 11-2 shows the relationship between the EXIT PROGRAM, STOP RUN, and GOBACK statements.

Table 11-2.
Relationship Between EXIT PROGRAM, STOP RUN and GOBACK Statements

Termination Statements	In a Main Program:	In a Subprogram:
EXIT PROGRAM	Non-operational. Treated as EXIT.	Returns to calling program.
STOP RUN	Logical end of run. Returns control to operating system.	Logical end of run for both the subprogram and the calling program. Returns control to operating system.
GOBACK	Logical end of run. Returns control to operating system.	Returns to calling program.

12

SORT/MERGE Operations

The sort/merge capabilities of HP COBOL II allow you to sort one or more files of records, or to combine two or more identically ordered files of records one or more times within a given execution of a COBOL program.

Additionally, you have the ability to specially process individual records by input or output procedures that are part of the sort or merge operation. For a sort operation, this special processing may be applied before, as well as after the records have been sorted. For a merge operation, this processing may be applied after the records have been combined. The RELEASE and RETURN statements are used in these input procedures to release or return records that are to be, or have been sorted or merged.

MERGE Statement

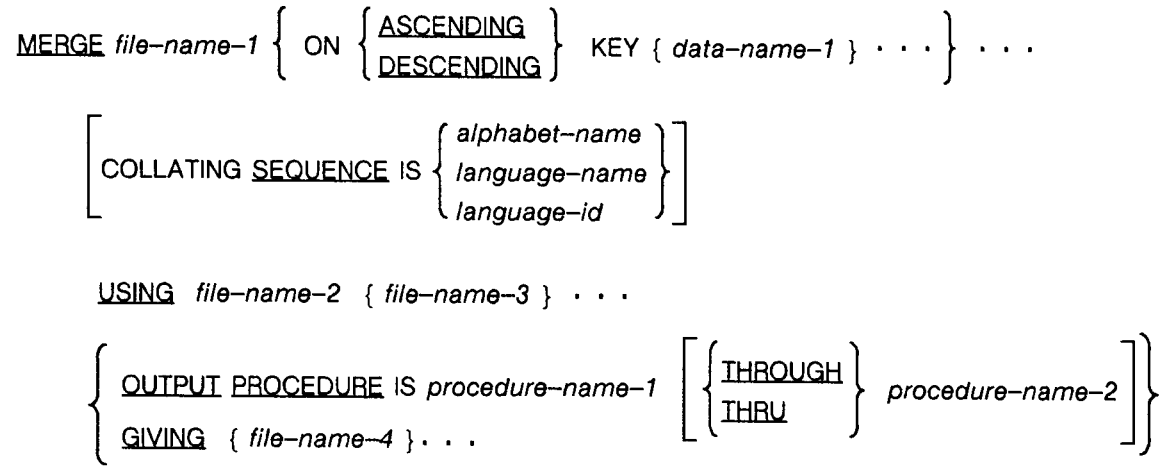
The MERGE statement combines two or more identically sequenced files on a set of specified keys. As part of the merge operation, it makes records available in their merged order to an output procedure or an output file.

The records are made available following the actual merging of the files. The output procedure or the moving of records to an output file is considered part of the merge process.

MERGE statements may appear anywhere within the PROCEDURE DIVISION except in the declaratives portion or in an input or output procedure associated with a SORT or MERGE statement.

Syntax

MERGE Statement Format



LG200026_157

Parameters

file-name-1 — sort/merge file, and is described in a sort/merge file description (SD level) entry in the DATA DIVISION.

data-name-1 — data items described in records associated with *file-name-1*. Each may be qualified and may vary in length. None of these data names can be described by an entry either containing an OCCURS clause, or subordinate to an entry containing such a clause. If *file-name-1* has more than one record description, then the data items represented by the data names need be described in only one of the record descriptions.

alphabet-name — either EBCDIC, STANDARD-1, NATIVE, or an alphabet name as defined by you in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

language-name — alphanumeric data item containing the name of the language whose collating sequence should be used. This parameter is an HP extension to the ANSI COBOL standard.

language-id — integer data item containing the identification number of the language to use. This parameter is an HP extension to the ANSI COBOL standard.

file-name-2
file-name-3 — files to be merged. These files must not be open at the time the MERGE statement is executed. Each must be a file described in an FD level file description in the DATA DIVISION. No more than one such file name can name a file from a multiple file reel. Any given file name can be used only once in any given MERGE statement.

The actual size of the logical record or records described for these files must be equal to the actual size of the logical record or records described for *file-name-1*. If the data descriptions of the elementary items that make up these records are not identical, it is your responsibility to describe the corresponding records in such a way as to cause an equal number of character positions to be allocated for the corresponding records.

procedure-name-1 — name of the first `paragraph` or section in an output procedure.

procedure-name-2 — name of the last `paragraph` or section in an output procedure.

file-name-4 — names of output files. These files are subject to the same restrictions as *file-name-2*.

Description

The words THROUGH and THRU are equivalent, and can be used interchangeably.

The MERGE statement merges all records contained in the files named in the USING phrase. The files to be merged are automatically opened and closed by the merge operation with all implicit functions performed, such as the execution of any associated USE procedures. The files described by *file-name-2* and *file-name-3* must not be open when the MERGE verb is executed, and may not be opened or closed during an output procedure if one is specified.

Following the actual merging of the files, but before they have been closed, the merged records are released in the order in which they were merged. They are released to either the specified output procedure or the specified output file.

The results of a merge operation are predictable only when the records in the files to be merged are ordered as described in the ASCENDING or DESCENDING KEY phrase associated with the MERGE statement.

The data names following the word KEY are listed from left to right in order of decreasing significance. This decreasing significance is maintained from KEY phrase to KEY phrase. Thus, in the format shown, *data-name-1* is the most significant, and each successive data name is the next most significant.

When the MERGE statement is executed, the records of *file-name-2* and *file-name-3* are merged in the specified order (ASCENDING or DESCENDING) using the most significant key data item. Within the records having the same value for the most significant key data item, the records are merged according to the next most significant key data item; this kind of merging continues until all key data items named in the MERGE statement have been used.

When the ASCENDING phrase is used, the merged records are in a sequence starting from the lowest value of the key data items and ending with the highest value.

When the DESCENDING phrase is used, the merged records are in a sequence from the highest value of the key data item to the lowest.

Merging takes place using the rules for comparison of operands of a relation condition. If all corresponding key data items of records to be merged are equal, the records are written to *file-name-4*, or returned to the output procedure, in the order that the input files are specified in the MERGE statement.

COLLATING SEQUENCE Phrase

The COLLATING SEQUENCE phrase allows you to specify what collating sequence to use in the merging operation. This phrase is optional. The program collating sequence is used if none is specified in the MERGE statement.

See the alphabet clause of the SPECIAL-NAMES paragraph for information on defining and using collating sequences.

Refer to the *Native Language Support Reference Manual* for details about language names and language id's.

GIVING and OUTPUT PROCEDURE Phrases

You must specify either the GIVING or OUTPUT PROCEDURE phrase in a MERGE statement.

If you specify the GIVING phrase, all merged records are automatically written to one or more occurrences of *file-name-4*. Files named in the GIVING phrase can be sequential, `relative, or indexed`.

If you use the OUTPUT PROCEDURE phrase, there are several rules you must follow in writing the procedure.

The procedure must consist of one or more `paragraphs` or sections that appear contiguously in the source program, and that are not part of any other procedure.

Since the RETURN statement is the means of making sorted or merged records available for processing, at least one such statement must appear in the procedure. The procedure may consist of any procedures needed to select, modify, or copy records. The records are returned one at a time in merged order from *file-name-1*.

Control must not be passed to a sort/merge output procedure except during the execution of a SORT or MERGE statement. The procedure itself can contain no SORT or MERGE statements, nor can it contain any explicit transfers of control to points outside its bounds; ALTER, GO TO, and PERFORM statements in the output procedure must refer to procedure names within the bounds of the output procedure. Note that implied transfers of control to declarative procedures are permitted. Thus, for example, a WRITE statement without an AT END phrase is permitted, and will transfer control to some associated declarative procedure if an AT END condition occurs.

The remainder of the PROCEDURE DIVISION must not contain any transfers of control to points inside sort/merge output procedures. No ALTER, GO TO, or PERFORM statements outside an output procedure can refer to procedure names within the output procedure.

When an output procedure is specified in a MERGE statement, the output procedure is executed as part of the merge operation. The procedure is used after the records have been merged.

At compile-time, the compiler inserts a return mechanism at the end of the last section in the output procedure. When this return mechanism is reached, the merge operation is terminated, and control passes to the next executable statement following the MERGE statement.

Segmentation Considerations

The following restrictions apply to the MERGE statement when it is used in a segmented program.

If the MERGE statement appears in a section whose segment number is less than 50, any output procedure named in the MERGE statement must either be totally contained within a segment (or segments) whose segment number (or numbers) is less than 50 or be wholly contained in a single segment whose segment number is greater than 49.

If the MERGE statement appears in a section whose segment number is greater than 49, any output procedure referenced by the MERGE statement must either be entirely contained within the same segment or be entirely contained within segments whose segment numbers are less than 50.

RELEASE Statement

The RELEASE statement can be used in an input procedure of a SORT statement to transfer records from your program to the initial phase of the sort operation.

Syntax

RELEASE *record-name-1* [FROM *identifier-1*]

Parameters

record-name-1 — the name of a logical record in the sort/merge file description entry of the file referenced in the associated SORT statement. It may be qualified. *Record-name-1* must not refer to the same storage area as *identifier-1*.

identifier-1 — the name of a data item described in your program, or a function-identifier. If it is a function, it must be an alphanumeric function. If it is not a function, *identifier-1* and *record-name-1* must not refer to the same storage area.

Description

A RELEASE statement may only be used within the range of an input procedure associated with a SORT statement for a file whose sort- merge file description entry contains *record-name-1*.

When the RELEASE statement is executed, a single record is made available, from *record-name-1*, to the initial phase of the sort operation. After the execution of the RELEASE statement, the logical record is no longer available in the record area named by *record-name-1* unless the associated sort/merge file is named in a SAME RECORD AREA clause. If the sort/merge file is named in the clause, the logical record is available to the program as a record of the other files named in the SAME AREA clause, as well as to the sort/merge file.

If the FROM phrase is used in a RELEASE statement, the contents of *identifier-1* are moved to *record-name-1*, then the contents of *record-name-1* are released to the sort file.

Although the logical record named by *record-name-1* might no longer be available, the data in *identifier-1* remains available following execution of the RELEASE statement (as stated in the preceding paragraph).

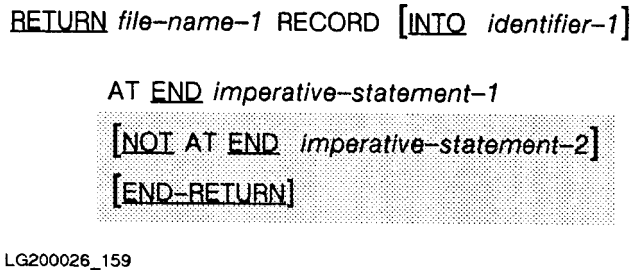
When control passes from the input procedure, the sort file consists of all those records placed in it by the execution of RELEASE statements.

RETURN Statement

The RETURN statement can be used in an output procedure of a SORT or MERGE statement. It cannot be used in any other type of procedure.

When used, the RETURN statement obtains either sorted records from the final phase of a sort operation, or merged records during a merge operation. Each record is obtained by a RETURN statement in the order specified by the keys listed in the SORT or MERGE statement. These records are made available for processing in the record area associated with the sort or merge file, and, optionally, to another data area.

Syntax



LG200026_159

Parameters

file-name-1	the name of the file used as the sort or merge file in the SORT or MERGE statement associated with the output procedure in which the RETURN statement appears. It must be described in a sort/merge file description entry (SD level) in the DATA DIVISION.
identifier-1	the name of a data item in your program. The storage area referenced by *identifier-1* must not be the same as the record area associated with *file-name-1*.
imperative-statement-1 and *imperative-statement-2*	one or more imperative statements.

Description

When logical records of a sort/merge file are described 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 the execution of the RETURN statement.

INTO Phrase

The INTO phrase, if specified in the RETURN statement, moves the current record into the record area associated with *file-name-1*, and then uses an implicit MOVE statement (without the CORRESPONDING phrase) to move a copy of the data from the record area to the storage area referenced by *identifier-1*. Thus, the data obtained from the SORT or MERGE statement is available in the data area associated with *identifier-1* as well as to the input record area.

Any subscripting or indexing associated with *identifier-1* is evaluated after the record has been returned to the file, and immediately before it is moved to the storage area referenced by *identifier-1*.

The INTO phrase must not be used when the input file contains logical records of various sizes as indicated by their record descriptions.

AT END Phrase

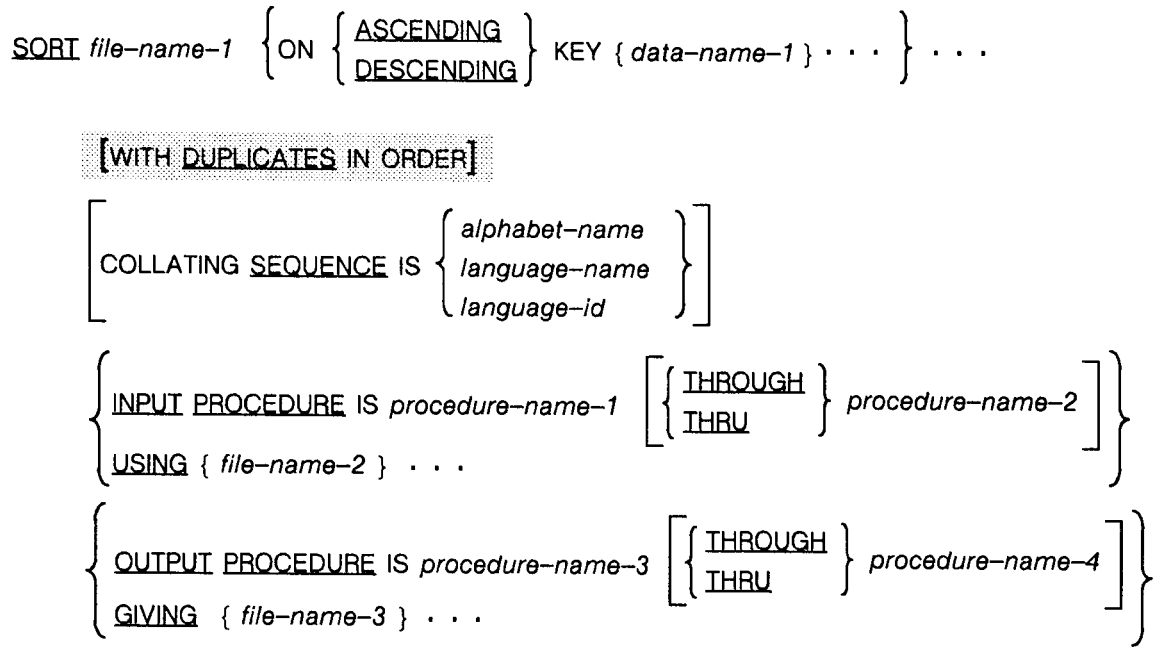
If no next logical record exits for the file at the time a RETURN statement executes, an AT END condition occurs. The contents of the record areas associated with the file when the AT END condition occurs are undefined; however, if the INTO phrase was used, the contents of *identifier-1* are the data moved into it by the preceding execution of the RETURN statement.

When the AT END condition occurs, the *imperative-statement-1* in the AT END phrase is executed. Following execution of *imperative-statement-1*, no RETURN statement may be executed as a part of the current output procedure. For more information on handling I/O errors, see "Input-Output Error Handling Procedures" in Chapter 8.

SORT Statement

The SORT statement creates a sort file either by executing an input procedure, or by transferring records from another file. The records of the sort file are then sorted using a specified set of keys, and are made available in sorted order to either an output procedure or an output file.

Syntax



LG200026_160

Parameters

file-name-1	a sort/merge file, and is described in a sort/merge file description entry in the DATA DIVISION.
data-name-1	data items described in records associated with *file-name-1*. Each may be qualified and may vary in length. None of these data names can be described by an entry that either contains an OCCURS clause, or is subordinate to an entry containing such a clause. If *file-name-1* has more than one record description, then the data items represented by these named need be described in only one of the record descriptions.
alphabet-name	a name defined by you in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.
language-name	an alphanumeric data item containing the name of the language whose collating sequence should be used. This parameter is an HP extension to the ANSI COBOL standard.

language-id — an integer data item containing the identification number of the language to use. This parameter is an HP extension to the ANSI COBOL standard.

file-name-2 — the files whose records are to be sorted. These files must not be open at the time the SORT statement is executed. Each must be a sequential, `relative`, or `indexed` file described in an FD level file description entry in the DATA DIVISION. No more than one of these file names may name a file on a multiple file reel. Any given file name can be used only once in a given SORT statement.

The actual size of the logical record or records described by these files must be equal to the actual size of the logical record or records described by *file-name-1*. If the data descriptions of the elementary items that make up these records are not identical, it is your responsibility to describe the corresponding records in such a way as to cause an equal number of character positions to be allocated for the corresponding records.

procedure-name-1 — the name of the first `paragraph` or section in an input procedure.

procedure-name-2 — the name of the last `paragraph` or section in an input procedure.

procedure-name-3 — the name of the first `paragraph` or section in an output procedure.

procedure-name-4 — the name of the last `paragraph` or section in an output procedure.

file-name-3 — the names of output files. These are subject to the same restrictions and rules as *file-name-2* above.

Description

The words THROUGH and THRU are equivalent, and can be used interchangeably.

The PROCEDURE DIVISION may contain more than one SORT statement appearing anywhere except in a declarative procedure, or in the input and output procedures associated with a SORT or MERGE statement. Files *file-name-2* and *file-name-3* must not be open when the SORT verb is executed or be opened or closed during the execution of an input or output procedure if such procedures are specified.

The data names following the word KEY are listed from left to right in order of decreasing significance. This decreasing significance is maintained from KEY phrase to KEY phrase.

When the SORT statement is executed, the records are first sorted in the specified order (ASCENDING or DESCENDING) using the most significant key data item. Next, within the groups of records having the same value for the most significant key data item, the records are sorted using the next most significant key data item, again, in ASCENDING or DESCENDING order as specified for that key. Sorting continues in this fashion until all key data items have been used.

Note Specifying the same file for the USING and GIVING file name is not recommended. The file contents may be contaminated if the SORT operation is abnormally terminated, for any reason. If the same file is to be used, you should ensure that a backup copy exists in case a file recovery becomes necessary.

For example, assume that the records to be sorted use the first three key data items, and that the unsorted records appear as shown below.

1	D	U	....
0	A	N	....
1	N	S	....
1	F	O	....
1	D	R	....
0	X	T	....
0	A	E	....
0	B	D	....
1	N	R	....
0	X	E	....
1	F	C	....
1	C	S	....

If the SORT statement uses the first character position as the most significant, and the third as the least significant, and the records are to be sorted in ascending order for the first two keys, and in descending order for the last key, then the results of each pass of the sort, as well as the SORT statement, are shown in the following.

Examples

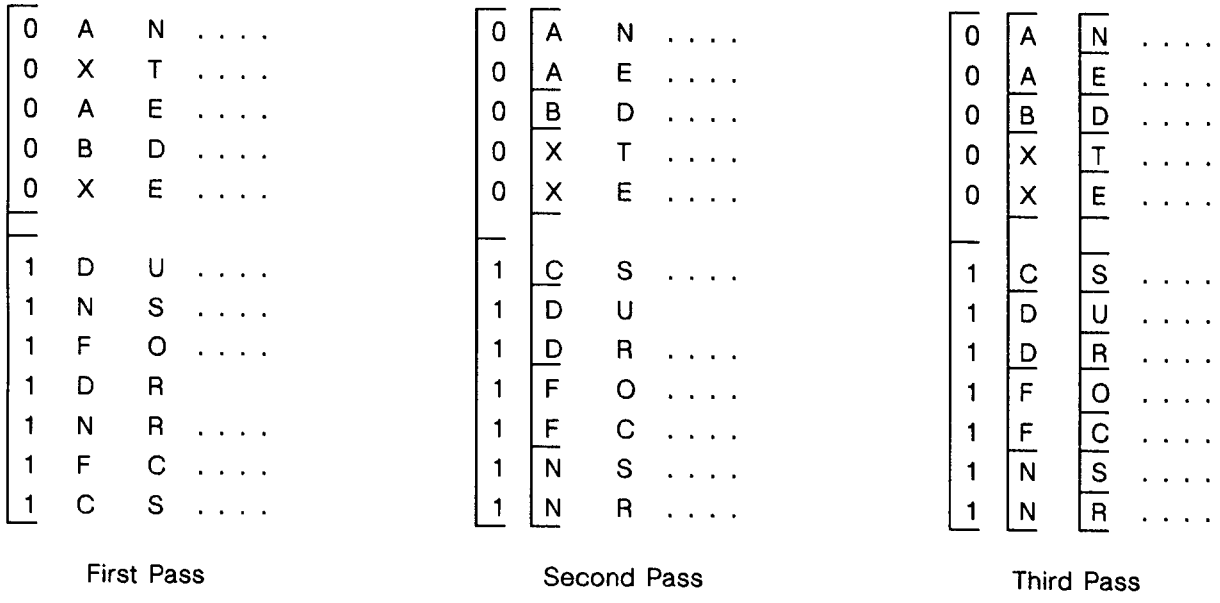
```
SORT TESTFILE ON ASCENDING KEY FIRSTCHAR, SECONDCHAR
                 ON DESCENDING KEY THIRDCHAR
                  USING INFILE
                   GIVING OUTFILE.
```

Where TESTFILE is, in part, described as:

```
SD TESTFILE.
01 TEST-REC.
   03 FIRSTCHAR   PIC 9.
   03 SECONDCHAR  PIC X.
   04 THIRDCHAR   PIC X.
        ⋮
```

and INFILE is described in part as:

```
FD INFILE.
01 IN-REC.
   03 FIRST    PIC 9.
   03 SECOND   PIC XX.
        ⋮
```



LG200026_162

Note that the third pass of the sort left the records unchanged from their order in the result of the second pass. The records are arranged in their proper sequences by chance. The SORT statement would not actually go through this third pass, as it recognizes the records as already being sorted. This saves execution time.

DUPLICATES Phrase

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:

1. 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.
2. 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.

ASCENDING and DESCENDING Phrases

When the ASCENDING phrase is used, the sorted records are in a sequence starting from the lowest value of the key data items, and continuing to the highest value.

When the DESCENDING phrase is used, the sorted records are in a sequence from the highest value of the key data items to the lowest value.

Sorting takes place according to the rules for comparison of operands of a relation condition.

COLLATING SEQUENCE Phrase

The COLLATING SEQUENCE phrase allows you to specify what collating sequence to use in the sorting operation. This phrase is optional. The program collating sequence is used if none is specified in the SORT statement. If you do not specify an alphabet name in the PROGRAM COLLATING SEQUENCE clause of the ENVIRONMENT DIVISION, the default is the ASCII collating sequence. See the alphabet name clause of the SPECIAL-NAMES paragraph in Chapter 6 for information on defining and using collating sequences.

USING and INPUT PROCEDURE Phrases

You must specify either the USING or the INPUT PROCEDURE phrase in a SORT statement.

If you specify the USING phrase, all records from *file-name-2* are automatically transferred to *file-name-1*. The files named by *file-name-2* must not be open when the SORT statement is executed. The file named in the USING phrase can be either relative or indexed.

The SORT process automatically opens these files, transfers their records, and then closes them. If USE procedures are specified for the files, they are executed at the appropriate times during these implicit operations. The files are closed as though a CLOSE statement, without any optional phrases, had been issued for them.

The records are then sorted in the file named by *file-name-1*, and are released to *file-name-3* or the specified output procedure during the final phase of the sort operation.

If you do not specify a USING phrase, then you must specify an INPUT PROCEDURE phrase.

If you do so, then *section-name-1* through *section-name-2* must define an input procedure.

Control is passed to this procedure before *file-name-1* is sorted. The compiler inserts a return mechanism at the end of the last statement in the statement in the section named by *section-name-2*, and when control passes to the last statement in the input procedure, the records that have been released to *file-name-1* are sorted.

The input procedure can consist of any procedure needed to select, modify, or copy the records that are made available contiguously by the RELEASE statement to the file referenced by *file-name-1*. This includes all statements executed by CALL, EXIT, GO TO, and PERFORM statements within the range of the input procedure. This also includes declarative procedure statements that are executed as a result of the execution statements within the range of the input procedure. The input procedure must not cause the execution of any MERGE, RETURN, or SORT statement.

GIVING and OUTPUT PROCEDURE Phrases

You must specify either the GIVING or the OUTPUT PROCEDURE phrase in a SORT statement.

If you specify the GIVING phrase, then, as a last step in the sort process, the sorted records in *file-name-1* are automatically written to each of the files referenced by *file-name-3*. The file can be relative or indexed.

File-name-2 and *File-name-3* must not be open when the sort process is executed.

The sort process automatically opens, writes records to, and closes *file-name-3*. If there are any USE procedures specified (whether implicitly or explicitly) for *file-name-3*, they are executed if and when appropriate, as part of the implicit function of the SORT statement. When all records have been written to *file-name-3*, the file is closed in a manner equivalent to issuing a CLOSE statement, with no optional phrases, for the file.

If you specify the OUTPUT PROCEDURE phrase, there are several rules you must follow in writing the procedure.

The output procedure can consist of any procedure needed to select, modify, or copy contiguously available records using the RETURN statement in sorted order from the file referenced by *file-name-1*. This includes all statements executed by CALL, EXIT, GO TO, and PERFORM statements within the range of the output procedure. This also includes declarative procedure statements executed as a result of executing statements within the range of the output procedure. The input procedure must not cause the execution of any MERGE, RELEASE, or SORT statement.

Since the RETURN statement is the means of making sorted or merged records available for processing, at least one such statement must appear in the procedure. The procedure may consist of any procedures needed to select, modify, or copy records. The records are returned one at a time in sorted order from *file-name-1*. The procedure itself can contain no RELEASE, SORT, or MERGE statements.

When an output procedure is specified in a SORT statement, the output procedure is executed as part of the sort operation. The procedure is used after the records have been sorted.

Output procedures, like input procedures, should be considered "slave" procedures designed exclusively for the use of the SORT statement. If the output procedures are not executed under the control of the SORT statement, the RETURN statement causes the job to abort.

Sorting Large Files

Normally, the file descriptions compiled into the object program (using information from the SELECT, FD, and SD clauses) provide all the information required to execute the object program. However, to override these file descriptions, you can supply MPE commands when executing the program. These commands are effective only for the current execution of the program.

You can use a :FILE command to specify the file size if the file to be sorted is not a disk file and contains more than 10,000 records. In this case, the :FILE command specifies the number of records in the SORT-FILE (the file named in the SD statement). For example, if a program contains the following statements:

```
SELECT SORT-FILE ASSIGN TO "SORT,DA".
SELECT NAME-FILE ASSIGN TO "INTAPE,UT".
                 ⋮
SD SORT-FILE RECORD CONTAINS 100 CHARACTERS.
                 ⋮
   SORT SORT-FILE DESCENDING KEY DATA-ONE USING NAME-FILE GIVING
           OUT-FILE.
```

And if the INTAPE magnetic tape file contains 15,000 records, enter the following :FILE command before executing the program:

```
:FILE SORT;DISC=15000    Maximum number of records SORT is expected to process.
```

You should also use this command if an input procedure passes more than 10,000 records to SORT.

An alternate method for specifying the file size is to include the file size parameter in the SELECT clause that defines the SORT-FILE (SD file name).

For example:

```
SELECT SORT-FILE ASSIGN TO "SORT,DA,,,15000".
```

If you do not use either of these methods and you are sorting more than 10,000 records, a "`TOO MANY INPUT RECORDS`" error condition results and the SORT program aborts. A disk file containing more than 10,000 records does not require a file size specification because SORT programmatically determines the file size.

Note

To determine the file size parameter value to be passed to the SORT subsystem with a CALL to SORTINIT, the compiler-generated code opens the specified SORT-FILE using FOPEN. The actual file size value that is used can be derived from the algorithm illustrated in Figure 12-1. SIZE-PARM is a temporary variable in Figure 12-1.

Once the file size parameter is determined, any attempt to send more records to the SORT subsystem with the RELEASE statement will cause a run-time abort with the SORT error message:

```
SORTLIB:   TOO MANY INPUT RECORDS
```

Note

It is possible that a permanent file with the same name as the SORT-FILE within job streams may cause undesired aborts. To avoid this, include the following command in the stream file:

```
:FILE SORT;NEW; DISC=15000
```

The file name used in the :FILE command is the one specified in the SELECT clause.

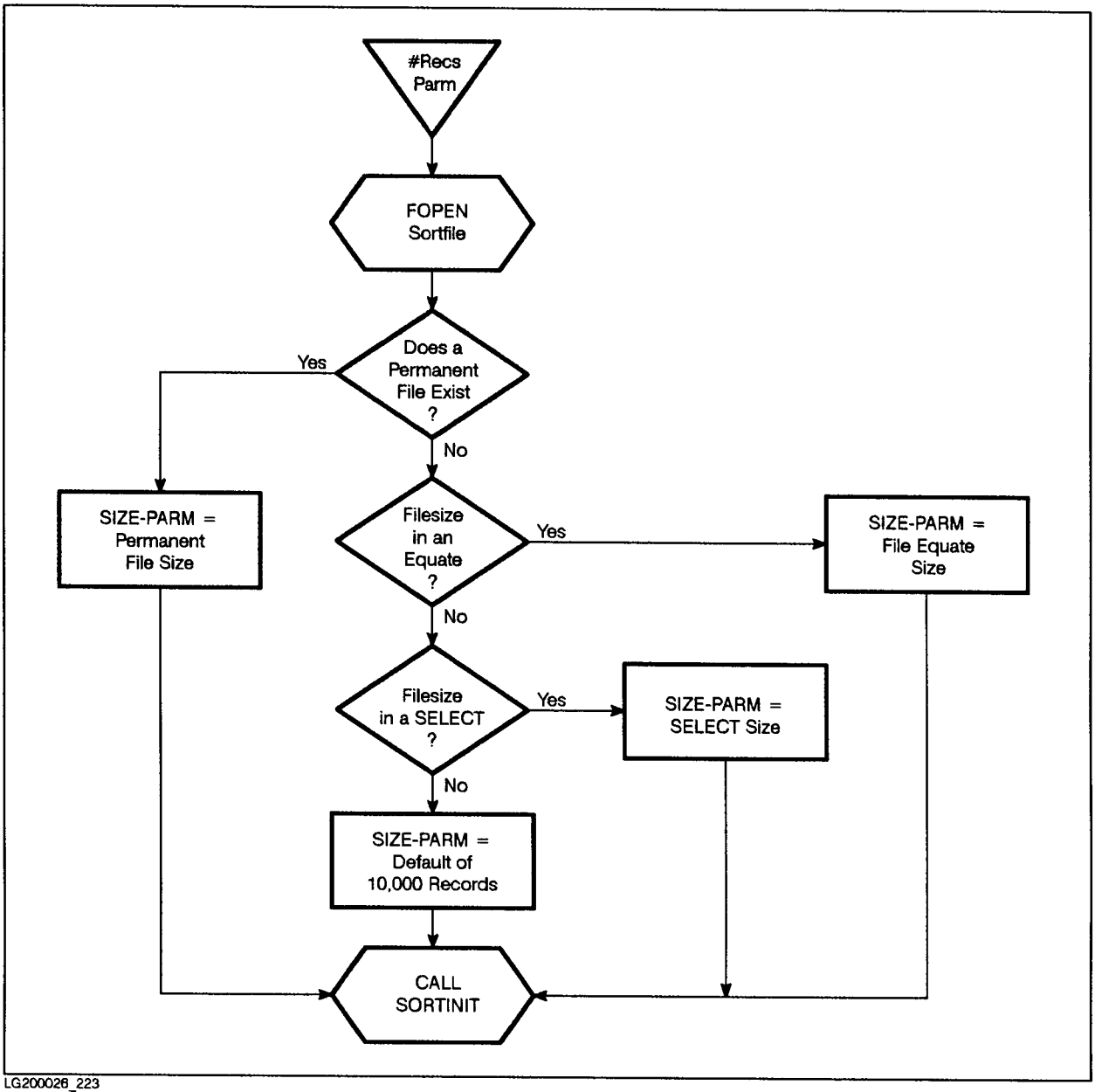


Figure 12-1. Determining Local File Size (SIZE-PARM) Used in FOPEN

Segmentation Considerations

The following restrictions apply to the SORT statement when it is used in a segmented program.

If the SORT statement appears in a section whose segment number is less than 50, then any output procedure named in the SORT statement either must be totally contained within a segment (or segments) whose segment number (or numbers) is less than 50, or must be entirely contained in a single segment whose segment number is greater than 49.

If the SORT statement appears in a segment whose segment number is greater than 49, then any output procedure referenced by the SORT statement must either be entirely contained within the same segment, or be entirely contained within segments whose segment numbers are less than 50.

13

Debug Module

The Debug Module provides a means by which you can describe a debugging algorithm including the conditions under which procedures are to be monitored during the execution of the object program.

Note With the exception of debug lines, this module is an obsolete feature of the 1985 ANSI COBOL standard.

The decisions of what to monitor and what information to display on the output device are explicitly yours. The COBOL debug facility simply provides a convenient access to pertinent information.

The HP COBOL II Debug Module implements the level 1 characteristics of the ANSI standard Debug Module. It provides a basic debugging capability, including the ability to specify: (a) selective or full procedure monitoring, and (b) optionally compiled debugging statements.

The features of the COBOL language that support the Debug Module are:

1. A compile-time switch: WITH DEBUGGING MODE

   The WITH DEBUGGING MODE clause is written as part of the SOURCE-COMPUTER paragraph. It serves as a compile-time switch over the debugging statements written in the program.

   When the WITH DEBUGGING MODE clause is specified in a program, all debugging sections and all debugging lines are compiled as specified in that section of the document. When the WITH DEBUGGING MODE clause is not specified, all debugging lines and all debugging sections are compiled as if they were comment lines.

2. An object-time switch: USE FOR DEBUGGING

   This switch dynamically activates the debugging code inserted by the compiler. This switch cannot be addressed in the program. It is controlled outside the COBOL environment. If the switch is "on", all the effects of the debugging language written in the source program are permitted. If the switch is "off", all the effects of the USE FOR DEBUGGING statement, are inhibited. Recompilation of the source program is required to provide or take away this facility.

   For information on setting the object-time debug module switch, refer to "System Dependencies" in Appendix H.

   This object-time switch has no effect on the execution of the object program if the WITH DEBUGGING MODE clause was not specified in the source program at compile time.

Note The object-time switch does not control the *debugging lines*. Recompilation without the "WITH DEBUGGING MODE" clause is necessary to disable the debugging lines.

3. A special register: DEBUG-ITEM

   The reserved word DEBUG-ITEM is the name for a special register generated automatically by the compiler. Only one DEBUG-ITEM is allocated per program. The names of the subordinate data items in DEBUG-ITEM are also reserved words.

4. Debugging lines

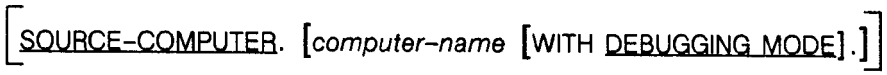
   A debugging line is any line with a "D" in column 7. These lines are added anywhere in the program after the OBJECT-COMPUTER paragraph and are used strictly for debugging purposes. Such lines make use of common COBOL statements such as "DISPLAY" in order to monitor program status.

   The contents of a debugging line must be totally independent of the program itself, so that a syntactically correct program is formed either with or without the debugging lines.

WITH DEBUGGING MODE Clause

The WITH DEBUGGING MODE clause indicates that all debugging sections and all debugging lines are to be compiled. If this clause is not specified, all debugging lines and sections are compiled as if they were comment lines. The WITH DEBUGGING MODE clause has the following format:

Syntax



LG200026_163

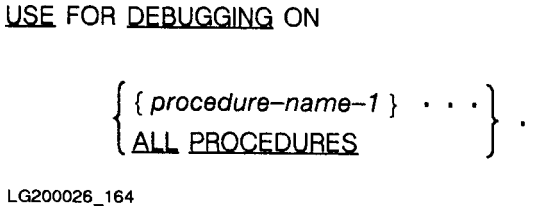
The following general rules apply to the WITH DEBUGGING MODE clause:

1. If the WITH DEBUGGING MODE clause is specified in the SOURCE-COMPUTER paragraph of the CONFIGURATION SECTION of a program, all USE FOR DEBUGGING statements and all debugging lines are compiled.
2. If the WITH DEBUGGING MODE clause is not specified in the SOURCE-COMPUTER paragraph of the CONFIGURATION SECTION of a program, any USE FOR DEBUGGING statements, all associated debugging sections, and any debugging lines are compiled as if they were comment lines.

USE FOR DEBUGGING statement

The USE FOR DEBUGGING statement identifies the user procedures that are to be monitored by the associated debugging section. The USE FOR DEBUGGING statement has the following format:

Syntax



The following syntax rules apply to the USE FOR DEBUGGING statement:

- Debugging section(s), if specified, must appear together immediately after the DECLARATIVES header.
- Except in the USE FOR DEBUGGING statement itself, there must be no reference to any nondeclarative procedure within the debugging section.
- Statements appearing outside of the set of debugging sections must not reference procedure names defined within the set of debugging sections.
- Except for the USE FOR DEBUGGING statement itself, statements appearing within a given debugging section may reference procedure names defined within a different USE procedure only with a PERFORM statement.
- Procedure names defined within debugging sections must not appear within USE FOR DEBUGGING statements.
- Any given procedure name may appear in only one USE FOR DEBUGGING statement and may appear only once in that statement.
- The ALL PROCEDURES phrase can appear only once in a program.
- When the ALL PROCEDURES phrase is specified, *procedure-name-1* must not be specified in any USE FOR DEBUGGING statement.
- References to the special register DEBUG-ITEM are restricted to references from within a debugging section.

USE FOR DEBUGGING

The following general rules apply to the USE FOR DEBUGGING statement:

1. When *procedure-name-1* is specified in a USE FOR DEBUGGING statement that debugging section is executed:

   a. Immediately before each execution of the named procedure.

   b. Immediately after the execution of an ALTER statement which references *procedure-name-1*.

2. The ALL PROCEDURES phrase causes the effects described in general rule #1 to occur for every procedure name in the program, except those appearing within a debugging section.

3. In the case of a PERFORM statement which causes iterative execution of a referenced procedure, the associated debugging section is executed once for each iteration.

   Within an imperative statement, each individual occurrence of an imperative verb identifies a separate statement for the purpose of debugging.

4. A reference to *procedure-name-1* as a qualifier does not constitute reference to that item for the debugging described in the general rules above.

5. Associated with each execution of a debugging section is the special register DEBUG-ITEM, which provides information about the conditions that caused the execution of a debugging section. DEBUG-ITEM has the following implicit description:

```
01 DEBUG-ITEM.
   02   DEBUG-LINE     PICTURE IS X(6).
   02   FILLER         PICTURE IS X    VALUE SPACE.
   02   DEBUG-NAME     PICTURE IS X(30).
   02   FILLER         PICTURE IS X    VALUE SPACE.
   02   DEBUG-SUB-1    PICTURE IS S9999   SIGN IS LEADING SEPARATE.
   02   FILLER         PICTURE IS X    VALUE SPACE.
   02   DEBUG-SUB-2    PICTURE IS S9999   SIGN IS LEADING SEPARATE.
   02   FILLER         PICTURE IS X    VALUE SPACE.
   02   DEBUG-SUB-3    PICTURE IS S9999   SIGN IS LEADING SEPARATE.
   02   FILLER         PICTURE IS X    VALUE SPACE.
   02   DEBUG-CONTENTS PICTURE IS X(30).
```

6. Prior to each execution of a debugging section, the contents of the data item referenced by DEBUG-ITEM are space filled. The contents of data items subordinate to DEBUG-ITEM are then updated, according to the following general rules, immediately before control is passed to that debugging section. The contents of any data item not specified in the following general rules remains spaces.

   Updating is accomplished in accordance with the rules for the MOVE statement, the sole exception being the move to DEBUG-CONTENTS. In this case, the move is treated exactly as if it was an alphanumeric to alphanumeric elementary move with no conversion of data from one form of internal representation to another.

7. The contents of DEBUG-LINE is the number of a particular source statement.

8. DEBUG-NAME contains the first 30 characters of the name that caused the debugging section to be executed.

   All qualifiers of the name are separated in DEBUG-NAME by the word "IN" or "OF"".

9. DEBUG-SUB-1, DEBUG-SUB-2, ... are treated only as filler items in HP COBOL II.
10. DEBUG-CONTENTS is a data item that is large enough to contain the data required by the following general rules.
11. If the first execution of the first nondeclarative procedure in the program causes the debugging section to be executed, the following conditions exist:
    a. DEBUG-LINE identifies the first statement of that procedure.
    b. DEBUG-NAME contains the name of that procedure.
    c. DEBUG-CONTENTS contains "START PROGRAM".
12. If a reference to *procedure-name-1* in an ALTER statement causes the debugging section to be executed, the following conditions exist:
    a. DEBUG-LINE identifies the ALTER statement that references *procedure-name-1*.
    b. DEBUG-NAME contains *procedure-name-1*.
    c. DEBUG-CONTENTS contains the applicable procedure name associated with the TO phrase of the ALTER statement.
13. If the transfer of control associated with the execution of a GO TO statement causes the debugging section to be executed, the following conditions exist:
    a. DEBUG-LINE identifies the GO TO statement whose execution transfers control to *procedure-name-1*.
    b. DEBUG-NAME contains *procedure-name-1*.
    c. DEBUG-CONTENTS contains spaces.
14. If reference to *procedure-name-1* in the INPUT or OUTPUT phrase of a SORT or MERGE statement causes the debugging section to be executed, the following conditions exist:
    a. DEBUG-LINE identifies the SORT or MERGE statement that references *procedure-name-1*.
    b. DEBUG-NAME contains *procedure-name-1*.
    c. DEBUG-CONTENTS contains:
        1. "SORT INPUT", if the reference to *procedure-name-1* is the INPUT phrase of a SORT statement.
        2. "SORT OUTPUT", if the references to *procedure-name-1* is in the OUTPUT phrase of a SORT statement.
        3. "MERGE OUTPUT", if the reference to *procedure-name-1* is in the OUTPUT phrase of a MERGE statement.
15. If the transfer of control from the control mechanism associated with a PERFORM statement caused the debugging section associated with *procedure-name-1* to be executed, the following conditions exist:
    a. DEBUG-LINE identifies the PERFORM statement that references *procedure-name-1*.
    b. DEBUG-NAME contains *procedure-name-1*.
    c. DEBUG-CONTENTS contains "PERFORM LOOP".

16. If *procedure-name-1* is a USE procedure that is to be executed, the following conditions exist:
    a. DEBUG-LINE identifies the statement that causes execution of the USE procedure.
    b. DEBUG-NAME contains *procedure-name-1*.
    c. DEBUG-CONTENTS contains "USE PROCEDURE".
17. If an implicit transfer of control from the previous sequential paragraph to *procedure-name-1* causes the debugging section to be executed, the following conditions exist:
    a. DEBUG-LINE identifies the previous executed statement at run time.
    b. DEBUG-NAME contains *procedure-name-1*.
    c. DEBUG-CONTENTS contains "FALL THROUGH".

Debugging Lines

A debugging line is any line with a "D" in column 7. It is only permitted in the program after the OBJECT-COMPUTER paragraph.

A debugging line will be considered to have all the characteristics of a comment line, if the WITH DEBUGGING MODE clause is not specified in the SOURCE-COMPUTER paragraph. Any debugging line that consists solely of spaces from margin A to margin R is treated the same as a blank line.

The contents of a debugging line must be such that a syntactically correct program is formed with or without the debugging line.

Successive debugging lines are allowed. Continuation of debugging lines is permitted, except that each continuation line must contain a "D" in column 7, and character strings may not be broken across two lines.

The ANSI Debug Module Example

```
001000 IDENTIFICATION DIVISION.
001100 PROGRAM-ID. EXAMPLE.
001200 ENVIRONMENT DIVISION.
001300 CONFIGURATION SECTION.
001400 SOURCE-COMPUTER. HP3000 WITH DEBUGGING MODE.
001500 OBJECT-COMPUTER. HP3000.
001600 DATA DIVISION.
001700 WORKING-STORAGE SECTION.
001800 01  GRP.
001900     05  N-1 PIC S9(4) VALUE 0
002000           SIGN IS LEADING SEPARATE.
002100     05  N-2 PIC S9(4) VALUE 0
002200           SIGN IS LEADING SEPARATE.
002300     05  N-3 PIC S9(4) COMP-3.
002400 PROCEDURE DIVISION.
002500 DECLARATIVES.
002600 DEBUG-SEC SECTION.
002700     USE FOR DEBUGGING ON ALL PROCEDURES.
002800 DEBUG-PAR-1.
002900     DISPLAY SPACE.
003000     DISPLAY "CURRENT PARA/SEC IS " DEBUG-NAME.
003100     DISPLAY "CONTROL FLOW WAS    " DEBUG-CONTENTS.
003200 END DECLARATIVES.
003300 SEC-1 SECTION 01.
003400 BEGIN-HERE.
003500     DISPLAY "ENTER N-1, 4 DIGITS".
003600     ACCEPT N-1 FREE.
003700     DISPLAY "N-1 = " N-1.
003800     PERFORM PAR-2.
003900     STOP RUN.
004000 PAR-2.
004100     DISPLAY "ENTER N-2, 4 DIGITS".
004200     ACCEPT N-2 FREE.
004300     DISPLAY "N-2 = " N-2.
004400     MOVE N-2 TO N-3.
004500D    DISPLAY "N-3 = " N-3.
```

Using the ANSI Debug Module Example

The above program displays the value of DEBUG-NAME and DEBUG-CONTENTS at the start of each section and paragraph. Note the use of the following source lines:

1. Source line 001400 - the WITH DEBUGGING MODE clause enables compilation of the debugging procedures and lines.
2. Source line 002700 - the USE FOR DEBUGGING ON ALL PROCEDURES sentence enables control within the Declaratives portion of the program and upon execution of ALL section and paragraph procedures.
3. Source line 004500 - the D, in column 7, declares a Source Debug Line. Debug lines are useful to display specific identifiers whenever the program is compiled with the WITH DEBUGGING MODE clause. When the clause is not specified, the lines are treated as comments.

To activate the debugging procedures at run time, the object-time switch must be set to 1. For example,

```
:RUN progname;PARM=1

CURRENT PARA/SEC IS SEC-1
CONTROL FLOW WAS    START PROGRAM

CURRENT PARA/SEC IS BEGIN-HERE
CONTROL FLOW WAS    FALL THROUGH
ENTER N-1, 4 DIGITS
123
N-1 = +0123

CURRENT PARA/SEC IS PAR-2
CONTROL FLOW WAS    PERFORM LOOP
ENTER N-2, 4 DIGITS
-1
N-2 = -0001
N-3 = -0001

END OF PROGRAM
```

14

Source Text Manipulation

This chapter describes the source text manipulation module, which is made up of the COPY statement `and the REPLACE statement.` These statements can function independently of or in conjunction with each other to provide an extensive capability to insert and replace source program text during source program compilation.

Note In this chapter, the term *word* implies *text word*.

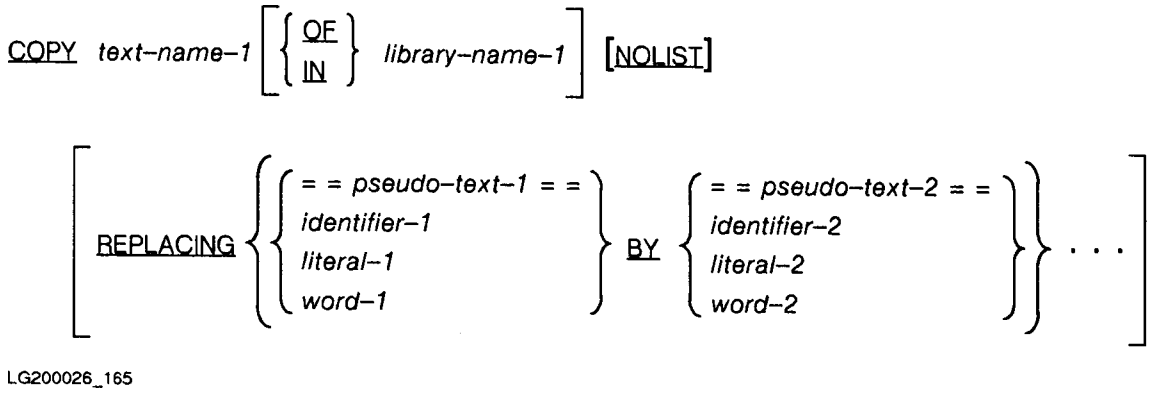
COPY Statement

The COPY statement is the method by which source records in a COBOL library are copied into your source program.

This statement may appear anywhere in your source program, from the IDENTIFICATION DIVISION to the end of the PROCEDURE DIVISION. Aside from allowing you to copy modules into your source file, it also allows you to replace occurrences of a string of words, a substring, an identifier, a literal, or a word appearing in the module being copied.

Syntax

The COPY statement has the following format:



LG200026_165

Parameters

text-name-1	the name of the module to be copied into your source program.
library-name-1	a name containing one to eight alphanumeric characters, the first of which must be alphabetic. This name is used to specify the library in which the module to be copied resides. *Library-name-1* must be used when you have more than one COBOL library in your log-on group. If *library-name-1* is not used, the COBOL compiler assumes that the library name is COPYLIB.
NOLIST	if used, indicates that the text of the module named by *text-name-1* is not included in the list file created by the compilation process. The NOLIST parameter is an HP extension to the ANSI COBOL standard.
==pseudo-text-1==	a sequence of words, comment lines, and spaces delimited on either end by double equal signs. It may consist of any text you wish, except that it must not consist of null text (that is,= = = =), all spaces, commas, semicolons, or all comment lines.

literal-1 and literal-2	each can be any COBOL literal.
identifier-1 and *identifier-2*	can each be any COBOL identifier and can be qualified.
word-1 and word-2	can each be any single COBOL word.

Note Where the sequence consists only of a single element, using the *identifier-1*, *literal-1*, or *word-1* format is more efficient.

==pseudo-text-2== a sequence of words, comment lines, or spaces delimited on either end by double equal signs. It may be any text you wish, including null text (that is, it may be of the form, ====).

Character strings within *pseudo-text-1* and *pseudo-text-2* may be continued. However, both equal signs forming the delimiters must be on the same line.

Description

A COPY statement may appear in a source program anywhere a character string or a separator is allowed. However, a COPY statement may not appear within another COPY statement. When a COPY statement is used, it must be preceded by a space and terminated by a period.

COPY statements are executed before source lines associated with them are sent to the compiler. Thus, only the lines of the copied module, including any replaced words, identifiers, and so forth, are sent to the compiler. The COPY statement itself is not.

Although the COPY statement is not sent to the compiler, it appears in the listing sent to the list file, along with the records of the copied module (unless NOLIST was specified).

If the REPLACING phrase is not used, the module is copied exactly as it appears in the library, including its sequence field and text-name, which appears in columns 73 through 80 of each record in the module.

REPLACING Phrase

To facilitate the following discussion, the REPLACING phrase is rewritten as shown below.

REPLACING `text-to-replace` BY `replace-text`

Before the comparison to determine which text, if any, is to be replaced in the copied module, spaces, commas, and semicolons to the left of the first word in the records of the module are moved into the source program. The first word of the record in a module is the first part of the module to be compared.

Starting with the left most word in the module being copied, and the first text-to-replace specified in the REPLACING phrase, text-to-replace is compared to an equivalent number of contiguous words in the module.

Text-to-replace matches the text in the module if and only if each character in text-to-replace equals the character in the corresponding position of the text in the module.

For purposes of matching, each occurrence of a separator comma, or semicolon in text-to-replace or in the text of the module is considered to be a single space. Each sequence of one or more spaces is considered to be a single space.

If no match occurs, the comparison is repeated with each next successive text-to-replace, if any, in the REPLACING phrase until either match is found or there is no next successive text-to-replace.

When all occurrences of text-to-replace have been compared to the text in the module and no match has occurred, the left most word in the text in the module is copied into your source program.

The next word of the text in the module is then considered as the left most word of the text in the module, and the comparison cycle is repeated, starting with the first text-to-replace in the REPLACING phrase.

Whenever a match occurs between text-to-replace and text in the module, replace-text is placed into the source program. The word immediately to the right of the text in the module which participated in the match is then considered as the left most word of the text in the module, and comparison begins again, starting with the first text-to-replace in the REPLACING phrase.

The comparison operation continues until the rightmost word in the last line of the module has either participated in a match or has been considered as the left most word of text in the module and has participated in a complete comparison cycle.

A comment line in text-to-replace or in the module is interpreted, for purposes of matching, as a single space. Comment lines appearing in replace-text and in the module are copied into the source program unchanged.

The syntactic correctness of the lines in a module cannot be independently determined. Nor can the syntactic correctness of the entire COBOL source program until all COPY statements have been completely processed.

The following are two examples of the COPY statement.

Example 1

This example uses a module named RITESTUF in a library called UTIL to illustrate the COPY statement.

```
001000 WRITE-ROUTINE.                                    RITESTUF
001100      OPEN OUTPUT CHECKS.                          RITESTUF
002000      WRITE AMOUNT BEF-AFT ADVANCING X LINES       RITESTUF
003000           AT EOP IMP-STAT.                        RITESTUF
```

The COPY statement appears as follows:

```
100000 PROCEDURE DIVISION.
                ⋮
102100       COPY RITESTUF OF UTIL
102200           REPLACING CHECKS BY FILEOUT
102300           ==AMOUNT== BY ==RECOUT==
102310           ==X== BY ==1==
102400           ==BEF-AFT== BY ==BEFORE==
102500           ==IMP-STAT== BY ==PERFORM PAGER==.
102600       CLOSE FILEOUT.
                ⋮
```

Results of the COPY statement:

```
102500
001000  WRITE-ROUTINE.                                RITESTUF
001100       OPEN OUTPUT FILEOUT.                     RITESTUF
002000       WRITE RECOUT BEFORE ADVANCING 1 LINES    RITESTUF
003000            AT EOP PERFORM PAGER.               RITESTUF
102600       CLOSE FILEOUT.
```

Example 2

This example illustrates how the COPY statement can copy substrings in library text. This is done by putting parentheses around the substring to be replaced and around *pseudo-text-1*. The name of the module is PRODUCT1 in library MFG1.

```
001000 01   (FIRST)-RECORD       PIC X(80).                PRODUCT1
```

The COPY statement appears as follows:

```
COPY PRODUCT1 OF MFG1 REPLACING ==(FIRST)== BY ==MAIN-INPUT==.
```

Below are the results of the COPY statement:

```
001000 01   MAIN-INPUT-RECORD    PIC X(80).                PRODUCT1
```

You cannot use COPY REPLACING to replace substrings within nonnumeric literals. For example, if the following DISPLAY statement were in a program, it would not be changed by the COPY REPLACING statement above:

```
DISPLAY  "(FIRST)-RECORD".
```

REPLACE Statement

The REPLACE statement is used to replace source program text. This statement may appear anywhere in a program, from the IDENTIFICATION DIVISION to the end of the PROCEDURE DIVISION. However, a better way to edit the source text permanently is by using an editor.

Syntax

Format 1

REPLACE ==*pseudo-text-1*== BY ==*pseudo-text-2*== . . .

Format 2

REPLACE OFF

LG200026_166a

Parameters

==pseudo-text-1== a sequence of words, comments, or spaces delimited on either end by double equal signs. It may consist of any text, except that the text may not consist of null text (that is,====), all spaces, commas, semicolons, or all comment lines.

Note Where the sequence consists only of a single element, using the *identifier-1*, *literal-1* or *word-1* format is more efficient.

==pseudo-text-2== a sequence of words, comment lines, or spaces delimited on either end by double equal signs. It can be any text, including null text (that is, ====).

Character strings within *pseudo-text-1* and *pseudo-text-2* may be continued. However, both equal signs forming the delimiters must be on the same line.

Description

A REPLACE statement can appear in a source program anywhere a character string or a separator can occur. However, a REPLACE statement cannot appear in another REPLACE statement. When a REPLACE statement is used, it must be preceded by a space and terminated by a period.

Any REPLACE statements contained in a source program are processed after the processing of the COPY statements contained in a source program.

The text produced as a result of processing a REPLACE statement cannot contain a REPLACE statement.

Note Although the REPLACE statement is not sent to the compiler, it appears in the listing sent to the listfile.

To facilitate the following discussion, the REPLACE statement is rewritten as shown below.

```
REPLACE text-to-replace BY replace-text
```

The format 1 REPLACE statement specifies the text of the source program to be replaced by the corresponding text. Each matched occurrence of the *text-to-replace* in the source program is replaced by the corresponding *replace-text*.

The format 2 REPLACE statement specifies that any text replacement currently in effect is discontinued.

A given occurrence of the REPLACE statement is in effect from the point at which it is specified until the next occurrence of the REPLACE statement or the end of the separately compiled program, respectively. For example, a REPLACE statement could be in effect until the END PROGRAM header of a program that is not contained in another program is encountered.

Starting with the leftmost word in the source program and the first text-to-replace, the corresponding text-to-replace is compared to an equivalent number of contiguous words in the source program.

Text-to-replace matches the source program text only if each character in text-to-replace equals the character in the corresponding position of the source program text.

For purposes of matching, each occurrence of a separator comma, or semicolon in text-to-replace or in the source program text, is considered to be a single space. Each sequence of one or more spaces is considered to be a single space.

If no match occurs, the comparison is repeated with each successive text-to-replace (if there is any), until either a match is found or there is no successive text-to-replace.

The next word of the source program text is then considered as the leftmost word of the source program text. The comparison is repeated, starting with the first text-to-replace.

Whenever a match occurs between text-to-replace and source program text, replace-text is placed into the source program. The word immediately to the right of the matching source program text is then considered as the leftmost word of the data. Comparison begins again, starting with the first text-to-replace.

The comparison operation continues until the rightmost word in the source program text (within the scope of the REPLACE statement) has either been matched or has been considered as a leftmost word in the source program and completed a comparison cycle.

A comment line in text-to-replace or in the source program is interpreted as a single space for purposes of matching.

The syntactic correctness of the lines in a source program cannot be independently determined, neither can the syntactic correctness of the entire COBOL source program be determined until all REPLACE statements have been completely processed.

The following are examples using the REPLACE statement.

Example 1

In this example, the 01 identifier "TEST" is replaced, but not the "TEST" in the PERFORM statement. This is due to the intervening REPLACE statement. A better way to produce the same result is to edit the source text permanently using an editor.

Before REPLACE is executed:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    PROG1.
DATA DIVISION.
REPLACE   ==TEST==  BY ==TESTT==
          ==TRUE==  BY ==TRUE-FLAG==.
    01  NAME   PIC   X(30).
    01  TEST   PIC   X.
        88   TRUE    VALUE  "T".
PROCEDURE DIVISION.
P1.
    ACCEPT TEST.
    IF TRUE   PERFORM P2.
    REPLACE ==ALPHABETIC==  BY ==ALPHABETIC-UPPER==.
    IF NAME IS ALPHABETIC THEN
       SET TRUE-FLAG TO TRUE.
    REPLACE OFF.
    PERFORM P3 WITH TEST AFTER
               UNTIL NAME IS NOT ALPHABETIC.
                  ⋮
```

The code sent to the compiler would be:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   PROG1.
DATA DIVISION.
    01 NAME   PIC  X(30).
    01 TESTT  PIC  X.
       88   TRUE-FLAG  VALUE "T".
PROCEDURE DIVISION.
P1.
    ACCEPT TESTT.
    IF TRUE-FLAG  PERFORM P2.
    IF NAME IS ALPHABETIC-UPPER THEN
       SET TRUE-FLAG TO TRUE.
    PERFORM P3 WITH TEST AFTER
             UNTIL NAME IS NOT ALPHABETIC.
               ⋮
```

Example 2

This example shows how the REPLACE statement can be used to replace substrings. This is done by putting parentheses around the substring to be copied and around *pseudo-text-1*.

Assume the source program contains the following text before the replacement:

```
          01   (FIRST)-RECORD        PIC X(80).
```

After the following REPLACE statement

```
REPLACE COPY-MODULE REPLACING ==(FIRST)== BY ==INPUT==.
```

the resultant text is:

```
          01   INPUT-RECORD          PIC X(80).
```

A

HP COBOL II Error Messages

This appendix explains how to read compiler error messages that occur when using the HP COBOL II compiler and run-time error messages that occur when your program is running. Error messages and their explanatory text are in the file named `COBCAT.PUB.SYS`. This appendix contains a complete listing of the COBCAT error messages. You can also obtain the most current listing of error messages for each compiler version update by listing the file `COBCAT.PUB.SYS`. For example, in MPE use the following command to list the file on the screen:

```
:PRINT COBCAT.PUB.SYS
```

Reading Error Messages from COBCAT

The file `COBCAT.PUB.SYS` contains the most current list of HP COBOL II error messages. Keep the following in mind when reading error messages from `COBCAT`:

- The error messages appear on the same line number as the corresponding error message numbers. For example, line number 23 contains the text for error message 23.
- A "$" in the left margin indicates a comment line containing explanatory text.
- An "!" indicates that an item is place holder. The specific item appears in the message text when the message is emitted by the compiler.
- Numbers within brackets "[ ]" indicate the corresponding file status codes for the run-time errors.
- When two items appear in brackets like this: `{xxx/yyy}`, the first item (`xxx`) refers to the HP COBOL II/XL compiler and the second item (`yyy`) refers to the HP COBOL II/V compiler.

Example

```
001 ILLEGAL CHARACTER IN COLUMN 7.

$  Only space, *, -, $, D, or / allowed.
```

Note All requests to "contact HP" in error messages should be interpreted to mean "Please submit a service request (SR) and the necessary source and object files to allow duplication of the problem being reported." For more details, see the file named `COBCAT.PUB.SYS`.

HP COBOL II error messages have been divided into seven categories:

Table A-1. Kinds of Error Messages

Message Number	Classification	Meaning
1-99	Warnings (W)	Something is incorrect in the code, but the compiler can probably fix it to produce what you intended.
100-399	Questionable Errors (Q)	An error has occurred that will be fixed, but the compiler probably will not produce what you intended.
400-449	Serious Errors (S)	The error is either too difficult or impossible to fix. No code is generated.
450-499	Disastrous Errors (D)	The error makes further processing risky or impossible. All files are closed and processing is stopped immediately.
500-539	Nonstandard Warnings (N)	The program uses a construct that is not part of ANSI COBOL'85 standard or is an obsolete or incompatible feature.
540-899	Run-Time Errors	These are errors that occur when your program is executing.
900-999	Informational Messages (I)	Other messages usually giving status after other errors.

Compile-Time Error Messages

Error messages numbered from 1 through 539 and 900 through 999 are compile-time errors. Messages numbered from 540 through 899 are run-time errors. See the following section for information about run-time errors.

HP COBOL II is a multiple pass compiler, which has the following implications for compile-time error messages:

- If the control option CHECKSYNTAX is used, errors checked in the last pass (the object code generation phase) of the compiler are not diagnosed.
- Serious errors found in the first pass inhibit code generation and the execution of the last compiler pass. So if all serious errors are fixed and the program is recompiled, additional errors may be diagnosed during the processing of the final pass.
- Errors found by the final pass of the compiler show the line number of the statement in error and a column number of 80. This occurs because column number information is not carried along to the final pass, and it is often not meaningful for errors detected in this pass.

Run-Time Error Messages

Run-time error messages are numbered from 540 through 899. They are at the end of the `COBCAT.PUB.SYS` file within the $SET 30 and $SET 31 catalog sets. These error messages are issued when run-time procedures encounter errors while performing the following:

- Input-output functions.
- Converting or moving illegal data.
- Detecting invalid index, subscript, or reference modification values.
- Performing exponential computations.
- When size error conditions occur and a SIZE ERROR clause has not been used.

Refer to "Run-Time Errors" later in this appendix for a complete list of run-time errors.

Note For additional information on error messages, refer to "System Dependencies" in Appendix H.

Warnings

1	ERROR MESSAGE	ILLEGAL CHARACTER IN COLUMN 7.
	CAUSE	Only space, *, -, $, D, or / allowed.
2	ERROR MESSAGE	DEBUGGING LINE ILLEGAL BEFORE OBJECT-COMPUTER PARAGRAPH.
	CAUSE	
3	ERROR MESSAGE	TOO MANY CHARACTERS IN SYMBOL !.
	CAUSE	Symbol ! is limited to 30 characters.
4	ERROR MESSAGE	MISSING SPACE.
	CAUSE	Separator space is needed here.
5	ERROR MESSAGE	CONTINUATION RECORD NOT ALLOWED HERE.
	CAUSE	
6	ERROR MESSAGE	MISSING QUOTE.
	CAUSE	QUOTE is needed in nonnumeric literal.
7	ERROR MESSAGE	! NOT IMPLEMENTED
	CAUSE	MULTIPLE SYSTEM FILE NAMES or INTEGER preceding SYSTEM FILE NAME or RERUN clause are all ignored.
8	ERROR MESSAGE	CALL ! ASSUMED TO BE CALL INTRINSIC
	CAUSE	An intrinsic will be called instead of a subprogram. Only occurs when $CONTROL CALLINTRINSIC specified.
9	ERROR MESSAGE	FILES IN MULTIPLE FILE TAPE CLAUSE MUST BE SEQUENTIAL.
	CAUSE	
13	ERROR MESSAGE	ERROR CONVERTING THE WRITE ADVANCING COUNT TO AN INTEGER.
	CAUSE	A "WRITE ADVANCING" specified a number of lines to advance that caused an error when converting to integer. A value of one is used in this case.
15	ERROR MESSAGE	DELETE VALID ONLY WITH RELATIVE OR INDEXED.
	CAUSE	The "DELETE" verb can only be used with RELATIVE or INDEXED I/O.

16	ERROR MESSAGE	START VALID ONLY WITH RELATIVE, RANDOM OR INDEXED.
	CAUSE	The "START" verb can only be used with RELATIVE, INDEXED or RANDOM I/O. Not valid with ACCESS mode RANDOM.
18	ERROR MESSAGE	SEEK VALID ONLY WITH RELATIVE OR RANDOM.
	CAUSE	The "SEEK" verb can only be used with RANDOM or RELATIVE files.
20	ERROR MESSAGE	INVALID DATA TYPE FOR KEY.
	CAUSE	Sort or Merge keys may only be of the following types: ALPHABETIC, ALPHANUMERIC, NUMERIC, or DISPLAY.
24	ERROR MESSAGE	A SECTION NAME IS REQUIRED IN DECLARATIVES.
	CAUSE	Use of DECLARATIVES requires sections.
25	ERROR MESSAGE	COLLATING SEQUENCE ! HAS NOT BEEN DEFINED.
	CAUSE	ALPHABET name for COLLATING SEQUENCE is not specified.
26	ERROR MESSAGE	HYPHEN NOT ALLOWED AT END OF WORD.
	CAUSE	
27	ERROR MESSAGE	SPACE NOT ALLOWED IN THIS POSITION.
	CAUSE	Embedded space in numeric literal not allowed.
28	ERROR MESSAGE	HIGH-VALUE/LOW-VALUE HAS NOT BEEN DEFINED.
	CAUSE	HIGH-VALUE/LOW-VALUE not defined because of undefined collating sequence.
29	ERROR MESSAGE	OPEN REVERSED NOT SUPPORTED.
	CAUSE	Open is generated but "REVERSED" is ignored.
30	ERROR MESSAGE	NON-88 LEVEL ITEM IN FILE SECTION HAS VALUE CLAUSE.
	CAUSE	The value clause is accepted by this compiler.
31	ERROR MESSAGE	VALUE CLAUSE ON NON-88 LEVEL ITEM IN LINKAGE SECTION OR ON EXTERNAL ITEM, IGNORED.
	CAUSE	These items do not belong to the current program and so cannot be initialized here with a VALUE clause.

32	ERROR MESSAGE	OCCURS CLAUSE USED ON 01 LEVEL ITEM.
	CAUSE	
33	ERROR MESSAGE	VALUE CLAUSE IN AN ITEM SUBORDINATE TO AN OCCURS TABLE.
	CAUSE	The VALUE clause is not permitted within an entry subordinate to an OCCURS. ANSI74 entry point only.
34	ERROR MESSAGE	MISSING AT END/INVALID KEY PHRASE.
	CAUSE	AN AT END or INVALID KEY phrase is required since no applicable USE procedure is specified.
35	ERROR MESSAGE	ILLEGAL INVALID KEY PHRASE.
	CAUSE	An INVALID KEY PHRASE is not legal with the access mode specified for the file. Instead, use either FILE STATUS or a USE procedure.
36	ERROR MESSAGE	PARAMETER # ! CONVERTED TO COMP.
	CAUSE	An alphanumeric item is passed by value to an INTEGER, LOGICAL or a DOUBLE. The alphanumeric item is treated as numeric DISPLAY and is converted to COMP (binary).
37	ERROR MESSAGE	PARAMETER ! IS LITERAL PASSED BY REFERENCE.
	CAUSE	A literal is passed to an intrinsic which expects a reference parameter. The literal is stored in a temporary location whose address is then passed to the intrinsic.
38	ERROR MESSAGE	\\ IGNORED FOR PARAMETER # !. INTRINSIC EXPECTS REFERENCE PARAMETER.
	CAUSE	A parameter enclosed in \\ is passed to an intrinsic which expects a reference parameter. The \\ are ignored and the parameter is passed by reference.
39	ERROR MESSAGE	@ IGNORED FOR PARAMETER # !. INTRINSIC EXPECTS WORD ADDRESS.
	CAUSE	A parameter preceded by an @ is passed to an intrinsic which expects a word address. The @ is ignored and the word address of the item is passed.
40	ERROR MESSAGE	BLANK LINE WITH CONTINUATION CHARACTER WAS NOT PROCESSED.
	CAUSE	
41	ERROR MESSAGE	MISSING PERIOD IN IDENTIFICATION DIVISION.
	CAUSE	

42	ERROR MESSAGE	INVALID COMMENT ENTRY.
	CAUSE	Comment entry is not allowed here.
43	ERROR MESSAGE	ILLEGAL IDENTIFICATION DIVISION PARAGRAPH.
	CAUSE	Check against legal IDENTIFICATION DIVISION paragraphs.
45	ERROR MESSAGE	UNPROCESSED SOURCE ON SUBSYSTEM COMMAND LINE.
	CAUSE	Unprocessed source is ignored.
46	ERROR MESSAGE	SUBSYSTEM COMMAND CHARACTER STRING WAS TRUNCATED.
	CAUSE	String exceeds maximum allowed length, which is approximately 110 characters for $PAGE and $TITLE, 116 characters for $COPYRIGHT, and 255 characters for $VERSION.
47	ERROR MESSAGE	SEQUENCING ERROR.
	CAUSE	Sequence numbers are out of order.
48	ERROR MESSAGE	MISSING "BY" IN COPY OR REPLACE STATEMENT.
	CAUSE	COPY or REPLACE statement is incomplete.
49	ERROR MESSAGE	INVALID SUBSYSTEM COMMAND DELIMITER; EXPECTED !.
	CAUSE	
50	ERROR MESSAGE	ARITHMETIC OVERFLOW MAY OCCUR.
	CAUSE	An intermediate or final result may have more than 31 digits on HP COBOL II/XL and 28 digits on HP COBOL II/V, when maximum operands and intermediate results are assumed in the arithmetic statement.
51	ERROR MESSAGE	REDEFINING ITEM ! IS SMALLER THAN REDEFINED ITEM.
	CAUSE	Except for level 01, a redefining item must be the same size as the item it redefines. ANSI85 allows it to be smaller.
52	ERROR MESSAGE	REDEFINING ITEM ! IS BIGGER THAN REDEFINED ITEM.
	CAUSE	Except for level 01, a redefining item must be the same size as the item it redefines.
54	ERROR MESSAGE	CODE-SET CLAUSE SPECIFIED FOR A MASS-STORAGE FILE.
	CAUSE	The code-set clause may not be specified for mass-storage files. ANSI74 entry point only.

55	ERROR MESSAGE	LEFT TRUNCATION MAY OCCUR.
	CAUSE	This warning is generated whenever significant digits will be truncated in a numeric move.
56	ERROR MESSAGE	VALUE OF CLAUSE NOT APPLICABLE TO NON-SEQUENTIAL FILES, IGNORED.
	CAUSE	The VALUE OF clause is meaningful only for sequential files. If it occurs in any other type of file it is ignored.
57	ERROR MESSAGE	REDEFINING ITEM ! CONTAINS A VALUE CLAUSE.
	CAUSE	An item with a VALUE clause contains a REDEFINES clause or is subordinate to a REDEFINES clause. The latter VALUE clause is used.
58	ERROR MESSAGE	UNABLE TO OPEN COBCNTL FILE.
	CAUSE	The file COBCNTL.PUB.SYS was not found.
59	ERROR MESSAGE	MISSING PROCEDURE DIVISION.
	CAUSE	No code will be generated. Syntax checking is done.
60	ERROR MESSAGE	OVERLAPPING PARAMETERS IN CALL; CALLED SUBPROGRAM MUST NOT SPECIFY $CONTROL PARMSNEVEROVERLAP.
	CAUSE	If overlapping parameters exist, the linkage section of called subprogram must be aliased conservatively.
61	ERROR MESSAGE	NO OPTIMIZATION DONE IF ALTER STATEMENT OCCURS.
	CAUSE	No alias sets constructed, optimization is turned off.
62	ERROR MESSAGE	NO OPTIMIZATION DONE IF SYMDEBUG IS SPECIFIED.
	CAUSE	Listing/object correspondence altered with optimization. Optimization turned off.
65	ERROR MESSAGE	OCTAL LITERAL CONVERTED TO DECIMAL.
	CAUSE	A condition name value clause specifies octal literal when numeric conditional variable has usage display or comp-3. The literal was converted to an equivalent base 10 number.
66	ERROR MESSAGE	RECURSION DETECTED; CALL TO EXTERNAL PROGRAM ! ASSUMED.
	CAUSE	Illegal recursive call to a nested or batched program was encountered and the compiler responded by generating code for a legal call to a separately compiled program.

Questionable Errors

100	ERROR MESSAGE	ILLEGAL INTEGER.
	CAUSE	A non-integer literal or an integer which is too large has occurred in a context where the compiler expects an integer.
101	ERROR MESSAGE	ALPHABET-NAME IN CODE-SET CLAUSE MAY NOT SPECIFY LITERAL PHRASE.
	CAUSE	An alphabet-name which was defined using the literal phrase has been used in a code set clause.
102	ERROR MESSAGE	DATA-NAME ! IS NOT AN ALPHABET-NAME OR CLASS-NAME.
	CAUSE	The data-name must be declared to be alphabet-name or class-name.
103	ERROR MESSAGE	ALPHABET-NAME OR CLASS-NAME ! IS NOT DECLARED.
	CAUSE	The data name must be declared to be alphabet-name or class-name.
104	ERROR MESSAGE	NULL LITERAL NOT ALLOWED; REPLACED BY SPACE.
	CAUSE	A nonnumeric literal was found which did not contain any characters between the opening and closing quotation marks. It was replaced with a one-character literal consisting of the space character.
105	ERROR MESSAGE	NUMBER OF SYMBOLIC CHARS ! INTEGERS.
	CAUSE	The number of symbolic characters must equal the # of integers.
106	ERROR MESSAGE	CLAUSES IN SPECIAL-NAMES OUT OF ORDER.
	CAUSE	The clauses in SPECIAL-NAMES paragraph must be in order: mnemonic, alphabet, symbolic character, class, currency and decimal-point.
107	ERROR MESSAGE	ILLEGAL CONFIGURATION SECTION.
	CAUSE	Nested programs must not contain a CONFIGURATION SECTION.
108	ERROR MESSAGE	ERROR ! GETTING NLS INFORMATION.
	CAUSE	Language not installed or system variable not set.

109	ERROR MESSAGE	BAD RECORDING MODE SPECIFICATION!.
	CAUSE	The RECORDING MODE specification must be “F”, “V”, “U”, or “S”.
112	ERROR MESSAGE	NEGATIVE NUMBER OF OCCURRENCES SPECIFIED.
	CAUSE	A table may not contain a negative number of occurrences.
113	ERROR MESSAGE	MINIMUM NUMBER OF OCCURRENCES IS GREATER THAN MAXIMUM NUMBER OF OCCURRENCES.
	CAUSE	The minimum number of occurrences in a table must be greater than the maximum number of occurrences of the table.
121	ERROR MESSAGE	MORE THAN {500/123} PAIRS IN A VALUE CLAUSE.
	CAUSE	Level 88 value clauses can have at most 500 pairs on HP COBOL II/XL and 123 pairs on HP COBOL II/V.
122	ERROR MESSAGE	MORE THAN 18 DIGITS IN A NUMERIC PICTURE.
	CAUSE	Numeric data items have at most 18 digits.
123	ERROR MESSAGE	MULTIPLE OCCURRENCES OF !.
	CAUSE	For data items, only one BLANK WHEN ZERO, JUSTIFIED, OCCURS, PICTURE, SYNCHRONIZED, USAGE, SIGN, or VALUE clause may appear. For file descriptions, only one BLOCK CONTAINS, DATA RECORDS, LABEL RECORDS, RECORD CONTAINS, RECORDING MODE, REPORT, VALUE OF, LINKAGE, or CODE-SET clause may appear. For the VALUE OF clause, only one VOL, LABELS, SEQ, or EXDATE phrase may appear. For the LINKAGE clause, only one FOOTING, TOP, or BOTTOM phrase may appear. For a CD only one of each INPUT or OUTPUT clause may appear.
124	ERROR MESSAGE	OCCURS DEPENDING ON ITEM FOLLOWED BY !.
	CAUSE	The OCCURS DEPENDING ON item must be the last item in a record.

125	ERROR MESSAGE	ILLEGAL EXTERNAL CLAUSE. (!)
	CAUSE	External clause must be on a FD or 01 in working storage section. The name must be unique. (See error 351). The item must be contain a redefines or renames clause. For files, it must not have the same area or multiple file clause. Redefining a 01 external item can't be larger. (See warning 52.)
126	ERROR MESSAGE	ILLEGAL GLOBAL CLAUSE. (!)
	CAUSE	Global clause must be a FD or on a 01 not in linkage section. The name must be unique and can't be FILLER. For files, it must not have the same record area clause. A GLOBAL USE procedure is not allowed on a local file. A GLOBAL and a local USE procedure declared in the same scope may not reference the same file or io-mode.
132	ERROR MESSAGE	88 LEVEL ON GROUP ITEM ! CONTAINING NON-DISPLAY USAGE, JUST, OR SYNC
	CAUSE	A condition-name cannot be associated with a group containing items whose descriptions include JUSTIFIED, SYNCHRONIZED, or USAGE other than DISPLAY. The compiler treats the group as though there were no USAGE, JUSTIFIED, or SYNCHRONIZED clause.
133	ERROR MESSAGE	! CLAUSE MAY NOT BE SPECIFIED IN 88 LEVEL ENTRIES.
	CAUSE	A BLANK WHEN ZERO, JUSTIFIED, OCCURS, PICTURE, SYNCHRONIZED, USAGE, or SIGN clause has been specified for a condition-name.
140	ERROR MESSAGE	MULTIPLE SIGN DESIGNATORS IN PICTURE.
	CAUSE	"+", "-", "S", "CR", and "DB" are mutually exclusive in a picture string.
141	ERROR MESSAGE	C NOT FOLLOWED BY R IN PICTURE.
	CAUSE	"C" must be immediately followed by a "R" in a picture string.
142	ERROR MESSAGE	D NOT FOLLOWED BY B IN PICTURE.
	CAUSE	"D" must be immediately followed by a "B" in a picture string.
143	ERROR MESSAGE	MULTIPLE POINT CHARACTERS IN PICTURE.
	CAUSE	The decimal point location may be specified at most once in a picture string.

144	ERROR MESSAGE	MULTIPLE FLOAT CHARACTERS IN PICTURE.
	CAUSE	Either "*" or "Z" has been encountered in a picture string and some other character has already been determined to be the floating insertion character ("*", "Z", "+" used as floating insertion character, "-" used as floating insertion character, and currency sign used as floating insertion character, are mutually in a picture string).
145	ERROR MESSAGE	MULTIPLE REPETITION FACTORS IN PICTURE.
	CAUSE	"(" immediately follows ")" in a picture string.
146	ERROR MESSAGE	MISSING ")" IN PICTURE REPETITION FACTOR.
	CAUSE	"(" has occurred in a picture string without a following ")".
147	ERROR MESSAGE	ILLEGAL CHARACTER IN PICTURE REPETITION FACTOR.
	CAUSE	Only numeric digits ("0" - "9") may occur between "(" and ")" in a picture string.
148	ERROR MESSAGE	REPETITION FACTOR WITH NO CHARACTER TO REPEAT IN PICTURE.
	CAUSE	A picture string may not begin with the character "(".
149	ERROR MESSAGE	REPETITION FACTOR IN PICTURE MAY NOT BE ZERO.
	CAUSE	The number between a "(" and a ")" in a picture string may not be zero.
150	ERROR MESSAGE	ILLEGAL REPETITION FACTOR IN PICTURE.
	CAUSE	A repetition factor greater than one follows a character that may not occur more than once in a picture string. Or the repetition factor is too big, that is, more than 1G bytes on HP COBOL II/XL or 64K bytes on HP COBOL II/V.
151	ERROR MESSAGE	ILLEGAL CHARACTER IN PICTURE.
	CAUSE	Only the characters "B", "0", "/", ",", ".", "+", "-", "C", "R", (following a "C"), "D", "B" (following a "D"), the currency sign character, "Z", "*", "9", "A", "X", "S", "V", "P", "(", numeric digits following a "(", and ")" following the numeric digits following a "(" may occur in a picture string.
152	ERROR MESSAGE	MULTIPLE NON-FLOATING CURRENCY SIGNS IN PICTURE.
	CAUSE	More than one currency sign character has occurred in a picture string and some other character has been determined to be the float character.

153	ERROR MESSAGE	NO DIGIT OR CHARACTER POSITIONS IN PICTURE.
	CAUSE	At least one of the characters "A", "X", "Z", "9", or "*" or at least two of the characters "+", "-", or the currency symbol must occur in a picture string.
154	ERROR MESSAGE	ILLEGAL COMBINATION OF PICTURE CHARACTERS.
	CAUSE	Certain combination of characters may not occur in a picture string.
155	ERROR MESSAGE	ILLEGAL SEQUENCE OF PICTURE CHARACTERS.
	CAUSE	Certain sequences of characters may not occur in a picture string.
156	ERROR MESSAGE	66-LEVEL ENTRY HAS NO RENAMES CLAUSE.
	CAUSE	66-level data items must have a renames clause.
157	ERROR MESSAGE	88-LEVEL ENTRY HAS NO VALUE CLAUSE.
	CAUSE	A value clause must be specified for every condition-name.
158	ERROR MESSAGE	ELEMENTARY ITEM HAS NO PICTURE.
	CAUSE	A 77-level or 49-level item is not usage index and has no picture clause or an empty group description exists.
159	ERROR MESSAGE	IMPROPER LEVEL NUMBER.

	CAUSE	The cause is one of the following:
		1. A level number is not 66, 77, 88, or between 01 and 49.
		2. A level number is immediately subordinate to a level whose subordinates are not equal to it. For example, in the following, the level 04 is improper:
		`03...`
		`05...`
		`....`
		`05...`
		`....`
		`04`
		3. A level 77 is in the file section.
		4. A level 88 is subordinate to an index item.
		5. The first level number in a section is not 01 or 77.
		6. The first level number subordinate to an FD or SD is not 01.
		7. A level 88 is subordinate to a 66 level item.
		8. A level 66 is subordinate to a 77 level item.

160	ERROR MESSAGE	ILLEGAL CLAUSE FOR 66-LEVEL ENTRY.
	CAUSE	The only clause which may be specified for a 66-level entry is the RENAMES clause.
161	ERROR MESSAGE	ILLEGAL REDEFINES CLAUSE.
	CAUSE	The item being redefined is not declared immediately subordinate to the item containing the redefines clause, or the item being redefined is a table or variable size item or, the item being redefined is a 66-level or 88-level item.
162	ERROR MESSAGE	PICTURE CLAUSE IS ILLEGAL IN 66 AND 88 LEVEL ENTRIES.
	CAUSE	A picture clause has occurred in a 66-level or 88-level entry.
163	ERROR MESSAGE	USAGE CLAUSE CONFLICTS WITH GROUP USAGE CLAUSE.
	CAUSE	The usage clause of an item must specify the same usage as any usage in any group containing it.
164	ERROR MESSAGE	SIGN CLAUSE CONFLICTS WITH GROUP SIGN CLAUSE.
	CAUSE	The sign clause may not be specified for any item whose group contains a sign clause. ANSI74 entry point only.
165	ERROR MESSAGE	SIGN CLAUSE CONFLICTS WITH USAGE.
	CAUSE	The sign clause may only be specified for items whose usage is display.
166	ERROR MESSAGE	! CLAUSE IS ILLEGAL IN INDEX ITEMS.
	CAUSE	The JUSTIFIED, PICTURE, VALUE or BLANK WHEN ZERO clauses may not be specified for items whose usage is index.
167	ERROR MESSAGE	REDEFINING ITEM ! DOES NOT IMMEDIATELY FOLLOW REDEFINED ITEM.
	CAUSE	Between an item containing a redefined clause and the item it redefines there must be any entries which define new character positions.
168	ERROR MESSAGE	! CLAUSE IS ILLEGAL IN POINTER ITEMS
	CAUSE	The JUSTIFIED, PICTURE, SIGN, or BLANK WHEN ZERO clauses may not be specified for items whose usage is pointer.

170	ERROR MESSAGE	ILLEGAL RENAMES CLAUSE.
	CAUSE	A RENAMES clause may only be specified in a 66-level item. The item(s) it renames must be defined in the immediately preceding record and must be special level (66,77,88) items, table items, table elements, or variable size items. If the THRU phrase is used, the names must specify different items, the beginning of the second item may not be before the beginning of the first item, and the end of the second item be after the end of the first item.
172	ERROR MESSAGE	JUSTIFIED CLAUSE IS ILLEGAL IN DATA ITEMS WHICH ARE NOT EITHER ALPHABETIC OR ALPHANUMERIC.
	CAUSE	The justified clause may only be specified for alphabetic and alphanumeric items.
173	ERROR MESSAGE	BLANK WHEN ZERO CLAUSE IS ILLEGAL FOR THIS ITEM.
	CAUSE	The blank when zero clause is legal only for items whose picture is numeric or numeric edited and does not contain a "*".
174	ERROR MESSAGE	BLANK WHEN ZERO CLAUSE IS REDUNDANT FOR THIS ITEM.
	CAUSE	If all of the numeric character positions of a numeric edited item are represented by "Z" then the item is implicitly BLANK WHEN ZERO.
175	ERROR MESSAGE	EDIT PROGRAM FOR THIS PICTURE IS TOO BIG.
	CAUSE	If a picture is excessively complicated, it can generate an edit program which will be too long to fit into a data table entry.
176	ERROR MESSAGE	OCCURS CLAUSE IS ILLEGAL IN 77-LEVEL ITEMS.
	CAUSE	The occurs clause may not be specified in 77 level items.
177	ERROR MESSAGE	ILLEGAL PICTURE FOR NON-DISPLAY USAGE.
	CAUSE	If the usage of an item is comp, binary, packed-decimal, or comp-3 then the picture must be numeric.
178	ERROR MESSAGE	! CLAUSE IS ILLEGAL IN GROUP ITEMS.
	CAUSE	The BLANK WHEN ZERO, JUSTIFIED or SYNCHRONIZED clauses may only be specified for elementary items.
183	ERROR MESSAGE	ILLEGAL SIGN IN LITERAL.
	CAUSE	A numeric literal in a value clause may not contain a sign if the corresponding data item is an unsigned numeric data item.

184	ERROR MESSAGE	ILLEGAL LITERAL.
	CAUSE	A literal has occurred in a value clause and either the corresponding data item is an index item, the corresponding data item is numeric and the literal is nonnumeric, or the corresponding data item is nonnumeric and the literal is numeric.
185	ERROR MESSAGE	MULTIPLE INITIAL VALUES FOR A DATA ITEM.
	CAUSE	When the value clause is used to specify an initial value for a data item it may only specify one value. VALUE clause must not be specified on a group and also a subordinate item.
186	ERROR MESSAGE	!! FOR !.
	CAUSE	This error message is for illegal or missing forward references. The insertions are for the forward reference type, the name of the forward reference, and the name of the file or table containing the forward reference. The forward reference types are table keys, alternate keys, depending on identifiers, file status identifiers, volume identifiers, labels identifiers, seq identifiers, exdate identifiers, linage identifiers, footing identifiers, top identifiers, bottom identifiers, padding character identifiers and record varying identifiers. Illegal forward references could occur because the key or identifier is in the wrong section or has wrong usage or size. This error could also occur because it has the wrong scope, that is, external files must have external linage or padding characters, depending on identifiers must have global or external the same as the table and be in the same DATA DIVISION.
188	ERROR MESSAGE	LITERAL REQUIRES TRUNCATION OF NON-ZERO DIGITS.
	CAUSE	A numeric literal in a value clause specified a value outside the range of values possible for the associated data item.
189	ERROR MESSAGE	LITERAL TOO LONG, TRUNCATED.
	CAUSE	A nonnumeric literal in a value clause is longer than the associated data item.
200	ERROR MESSAGE	MULTIPLE OCCURRENCES OF FD-SD ENTRIES FOR FILENAME.
	CAUSE	The same name has been used in more than one FD or SD entry.
201	ERROR MESSAGE	AREA A MUST BE BLANK IN A CONTINUATION RECORD.
	CAUSE	

202	ERROR MESSAGE	ILLEGAL COBOL CHARACTER IGNORED.
	CAUSE	Check list of legal COBOL characters.
205	ERROR MESSAGE	RESERVED WORD ! NOT LEGAL IN THIS DIVISION.
	CAUSE	The specified word is a Reserved word used in another division.
206	ERROR MESSAGE	PICTURE CHARACTER STRING TOO LONG.
	CAUSE	Picture character string is limited to 30 characters.
207	ERROR MESSAGE	ILLEGAL OCTAL DIGIT.
	CAUSE	
208	ERROR MESSAGE	LITERAL TOO LONG.
	CAUSE	Nonnumeric or octal literal must not be longer than data item.
210	ERROR MESSAGE	ILLEGAL DUPLICATION OF CLAUSES IN ! PARAGRAPH.
	CAUSE	In SPECIAL-NAMES OR OBJECT-COMPUTER paragraph.
211	ERROR MESSAGE	ALPHABET-NAME ! HAS ALREADY BEEN USED.
	CAUSE	In program COLLATING SEQUENCE clause.
212	ERROR MESSAGE	ILLEGAL IMPLEMENTOR-NAME !.
	CAUSE	An unknown name was used in mnemonic or alphabet clause. Implementor-names are switch names, function names or alphabet names. Valid function names are: SYSIN, SYSOUT, CONSOLE, C01-C16, TOP, or NO SPACE CONTROL. Valid alphabet names are: EBCDIC, EBCDIK, NATIVE, STANDARD-1 or STANDARD-2. Valid switch names are SW0 - SW15.
213	ERROR MESSAGE	MNEMONIC-NAME ! HAS ALREADY BEEN USED.
	CAUSE	
214	ERROR MESSAGE	ON-OFF CONDITION DOES NOT REFER TO A SWITCH.
	CAUSE	
216	ERROR MESSAGE	NONNUMERIC LITERAL IN "!" PHRASE HAS MORE THAN ONE CHARACTER.
	CAUSE	The THRU or ALSO phrase in alphabet-name.

218	ERROR MESSAGE	CURRENCY SIGN HAS MORE THAN ONE CHARACTER.
	CAUSE	
219	ERROR MESSAGE	ILLEGAL SUBSTITUTE CURRENCY SIGN.
	CAUSE	
220	ERROR MESSAGE	ILLEGAL COMBINATION OF FILE ORGANIZATION AND ACCESS METHODS.
	CAUSE	
221	ERROR MESSAGE	MISSING FILE POSITION NUMBER(S).
	CAUSE	Missing file position numbers in MULTIPLE FILE clause.
222	ERROR MESSAGE	DUPLICATE FILE NAME ! IN MULTIPLE FILE TAPE CLAUSE.
	CAUSE	
223	ERROR MESSAGE	DUPLICATE FILE POSITION IN MULTIPLE FILE TAPE CLAUSE.
	CAUSE	
224	ERROR MESSAGE	FILE NAME ! NOT DEFINED IN SELECT CLAUSE.
	CAUSE	
225	ERROR MESSAGE	FILE POSITION HAS MORE THAN 4 DIGITS.
	CAUSE	
226	ERROR MESSAGE	FILE POSITION NUMBERS MUST START WITH 1.
	CAUSE	
227	ERROR MESSAGE	FILE NAME ! IN MORE THAN ONE SAME RECORD AREA.
	CAUSE	
228	ERROR MESSAGE	FILE NAME ! IN MORE THAN ONE SAME SORT/MERGE AREA.
	CAUSE	
229	ERROR MESSAGE	FILE NAME ! IN MORE THAN ONE SAME AREA.
	CAUSE	

230	ERROR MESSAGE	FILE NAME ! HAS ALREADY BEEN USED.
	CAUSE	
232	ERROR MESSAGE	DUPLICATE CHARACTER IN ALPHABET OR CLASS DEFINITION.
	CAUSE	
233	ERROR MESSAGE	NUMERIC LITERAL IN ALPHABET/CLASS/SYMBOLIC CHARACTER HAS SIGN ,SIGN DROPPED.
	CAUSE	
234	ERROR MESSAGE	NUMERIC LITERAL IN ALPHABET/CLASS/SYMBOLIC CHARACTER MUST BE 1 THRU !.
	CAUSE	For alphabets and classes the value is 256. For symbolic character the value depends on the alphabet.

Note The term SYSTEM FILE NAME in ERROR MESSAGES 238 through 252 is the same as the term FILE-INFO in the manual.

238	ERROR MESSAGE	MISSING COMMA IN SYSTEM FILE NAME.
	CAUSE	See FILE-INFO for the ASSIGN clause.
239	ERROR MESSAGE	ILLEGAL FORMAL FILE DESIGNATOR IN SYSTEM FILE NAME.
	CAUSE	See FILE-INFO for the ASSIGN clause.
240	ERROR MESSAGE	FIRST CHARACTER OF FORMAL FILE DESIGNATOR MUST BE '$' OR ALPHABETIC.
	CAUSE	See MPE restrictions for file name.
241	ERROR MESSAGE	FORMAL FILE DESIGNATOR ! HAS MORE THAN 8 CHARACTER !.
	CAUSE	See MPE restrictions for file name.
242	ERROR MESSAGE	ILLEGAL CHARACTER IN FORMAL FILE DESIGNATOR !.
	CAUSE	See MPE restrictions for file name.
243	ERROR MESSAGE	ILLEGAL DEVICE CLASS !.
	CAUSE	

244	ERROR MESSAGE	ILLEGAL RECORDING MODE !.
	CAUSE	
245	ERROR MESSAGE	ILLEGAL DEVICE NAME !.
	CAUSE	
246	ERROR MESSAGE	DEVICE CODE MUST CONTAIN 3 DIGITS.
	CAUSE	
247	ERROR MESSAGE	FILE SIZE HAS MORE THAN 9 DIGITS.
	CAUSE	
249	ERROR MESSAGE	FORMS MESSAGE HAS MORE THAN 49 CHARACTERS.
	CAUSE	
250	ERROR MESSAGE	FORMS MESSAGE MUST END WITH A PERIOD.
	CAUSE	
251	ERROR MESSAGE	ILLEGAL LOCKING PARAMETER !.
	CAUSE	'L' is locking parameter.
252	ERROR MESSAGE	SYSTEM FILE NAME CONTAINS TOO MANY FIELDS.
	CAUSE	See FILE-INFO for the ASSIGN clause.
253	ERROR MESSAGE	ILLEGAL COMBINATION OF SAME AREA AND SAME RECORD AREA FILES.
	CAUSE	Inconsistent combination of these clauses.
254	ERROR MESSAGE	ILLEGAL COMBINATION OF SAME AREA AND SAME SORT AREA FILES.
	CAUSE	Inconsistent combination of these clauses.
258	ERROR MESSAGE	! NOT IMPLEMENTED.
	CAUSE	MULTIPLE REEL/UNIT or REPORT WRITER, COMMUNICATION, or level 2 DEBUG modules or the ENTER statement.
259	ERROR MESSAGE	DEFAULT FILE NAME IS TEMPORARY NAMELESS FILE.
	CAUSE	

260	ERROR MESSAGE	FILE NAME ! IS NOT DEFINED IN THIS SCOPE.
	CAUSE	A file access is being attempted in a scope in which the file is not defined. The 'FD' may require the 'GLOBAL' clause.
262	ERROR MESSAGE	KEY WORD "!" IGNORED.
	CAUSE	For CLOSE REEL/UNIT the close will not be performed. Otherwise the keyword is ignored (END & BEFORE).
266	ERROR MESSAGE	NAME OF "SD" ENTRY FOUND, EXPECTED "FD" ENTRY.
	CAUSE	I/O statements should reference an "FD" name.
267	ERROR MESSAGE	NAME OF "FD" ENTRY FOUND, EXPECTED "SD" ENTRY.
	CAUSE	SORT and MERGE statements should reference an "SD" name.
268	ERROR MESSAGE	LINAGE MUST BE SPECIFIED TO USE "WRITE AT END-OF-PAGE".
	CAUSE	A "WRITE AT END-OF-PAGE" was found for a file without a LINAGE clause specified.
269	ERROR MESSAGE	LINAGE MUST BE SPECIFIED TO USE "LINAGE-COUNTER."
	CAUSE	"LINAGE-COUNTER" was reference for a file which does not have a LINAGE clause.
272	ERROR MESSAGE	INTRINSIC RETURN VALUE MISMATCH.
	CAUSE	The GIVING operand does not match intrinsic definition.
273	ERROR MESSAGE	INCOMPATIBLE NUMBER OF PARAMETERS FOR INTRINSIC !.
	CAUSE	The number of parameters in the USING phrase of the CALL statement is incompatible with the parameter count specified for the intrinsic in the SYSINTR file. The intrinsic does not have OPTION EXTENSIBLE/VARIABLE.
274	ERROR MESSAGE	INVALID SCOPE FOR THIS $ COMMAND.
	CAUSE	This command is in effect for all programs in the file and, therefore, must appear within or before the IDENTIFICATION DIVISION of the first program in the file. This applies to the $CONTROL CODE, NOCODE, VERBS, NOVERBS, SYMDEBUG, and OPTIMIZE options.
281	ERROR MESSAGE	INVALID MACRO NAME, EXPECTED ! AS FIRST CHARACTER.
	CAUSE	

282	ERROR MESSAGE	INVALID SUBSYSTEM COMMAND PARAMETER.
	CAUSE	Invalid or missing parameter.
283	ERROR MESSAGE	INVALID SUBSYSTEM COMMAND.
	CAUSE	
284	ERROR MESSAGE	TOO MANY PARAMETERS IN MACRO CALL.
	CAUSE	Check macro definition.
285	ERROR MESSAGE	INVALID FILENAME IN $INCLUDE COMMAND.
	CAUSE	
286	ERROR MESSAGE	INVALID CHARACTER IN $PREPROCESSOR COMMAND.
	CAUSE	
287	ERROR MESSAGE	PSEUDO-TEXT CAN'T BE NULL, BLANK, COMMA, OR SEMICOLON.
	CAUSE	Pseudo-text-1 ignored in COPY REPLACING or REPLACE statement.
288	ERROR MESSAGE	INVALID IDENTIFIER USED AS QUALIFIER.
	CAUSE	Qualifier in COPY REPLACING or REPLACE identifier.
289	ERROR MESSAGE	INVALID TOKEN IN COPY ... REPLACING OR REPLACE CLAUSE.
	CAUSE	
290	ERROR MESSAGE	ILLEGAL TERMINATION OF NONNUMERIC LITERAL !.
	CAUSE	
291	ERROR MESSAGE	UNABLE TO FIND LIBRARY MODULE !.
	CAUSE	The specified library module was not found in the COPYLIB file. The specified name is not valid.
306	ERROR MESSAGE	VOL MUST BE A NONNUMERIC LITERAL <= 6 CHARACTERS LONG.
	CAUSE	When the VOL phrase of the value of clause specifies a literal, the literal must be nonnumeric and no more than six characters long.

307	ERROR MESSAGE	LABELS MUST BE NONNUMERIC LITERAL = "IBM" OR "ANS".
	CAUSE	When the LABELS phrase of the value of clause specifies a literal, the literal must be "IBM" or "ANS".
308	ERROR MESSAGE	SEQ MUST BE AN UNSIGNED NUMERIC LITERAL.
	CAUSE	When the SEQ phrase of the value of clause specifies a literal, the literal must be numeric and unsigned.
309	ERROR MESSAGE	EXDATE MUST BE NONNUMERIC LITERAL OF THE FORM "MM/DD/YY".
	CAUSE	When the EXDATE phrase of the value of clause specifies a literal, the literal must be nonnumeric and of the form "MM/DD/YY".
310	ERROR MESSAGE	LINAGE MUST BE AT LEAST 1.
	CAUSE	When the lines phrase of the LINAGE clause specifies an integer it must be >=1.
311	ERROR MESSAGE	FOOTING MAY NOT BE LESS THAN 1.
	CAUSE	When the footing phrase of the LINAGE clause specifies an integer, it is necessary that 1<= footing integer <= LINAGE integer (if specified).
312	ERROR MESSAGE	FOOTING MAY NOT BE GREATER THAN LINAGE.
	CAUSE	When the footing phrase of the LINAGE clause specifies an integer, it is necessary that 1 <= footing integer <= LINAGE integer (if specified).
314	ERROR MESSAGE	! MAY NOT BE LESS THAN 0.
	CAUSE	When the TOP or BOTTOM phrase of the LINAGE clause specifies an integer it must be >=0.
318	ERROR MESSAGE	FILE RECORD ! SMALLER THAN MINIMUM SIZE IN RECORD CONTAINS CLAUSE.
	CAUSE	When the record contains clause is specified in an FD or SD every record subordinate to that file description must have a size that is within the specified bounds.
319	ERROR MESSAGE	FILE RECORD ! LARGER THAN MAXIMUM SIZE IN RECORD CONTAINS CLAUSE.
	CAUSE	When the record contains clause is specified in an FD or SD every record subordinate to that file description must have a size that is within the specified bounds.

320	ERROR MESSAGE	FILE ! RECORD SIZE IS ZERO.
	CAUSE	When the largest record for a file is 0 characters and there is a block contains clause in the file description which contains the characters phrase, it is impossible to compute the blocking factor.
321	ERROR MESSAGE	INTEGER-1 MUST BE <= INTEGER-2 IN 'TO' PHRASE.
	CAUSE	
322	ERROR MESSAGE	INVALID REFERENCE MODIFICATION.
	CAUSE	Either the starting position or length is incorrect or the item is not USAGE DISPLAY.
330	ERROR MESSAGE	ILLEGAL INTRINSIC FUNCTION NAME !.
	CAUSE	The specified name is not valid.
331	ERROR MESSAGE	WRONG NUMBER OF ARGUMENTS FOR FUNCTION !.
	CAUSE	An incorrect number of arguments was specified or the function doesn't take ALL for subscripts.
337	ERROR MESSAGE	NEXT SENTENCE IS ILLEGAL WITH END-IF OR END-SEARCH.
	CAUSE	The NEXT SENTENCE phrase must not be used when an explicit scope terminator exists.
338	ERROR MESSAGE	AFTER PHRASE IS ILLEGAL WITH INLINE PERFORM.
	CAUSE	
347	ERROR MESSAGE	PROCEDURE ! HAS MORE THAN ONE DEBUGGING SECTION.
	CAUSE	The specified procedure-name is included in more than one USE FOR DEBUGGING Statement.
348	ERROR MESSAGE	DEBUGGING SECTION AFTER NON-DEBUGGING SECTION.
	CAUSE	A Debugging Section was found after a non-debugging Section. If the ALL PROCEDURES phrase is specified, the sections prior to the Debugging Section would not cause the Debugging Section to be executed.

349	ERROR MESSAGE	REFERENCE TO A DEBUG-ITEM OUTSIDE OF A DEBUG SECTION.
	CAUSE	A reference to DEBUG-ITEM or one of its subordinate items was made outside of a Debugging Section.
350	ERROR MESSAGE	NO CORRESPONDING OR INITIALIZE ITEMS FOUND.
	CAUSE	A CORRESPONDING statement has no CORRESPONDING pairs. An INITIALIZE statement didn't find any items.
351	ERROR MESSAGE	REFERENCE TO ! IS NOT UNIQUE.
	CAUSE	Reference needs additional qualification(s) to make it unique.
352	ERROR MESSAGE	ILLEGAL CORRESPONDING OPERAND.
	CAUSE	The operands are of the wrong type, that is, non-numeric for ADD or SUBTRACT.
354	ERROR MESSAGE	COMPILER ERROR: INVALID SYMBOL/DATA TABLE REFERENCE.
	CAUSE	An invalid symbol or data table reference has occurred internal to compiler. See directions on line .3 of the file COBCAT.PUB.SYS if it is the *only* error or warning.
355	ERROR MESSAGE	SECTION HEADER MUST PRECEDE PARAGRAPH.
	CAUSE	The program has an improper construct, for example, it has a section appearing in a PROCEDURE DIVISION starting with a paragraph.
356	ERROR MESSAGE	UNDEFINED DATA NAME !.
	CAUSE	The referenced data name is undefined, replaced by TALLY.
357	ERROR MESSAGE	DUPLICATE PARAGRAPH OR SECTION NAME: !.
	CAUSE	A duplicate paragraph or section name appears.
358	ERROR MESSAGE	ILLEGAL PARAGRAPH OR SECTION NAME !.
	CAUSE	A paragraph or section name is illegal, for example, a signed numeric literal.
359	ERROR MESSAGE	UNDEFINED OR IMPROPER PROCEDURE NAME: !.
	CAUSE	The referenced procedure name is undefined or improper, for example, data name, signed numeric literal, or paragraph name where section name is required. It could also occur if the paragraph name does not appear in the current section and duplicate names appear in other sections. No code is generated for the statement.

360	ERROR MESSAGE	ILLEGAL GO TO STATEMENT.
	CAUSE	The GO TO statement is incorrect, for example, GO TO. appears, but it is not the first statement in a paragraph, so it can't be ALTERed.
361	ERROR MESSAGE	ILLEGAL ALTER STATEMENT.
	CAUSE	The ALTER statement references a non-existent paragraph name or it does not reference an alterable GO TO.
362	ERROR MESSAGE	ILLEGAL PERFORM TIMES COUNT.
	CAUSE	The count must be an unsigned, positive integer.
363	ERROR MESSAGE	RECURSIVE PERFORM.
	CAUSE	The PERFORM statement is recursive through itself or its parent. Use of recursive PERFORMs, although an ANSI extension in HP COBOLII, is not recommended for upward compatibility to future HP systems.
364	ERROR MESSAGE	TOO MANY PARAMETERS.
	CAUSE	A CALL statement or USING option contains too many parameters. The maximum number of parameters on HP COBOL II/XL is 255. The maximum number of parameters on HP COBOL II/V is 60.
365	ERROR MESSAGE	ILLEGAL RELATIONAL COMPARE.
	CAUSE	For example, comparing CONDITION-CODE against non-zero literal. Also, CLASS test on the wrong category operands, or abbreviated relation condition without subjects.
366	ERROR MESSAGE	COMPOSITE OF OPERANDS TOO BIG.
	CAUSE	The composite of operands for the statement is more than 18 digits.
367	ERROR MESSAGE	ILLEGAL NUMERIC OPERAND !.
	CAUSE	The operand is not an allowed numeric operand. The reference is replaced with a reference to TALLY.
368	ERROR MESSAGE	REFERENCE TO ! BY ! IS NOT UNIQUE.
	CAUSE	The reference to the data item named requires further qualification to be unique. Reference is used as a key or DEPENDING ON variable.

369	ERROR MESSAGE	ILLEGAL FUNCTION NAME !.
	CAUSE	Mnemonic-name is ACCEPT/DISPLAY statements may only be SYSIN, SYSOUT or CONSOLE. The function name is changed to SYSIN/OUT.
370	ERROR MESSAGE	ILLEGAL STATEMENT FORMAT.
	CAUSE	For example, PERFORM-VARYING statement with more than 7 levels. ACCEPT ... ON INPUT ERROR without the FREE option. SEARCH ALL with missing keys or referencing the same key twice.
371	ERROR MESSAGE	INTRINSIC ! NOT FOUND IN INTRINSIC FILE.
	CAUSE	The system intrinsic file (SYSINTR.PUB.SYS) does not contain the intrinsic.
372	ERROR MESSAGE	INVALID RECORD NAME !.
	CAUSE	The record name in this WRITE, REWRITE, or RELEASE statement is not an "01" level item. No code is generated for this statement.
373	ERROR MESSAGE	LINKAGE SECTION WITHOUT $CONTROL SUBPROGRAM/DYNAMIC; SUBPROGRAM ASSUMED.
	CAUSE	A LINKAGE SECTION header was found in a program which does not contain a $CONTROL command with either the SUBPROGRAM or DYNAMIC parameter specified. Compilation continues as if $CONTROL SUBPROGRAM has been specified.
374	ERROR MESSAGE	UNINITIALIZED GO TO IN ! NOT REFERENCED IN AN ALTER STATEMENT.
	CAUSE	A GO TO statement without a procedure name specified was found without a corresponding ALTER statement.
377	ERROR MESSAGE	CALL TO ! HAS A DIFFERENT PARAMETER COUNT THAN A PREVIOUS CALL.
	CAUSE	The number of parameters listed in the USING phrase of a CALL statement is different than a previous call to the same routine.
378	ERROR MESSAGE	NON-UNIQUE INTERNAL (RBM) NAME !.
	CAUSE	Two non-contiguous sections have the same internal name. The internal name is formed by taking the first 12 non-hyphen characters and adding the segment number and a single quote ('). For example, PROCESS-INPUT-RECORD SECTION 01 has the internal name PROCESSINPUT01'.

379	ERROR MESSAGE	FROM/INTO IDENTIFIER ! SHARES SAME AREA AS RECORD-NAME.
	CAUSE	The compiler has found a READ, REWRITE, RELEASE, RETURN, or WRITE statement that refers to the same area as the record-name reserved for the file.
380	ERROR MESSAGE	INVALID USE OF A SEPARATOR CHARACTER.
	CAUSE	
381	ERROR MESSAGE	MISSING PERIOD IN COPY OR REPLACE STATEMENT.
	CAUSE	The statement must be terminated by a period.
382	ERROR MESSAGE	ERROR OR MISSING PROGRAM-ID PARAGRAPH.
	CAUSE	
383	ERROR MESSAGE	INCORRECT TABLE REFERENCE OR IMPROPER SUBSCRIPTING.
	CAUSE	For example, illegal subscript is used, or table reference as simple variable, or index-name does not belong to table-name or too many or too few subscripts, or subscripts beyond 7. This will also occur when ALL is incorrectly used as a subscript.
384	ERROR MESSAGE	CODEGEN INTERNAL ERROR IN !.
	CAUSE	Some type of error occurred in the code generator of the compiler. Errors 5000-7999 are substituted for !. For 5209 through 5211, this is probably due to some type of user error on the COBOBJ file. For 7204 through 7207, the object file, COBOBJ, is too small or there was some kind of error writing to it, such as $NULL. For errors 5213 through 5214, this is probably due to some type of user error on the COBASSM file. For errors 5380 through 5383, 5980 or 5985 through 5990, this is probably some type of user error on the COBOBJ file as a NMRL. For error 6305, and this occurs with $OPTIMIZE, this is a compiler limitation; you must remove $OPTIMIZE. For all other errors, see directions on line .3 of the file `COBCAT.PUB.SYS` if it is the *only* error or warning.
387	ERROR MESSAGE	ILLEGAL MOVE.
	CAUSE	A move statement (or an implied move statement, for example, a write from) has an illegal combination of "from" and "to" operands, or the number of digits to move is greater than 31 on HP COBOL II/XL or 28 on HP COBOL II/V.
388	ERROR MESSAGE	ILLEGAL COMPARE.
	CAUSE	The subject and object of a relational operator are incompatible.

389	ERROR MESSAGE	PARAMETER ! IS NOT ALIGNED PROPERLY.
	CAUSE	A data item being passed by reference is required to be on a word or half-word boundary. Use SYNC or move to 01/77. For CM, it is 16-bit word boundary only.
390	ERROR MESSAGE	COMPILER ERROR: UNIMPLEMENTED CASE ! IN PROCEDURE !.
	CAUSE	The compiler was not prepared for a given situation. See directions on line .3 of the file COBCAT.PUB.SYS if it is the *only* error or warning.
391	ERROR MESSAGE	ENTRY STATEMENT NOT ALLOWED IN MAIN PROGRAM OR IN DECLARATIVES SECTION.
	CAUSE	An ENTRY statement has been found in the Declaratives Section of the program. The compiler ignores the ENTRY statement.
392	ERROR MESSAGE	'DELIMITER IN' OR 'COUNT IN' PHRASE WITHOUT 'DELIMITED BY' PHRASE.
	CAUSE	The DELIMITER IN and COUNT IN phrases of the UNSTRING statement may only be used if a DELIMITED BY phrase is specified.
393	ERROR MESSAGE	DUPLICATE ENTRY POINT NAME !.
	CAUSE	The same name is used in more than one ENTRY statement or the name used in an ENTRY statement is the same as the name in the PROGRAM-ID paragraph. Entry names are formed from the first 30 non-hyphen characters.
394	ERROR MESSAGE	!! EXPECTED FOR PARAMETER #!.
	CAUSE	The parameter passed to the intrinsic cannot be mapped to the type of parameter the intrinsic expects.
395	ERROR MESSAGE	NUMBER OF SUBJECTS ! OBJECTS IN EVALUATE.
	CAUSE	The number of objects on each WHEN must match the number of subjects.
396	ERROR MESSAGE	BEFORE/AFTER CLAUSE SPECIFIED TWICE IN INSPECT.
	CAUSE	

397	ERROR MESSAGE	ILLEGAL OBJECT OPERAND FOR EVALUATE SUBJECT.
	CAUSE	The type of the object operand does not match its corresponding subject. For example, comparing TRUE with an item instead of a condition. Or comparing an item or literal with a condition.
398	ERROR MESSAGE	ILLEGAL OPERAND ! FOR INITIALIZE.
	CAUSE	The operand does not exist or is illegal.
399	ERROR MESSAGE	REPLACING CATEGORY IS SPECIFIED TWICE IN INITIALIZE.
	CAUSE	

Serious Errors

401	ERROR MESSAGE	ASSIGN CLAUSE REQUIRED WITH SELECT STATEMENT.
	CAUSE	
402	ERROR MESSAGE	MISSING ! KEY CLAUSE.
	CAUSE	RELATIVE, RECORD or ACTUAL KEY is missing.
403	ERROR MESSAGE	FUNCTION NOT ALLOWED AS OPERAND.
	CAUSE	Functions are not allowed in the following cases: 1. Target of any statement. 2. Operand of a call. 3. Where specific rules of the statement disallow functions.
404	ERROR MESSAGE	NUMERIC FUNCTION NOT ALLOWED AS OPERAND.
	CAUSE	Numeric functions are only allowed in arithmetic expressions.
405	ERROR MESSAGE	TOO MANY KEY CLAUSES.
	CAUSE	More than one RELATIVE clause, or RECORD clause, or ACTUAL KEY clause appears. Or the file organization does not support keys. That is, the organization is sequential.
406	ERROR MESSAGE	! ! FOR !.
	CAUSE	This error message is for illegal or missing forward references. The insertions are for the forward reference type, the name of the forward reference, and the name of the file or table containing the forward reference. The forward reference types are relative keys, record keys, and actual keys.
409	ERROR MESSAGE	ILLEGAL OPERAND ! IN SET STATEMENT.
	CAUSE	One of the operands must be an INDEX-NAME, MNEMONIC-NAME or CONDITION-NAME. Any identifiers must be numeric integers.

410	ERROR MESSAGE	SYNTAX ERROR. FOUND: !; EXPECTING ONE OF THE FOLLOWING:!!.
	CAUSE	An item appears in some context that the compiler cannot recognize. The items which are allowed at this point are listed. EXPECTING ... Disabled_reserved_word probably means the wrong entry was used to invoke the compiler. The compiler attempts to recover from this error.
412	ERROR MESSAGE	PARAMETER ! MUST BE 01 OR 77 LEVEL ITEM IN LINKAGE SECTION.
	CAUSE	An illegal parameter was used in a PROCEDURE DIVISION USING statement.
413	ERROR MESSAGE	FILE NAME ! DOES NOT APPEAR IN A SELECT CLAUSE AND/OR FD/SD ENTRY IS NOT UNIQUE.
	CAUSE	A name that appears in an FD or SD entry in the DATA DIVISION did not appear in a SELECT clause in the ENVIRONMENT DIVISION or the the name is not unique.
414	ERROR MESSAGE	! IS AN ILLEGAL KEY.
	CAUSE	The data names identified as sort/merge keys must be described within the records associated with the Sort/Merge file. For READ and START statements, the KEY phrase must reference a valid key for the Relative or Indexed file.
415	ERROR MESSAGE	INVALID LIBRARY OR TEXT-NAME.
	CAUSE	
416	ERROR MESSAGE	COPYLIB OR $INCLUDE NESTED TOO DEEP.
	CAUSE	The maximum $INCLUDE/COPY nesting is 10.
417	ERROR MESSAGE	MACRO DEFINITION MAXIMUM LENGTH EXCEEDED.
	CAUSE	Errors 417 and 418 may cause error 461.
418	ERROR MESSAGE	PSEUDO-TEXT-BUFFER OVERFLOW.
	CAUSE	Errors 417 and 418 may cause error 461.
419	ERROR MESSAGE	MOVE OR COMPARE >32 K BYTES.
	CAUSE	A MOVE statement or condition clause exceeds the maximum character limit.

420	ERROR MESSAGE	SECTION MISPLACED OR DUPLICATED.
	CAUSE	FILE SECTION, WORKING-STORAGE SECTION and LINKAGE SECTION must appear in that order when used.
421	ERROR MESSAGE	OPERAND ! HAS ILLEGAL FORMAT FOR STATEMENT.
	CAUSE	■ This type of operand may not be used in the ACCEPT or DISPLAY statements. ■ Bad identifier-1 for CALL or identifier for CALL INTRINSIC. ■ EXAMINE identifier not usage DISPLAY or literals not one character. ■ GO TO ... DEPENDING identifier not an integer. ■ INSPECT operands not usage DISPLAY or literals. (All but tallying identifier-2, which must be numeric.) ■ INSPECT identifier-1 not usage DISPLAY or a group. ■ CHARACTERS operands must be one character. ■ Replacing operands must be of the same size. ■ Invalid identifier-1 or identifier-2 for SEARCH. SEARCH ALL must have an INDEXED or KEY clause. ■ SEARCH ALL must have a key referenced in each condition. ■ STRING and UNSTRING POINTER must be numeric and large enough. ■ STRING INTO operands must be alphanumeric without JUST. ■ STRING source and delimiter operands must be usage DISPLAY or nonnumeric literals without ALL. ■ UNSTRING identifier-1 must be usage DISPLAY. ■ UNSTRING DELIMITED BE operands must be usage DISPLAY or nonnumeric literals without ALL. ■ UNSTRING identifier-4 must be usage DISPLAY and not edited. ■ UNSTRING identifier-5 must be usage DISPLAY and not edited. ■ POINTER, COUNT, TALLYING operands must be integers with no P. ■ Parameters to a FUNCTION must be of the correct type. That is, numeric or integer passed to alphanumeric or numeric passed to integer.
422	ERROR MESSAGE	SIZE OF DATA SEGMENT GREATER THAN {1G OR 64K} BYTES.
	CAUSE	The upper limit on the size of the data area has been reached. The maximum is 1G bytes on HP COBOL II/XL and 64K bytes on HP COBOL II/V.

423	ERROR MESSAGE	MORE THAN 500 ENTRIES IN A GO TO DEPENDING ON STATEMENT.
	CAUSE	A GO TO DEPENDING ON statement has too many paragraph/section names in the list.
424	ERROR MESSAGE	MORE THAN {255/57} NON-REDEFINED 01/77 LEVEL ITEMS IN LINKAGE SECTION.
	CAUSE	The limit of independent 01/77 level items in the LINKAGE SECTION has been exceeded. The maximum is 255 on HP COBOL II/XL and 57 on HP COBOL II/V. Use REDEFINES if possible.
425	ERROR MESSAGE	PARAMETER #! MAY NOT BE A LITERAL.
	CAUSE	A literal is attempted to be passed to an intrinsic which expects an array or a pointer.
426	ERROR MESSAGE	ILLEGAL END PROGRAM HEADER.
	CAUSE	Check that the program-name here matches the one in the IDENTIFICATION DIVISION.
427	ERROR MESSAGE	PROGRAM-ID ! NOT UNIQUE.
	CAUSE	Program names in a compilation unit must be unique.
428	ERROR MESSAGE	MISSING END PROGRAM HEADER FOR PROGRAM !.
	CAUSE	Batched or nested programs require END PROGRAM headers.

Disastrous Errors

450	ERROR MESSAGE	USL FILE OVERFLOW.
	CAUSE	The USL file may overflow under the following conditions: A $CONTROL USLINIT command may be missing, the default size of 1023 records may be too small, or there may not be enough records left in the usl file.
451	ERROR MESSAGE	PARSE STACK OVERFLOW; POSSIBLE LIMIT EXCEEDED.
	CAUSE	The parse stack in compiler overflowed. This may have been caused by too many levels of nesting of IF ... THEN ... ELSE statements. This could also be caused by error 410. This could occur instead of error 423.
452	ERROR MESSAGE	EARLY END OF FILE ON COBOL SOURCE.
	CAUSE	This could occur with the following errors: 1. The syntax of a COPY or REPLACE statement is incorrect. 2. A syntax error in the program. For example, a period may be missing making the IDENTIFICATION DIVISION paragraph incorrect.
453	ERROR MESSAGE	BAD INTRINSIC FILE.
	CAUSE	The system intrinsic file (SYSINTR.PUB.SYS) is not in the proper format.
454	ERROR MESSAGE	READ ERROR ON IDS FILE.
	CAUSE	An error has occurred while trying to do a read on the IDS file (an internal temporary file). The most likely cause is a serious compiler problem. See directions on line .4 of the file COBCAT.PUB.SYS.
455	ERROR MESSAGE	WRITE ERROR ON IDS FILE.
	CAUSE	An error has occurred while trying to do a write to the IDS file (an internal temporary file). See line .4 of the file COBCAT.PUB.SYS, if compiler problem. This can be caused by a serious compiler problem, an excessively large source file, or a lack of disk space.
456	ERROR MESSAGE	OPEN ERROR ON IDS FILE.
	CAUSE	An error has occurred while trying to open the IDS file (an internal temporary file). The most likely cause of this is lack of disk space.
457	ERROR MESSAGE	COMPILER ERROR: OUT OF IDS FILE BUFFERS.
	CAUSE	This can be caused by a compiler problem or by statements too complex or with too many operands.

458	ERROR MESSAGE	COMPILER ERROR: INVALID INTERNAL LABEL.
	CAUSE	The compiler has generated or referenced an invalid internal label number. This could also be caused by too many VALUE clauses on table elements. If the error is not in the DATA DIVISION or it is the only error, then see directions on line .3 of the file COBCAT.PUB.SYS.
459	ERROR MESSAGE	TOO MANY VALUE CLAUSES.
	CAUSE	USL entry overflows maximum size. Reduce the number of VALUE clauses, for example, by combining at group level.
460	ERROR MESSAGE	MISSING IDENTIFICATION DIVISION, COMPILATION TERMINATED.
	CAUSE	
461	ERROR MESSAGE	DYNAMIC ARRAY ERROR, OUT OF SPACE.
	CAUSE	The number of macros, size of one macro or copylib member is too big. May be caused instead of 417 and 418.
462	ERROR MESSAGE	AVAILABLE MEMORY INSUFFICIENT FOR COMPILATION.
	CAUSE	Refer to error 471 for possible problem cause. If not applicable, see directions on line .3 of the file COBCAT.PUB.SYS.
463	ERROR MESSAGE	READ ERROR ON SYMBOL TABLE FILE.
	CAUSE	An error has occurred while trying to read from the symbol table file (an internal temporary file). See directions on line .4 of the file COBCAT.PUB.SYS. The most likely cause of this is a serious compiler error.
464	ERROR MESSAGE	READ ERROR ON DATA TABLE FILE.
	CAUSE	An error occurred while trying to read from the data table file (an internal temporary file). See directions on line .4 of the file COBCAT.PUB.SYS. The most likely cause of this is a serious compiler error.
465	ERROR MESSAGE	WRITE ERROR ON SYMBOL TABLE FILE.
	CAUSE	An error has occurred while trying to write to the symbol table file (an internal temporary file). See directions on line .4 of the file COBCAT.PUB.SYS. This can be caused by a compiler error or by a lack of disk space.
466	ERROR MESSAGE	WRITE ERROR ON DATA TABLE FILE.
	CAUSE	An error has occurred while trying to write to the data table file (an internal temporary file). See directions on line .4 of the file COBCAT.PUB.SYS. This can be caused by a compiler error or by a lack of disk space.

467	ERROR MESSAGE	OPEN ERROR ON SYMBOL TABLE FILE.
	CAUSE	An error has occurred while trying to open the symbol table file (an internal temporary file). See directions on line .4 of the file COBCAT.PUB.SYS. The most likely cause of this is a lack of disk space.
468	ERROR MESSAGE	OPEN ERROR ON DATA TABLE FILE.
	CAUSE	An error has occurred while trying to open the data table file (an internal temporary file). See directions on line .4 of the file COBCAT.PUB.SYS. The most likely cause of this is a lack of disk space.
470	ERROR MESSAGE	USL FILE (DIRECTORY) OVERFLOW.
	CAUSE	The directory area of the USL file does not have enough space for the current entry.
471	ERROR MESSAGE	CODE SEGMENT EXCEEDS 16 K.
	CAUSE	A compiled code segment is too large. Use COBOL SECTION entries to break up the code segments. (If not in initialization section, that is, for VALUE clauses.)
491	ERROR MESSAGE	UNABLE TO OPEN FILE !.
	CAUSE	
492	ERROR MESSAGE	UNABLE TO USE FILE !.
	CAUSE	
493	ERROR MESSAGE	READ FAILURE ON FILE !.
	CAUSE	
494	ERROR MESSAGE	WRITE FAILURE ON FILE !.
	CAUSE	This could occur on writes to COBXREF. The disc file is not big enough to contain all of its data. The filesize can be increased with the file equation :FILE COBXREF;DISC=nnnnn
495	ERROR MESSAGE	UNABLE TO CLOSE FILE !.
	CAUSE	

496	ERROR MESSAGE	EARLY END OF FILE ON FILE !.
	CAUSE	This could occur with the following errors: ■ Syntax of COPY or REPLACE statement is bad. ■ Syntax in program, such as a period is missing. ■ IDENTIFICATION DIVISION paragraph is incorrect. This could occur on writes to COBLIST, COBTEMP, COBMAC, or COBXDB. The disc file is not big enough to contain all of its data. The filesize can be increased with a file equation :FILE COBTEMP;DISC=nnnnn
498	ERROR MESSAGE	UNABLE TO SAVE FILE !.
	CAUSE	

Nonstandard Warnings

500	ERROR MESSAGE	INTERNAL SORT/MERGE ERROR ! (COBERR 500)
	CAUSE	This is a run-time error. See the section "Run-Time Errors" later in this appendix for more information.
503	ERROR MESSAGE	NONSTANDARD ORDERING OF CLAUSES OBJECT-COMPUTER PARAGRAPH.
	CAUSE	
505	ERROR MESSAGE	PROCESSING MODE CLAUSE IS NONSTANDARD.
	CAUSE	
509	ERROR MESSAGE	STATEMENT MUST BEGIN IN AREA A IN STANDARD COBOL.
	CAUSE	
510	ERROR MESSAGE	STATEMENT MUST NOT BEGIN IN AREA A IN STANDARD COBOL.
	CAUSE	
512	ERROR MESSAGE	FILE LIMITS CLAUSE IS NONSTANDARD.
	CAUSE	
514	ERROR MESSAGE	A SPACE IS REQUIRED AFTER COMMA (,) OR SEMICOLON (;).
	CAUSE	
515	ERROR MESSAGE	! IS IN THE INTERMEDIATE FIPS LEVEL.
	CAUSE	
516	ERROR MESSAGE	! IS IN THE HIGH FIPS LEVEL.
	CAUSE	
517	ERROR MESSAGE	! IS A HEWLETT-PACKARD COBOL II EXTENSION.
	CAUSE	
519	ERROR MESSAGE	! (OPTIONAL MODULE).
	CAUSE	Occurs for DEBUG and SEGMENTATION FIPS flagging.

520	ERROR MESSAGE	ITEM REDEFINES AN ITEM CONTAINING A REDEFINES CLAUSE.
	CAUSE	An item may not redefine an item that contains a redefines clause.
		See also error 520 ITEM REDEFINES AN ITEM CONTAINING A REDEFINES CLAUSE in the section "Run-Time Errors."
521	ERROR MESSAGE	REDEFINE OF FILE RECORD IGNORED.
	CAUSE	The redefines clause is illegal at the 01-level in the file section.
522	ERROR MESSAGE	!! FOR !.
	CAUSE	This error message is for non-standard forward references. The insertions are for the forward reference type, the name of the reference, and the name of the file or table containing the forward reference. The forward reference types are signed relative keys and depending on identifiers, non-alphanumeric record keys, and non-alphanumeric alternate record keys. These are incompatibility warnings.
526	ERROR MESSAGE	! IS AN OBSOLETE LANGUAGE ELEMENT FOR COBOL85.
	CAUSE	
527	ERROR MESSAGE	USE OF ! FEATURE MAY BE INCOMPATIBLE IN COBOL85.
	CAUSE	

Run-Time Errors

Errors that can occur while your program is executing are listed here. Some error messages include a number in square brackets. These numbers indicate the file status codes.

Unless otherwise indicated, the following action is taken for a run-time I/O error (messages 520 through 699 are I/O errors).

- The error message is printed.
- If the error is file related (most are) the file system error is printed and a PRINTFILEINFO is executed.
- If a USE procedure, an AT END clause, an INVALID KEY clause, or a FILE STATUS word is specified (that is, you have some programmatic way to detect the error), then execution is allowed to continue. In most cases the error message is suppressed. Otherwise a QUIT is issued to terminate the program. For more information see "Input-Output Error Handling Procedures" in Chapter 8.

See also Appendix H for more information on handling run-time errors.

500	ERROR MESSAGE	INTERNAL SORT/MERGE ERROR ! (COBERR 500)
	CAUSE	See previous errors, for the detailed HPSORT error. The substitution is the status returned by HPSORT.
520	ERROR MESSAGE	ATTEMPT TO CLOSE A FILE THAT IS NOT OPEN [42] (COBERR 520)
	CAUSE	An attempt was made to CLOSE a file which is not open. This is usually the result of a program logic error. Remove the redundant CLOSE statement or insert any necessary OPEN statement and recompile the source program. See also error 520 ITEM REDEFINES AN ITEM CONTAINING A REDEFINES CLAUSE in the section "Nonstandard Warnings".
540	ERROR MESSAGE	ATTEMPT TO OPEN A FILE THAT IS OPEN [41] (COBERR 540)
	CAUSE	An attempt was made to OPEN a file which is currently open. This is usually the result of a program logic error. Remove the redundant OPEN statement or insert any necessary CLOSE statement and recompile the source program.
551	ERROR MESSAGE	READ ERROR ON ACCEPT (COBERR 551)
	CAUSE	An attempt to execute a ACCEPT statement resulted in an error.
611	ERROR MESSAGE	WRITE ERROR ON DISPLAY (COBERR 611)
	CAUSE	An attempt to execute a DISPLAY statement resulted in an error. IT could also occur for ACCEPT from CONSOLE.
631	ERROR MESSAGE	USER LABEL SPACE UNALLOCATED OR ATTEMPT TO WRITE BEYOND LABEL LIMIT [9x] (COBERR 631)
	CAUSE	An error occurred while reading/writing a label. Refer to the file system error.

632	ERROR MESSAGE	READ MUST PRECEDE ! [43] (COBERR 632)
	CAUSE	In sequential access mode, a READ must be the previous operation on the file, before REWRITE or DELETE.
633	ERROR MESSAGE	UNABLE TO CLOSE FILE, SEE FILE SYSTEM ERROR [9x] (COBERR 633)
	CAUSE	An error occurred while attempting to CLOSE a file. Refer to the file system error.
636	ERROR MESSAGE	RECORD NOT FOUND [23] ! (COBERR 636)
	CAUSE	An I/O statement was executed which addressed a nonexistent record.
637	ERROR MESSAGE	DUPLICATE KEY [22] ! (COBERR 637)
	CAUSE	A WRITE statement was executed which would have created a duplicate key. The record already exists.
638	ERROR MESSAGE	FILE NOT OPENED OR MODE INCORRECT [47,48,49,9x] ! (COBERR 638)
	CAUSE	The file was not open, or was open with the incorrect mode. For example, trying to WRITE to a file opened in INPUT mode. Or a DELETE or REWRITE was executed for a file which was not opened in I-O mode.
639	ERROR MESSAGE	ATTEMPT TO ! A FILE THAT IS NOT OPEN [9x] (COBERR 639)
	CAUSE	An EXCLUSIVE, UN-EXCLUSIVE, or SEEK statement was executed for a file which was not OPEN. This is usually the result of a program logic error. Remove the redundant statement or insert any necessary OPEN statement and recompile the program.
640	ERROR MESSAGE	ATTEMPT TO OPEN A FILE WITH AN INVALID DYNAMIC FILE NAME [31] & (COBERR 640).
	CAUSE	The contents of the data item specified as containing the dynamic file name in the USING phrase of the SELECT clause do not form a legal file name or are not consistent with the name or literal in the ASSIGN phrase of that SELECT clause. This is usually the result of improper or missing initialization. Verify that the data item is initialized to a valid file name.
641	ERROR MESSAGE	FILE IN USE BY ANOTHER PROCESS [9x] (COBERR 641)
	CAUSE	An EXCLUSIVE CONDITIONALLY was executed for a file for which some other process had already executed an EXCLUSIVE. No USE procedure or FILE STATUS word was specified.
642	ERROR MESSAGE	FILE IS LOCKED BY A CLOSE [38] (COBERR 642)
	CAUSE	An OPEN statement was executed for a file which was locked by a previous CLOSE statement. The CLOSE WITH LOCK option prevents a file from being reopened within the current run-unit. Correct the program and recompile.

644	ERROR MESSAGE	UNABLE TO OPEN FILE, SEE FILE SYSTEM ERROR [9x] (COBERR 644)
	CAUSE	An attempt to execute an OPEN statement failed. Refer to the file system error.
645	ERROR MESSAGE	I/O ERROR ON OPEN [9x] (COBERR 645)
	CAUSE	An I/O error occurred while attempting to write CCTL information during an OPEN. Refer to the file system error.
646	ERROR MESSAGE	TOP TOO LARGE IN LINAGE CLAUSE (COBERR 646)
	CAUSE	The value of the TOP specification within the LINAGE clause was larger than 63. Zero is assumed.
647	ERROR MESSAGE	FILE NOT FOUND [35] (COBERR 647)
	CAUSE	An OPEN statement was executed for a file that could not be found.
648	ERROR MESSAGE	FILE'S FIXED ATTRIBUTES DIFFER FROM PROGRAM [39]! (COBERR 648)
	CAUSE	The file organization declaration within the ENVIRONMENT DIVISION is different than the organization of the actual file for which an OPEN statement was executed. Compare the two organizations and adjust as necessary to correct the problem. This could also occur if the record size does not match the size of the FD records. It could also occur if the KEY types in an INDEXED file do not match or the presence or absence of the WITH DUPLICATES clause does not match. Use the RECORD CONTAINS clause to specify the minimum and maximum valid record sizes.
649	ERROR MESSAGE	RECORDS MUST BE IN ASCENDING ORDER BY KEY [21] (COBERR 649)
	CAUSE	If a INDEXED file was created by COBOL, then whenever a OPEN OUTPUT with ACCESS MODE SEQUENTIAL is done, the key values must be written in primary key order.
650	ERROR MESSAGE	END OF FILE ENCOUNTERED UPON READ [10,46] (COBERR 650)
	CAUSE	A READ statement was executed and no record was found.
653	ERROR MESSAGE	FILE ALREADY OPEN WITH SAME AREA OR IS MULTIPLE FILE (COBERR 653)
	CAUSE	An OPEN statement was executed for a file which was specified in a SAME AREA clause that was in use or for a file which was listed in a MULTIPLE FILE clause which specified a file which was already open.

655	ERROR MESSAGE	SEQ ERROR ON LABELED TAPE (COBERR 655)
	CAUSE	An OPEN statement resulted in a SEQ error for a labeled tape. Refer to the file system error.
658	ERROR MESSAGE	RELATIVE KEY OVERFLOW [14,24] ! (COBERR 658)
	CAUSE	The value for a RELATIVE KEY item has overflowed while attempting to execute WRITE/READ statement to a relative file. The PICTURE for the RELATIVE KEY needs more digits.
659	ERROR MESSAGE	FILE/ORG/DEVICE DOES NOT SUPPORT OPEN MODE [37] ! (COBERR 659)
	CAUSE	An attempt was made to execute an OPEN statement on a device that does not support the OPEN mode specified, for example, OPEN INPUT for a line printer device. This could also occur if the file system, possibly through a file equation or security, does not honor the open mode. Or, use of a :FILE equation ACCESS= option conflicts with the OPEN mode. (Such as OPEN I-O versus ACCESS=IN.)
664	ERROR MESSAGE	I/O ERROR ON ! [30,9x] (COBERR 664)
	CAUSE	An I/O error occurred. Refer to the file system error. Or an attempt was made to execute an EXCLUSIVE statement for more than one file at a time. This requires that you have the MR capability. Database locks are equivalent to EXCLUSIVE.
665	ERROR MESSAGE	END OF FILE ENCOUNTERED UPON WRITE [24] (COBERR 665)
	CAUSE	An attempt was made to WRITE beyond the end of file. Refer to the file system error.
666	ERROR MESSAGE	ERROR WHILE READING/WRITING USER LABEL [9x] (COBERR 666)
	CAUSE	An error occurred while reading/writing the label. Refer to the file system error.
668	ERROR MESSAGE	LINAGE, TOP, BOTTOM, FOOTING > 32767 NOT ALLOWED, RESET TO 32767 (COBERR 68)
	CAUSE	Check the LINAGE, TOP, BOTTOM and FOOTING clauses.
671	ERROR MESSAGE	RECORDS LARGER THAN FD DESCRIPTION, TRUNCATED [44] (COBWARN 671)
	CAUSE	The record length found when the input file was opened was larger than the amount of space specified in the largest FD description for this file. Any records which are too long will be truncated.
672	ERROR MESSAGE	RECORD SIZE DOES NOT MATCH FD DESCRIPTION [44] ! (COBERR 672)
	CAUSE	The record size of the file does not match the minimum and maximum valid sizes in the RECORD CONTAINS clause.

673	ERROR MESSAGE	FD DESCRIPTION LARGER THAN RECORD SIZE, TRUNCATED [44] (COBWARN 673)
	CAUSE	The record length found when the file was opened for output was smaller than the FD record description. Data written to this file will be truncated.
706	ERROR MESSAGE	UNDERFLOW IN EXPONENTIATE (COBERR 706)
	CAUSE	The result value of an exponentiation operation has resulted in an underflow. The result falls in the range -10**-28 to +10**-28.
707	ERROR MESSAGE	OVERFLOW IN EXPONENTIATE (COBERR 707)
	CAUSE	The result value of an exponentiation operation has resulted in an overflow. The result has exceeded the range -10**-28 to +10**-28.
708	ERROR MESSAGE	UNDEFINED RESULT FROM EXPONENTIATE (COBERR 708)
	CAUSE	The result value of an exponentiation operation is undefined. An attempt was made to compile A**B, where A is EQUAL or LESS THAN 0 and B is not an integer.
709	ERROR MESSAGE	ILLEGAL DIGIT IN NUMERIC DATA ITEM, TREATED AS 0 ! (COBERR 709)
	CAUSE	There was a bad digit in an item input with ACCEPT FREE or in a source item for UNSTRING. This digit was changed to zero.
710 *	ERROR MESSAGE	ILLEGAL DECIMAL DIGIT (COBERR 710)
	CAUSE	An illegal packed decimal digit was encountered in execution. Correct the data and re-run the program. NM : The invalid data and the location is printed. The program may abort or continue to execute based on the VAR COBRUNTIME. CM : Possible instructions are: ADDD, SUBD, CMPD, SLD, NSLD, SRD, DIVD, MPYD. The invalid data and the instruction type is printed on the list device and the program is aborted.

* See the section "Run-Time Trap Handling" in Appendix H for more information.

711 *	ERROR MESSAGE	ILLEGAL ASCII DIGIT (COBERR 711)
	CAUSE	An illegal ASCII digit was encountered in execution. The invalid data and the location is printed. Correct the data, if necessary, and re-run the program. NM : The program may abort or continue to execute based on the VAR COBRUNTIME. CM : Possible instructions are: CVDA, CVAD, CVDB, EDIT, CVND. The invalid data and the instruction type is printed on the list device and the program may abort or continue to execute based on the ability to successfully FIXUP the bad digit. Refer to the article entitled "COBOL II/3000 Programs: Tracing the Illegal Data Using Error 710/711 Documentation," published in Issue #28 of the *Communicator* for additional details on error 710/711 processing.
745	ERROR MESSAGE	BAD PARAMETER TO !, ERROR ! (COBERR 745)
	CAUSE	■ A bad input value was detected for NUMVAL or NUMVAL-C functions. Check for comma, decimal point, and currency values. ■ A bad input value was detected for INTEGER-OF-DATE or INTEGER-OF-DAY functions. Check that YYY is > 1600 and that MM and DD or DDD are valid. ■ A bad input value was detected for DATE-OF-INTEGER or DAY-OF-INTEGER functions. Check that the input value > 0. Error Cause 1 Alpha char in beginning. 2 Embedded space in numeric string. 3 Too many signs (includes +, -, CR, and DB). 4 Too many decimal points. 5 Numbers and alphabetic characters mixed. 6 Number of digits > 18. 7 Something other than CR or DB found in string. 8 Illegal currency sign. 9 Illegal input format for the string. 11 Error in date.
746	ERROR MESSAGE	NO EXCEPTION PHRASE ON CALL, ERROR ! (COBERR 746)
	CAUSE	An error occurred trying to dynamically load a procedure. The parameter is the loader error number.
747 *	ERROR MESSAGE	NO SIZE ERROR PHRASE (COBERR 747)
	CAUSE	This message occurs for any SIZE ERROR condition on a verb that does not have a SIZE ERROR phrase. It also may occur on MOVEs if the data in a BINARY item does not match its picture. This error will also occur for any type of IEEE floating point exceptions that occur when evaluating FUNCTIONs. That is, divide by zero or taking a square root of a negative number.

* See the section "Run-Time Trap Handling" in Appendix H for more information.

748 *	ERROR MESSAGE	PARAGRAPH STACK OVERFLOW (COBERR 748)
	CAUSE	Recursive performs, or performs with common exit points. Or illegal GOTOs out of a PERFORMed procedure.
749	ERROR MESSAGE	COLLATING SEQUENCE NOT SUPPORTED, ERROR ! (COBERR 749)
	CAUSE	NLINFO (Native Language) returned an error.
750	ERROR MESSAGE	DEPENDING-ON IDENTIFIER OUT OF BOUNDS (COBERR 750)
	CAUSE	The depending on identifier was not between occurs-1 and occurs-2. This error only occurs when you select the BOUNDS option of the $CONTROL compiler subsystem command. Correct the program by removing the bounds violation.
751 *	ERROR MESSAGE	SUBSCRIPT/INDEX/REFMOD/DEP-ON OUT OF BOUNDS (COBERR 751)
	CAUSE	This error only occurs when you select the BOUNDS option of the $CONTROL compiler subsystem command. Correct the program by removing the bounds violation. For CM, only the first two errors apply. This could also occur for the function CHAR.
752	ERROR MESSAGE	REFERENCE MODIFICATION OUT OF BOUNDS (COBERR 752)
	CAUSE	This error only occurs when you select the BOUNDS option of the $CONTROL compiler subsystem command. Correct the program by removing the bounds violation.
753 *	ERROR MESSAGE	ADDRESS ALIGNMENT ERROR (COBERR 753)
	CAUSE	Parameter is not on a 32- or 16-bit boundary.
754 *	ERROR MESSAGE	INVALID GOTO (COBERR 754)
	CAUSE	GOTO. was not ALTERed before use. Could also occur for RETURN without SORT/MERGE or GOTO MORE-LABELS without OPEN.

* See the section "Run-Time Trap Handling" in Appendix H for more information.

Informational Messages

980	ERROR MESSAGE	ATTEMPTING TO RECOVER FROM SYNTAX ERROR.
	CAUSE	The indicated item is where the compiler is attempting to recover from the previous syntax error. The items between the two messages were ignored by the compiler.
981	ERROR MESSAGE	DEBUGGING SECTION ! IGNORED, NO DEBUGGING MODE CLAUSE.
	CAUSE	The indicated Debugging Section is ignored because the WITH DEBUGGING MODE clause was not specified in the SOURCE-COMPUTER paragraph.
982	ERROR MESSAGE	UNPREDICTABLE OPTIMIZATION RESULTS IF ERRORS OCCUR.
	CAUSE	
992	ERROR MESSAGE	COMPILER ERROR: ERROR MESSAGE #! OVERFLOWS BUFFER.
	CAUSE	Only first ~500 bytes are printed.
993	ERROR MESSAGE	FAILURE IN INTERNAL CALL TO INTRINSIC GENMESSAGE.
	CAUSE	See directions on line .3 of the file COBCAT.PUB.SYS.
994	ERROR MESSAGE	TOO MANY ERRORS FOR ERROR FILE. SOME ERRORS MAY NOT BE LISTED.
	CAUSE	
997	ERROR MESSAGE	MISSING ERROR MESSAGE #!.
	CAUSE	Check compiler and COBCAT (this file) for proper versions.
998	ERROR MESSAGE	TOO MANY ERRORS.
	CAUSE	The number of errors has exceeded 100 or value in $ERRORS=.
999	ERROR MESSAGE	UNABLE TO CONTINUE. COMPILATION TERMINATED.
	CAUSE	

B

Preprocessor Commands and $CONTROL Options

HP COBOL II contains special language independent compile-time features designed to simplify the compilation process. These features allow you to modify input text, set certain compilation options, or control the contents of the list output.

All these options are implemented by a simple compile-time language, consisting of commands, statements, variables or identifiers, and macro calls for text substitution.

Types of Processes

The compile-time processes are divided into three types. These types are edit processing, source input modification, and list and compilation options.

The first two processes mentioned above are independent of the compilation process. In effect, they constitute a "first pass" on your source program. The output of these preprocessor functions is released to the COBOL compiler. This output is referred to as the expanded source program.

The editing process merges lines from an optional textfile into your masterfile and provides other functions such as deletion and replacement of lines of the masterfile. The end product of the editing process is a single input stream.

The operating system's COBOL command, as well as the $EDIT and $INCLUDE preprocessor commands, are used to perform these editing operations.

The source input modification process uses macros to modify your source code. This is done by using the $DEFINE preprocessor command to define the macros, and then placing the macro name in your source code. When the source code is compiled, the definition of the macro replaces the macro name.

List and compilation options are provided to allow you to obtain compiler listings, source listings of the expanded source records, a symbol table map, the object code, a cross reference listing of symbols and procedures used in the source program, and a listing of statements appearing in the PROCEDURE DIVISION (as well as the line numbers in which they appear in the source code and their code offsets). These options are available through use of the $CONTROL preprocessor command.

Compilation options include warnings of possible error conditions, limiting the number of compilation errors before halting compilation, initializing the RL file, identifying the compiled code as being a subprogram (dynamic or nondynamic etc.), enabling the operating system's debug facility, setting bounds checking on table indices, subscripts, and reference modification, and flagging of HP extensions in the source program. These options are also available through use of the $CONTROL preprocessor command.

Preprocessor Programming Language

The preprocessor programming language is a simple language consisting of variables and commands. All of the preprocesses are performed at compile-time. None is performed at run-time.

Table B-1 lists all of the preprocessor commands. They are described in the remainder of this appendix.

Table B-1. Preprocessor Commands

Command	Purpose
$COMMENT	Writing comment lines.
$DEFINE $PREPROCESSOR	Defining and using macros.
$IF $SET	Conditional compilation.
$INCLUDE $EDIT	File insertion, merging and editing operations.
$COPYRIGHT $PAGE $TITLE $VERSION $CONTROL	Options affecting compiler output (code and listing).

The preprocessor commands have the following format:

$*commandname* [*parameterlist*]

Parameters

commandname — one of the command names shown in the list above.

parameterlist — a list of parameters for a given preprocessor command. The specific parameters (if any) allowed for a given preprocessor command are listed later in this section where the command is described. A list of parameters in a command must be separated from the command by one or more spaces, and each parameter specified must be separated from any succeeding parameter by a comma optionally followed or preceded by one or more spaces.

Description

The required dollar sign is used to identify a preprocessor command. This symbol must appear in the first column of a line of source code, immediately following the sequence number field.

The second, optional, dollar sign is used to indicate that the preprocessor command of which it is a part is to be executed, but is not to be copied (in a merging process) to the newfile.

Continuation Lines

A preprocessor command can be continued to the next line by using a continuation character as the last nonblank character in the line where the command is. The continuation character is the ampersand (&).

A continuation character can be used anywhere in a preprocessor command where a blank can be used and will not change the effect of the command.

Example. The following is a valid use of the continuation character:

```
001000$CONTROL CROSSREF,&
002000$MAP,LIST
```

When you continue a preprocessor command to another line, the new line must contain a single dollar sign in the first column following the sequence number field of that line, as shown in the example above.

The effect of using a continuation character in a preprocessor command is equivalent to replacing the continuation character and subsequent dollar sign with a blank, and concatenating the two lines. So the example above is equivalent to the following:

```
001000$CONTROL CROSSREF,MAP,LIST
```

Continuation lines of a preprocessor command are not copied to a newfile during a merging operation if the preprocessor command begins with two dollar signs.

Example. The following preprocessor command is executed but neither line 003000 nor line 004000 is copied to the newfile created by the merging process.

```
003000$$EDIT VOID=005000, &
004000$ SEQNUM=001000
```

$COMMENT Command

The $COMMENT command has the following format:

$COMMENT [*comment-text*]

where *comment-text* is a string containing anything you want to enter. *comment-text* requires no delimiters. It ends at the end of the line where the $COMMENT command is issued unless a continuation character is used. Use of COBOL comments, "*", is preferred.

The example below illustrates the $COMMENT command:

```
000100$COMMENT THIS LINE IS AN EXAMPLE OF THE $COMMENT PRE- &
000200$ PROCESSOR COMMAND.
```

Defining and Using Macros

One of the most powerful facilities of the preprocessor is the macro processor. This processor associates a macro name with a string of text. When the macro name is used by itself in a COBOL sentence, the preprocessor sends the definition of the macro to the compiler. See also the COPY and REPLACE statements.

Macros can have up to nine formal parameters. If formal parameters are used in the definition, actual parameters are supplied to replace them when the macro is called in the source program.

$DEFINE Command

Use the $DEFINE command to define a macro. When you define a macro, you associate a macro name with a string of text.

The $DEFINE command can redefine a previously defined macro. However, all macro definitions are global. If you use the $DEFINE command to redefine a macro, the old definition is lost, and can only be recovered by issuing another $DEFINE command that repeats the old definition.

Syntax

$DEFINE *macro-name*=[*string-text*]#

Parameters

macro-name the name of the macro being defined, and consists of an initial non-alphanumeric character (default is the percent sign, %), followed by an alphabetic character, followed by zero or more alphanumeric characters.

The length of the macro name may be any number of characters, but only the first fifteen are recognized by the preprocessor. Note that care must be taken to assure uniqueness of such names.

string-text can be any text you choose. However, because this text is sent to the compiler, it must be a valid COBOL statement or sentence, with one exception. This exception is the use of formal parameters in the *string-text*. Formal parameters are described later in this section.

Description

Note that *string-text* is delimited by an equal sign and a pound sign. The pound sign is the delimiter of the entire definition. This is a default delimiter, and can be changed by the $PREPROCESSOR command.

Nested $DEFINE commands and recursive macros can be used; however, you must take care in using recursive macros, as there is no specific method for terminating the macro call sequence when used in this manner.

If a continuation character is used in a macro definition, it is assumed to be a part of the macro definition, and not a continuation character. The following illustrates the use of a continuation character in defining a macro.

Example.

```
000100 $DEFINE %INCA=ADD 1 TO ALPHA-COUNTER.
000200       ADD ALPHA-COUNTER TO BETA-COUNTER.
000300*      INCREMENT THE VALUES OF COUNTERS ALPHA-COUNTER &
000400*      BETA-COUNTER.#
```

In the above example, the entire definition of the macro, %INCA, is the following set of sentences:

```
000100          ADD 1 TO ALPHA-COUNTER.
000200    ADD ALPHA-COUNTER TO BETA-COUNTER.
000300*   INCREMENT THE VALUES OF COUNTERS ALPHA-COUNTER &
000400*   BETA-COUNTER.#
```

The dollar symbol is not needed in continuation lines of a macro definition. The only time that a dollar sign would appear in the first column after the sequence number field is when the macro definition contains preprocessor commands.

When the $DEFINE command is processed, the *string-text* that makes up the definition of the macro is stored exactly as it appears in the command. All end-of-line markers and sequence numbers are saved. The only exception to this rule is that if all characters between the equal sign in the macro definition and the end of the text line in which the definition begins are blank, then only the end-of-line marker is saved, and the macro definition begins in the first column of the next line. This allows you to control the initial column of the macro definition without worrying about the column position of the macro name when it is called. Below is an illustration:

Macro definition:

```
000300$DEFINE %ADDIT=MOVE SPACES TO DISPLY-ITM.#
000400
                ⋮
```

Macro call:

```
001200     DISPLAY DISPLY-ITM. %ADDIT
                ⋮
```

Expanded source code:

```
001200     DISPLAY DISPLY-ITM. MOVE SPACES TO DISPLY-ITM.
                ⋮
```

Because the macro definition starts on the same line as the macro name, the definition of the macro replaces the macro call starting in exactly the same column as the macro call.

```
000100$DEFINE %CHECKIF=
000200      IF STAT-ITEM EQUAL "10"
000300         THEN PERFORM STATUS-REPORT
000400      ELSE NEXT SENTENCE.#
```

In this case, because the characters between the equal sign and the end of line 000100 are blanks, the macro definition begins on line 000200.

Thus, when the macro call to %CHECKIF is made, the expanded source appears as shown below.

Macro call:

```
002400     WRITE FILE-OUT. %CHECKIF
002500           ⋮
```

Replacement:

```
002400      WRITE FILE-OUT.
000200      IF STAT-ITEM EQUAL "10"
000300         THEN PERFORM STATUS-REPORT
000400      ELSE NEXT SENTENCE.
```

Formal Parameters

A macro definition may contain up to nine formal parameters. A formal parameter is designated by an exclamation point followed immediately by an integer from the range 1 to 9.

Formal parameters in a macro definition are replaced by values you assign when you call the macro in your source program, as illustrated below.

Source program:

```
000100$DEFINE %PERFORM=
000200      PERFORM !1
000300         VARYING !2 FROM !3 BY !4
000400         UNTIL !5.#
                  ⋮
001200  %PERFORM(CHEK-PARA#,CTROL#,INIT#,OFFSET#,A = B#)
                  ⋮
```

Expanded source program:

```
                  ⋮
000200      PERFORM CHEK-PARA
000300         VARYING CTROL FROM INIT BY OFFSET
000400         UNTIL A = B.
                  ⋮
```

Using Macros

Macro Calls

There are two different forms of a macro call:

macro-name

and

macro-name(p1#,p2#,p3#,...,pn#)

Parameters

macro-name	the name of a macro which has been previously defined in the source program, using a $DEFINE command.
p1,p2,...,pn	the actual parameters. Each of *p1*, *p2*, and so on may be either a null string or any combination of characters and numerals, including spaces. Each actual parameter begins with the first character after a preceding comma (except *p1*, which begins after the left parenthesis), and ends with the pound sign. If no characters are specified for an actual parameter (that is, if an actual parameter is specified by "#"), then a null string replaces its corresponding formal parameter in the macro definition.

Note from the above format that there can be no intervening spaces between the end of the macro name and the left parenthesis of the actual parameter list.

Continuation of lengthy parameters in a macro call is established by starting the continued line with a comma. The following illustrates this:

```
macro-name(p1#,p2#,p3#,p4#
          ,p5#,p6#,pn#)
```

The first method of calling a macro is used when the macro definition has no formal parameters.

The second method must be used when formal parameters are specified in the definition of the macro.

When a macro name is encountered in a source program, it is deleted, and the associated macro definition is sent to the compiler in place of the name. Any formal parameters are replaced by actual parameters listed in the macro call.

With two exceptions, macro names are replaced wherever they occur in the source program, including quoted strings. Macro names are not expanded 1) in a comment, unless the comment itself is found in a macro, and 2) in list and compilation preprocessor commands (such as $CONTROL), where they are not recognized.

Relationship of Formal Parameters to Actual Parameters

The numeric value of a formal parameter determines which actual parameter in the macro call is to replace it.

That is, for a formal parameter, *!n* (where *n* is 1 through 9), the *n*th actual parameter from the left in the macro call replaces *!n*, as illustrated below.

Examples. Macro definition:

```
$DEFINE %OPENSTATE=
         OPEN INPUT !1.
         DISPLAY !2,!3.#
```

Macro call:

```
%OPENSTATE(FILE-IN#,"FILE STATISTICS"#,   OPEN-STATS#)
```

Result of macro call:

```
OPEN INPUT FILE-IN.
DISPLAY "FILE STATISTICS",  OPEN-STATS.
```

Macro definition:

```
090000$DEFINE %COMPUTESUM=   COMPUTE  !1 = !2 + !3.#
```

Macro call:

```
091000 %COMPUTESUM(INCREMEN#,OFFSETTER#,7#)
```

Result of macro call:

```
091000    COMPUTE INCREMEN=OFFSETTER + 7.
```

In the second example above, `!3` is replaced by the third parameter in the macro call, which is 7; `!2` is replaced by the second parameter, `OFFSETTER`, and `!1` is replaced by the first parameter in the call, `INCREMEN`.

For a given macro definition, if there is a formal parameter, *!n*, and if there are less than *n* actual parameters in the macro call, then any formal parameter whose numeric value is greater than the number of parameters in the macro call is ignored. This is illustrated in the following example.

Macro definition:

```
001100$DEFINE  %SHOWIT=     DISPLAY !1,!2,!3,!5.#
```

Macro call:

```
002500 %SHOWIT("A"#," WORD "#,"IS "#,"MISSING"#)
```

Result of macro call:

```
002500     DISPLAY "A"," WORD ","IS ", .
```

When you specify a formal parameter in a macro definition, you can choose not to use it in the macro call by entering only a comma and a pound sign in the appropriate position within the macro call. This is shown in the next example:

Macro definition:

```
000100$DEFINE %INITSTUFF=
000200 IDENTIFICATION DIVISION.
000300 PROGRAM-ID. !1
000400 AUTHOR. !2
000500 DATE-COMPILED. !3#
```

Macro call:

```
001000 %INITSTUFF(MACRO-TEST.#,#,#)
```

Result of macro call:

```
000200 IDENTIFICATION DIVISION.
000300 PROGRAM-ID. MACRO-TEST.
000400 AUTHOR.
000500 DATE-COMPILED.
```

The last two actual parameters were specified by ",#". When the replacement code compiled, no author name or compile date was supplied.

The format for a macro definition assures that the initial column of each line in the macro definition maps onto the same column when the macro definition is inserted into the source program at macro call time. This could cause a wraparound of the replacement text when actual parameters are substituted in the definition. However, to ensure that this does not happen, blanks are removed from the executable text field to make it the correct size. If there are no trailing blanks to remove, the overflow portion of the executable text field is used to create a new record on the next line. The sequence number field is left blank for this new record.

Nested Macro Calls

You can pass the result of one macro call as an actual parameter to another macro by using the $PREPROCESSOR command discussed in the next section. By redefining the macro delimiter for the nested macro, you can use a call to that macro as a parameter to another macro. The following examples illustrate two slightly different ways of nesting macro calls.

Example 1. This example defines two macros, *%M1* and *%M2*. The macro *%M2* is passed as a parameter to *%M1*.

Macro definition:

```
$PREPROCESSOR DELIMITER=?              Change the delimiter to define M1.
$DEFINE %M1=                           Start the definition of M1.
$PREPROCESSOR DELIMITER=~              Change the delimiter within M1.
 DISPLAY "!1"                          Body of macro M1.
$PREPROCESSOR DELIMITER=#              Restore the delimiter to # within M1.
 ?                                     End of definition of M1.
$PREPROCESSOR DELIMITER=#              Restore the delimiter to #.

$DEFINE %M2=ADD !1 TO !2 GIVING !3#    Define macro M2.
```

In the following macro call, macro *%M2* is passed as a parameter to macro *%M1*. Notice that the "~" character is the delimiter for the parameters to *%M2*:

```
%M1(%M2(A~,B~,C~)#)
```

Result of macro call:

```
DISPLAY "ADD A TO B GIVING C"
```

Example 2. This example also defines two macros, *%M1* and *%M2*. The macro *%M2* is again passed as a parameter to *%M1*. In this case, however, a third macro, *%NESTEDM2*, is created that consists of the call to *%M2*. This third macro is passed to *%M1* instead of passing *%M2* directly.

Macro definition:

```
$DEFINE %M1=DISPLAY "!1"#              Define macro M1.
$DEFINE %M2=ADD !1 TO !2 GIVING !3#    Define macro M2.
```

In the following nested macro call, the delimiter is changed to create the third macro, *%NESTEDM2*. This third macro is passed to macro *%M1*:

```
$PREPROCESSOR DELIMITER=~              Change the delimiter.
$DEFINE %NESTEDM2=%M2(A#,B#,C#)~       Define macro NESTEDM2.
$PREPROCESSOR DELIMITER=#              Restore the delimiter to #.
%M1(%NESTEDM2#)                        Call M1, passing NESTEDM2.
```

Result of macro call:

```
DISPLAY "ADD A TO B GIVING C"
```

$PREPROCESSOR Command

The $PREPROCESSOR command allows you to change the default characters used in macro definitions and names. The default characters are:

% used as the initial character in a macro name.

! used as the first character of a formal parameter in macro definitions.

used to delimit string-text in a macro definition, and actual parameters.

To specify a different character to be used in place of one of these, use the $PREPROCESSOR command in the following format:

$PREPROCESSOR *parameter=subparameter* [, *parameter=subparameter*]

Parameters

parameter — one of the keywords shown below. Each may be used only once in a given $PREPROCESSOR command.

KEYCHAR specifies that the initial character of all macro names is to be subparameter.

PARMCHAR specifies that the initial character of all formal parameters in macro definitions is to be subparameter.

DELIMITER specifies that the delimiting character in a macro string-text is to be subparameter.

subparameter — the character to be used in replacing the currently used initial character or delimiter.

Example

```
000100$PREPROCESSOR KEYCHAR= ,PARMCHAR=?,DELIMITER=^
000200$DEFINE  MOVEIT=
000300      MOVE ?1 TO ?2.^
```

Note that care must be taken when you redefine the initial characters and the string-text delimiter, because there may be cases when you use one of the newly defined characters in your string-text as part of the string-text itself, and not as a delimiter or an initial character.

Conditional Compilation

Usually, when you compile a source file, you want the entire program compiled. However, there may be occasions when you want only part of the program compiled. Conditional compilation, that is, compilation of source code contingent upon whether a switch is on or off, is accomplished by using the $SET and $IF preprocessor commands.

$SET Command

The $SET preprocessor command may be used to turn ten compilation switches on or off. The ten software switches are of the form, Xn, where n is an integer in the range 0 through 9.

The $SET command has the following format:

```
$SET [ Xn={ ON  } [ , Xr={ ON  } ] ... ]
      {     }   {      }
      { OFF }   { OFF }
```

where X*n* and X*r* are compilation switches as described above.

Initially, all compilation switches are set to OFF.

A $SET command may appear anywhere in the source text. If used without parameters, that is, in the form $SET, it sets all switches to OFF.

$IF Command

The $IF command interrogates any of the ten compilation switches. If the condition specified in the $IF command is true, source records are sent to the compiler, beginning with the first one following the $IF command, and continuing until another $IF command is encountered which is false.

If the condition specified by the $IF command is false, no source records are sent to the compiler until a $IF command which is true is encountered. Note also, that during this processing any $SET and $CONTROL commands encountered are ignored by the compiler.

The $IF command has the following format:

```
$IF [ Xn= { ON  } ]
          {     }
          { OFF }
```

where X*n* is a compilation switch as described under the $SET command in the preceding paragraphs.

Conditional Compilation

The $IF command may appear anywhere in the source text.

The appearance of a $IF command always terminates the influence of any preceding $IF command.

When a $IF command is entered without a parameter, it has the same effect as a $IF command whose condition is true. That is, the text following the command is compiled, and any previous $IF command is canceled.

Note that the merging of a text and master source file, and copying of this merged file to a newfile is unaffected by $IF commands. Also, the $EDIT, $PAGE, and $TITLE commands are executed even when they appear in a portion of source code that is not to be sent to the compiler. However, all other preprocessor commands are ignored in such a portion of source code, that is, $SET, $CONTROL, etc.

If you do not want to list source records that are not compiled, you must use the CONTROL preprocessor command, specifying the NOMIXED parameter.

Example

```
$SET X1=ON, X3=ON
            ⋮
$IF X1=ON
$COMMENT SINCE X1 IS ON, CONTINUE SENDING RECORDS TO THE &
$        COMPILER.
            ⋮
$IF X3=OFF
$COMMENT THIS $IF COMMAND CANCELS THE PRECEDING ONE.  SINCE &
$         X3 IS SET TO "ON", DO NOT SENT THE FOLLOWING RECORDS &
$         TO THE COMPILER.

$SET X2=ON
$CONTROL NOLIST.
$COMMENT NOTE THAT THE $SET AND $CONTROL COMMANDS ARE &
$         IGNORED, SINCE THE PREVIOUS $IF WAS FALSE.


$IF
$COMMENT PREVIOUS $IF CONDITION IS TERMINATED, AND COMPILATION &
$        RESUMES.
```

File Insertion, and Merging and Editing Operations

There are essentially two types of file merging functions available to you through the preprocessor. The first uses the preprocessor command, $INCLUDE. The second merging function is done using any of the operating system's COBOL commands, and optionally, the $EDIT preprocessor command.

$INCLUDE Command

The $INCLUDE command allows you to specify an entire file to be sent, line by line, to the compiler as part of your source file. See also COPY statement.

The $INCLUDE command has the following format:

$INCLUDE *filename*

where *filename* is the name of the file whose records are to be sent to the compiler.

$INCLUDE commands may be nested. That is, the file that is being included may itself have an $INCLUDE command in it. This nesting may go to a depth of ten.

When the $INCLUDE command is encountered in a source file, the following actions take place:

1. The file named in the $INCLUDE command is opened.
2. This file then becomes the input file. Each line of the file is processed and the results are sent to the compiler. If sequence numbers exist for the lines (records) of the file, they are preserved.
3. When the end-of-file is reached for the included file, it is closed.
4. The next line in the original source code is processed and sent to the compiler.

Note that the $INCLUDE command has no effect upon the text, master or newfiles. The $INCLUDE command, if used, and the included file data is sent to the listfile, while only the $INCLUDE command is sent to the newfile resulting from the compilation process.

Example

```
000100$INCLUDE INITFILE
001000 ENVIRONMENT DIVISION.
001100 CONFIGURATION SECTION.
            ⋮
```

If INITFILE contains the records,

```
000200$CONTROL SUBPROGRAM
000300 IDENTIFICATION DIVISION.
000400 PROGRAM-ID. SUBDUMMY.
000500 AUTHOR. JPW.
000600 INSTALLATION. GSD.
000700 DATE-WRITTEN. DUMMY-WRITTEN-05/08/85.
000800 DATE-COMPILED. WHEN-USED.
000900 SECURITY. NONE-ON-DUMMY.
```

Then when the $INCLUDE statement above is executed, the results are as follows.

```
000100$INCLUDE INITFILE
000200$CONTROL SUBPROGRAM
000300 IDENTIFICATION DIVISION.
000400 PROGRAM-ID. SUBDUMMY.
000500 AUTHOR. JPW.
000600 INSTALLATION. GSD.
000700 DATE-WRITTEN. DUMMY-WRITTEN-05/08/85.
000800 DATE-COMPILED. WHEN-USED.
000900 SECURITY. NONE-ON-DUMMY.
001000 ENVIRONMENT DIVISION.
001100 CONFIGURATION SECTION.
            ⋮
```

Merging Files and the $EDIT Command

This section is obsolete and its use is not recommended.

Merging and editing operations are done prior to other preprocessor functions. The following editing and merging operations are available to you at compile-time:

1. Merge corrections or additional source text in the textfile with an existing source program (masterfile) to produce a new source program and commands.
2. Omit sections of the old source program during merging.
3. Check source-record sequence numbers for ascending order.

Merging Files

Merging is done at compile-time by using the textfile, masterfile, and newfile parameters of the operating system's COBOL commands.

(For a detailed description of these commands, refer to "MPE XL System Dependencies" in Appendix H.)

Prior to merging, the records in both textfile and masterfile must be arranged in ascending order according to the values in their sequence fields, or sequence fields may be blank.

The order of sequencing is based on the ASCII collating sequence.

The merging operation is based on nonblank ascending sequence fields. During merging, nonblank sequence fields of records in both files are checked for ascending order. If their order is improper, the appropriate diagnostic messages are sent to your listfile.

Blank sequence fields are never considered out of sequence. They are assumed to have the same sequence value as the last preceding record which contains a nonblank sequence value. The sequence fields of some or all of the records in either file may be blank, and such records may appear anywhere in the files.

During each comparison in the merging process, one record is read from each file, and these records are compared, with one of three results, depending upon whether the value of the field in the record of the textfile is equal to, greater than, or less than the value of the sequence field in the record of the masterfile.

If the values of the sequence fields are equal, the textfile record is sent to the compiler, and the record of the masterfile is ignored.

If the value of the sequence field in the textfile is greater than the value of the sequence field in the masterfile, the record of the masterfile is sent to the compiler, and the record of the textfile is retained for comparison with the next record of the masterfile.

If the value of the sequence field in the textfile is less than the value of the sequence field in the masterfile, the record of the textfile is sent to the compiler, and the masterfile record is retained for comparison with the next textfile record.

Records with a blank sequence field (from either file) are assumed to have the same sequence field value as that of the last record with a nonblank sequence field read from the same file. If no record with a nonblank sequence field has yet been encountered, the blank records are assumed to have a null sequence field.

Merging Files

The above description implies that records from the masterfile with blank sequence fields are always compiled. This is because the records of the masterfile with blank sequence fields will either eventually be less than a sequence field for a textfile record, or the entire textfile will have been used.

When an end-of-file condition is encountered on either textfile or masterfile, merging terminates (except for the continuing influence of an unterminated VOID parameter in an EDIT command). At this point, the subsequent records on the remaining file are checked for proper sequence, and are then compiled (except for masterfile records within the range of a VOID parameter in a $EDIT command).

Note that masterfile records that were replaced in the merging process are not listed in your listfile during compilation. Also, files sent to the compiler by a $INCLUDE command have no sequence field checking performed on them.

Sequence Field Checking

Sequence fields are checked for proper order during the merging process provided that both a text and a masterfile are present.

If you do not have a textfile, or if you have no masterfile, and you still wish to have sequence fields checked, you can equate the missing file to $NULL. If the sequence fields are out of order, a warning message is generated and sent to the listfile. However, improperly arranged sequence fields will not cause the compilation of the specified file to fail.

$EDIT Command

The $EDIT preprocessor command can be used to bypass all records of the masterfile whose sequence fields contain a certain value, and renumber the numeric sequence fields of records in the newfile created by merging a textfile and a masterfile.

The $EDIT command has the following format:

$EDIT [*parameter=subparameter*] [, *parameter=subparameter*] [...]

Parameters

parameter	either VOID, SEQNUM, NOSEQ, or INC.
subparameter	either a sequence value, a sequence number, or an increment number. Which one is used depends on the parameter.

VOID Parameter

The VOID parameter is the only parameter of the $EDIT command that has any effect upon the compilation process. The other parameters have an effect only upon the newfile created by the merge process. If no newfile is specified in the operating system command that initiates the compilation process, all $EDIT commands other than $EDIT VOID=sequence value are ignored.

If VOID is specified, then the subparameter must be a sequence value. This parameter indicates that all records on the masterfile whose sequence values are less than or equal to the specified sequence value, and any subsequent records with blank sequence fields, are to be ignored. Thus, such records are not sent to the compiler.

This voiding of masterfile records continues until a record of the masterfile with a sequence value higher than that specified in the $EDIT VOID command is encountered. When this occurs, the merging process continues as described on earlier pages.

The sequence value of the VOID parameter can be either a sequence number or a character string. If the sequence value is less than the length of the sequence field for the masterfile, it is left filled with zeroes (if a number), or it is left filled with blanks (if a character string).

Note that if the sequence fields for records in the masterfile contain only numerals, and the sequence value of the VOID parameter is a character string, all of the masterfile is voided.

SEQNUM Parameter

SEQNUM allows you to renumber sequence fields in the newfile. If SEQNUM is used as a parameter in the $EDIT command, it has no effect upon the master or textfile.

Initially, renumbering of records in a newfile is disabled.

SEQNUM requires a sequence number as its subparameter. This sequence number is the value used for the first record when resequencing begins. Subsequent records are renumbered using an increment of 100, unless the INC parameter of $EDIT is used to specify a different one. The records to be renumbered start with the first record to be sent to the newfile following the $EDIT SEQNUM command. This renumbering continues until a $EDIT NOSEQ command is encountered in the file where the $EDIT SEQNUM command appears.

If the sequence number subparameter of the SEQNUM command is of insufficient length to fill the sequence field of the newfile, sufficient zeroes are appended to the left of the sequence number to fill the sequence field of the records in the newfile.

NOSEQ Parameter

The NOSEQ parameter of the $EDIT command is used to terminate the resequencing of a newfile initiated by a $EDIT SEQNUM command. If a $EDIT NOSEQ command is issued after a $EDIT SEQNUM command, resequencing of newfile records is terminated, and remaining records sent to the newfile retain their old sequence values until a new $EDIT SEQNUM command is encountered.

If a $EDIT NOSEQ command is issued when no resequencing has been specified, it is ignored. Therefore, because no resequencing takes place unless you issue a $EDIT SEQNUM command, the $EDIT NOSEQ command is the default. In this case, sequence fields are retained and passed to the newfile as records are sent to the compiler.

INC Parameter

If INC is used as a parameter of the $EDIT command, it must have an increment value associated with it. This increment value is used when a $EDIT SEQNUM command is issued to renumber lines of the newfile. The increment value may range from 1 through 999999. Note, however, that a very large increment is of limited value, because it may cause the sequence number to be long for the sequence field.

If no $EDIT INC command is issued, the default value is 100. This default value stays in effect until an INC parameter specifying a new increment value is encountered.

In general, if you wish to provide for a different increment during a resequencing operation, the $EDIT INC command must be specified before the $EDIT SEQNUM command is executed. As with the default value, the increment value specified in the $EDIT INC command stays in effect until a new increment value is specified.

$EDIT commands are normally part of the textfile. You may use them in the masterfile, but it is not recommended because any $EDIT command using the VOID parameter in the masterfile could void its own continuation records. $EDIT commands themselves are never sent to the newfile. While sequence fields are allowed, and indeed usually necessary, on records containing $EDIT commands, continuation records for such commands should have blank sequence fields.

During merging, a group of one or more masterfile records with blank sequence fields are never replaced by lines from the textfile. They can only be deleted by using the VOID parameter of $EDIT. This is accomplished by using a sequence value subparameter at least as great as the contents of the last nonblank sequence field preceding the group of records with blank sequence fields. Because voided records are never passed to the compiler or newfile, their sequence fields are never checked for proper sequence, and they never generate an out-of-sequence diagnostic message.

Any masterfile record replaced by a textfile record is treated as if voided, except that following records with blank sequence fields are not also voided. If a replaced record is out of sequence, the textfile record that replaces it produces an out-of-sequence diagnostic message.

In general, whenever a record sent to the newfile has a nonblank sequence field lower in value than that of the last record with a nonblank sequence field, a diagnostic message is printed.

Compiler-Dependent Options

This section covers the five preprocessor commands that are compiler-dependent. This means that they are processed by the HP COBOL II compiler.

The compiler-dependent preprocessor commands are $COPYRIGHT, $PAGE, $TITLE, $VERSION, and $CONTROL.

$COPYRIGHT Command

With the $COPYRIGHT command you can put a copyright string into your object file.

Syntax

$COPYRIGHT [*string* [, *string*] ...]

Parameter

string — the data to be placed into the object file. The characters of *string* must be preceded and followed by a quotation mark. The total number of characters used in the strings is limited to 116. This includes any blanks appearing in strings, but does not include the quotation marks used to delimit the strings.

Description

$COPYRIGHT places the specified strings into the object file when the program is compiled and linked. If you have more than one $COPYRIGHT command in your program, only the last one is used. If multiple source files make up a program file and there is a $COPYRIGHT for each, one $COPYRIGHT for each source file is used.

Examples

The following is an example copyright string:

```
$COPYRIGHT "Copyright 1991 My Company, Inc. All rights reserved."
```

The following shows a copyright string continued onto a second line:

```
$COPYRIGHT "Copyright 1991 My Company, Inc. ",&
$"All rights reserved."
```

$PAGE Command

The $PAGE command allows you to replace the first line of the title portion of the standard page heading in a listfile, and to advance to the next logical page of the listfile.

Syntax

$PAGE [*string* [, *string*] ...]

Parameter

string — the data to be used in replacing the first line of the title. The characters of *string* must be preceded and followed by a quotation mark. The total number of characters used in the strings is limited to 97. This includes any blanks appearing in strings, but does not include the quotation marks used to delimit the strings.

Description

The title line resulting from execution of the $PAGE command is a concatenation of all characters in all strings used in the command, in the order in which the strings are specified.

If no string is specified, $PAGE does not change the first line of the title, but, if $CONTROL LIST is in effect, causes the listfile to be advanced to the next logical page.

If the $CONTROL LIST command is in effect when the $PAGE command is encountered, the listfile is advanced to the next logical page and the standard page heading, including the new title, is printed, followed by one or two blank lines, depending upon whether the title is one or two lines long.

If the $CONTROL NOLIST command is in effect when the $PAGE command is encountered, the first line of the title is replaced with the specified string (or strings), but no page is advanced, and no listing of title or text occurs.

Note that $PAGE is never listed in the listfile.

$TITLE Command

The $TITLE command is similar to $PAGE in that it can be used to replace the first line of a title in the listfile. However, it can also be used to replace the second line of the title as well as the first, or only the second line; unlike $PAGE, it does no page advancement on the logical page of the listfile.

Syntax

$TITLE [(*n*)] [*string* [, *string*] ...]

Parameters

n specifies which line of the title is to be replaced. Thus, *n* can be either 1 or 2, and must be preceded and followed by a space. The default is 1.

string has the same format, restrictions and use as in the $PAGE command.

Description

If the $TITLE command is used with no parameters, it is equivalent to replacing the first line of the title with blanks.

$VERSION Command

With the $VERSION command you can put a version string into your object file.

Syntax

$VERSION [*string* [, *string*] ...]

Parameter

string — the data to be placed into the object file. The characters of *string* must be preceded and followed by a quotation mark (") or an apostrophe ('). The total number of characters used in the strings is limited to 255. This includes any blanks appearing in strings, but does not include the quotation marks used to delimit the strings.

Description

$VERSION places the specified strings into the object file when the program is compiled and linked. If you have more than one $VERSION command in your program, only the last one is used. If multiple source files make up a program file and there is a $VERSION command for each, one $VERSION for each source file is used.

To display the version strings from an object file or a program file, run the program VERSION.PUB.SYS.

Examples

The following is an example version string:

```
$VERSION "ABC Application, XYZ Company, Revision D.01.12"
```

The following shows a version string continued onto a second line:

```
$VERSION "ABC Application, XYZ Company, ",&
$"Revision D.01.12"
```

The following shows how to add the $VERSION command using the INFO string (note the use of an apostrophe instead of a quotation mark):

```
:COB85XL SRC,OBJ,$NULL;INFO="$VERSION 'REV'"
```

$CONTROL Command

The $CONTROL command controls compilation and list options. It has the following format:

$CONTROL *option* [, *optionlist*]

Parameters

optionlist — one or more valid options, each separated from the preceding option by a comma and zero or more optional spaces. See Table B-2 for a complete list of $CONTROL options.

option — a valid option for the $CONTROL command. Table B-2 lists all the $CONTROL options.

The following lists all the $CONTROL options:

Table B-2. $CONTROL Options

ANSISORT	INDEX32[1]	RLINIT[1]
ANSISUB	LINES=*number*	SOURCE
BOUNDS	LIST	NOSOURCE
CALLINTRINSIC[1]	NOLIST	STAT74
CHECKSYNTAX	LOCKING	STDWARN[=*level*]
CMCALL[1]	LOCOFF	NOSTDWARN
CODE	LOCON	SUBPROGRAM
NOCODE	MAP	SYMDEBUG
CROSSREF	NOMAP	SYNC16
NOCROSSREF	NLS=*options*[1]	SYNC32
DEBUG	MIXED	USLINIT
DIFF74	NOMIXED	VALIDATE[1]
DIFF74=0BS	OPTFEATURES=*options*[1]	NOVALIDATE[1]
DIFF74=INC	OPTIMIZE[=*number*][1]	VERBS
DYNAMIC	POST85[1]	NOVERBS
ERRORS=*number*	QUOTE=	WARN
INDEX16[1]	RLFILE[1]	NOWARN

1 This option is available only on HP COBOL II/XL. See Appendix H, "MPE XL System Dependencies", for more information.

The default $CONTROL options are shown below:

```
$CONTROL NOCODE, NOCROSSREF, ERRORS=100, LINES = 60, QUOTE='', LIST, LOCON, &
$ NOMAP, MIXED, SOURCE, NOSTDWARN, NOVERBS, WARN
```

Note For a description of other $CONTROL commands, refer to "MPE XL System Dependencies" in Appendix H.

ANSISORT

This option is provided to allow you to open the files specified in the USING or GIVING clause of the SORT statement in the input or output procedure of the same SORT statement. This capability will degrade the performance of SORT statements in the program. It is also not recommended to use the USING file as an output file in the output procedure of the SORT. If you do, you will not be able to recover the data that went into the sort if the program should abnormally terminate.

ANSISUB

The ANSISUB option can be seen as a combination of the DYNAMIC and SUBPROGRAM options. It determines that the expanded source code currently being compiled is to be a subprogram which strictly conforms to the ANSI standard in the following ways:

1. It can be called with the CALL *identifier-1* form of the CALL statement (as with DYNAMIC).
2. It maintains values of data items from one call to the next (as with SUBPROGRAM).
3. You can use the CANCEL statement to cause subsequent calls to the subprogram to reset data items to their initial values.

Refer to "MPE XL System Dependencies" in Appendix H for more information.

BOUNDS

The BOUNDS option requests the compiler to generate code for both the validation of indices and subscripts and the validation of start and length for reference modification used in tables. When an invalid index is detected, the program continues to execute, enabling you to continue testing the program for other conditions. Initially, no bounds checking is enabled. This option, if used, must appear in a $CONTROL command before the PROCEDURE DIVISION is encountered. Refer to "MPE XL System Dependencies" in Appendix H for further details.

CHECKSYNTAX

The CHECKSYNTAX option checks the syntax of the program without producing an object program. This option can also be used to produce a complete compiler listing that lists COPYLIB modules called by the program.

CODE

The CODE option requests a copy of the object code to be included. This object code is a listing of the machine code generated by the compilation of your expanded source code. This option must appear before the IDENTIFICATION DIVISION. Refer to "MPE XL System Dependencies" in Appendix H for more information.

NOCODE

The NOCODE option requests that no object code be included. NOCODE is the default. It is used to negate a previously issued $CONTROL CODE command. This option must appear before the IDENTIFICATION DIVISION. See "CODE" above.

CROSSREF

The CROSSREF option of $CONTROL requests a cross reference if symbols and procedures used on the expanded source file. This cross reference is sent to the listfile.

NOCROSSREF

The NOCROSSREF option of $CONTROL requests that no cross reference of symbols and labels used in the expanded source file be listed in the listfile.

This is the default; thus, its use is to cancel a previously issued $CONTROL CROSSREF command.

DEBUG

The DEBUG option requests that HP COBOL II generate a call in the initialization segment of your main program to the XCONTRAP intrinsic. This operating system intrinsic allows you to transfer control to Debug at any time during execution of the prepared code. To transfer control to Debug, you must press the control (CNTL) key, and while holding it down, press the Y key.

DIFF74, DIFF74=OBS, and DIFF74=INC

The DIFF74 option turns on flagging of differences between the ANSI COBOL'74 and the ANSI COBOL'85 standards. Specifying DIFF74=OBS flags all obsolete features. Specifying DIFF74=INC flags incompatible features that can be detected at compile-time, as well as the obsolete features. Specifying DIFF74 has the same effect as specifying DIFF74=INC. Use this option with COBOL'74 code that you want to migrate to COBOL'85.

See Appendix C, "Differences Between ANSI COBOL'74 and ANSI COBOL'85" for a list of differences.

DYNAMIC

The DYNAMIC option indicates that the expanded source code currently being compiled is to be a subprogram. This or the SUBPROGRAM option must be used if you are compiling a subprogram; otherwise, the compiler assumes that the expanded source file is a main program.

ERRORS=*number*

The ERRORS option uses an integer suboption, *number*, to specify the maximum number of errors to be allowed before compilation of the expanded source file is terminated. Thus, for example, if *number* is set to 10, then when compilation of a source file begins, it will terminate either after all records have been compiled with less than ten errors, or after the tenth compilation error has been found and listed to the listfile.

LINES=*number*

The LINES option of the $CONTROL command allows you to define the number of lines to be written on the logical page of your listfile. For example, if the line printer is your listfile, then the command, $CONTROL LINES=30, causes thirty lines to be listed on each page of the hard copy produced by the line printer.

$CONTROL Options

LIST

The LIST option of the $CONTROL command enables the listing of all source text, as well as error and warning messages, subsystem initiation and completion messages, and all other listings requested by the $CONTROL command (as for example, $CONTROL MAP).

Initially, this listing option is in effect by default. You might wish to use it to cancel a previously issued $CONTROL NOLIST, or $CONTROL LOCOFF command.

NOLIST

The NOLIST option of $CONTROL disables the listing of source text, and all other listings requested by previous $CONTROL commands. It does not, however, disable the listing of erroneous source records, error and warning messages, and subsystem initiation and completion messages.

LOCKING

This option is not required if the EXCLUSIVE and UN-EXCLUSIVE statements are used. The LOCKING option of the $CONTROL option enables dynamic locking of all files opened during the execution of your program. Note that a $CONTROL LOCKING command does not lock your file; it simply makes it possible for you to do so from within your program.

LOCOFF

The LOCOFF option of $CONTROL has the same effect as the NOLIST option. Only error and warning messages, and subsystem initiation and completion messages are listed.

A $CONTROL LOCOFF command remains in effect until either a $CONTROL LOCON, a $CONTROL LIST option, or the last source line is encountered.

A $CONTROL LOCOFF command can be nested.

LOCON

The LOCON option of the $CONTROL command negates the effect of any $CONTROL LOCOFF command issued previously. If a $CONTROL LIST command has been issued before $CONTROL LOCOFF, then a following $CONTROL LOCON restores listing of output to the listfile. If a $CONTROL NOLIST command was issued before a $CONTROL LOCOFF, then a following $CONTROL LOCON has no effect upon the listfile.

$CONTROL LOCON and $CONTROL LOCOFF commands can be nested.

You can use $INCLUDE to copy a file into your source file, and $CONTROL LOCOFF to suppress the listing of that file. Use $CONTROL LOCOFF as the first command and $CONTROL LOCON as the last command in the file to be copied. This suppresses the listing of the copied file and restores the listing option in effect before the $INCLUDE command was encountered.

MAP

The MAP option of $CONTROL requests a symbol table map to be included in your listfile. When certain compile-time errors occur, the symbol table map is not printed.

NOMAP

The NOMAP option requests that no symbol table map be included in your listfile. This is the default; thus, the purpose of the NOMAP option is to negate a previously issued $CONTROL MAP command.

MIXED

The MIXED option of the $CONTROL command requests that the listfile include all preprocessor commands used in the expanded source file. Note that the $PAGE and $TITLE preprocessor commands are not listed even if this option is in effect. This option is the default; thus, if you do not wish to have preprocessor commands included as part of the listfile, you should use the NOMIXED option of the $CONTROL command.

NOMIXED

The NOMIXED option of $CONTROL requests that no preprocessor commands be listed in the listfile.

QUOTE = {" '}

The QUOTE option in HP COBOL II is usually not necessary. It is provided only for the purpose of defining figurative constant QUOTE and simplifying conversion from COBOL/3000 source programs (based on ANSI COBOL'68) to HP COBOL II source programs.

SOURCE

The SOURCE option of $CONTROL command requests the listing of the expanded source file to the listfile.

Initially, this listing option is in effect by default; however, if the input file is your terminal, this option is disabled and NOSOURCE is initially the default, unless you explicitly request SOURCE. Note that if you do request the SOURCE option while using your terminal as the input device, the result is an echoing of each line entered.

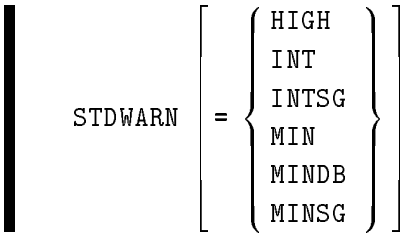
NOSOURCE

The NOSOURCE option of $CONTROL disables listing of the expanded source file generated by the preprocessor.

STAT74

This option allows you to specify that file status codes should be returned in the manner specified in ANSI COBOL 1974 (the way the ANSI74 entry point works) when the ANSI85 entry point is used. This allows you to use the ANSI COBOL 1985 features together with the old file status codes.

STDWARN



The STDWARN option requests that the compiler detect and flag features in your source file that are part of HP COBOL II but not part of one of three levels of Federal Standard COBOL (FIPS). The three levels are minimum, intermediate, and high. The default, high, flags HP COBOL II extensions. These flagged features are listed in your listfile. STDWARN is useful in converting your HP COBOL II source program to conform to one of the three Federal Standard levels of COBOL. The INTSG, MINDB, and MINSG flags are used to flag the minimum and intermediate levels of the optional modules Debug and Segmentation.

Table B-3 summarizes which of the subsets of the 12 standard COBOL modules are contained in each of the three federal levels of the language.

To illustrate how to interpret the table, examine the Relative I-O module entry.

When you request that your program be compared to the minimum level federal standard, any program construct using Relative I-O will be flagged, because the minimum level federal standard subset has no implementation of Relative I-O.

When compared to the intermediate federal standard, which has a level 1 subset of Relative I-O, only level 2 constructs would be flagged as not being part of the intermediate federal standard. When your program is compared to the high level federal standard for COBOL, no construct utilizing Relative I-O will be flagged, because the high level federal standard contains the level 2 subset of Relative I-O.

Note The HP COBOL II compiler does not implement every feature of all the modules. Refer to Table 1-2 for a summary of the subunits that are implemented in HP COBOL II.

Table B-3. FIPS COBOL Subsets

COBOL Module	Minimum Level	Intermediate Level	High Level
Nucleus	1	1	2
Sequential I-O	1	1	2
Relative I-O	-	1	2
Indexed I-O	-	1	2
Interprogram Communication	1	1	2
Sort-Merge	-	1	1
Source Text Manipulation	-	1	2
Report Writer	-	-	-
Segmentation	-	1[1]	2[1]
Debug	-	1[1]	-
Communication	-	-	-
Intrinsic Functions	-	-	1

1 Use INTSG, MINDB, or MINSG to flag

NOSTDWARN

The NOSTDWARN option requests that the compiler not flag nonstandard features of HP COBOL II. This is the default condition of the compiler. Thus, if you want to have features that are not part of Federal Standard COBOL noted, use the $CONTROL STDWARN option.

SUBPROGRAM

The SUBPROGRAM option indicates that the expanded source code currently being compiled is to be a subprogram using static storage. This option, or the DYNAMIC option, must be used when you are compiling a subprogram, because the compiler otherwise assumes the expanded source code to be the source for a main program.

SYMDEBUG

The SYMDEBUG option causes the compiler to put symbolic debug information into the object file for symbolic debugging. The main program should include this option if any subprogram includes the option. Refer to the *HP COBOL II/XL Programmer's Guide* for more information. This option must appear before the IDENTIFICATION DIVISION.

The SYMDEBUG option can output symbolic debug information into the program file for the HP Symbolic Debugger/XL. See Appendix H, "MPE XL System Dependencies," for more information.

Note With the SYMDEBUG option, the compiler places significantly more information in the object file.

SYNC16 and SYNC32

This option changes the alignment of SYNCHRONIZED data items. This alignment affects the number of slack bytes generated in a record.

If SYNC16 is in effect, index data items and items with the SYNCHRONIZED clause are aligned along 16-bit boundaries. If SYNC32 is in effect, these items are aligned along 32-bit boundaries.

This option can be used more than once in a program to align one record along 16-bit boundaries, and another record along 32-bit boundaries. Alignment cannot be changed within a record, only between records. This option is especially useful when developing files to be used on other computer architectures.

For information on the default, refer to "MPE XL System Dependencies" in Appendix H.

USLINIT

The USLINIT option initializes the USL file into which the compiled code is stored. This gives you a completely clear file. You should not use this option if you are compiling a source file into a USL file containing the object code of a main or subprogram to be used with the object code currently being compiled. Refer to "MPE XL System Dependencies" in Appendix H for more information.

VERBS

The VERBS option of the $CONTROL command requests that the listing of statements used in the PROCEDURE DIVISION and their code offsets be listed in the listfile.

This option must appear before the IDENTIFICATION DIVISION. When certain compile-time errors occur, the verb map is not printed.

NOVERBS

The NOVERBS option of the $CONTROL command prohibits the listing of the verb map. See "VERBS" above.

NOVERBS is the default and is used to cancel a previously issued $CONTROL VERBS command. This option must appear before the IDENTIFICATION DIVISION.

WARN

The WARN option enables the flagging of possible, but not clearly, erroneous conditions within your expanded source file. These flagged conditions are listed in your listfile, along with an appropriate warning message.

WARN is a default condition. Thus, you need only use it to negate the use of a previously issued $CONTROL NOWARN command.

NOWARN

The NOWARN option disables the flagging and listing of possible erroneous conditions and their associated warning messages.

The COBCNTL FILE

The file `COBCNTL.PUB.SYS` is used to override the compiler defaults for the compiler options. The compiler automatically includes a file named `COBCNTL.PUB.SYS` in each source textfile. You can define different system-wide defaults by adding new $CONTROL commands to the file.

For example, to define the defaults as SYNC32, MAP, and CROSSREF, change the `COBCNTL.PUB.SYS` file contents to the following:

```
000010$CONTROL SYNC32
000020$CONTROL MAP
000030$CONTROL CROSSREF
```

The contents of the INFO string override the `COBCNTL.PUB.SYS` file. Also you can use a file equation for `COBCNTL.PUB.SYS` to another file.

C

Differences Between ANSI COBOL'74 and ANSI COBOL'85

This appendix describes the differences and incompatibilities between the 1974 ANSI COBOL standard and the 1985 ANSI COBOL standard and identifies obsolete features that will be deleted from standard COBOL.

ANSI74 Entry Point Differences

The following is a list of the ANSI COBOL'74 features accessible through the ANSI74 entry point.

Exponentiation has been corrected as defined in the 1985 standard for the following special cases:

- If a value less than or equal to zero is raised to a power of zero, the size error condition results.
- If no real number exists as the result of an evaluation, the size error condition results.

The CANCEL statement closes all files.

The STOP RUN statement closes all files.

The EXIT PROGRAM statement is executed when there is no next executable statement in the called program.

The COPY statement follows ANSI COBOL'85 rules.

The following ANSI COBOL'85 features may be used with ANSI74. They are flagged as extensions if you specify $CONTROL STDWARN.

- De-edited MOVE's
- Non-numeric literals > 132 characters
- Use of < > as a relational operator
- Use of the double negative (NOT < >)

When compiling statements from the report writer or the communications module, the compiler issues a syntax error message.

The following new $CONTROL options may be used: SYNC16/SYNC32, DIFF74.

Note

COBOL'68 syntax that was treated as comments by the compiler, is implemented in the HP COBOL II compiler. However, use of this feature in future programming is not suggested. (Refer to "System Dependencies" in Appendix H for more information.) For example, the PROCESSING MODE IS SEQUENTIAL phrase of the SELECT statement is COBOL'68 syntax that is treated as *comments*.

Incompatibilities between ANSI COBOL'74 and ANSI COBOL'85

The following paragraphs summarize the syntax and run-time incompatibilities between ANSI COBOL'74 and ANSI COBOL'85.

Syntax Incompatibilities

Syntax incompatibilities are:

- The keyword ALPHABET must precede each alphabet name within the ALPHABET-NAME clause of the SPECIAL-NAMES paragraph.
- New reserved words have been added. Refer to Appendix F for more information.
- COPY REPLACING does not allow *pseudo-text-1* to contain only a comma or semicolon.

Run-time Incompatibilities

Run-time incompatibilities are:

- When a receiving item is a variable length data item and contains the object of the DEPENDING ON phrase, the maximum length is used.
- A data item appearing in the USING phrase of the PROCEDURE DIVISION header must not have a REDEFINES clause in its data description entry.
- The following cases of exponentiation are now defined:
  - If a value less than or equal to zero is raised to a power of zero, the size error condition occurs.
  - If no real number occurs as a result of the evaluation, the size error condition occurs. For example, -4**.5
- The CANCEL statement closes all open files.
- When there is no next executable statement in a called program, an implicit EXIT PROGRAM statement is executed.
- Within the VARYING ... AFTER phrase of the PERFORM statement, *identifier-2* is augmented before *identifier-5* is set. In HP COBOL II, *identifier-5* was set before *identifier-2* was augmented.
- New I-O status codes have been specified. Refer to Table C-1 for more information.
- STOP RUN closes all files.

Table C-1 describes the difference between the I/O status codes received using the ANSI74 and ANSI85 entry points. It shows the equivalent ANSI74 codes for the new ANSI85 codes and any difference in execution for these codes as well as codes whose definition has changed.

Table C-1. New I-O Status Codes

ANSI85 Status Code	ANSI74 Status Code	New, Old, or Changed	Execution Differences between the Entry Points
00		Old	
02		Old	
04	00	New	None
05	00	New	None
07	00	New	None
10		Old	
14	00	New	Using ANSI74, the READ successfully executes. Using ANSI85, the READ fails because the value of the data item is greater than the PICTURE that describes the key.
21		Old	
22		Old	
23		Old	
24	24	Changed	In addition to the old causes for 24, ANSI85 returns this code when the value of the data item is greater than the PICTURE that describes it.
30		Old	
34		Old	
35	9x	New	The file is not created using ANSI85, it is created in ANSI74 for an OPEN with the I-O or EXTEND phrase.
37	00	New	ANSI74 opens the file and the program continues to execute, however the program might abort later for another reason. ANSI85 does not open the file and a permanent error condition exists for that file.

Table C-1. New I-O Status Codes (continued)

ANSI85 Status Code	ANSI74 Status Code	New, Old, or Changed	Execution Differences between the Entry Points
38	00	New	Using ANSI74, the OPEN fails and a message is printed even though an error status code is not returned. Using ANSI85, the OPEN fails and a permanent error condition exists for that file.
39	00	New	ANSI74 entry point successfully opens the file and may print an error message. ANSI85 treats it as a permanent error.
41	9x	New	None
42	9x	New	None
43	9x or 00	New	No difference for a REWRITE statement. ANSI74 entry point executes a DELETE statement successfully.
44	00	New	ANSI74 executes the statement successfully. In ANSI85 the statement is unsuccessful due to a logic error.
46	10	New	ANSI74 entry point continues to return AT END condition or read error condition.
47	9x or 00	New	Using ANSI74, if the file is not open then a 9x status is returned. If the file is opened with the wrong mode, a 9x status is sometimes returned; sometimes it continues to execute correctly.
48	9x or 00	New	Using ANSI74, if the file is not open then a 9x status is returned. If the file is opened with the wrong mode, a 9x status is sometimes returned; sometimes it continues to execute correctly.
49	9x or 00	New	Using ANSI74, if the file is not open then a 9x status is returned. If the file is opened with the wrong mode, a 9x status is sometimes returned; sometimes it continues to execute correctly.

Obsolete Features

Obsolete features are elements of the language that are now part of the 1985 ANSI COBOL standard, but will be deleted from the full standard after ANSI COBOL'85. The ANSI85 entry point will continue to support these features.

The following items have been placed in the obsolete category:

- AUTHOR, DATE-WRITTEN, DATE-COMPILED, and SECURITY paragraphs in the IDENTIFICATION DIVISION.
- MEMORY-SIZE clause of the OBJECT-COMPUTER paragraph.
- RERUN clause of the I-O-CONTROL paragraph.
- LABEL RECORDS clause of the file description entry.
- VALUE OF clause of the file description entry.
- DATA RECORDS clause of the file description entry.
- ALTER statement.
- ENTER statement.
- REVERSED phrase of the OPEN statement.
- STOP literal statement.
- Segmentation module.
- Debug module.

D

ASCII and EBCDIC Character Sets

This appendix presents a table showing the ASCII (American Standard Code for Information Interchange) and EBCDIC (Extended Binary Coded Decimal Interchange Code) character sets.

How to Use This Table

1. The table is sorted by character code, each code being represented by its decimal, octal, and hexadecimal equivalent.
2. Each row of the table gives the ASCII and EBCDIC meaning of the character code.

The following examples show ways of using the table:

Example 1

Suppose you want to determine the ASCII code for the $ character. Scan down the ASCII column until you locate $, then look right on that row to find the character codes 36 (decimal), 044 (octal), and 24 (hexadecimal). This is the code used by an ASCII device (for example, a terminal, printer, or computer) to represent the $ character.

Example 2

Suppose you want to determine the EBCDIC and ASCII codes for the hexadecimal character code 5B. First locate 5B in the Hexadecimal Value column. Then move left on that row to the EBCDIC column which shows a $ and further left to the ASCII column which shows a [. These are the EBCDIC and ASCII characters represented by the hexadecimal character code 5B.

Table D-1. ASCII and EBCDIC Character Sets

ASCII	EBCDIC	Character Code Values		
Control/ Graphic	Control/ Graphic	Decimal Value	Octal Value	Hex Value
NUL	NUL	0	000	00
SOH	SOH	1	001	01
STX	STX	2	002	02
ETX	ETX	3	003	03
EOT	PF	4	004	04
ENQ	HT	5	005	05
ACK	LC	6	006	06
BEL	DEL	7	007	07
BS		8	010	08
HT		9	011	09
LF	SMM	10	012	0A
VT	SI	11	013	0B
FF	FF	12	014	0C
CR	CR	13	015	0D
SO	SO	14	016	0E
SI	SI	15	017	0F
DLE	DLE	16	020	10
DC1	DC1	17	021	11
DC2	DC2	18	022	12
DC3	TM	19	023	13
DC4	RES	20	024	14
NAK	NL	21	025	15
SYN	BS	22	026	16
ETB	IL	23	027	17
CAN	CAN	24	030	18
EM	EM	25	031	19
SUB	CC	26	032	1A
ESC	CU1	27	033	1B
FS	IFS	28	034	1C
GS	IGS	29	035	1D
RS	IRS	30	036	1E
US	IUS	31	037	1F

ASCII	EBCDIC	Character Code Values		
Control/ Graphic	Control/ Graphic	Decimal Value	Octal Value	Hex Value
SP	DS	32	040	20
!	SOS	33	041	21
”	FS	34	042	22
#		35	043	23
$	BYP	36	044	24
%	LF	37	045	25
&	ETB	38	046	26
’	ESC	39	047	27
(		40	050	28
)		41	051	29
*	SM	42	052	2A
+	CU2	43	053	2B
,		44	054	2C
-	ENQ	45	055	2D
.	ACK	46	056	2E
/	BEL	47	057	2F
0		48	060	30
1		49	061	31
2	SYN	50	062	32
3		51	063	33
4	PN	52	064	34
5	RS	53	065	35
6	UC	54	066	36
7	EOT	55	067	37
8		56	070	38
9		57	071	39
:		58	072	3A
;	CU3	59	073	3B
<	DC4	60	074	3C
=	NAK	61	075	3D
>		62	076	3E
?	SUB	63	077	3F

Table D-1. ASCII and EBCDIC Character Sets (continued)

ASCII	EBCDIC	Character Code Values			ASCII	EBCDIC	Character Code Values		
Control/ Graphic	Control/ Graphic	Decimal Value	Octal Value	Hex Value	Control/ Graphic	Control/ Graphic	Decimal Value	Octal Value	Hex Value
@	SP	64	100	40	‘	-	96	140	60
A		65	101	41	a	\	97	141	61
B		66	102	42	b		98	142	62
C		67	103	43	c		99	143	63
D		68	104	44	d		100	144	64
E		69	105	45	e		101	145	65
F		70	106	46	f		102	146	66
G		71	107	47	g		103	147	67
H		72	110	48	h		104	150	68
I		73	111	49	i		105	151	69
J	¢	74	112	4A	j	|	106	152	6A
K	.	75	113	4B	k	,	107	153	6B
L	<	76	114	4C	l	%	108	154	6C
M	(	77	115	4D	m	_	109	155	6D
N	+	78	116	4E	n	>	110	156	6E
O		79	117	4F	o	?	111	157	6F
P	&	80	120	50	p		112	160	70
Q		81	121	51	q		113	161	71
R		82	122	52	r		114	162	72
S		83	123	53	s		115	163	73
T		84	124	54	t		116	164	74
U		85	125	55	u		117	165	75
V		86	126	56	v		118	166	76
W		87	127	57	w		119	167	77
X		88	130	58	x		120	170	78
Y		89	131	59	y	‘	121	171	79
Z	!	90	132	5A	z	:	122	172	7A
[	$	91	133	5B	{	#	123	173	7B
\	*	92	134	5C	|	@	124	174	7C
]	)	93	135	5D	}	’	125	175	7D
^	;	94	136	5E	~	=	126	176	7E
_	^	95	137	5F	DEL	”	127	177	7F

Table D-1. ASCII and EBCDIC Character Sets (continued)

ASCII	EBCDIC	Character Code Values			ASCII	EBCDIC	Character Code Values		
Control/ Graphic	Control/ Graphic	Decimal Value	Octal Value	Hex Value	Control/ Graphic	Control/ Graphic	Decimal Value	Octal Value	Hex Value
		128	200	80			160	240	A0
	a	129	201	81		~	161	241	A1
	b	130	202	82		s	162	242	A2
	c	131	203	83		t	163	243	A3
	d	132	204	84		u	164	244	A4
	e	133	205	85		v	165	245	A5
	f	134	206	86		w	166	246	A6
	g	135	207	87		x	167	247	A7
	h	136	210	88		y	168	250	A8
	i	137	211	89		z	169	251	A9
		138	212	8A			170	252	AA
		139	213	8B			171	253	AB
		140	214	8C			172	254	AC
		141	215	8D			173	255	AD
		142	216	8E			174	256	AE
		143	217	8F			175	257	AF
		144	220	90			176	260	B0
	j	145	221	91			177	261	B1
	k	146	222	92			178	262	B2
	l	147	223	93			179	263	B3
	m	148	224	94			180	264	B4
	n	149	225	95			181	265	B5
	o	150	226	96			182	266	B6
	p	151	227	97			183	267	B7
	r	152	230	98			184	270	B8
		153	231	99			185	271	B9
		154	232	9A			186	272	BA
		155	233	9B			187	273	BB
		156	234	9C			188	274	BC
		157	235	9D			189	275	BD
		158	236	9E			190	276	BE
		159	237	9F			191	277	BF

Table D-1. ASCII and EBCDIC Character Sets (continued)

ASCII	EBCDIC	Character Code Values			ASCII	EBCDIC	Character Code Values		
Control/ Graphic	Control/ Graphic	Decimal Value	Octal Value	Hex Value	Control/ Graphic	Control/ Graphic	Decimal Value	Octal Value	Hex Value
	{	192	300	C0		\	224	340	E0
	A	193	301	C1			225	341	E1
	B	194	302	C2		S	226	342	E2
	C	195	303	C3		T	227	343	E3
	D	196	304	C4		U	228	344	E4
	E	197	305	C5		V	229	345	E5
	F	198	306	C6		W	230	346	E6
	G	199	307	C7		X	231	347	E7
	H	200	310	C8		Y	232	350	E8
	I	201	311	C9		Z	233	351	E9
		202	312	CA			234	352	EA
		203	313	CB			235	353	EB
		204	314	CC			236	354	EC
		205	315	CD			237	355	ED
		206	316	CE			238	356	EE
		207	317	CF			239	357	EF
	}	208	320	D0		0	240	360	F0
	J	209	321	D1		1	241	361	F1
	K	210	322	D2		2	242	362	F2
	L	211	323	D3		3	243	363	F3
	M	212	324	D4		4	244	364	F4
	N	213	325	D5		5	245	365	F5
	O	214	326	D6		6	246	366	F6
	P	215	327	D7		7	247	367	F7
	Q	216	330	D8		8	248	370	F8
	R	217	331	D9		9	249	371	F9
		218	332	DA			250	372	FA
		219	333	DB			251	373	FB
		220	334	DC			252	374	FC
		221	335	DD			253	375	FD
		222	336	DE			254	376	FE
		223	337	DF			255	377	FF

E

COBOL Glossary

The terms in this appendix are defined in accordance with their meaning as used in this document describing COBOL and may not have the same meaning for other languages.

These definitions are also intended to be either reference material or introductory material to be reviewed prior to reading the detailed language specifications. For this reason, these definitions are, in most instances, brief and do not include detailed syntactical rules.

Definitions

Abbreviated Combined Relation Condition. The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

Access Mode. The manner in which records are to be operated upon within a file.

Actual Decimal Point. The physical representation, using either of the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

Alphabet Name. A user-defined word, in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION, that assigns a name to a specific character set and/or collating sequence.

Alphabetic Character. A character that belongs to the following set of letters: A,B,C, ... X,Y,Z, a,b,c ... x,y,z and the space character.

Alphanumeric Character. Any character in the computer's character.

Alphanumeric Function. A function whose value is composed of a string of one or more characters from the computer's character set.

Alternate Record Key. A key, other than the prime record key, whose contents identify a record within an indexed file.

Argument See Parameter.

Arithmetic Expression. An arithmetic expression can be an identifier or a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

Arithmetic Operation. The process caused by the execution of an arithmetic statement, or the evaluation of an arithmetic expression, that results in a mathematically correct solution to the arguments presented.

Arithmetic Operator. A single character, or a fixed two-character combination, that belongs to the following set:

Character	Meaning
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Arithmetic Statement. A statement that causes an arithmetic operation to be executed. The arithmetic statements are the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements.

Ascending Key. A key upon the values of which data is ordered starting with the lowest value of key up to the highest value of key in accordance with the rules for comparing data items.

Assumed Decimal Point. A decimal point position which 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:

1. During the execution of a READ statement for a sequentially accessed file.
2. During the execution of a RETURN statement, when no next logical record exists for the associated sort or merge file.
3. During the execution of a SEARCH statement, when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

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 contained within the block or that overlap the block. The term is synonymous with physical record.

Body Group. Generic name for a report group of TYPE DETAIL, CONTROL HEADING, or CONTROL FOOTING. This term has no meaning in HP COBOL II.

Bottom Margin. An empty area which follows the page body.

Called Program. A program which is the object of a CALL statement combined at object time with the calling program to produce a run unit.

Calling Program. A program which executes a CALL to another program.

Cd-Name. A user-defined word that names an MCS interface area described in a communication description entry within the COMMUNICATION SECTION of the DATA DIVISION. This term has no meaning in HP COBOL II.

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 described as usage is DISPLAY. Further characteristics of the physical storage are defined by the implementor.

Character String. A sequence of contiguous characters which for COBOL word, a literal, a PICTURE character string, or a comment entry.

Class Condition. The proposition, for which a truth value can be determined, that the content of an item is wholly alphabetic or is wholly numeric, or consists exclusively of those characters listed in the definition of the class name.

Class Name. A user-defined word defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION that assigns a name to the proposition for which a truth value can be defined that the content of a data item consists exclusively of those characters listed in the definition of the class name.

Clause. A clause is an ordered set of consecutive COBOL character strings whose purpose is to specify an attribute of an entry.

COBOL Character Set. The complete COBOL character set consists 79 characters listed below:

Character	Meaning
0,1, . . . ,9	digit
A,B, . . . ,a,b . . . ,z	letter
	space
+	plus sign
-	minus sign (hyphen)
*	asterisk
/	stroke (virgule, slash)
=	equal sign
$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point)
”	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol
:	colon

Note The HP extended COBOL character set includes the @ (at), \ (back slash), and % (percent) characters.

COBOL Word. A character string of not more than 30 characters which forms a user-defined word, a system name, or a reserved word.

Collating Sequence. The sequence in which the characters that are acceptable in a computer are ordered for purposes of sorting, merging, and comparing.

Column. A character position within a print line. The columns are numbered from 1, by 1, starting at the leftmost character position of the print line and extending to the rightmost position of the print line.

Combined Condition. A condition that is the result of connecting two or more conditions with the 'AND' or the 'OR' logical operator.

Comment Entry. An entry in the IDENTIFICATION DIVISION that may be any combination of characters from the computer character set.

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 stroke (/) 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.

Common Program. A program which, despite being directly contained within another program, may be called from any program directly or indirectly contained in that other program.

Compile Time. The time at which a COBOL source program is translated, by a COBOL compiler, to a COBOL object program.

Compiler Directing Statement. A statement, beginning with a compiler directing verb, that causes the compiler to take a specific action during compilation. The compiler directing statements are the COPY, ENTER, `REPLACE`, and USE statements.

Complex Condition. A condition in which one or more logical operators act upon one or more conditions.

Computer Name. A system name that identifies the computer upon which the program is to be compiled or run.

Concatenated Programs. A source file that contains more than one COBOL program appended to the previous program, of which at least one program is not a nested program. The previous program must be terminated with `END PROGRAM` statement. In the following example, PROGRAM-1 and PROGRAM-2 are concatenated programs. PROGRAM-3 is a nested program.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGRAM-1.
       ⋮
END PROGRAM PROGRAM-1.

IDENTIFICATION DIVISION.
PROGRAM-ID. PROGRAM-2.
       ⋮
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGRAM-3.
END PROGRAM PROGRAM-3.
       ⋮
END PROGRAM PROGRAM-2.
```

Condition. A status of a program at execution time for which a truth value can be determined. Where the term 'condition' (condition-1, condition-2, ...) appears in these language specifications in or in reference to 'condition' (condition-1, condition-2, ...) of a general format, it is a conditional expression consisting of either a simple condition optionally parenthesized, or a combined condition consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses, for which a truth value can be determined.

Condition Name. A user-defined word that assigns a name to a subset of values that a conditional variable may assume; or a user-defined word assigned to a status of an implementor defined switch or device. When 'condition name' is used in the general formats, it represents a unique data item reference consisting of a syntactically correct combination of a condition name, together with qualifiers and subscripts, as required for uniqueness of reference.

Condition Name Condition. The proposition, for which a truth value can be determined, that the value of a conditional variable is a member of the sets of values attributed to a condition name associated with the conditional variable.

Conditional Expression. A simple condition or a complex condition specified in an `EVALUATE`, IF, PERFORM, or SEARCH statement.

Conditional Phrase. A conditional phrase specifies the action to be taken upon determination of the truth value of a condition resulting from the execution of a conditional statement.

Conditional Statement. A conditional statement specifies that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value.

Conditional Variable. A data item one or more values of which has a condition name assigned to it.

CONFIGURATION SECTION. A section of the ENVIRONMENT DIVISION that describes overall specification of source and object programs.

Containing and Contained Programs. Programs that are nested. For example, in the program below, PROGRAM-1 and PROGRAM-2 are containing programs because PROGRAM-1 contains PROGRAM-2 and PROGRAM-3, and PROGRAM-2 contains PROGRAM-3. PROGRAM-2 and PROGRAM-3 are contained programs.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGRAM-1.
      ⋮
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGRAM-2.
      ⋮
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGRAM-3.
     ⋮
END PROGRAM PROGRAM-3.
END PROGRAM PROGRAM-2.
END PROGRAM PROGRAM-1.
```

Contiguous Items. Items that are described by consecutive entries in the DATA DIVISION, and that bear a definite hierarchic relationship to each other.

Counter. A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

Currency Sign. The character '$' of the COBOL character set.

Currency Symbol. The character defined by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. If no CURRENCY SIGN clause is present in a COBOL source program, the currency symbol is identical to the currency sign.

Current Record. The record which is available in the record are associated with the file.

Current Record Pointer. A conceptual entity that is used in the selection of the next record.

Data Clause. A clause that appears in a data description entry the DATA DIVISION and provides information describing a particular attribute of a data item.

Data Description Entry. An entry in the DATA DIVISION that is composed of a level number followed by a data name, if required, and then followed by a set of data clauses, as required.

Data Item. A character or a set of contiguous characters (excluding in either case literals) defined as a unit of data by the COBOL

Data Name. A user-defined word that names a data item described in a data description entry in the DATA DIVISION. When used in the general formats, 'data name' represents a word which can neither be subscripted, indexed, nor qualified unless specifically permitted by the rules for that format.

Debugging Line. A debugging line is any line with 'D' in the indicator area of the line.

Debugging Section. A debugging section is a section that contains USE FOR DEBUGGING statement.

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 compiler directing sentence, followed by a set of zero, one or more associated paragraphs.

Declarative Sentence. A compiler-directing sentence consisting of a single USE statement terminated by the separator period.

`De-Edit.` The logical removal of all editing characters from a numeric-edited data item in order to determine that item's unedited numeric value.

`Delimited Scope Statement.` Any statement which includes its explicit scope terminator.

Delimiter. A character or a sequence of contiguous characters that identifies the end of a string of characters and separates that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

Descending Key. A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

Digit Position. A digit position is the amount of physical storage required to store a single digit. This amount may vary depending on the usage of the data item describing the digit position. Further characteristics of the physical storage are defined by the implementor.

Division. A set of zero, one or more sections or paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. There are four (4) divisions in a COBOL program: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE.

Division Header. A combination of words followed by a period an space that indicates the beginning of a division. The division headers are:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION [ USING data-name-1 [data-name-2]... ] .
```

Dynamic Access. An access mode in which specific logical 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.

Editing Character. A single character or a fixed two -character combination belonging to the following set:

Character	Meaning
B	space
0	zero
+	plus
-	minus
CR	credit
DB	debit
Z	zero suppress
*	check protect
$	currency-sign
,	comma (decimal-point)
.	period (decimal-point)
/	stroke (virgule, slash)

Elementary Item. A data item that is described as not being further logically subdivided.

End of PROCEDURE DIVISION. The physical position in a COBOL source program after which no further procedures appear.

Entry. Any descriptive set of consecutive clauses terminated by a period and written in the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, or DATA DIVISION of a COBOL source program.

Environment Clause. A clause that appears as part of an ENVIRONMENT DIVISION entry.

Execution Time. (See Object Time)

`Explicit Scope Terminator.` A reserved word that terminates the scope of a particular PROCEDURE DIVISION statement.

Expression. An arithmetic or conditional expression.

Extend Mode. The state of a file after execution of an OPEN statement, with the EXTEND phrase specified for that file, and before the execution of a CLOSE statement for that file.

External Name. An external name is a name that is visible to other programs. It is visible "outside" your program. External names are generated by the compiler using names in the source program. The compiler generates external names for ENTRY literals, PROGRAM-ID literals, and program names specified in CALL and CANCEL statements.

External Naming Convention. These are the rules used by the compiler to convert the name in the source program to the external name.

External Switch. A hardware or software device, defined and name by the implementor, which is used to indicate that one of two alternate states exists.

Figurative Constant. A compiler-generated value referenced through the use of certain reserved words.

File. A collection of records.

File Clause. A clause that appears as part of any of the follow DATA DIVISION entries:

File description (FD)
Sort-merge file Description (SD)
Communication description (CD) Not used in HP COBOL II.

File Connector. A storage area which contains information about a file and is used as the linkage between a file name and a physical file and between a file name and its associated record area.

FILE-CONTROL. The name of an ENVIRONMENT DIVISION paragraph in the data files for a given source program are declared.

File Control Entry. A SELECT clause and all its subordinate clauses that declare the relevant physical attributes of a file.

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 description entry or a sort-merge file description entry within the FILE SECTION of the DATA DIVISION.

File Organization. The permanent logical file structure establish at the time that a file is created.

`File Position Indicator.` A conceptual entity that contains the value of the current key within the key of reference for an indexed file, or the record number of the current record for a sequential file, or the relative record number of the current record for a relative file, or indicates that no next logical record exists, or that the number of significant digits in the relative record number is larger than the size of the relative key data item, or that an optional input file is not present, or that the at end condition already exists, or that no valid next record has been established.

FILE SECTION. The section of the DATA DIVISION that contains file description entries and sort-merge file description entries together with their associated record descriptions.

`Fixed File Attributes.` Information about a file which is established when a file is created and cannot subsequently be changed during the existence of the file. These attributes include the organization of the file (sequential, relative, or indexed), the prime record key, the alternate record keys, the code-set, the minimum and maximum record size, the record type (fixed or variable), the collating sequence of the keys for indexed files, the blocking factor, the padding character, and the record delimiter.

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.

Footing Area. The position of the page body adjacent to the bottom margin.

Format. A specific arrangement of a set of data.

Function. A temporary data item whose value is determined at the time the function is referenced during the execution of a statement.

Function-Identifier. A syntactically correct combination of character-strings and separators that references a function. The data item represented by a function is uniquely identified by a function name with its arguments, if any. A function-identifier may include a reference-modifier. A function-identifier that references an alphanumeric function may be specified anywhere that an identifier may be specified, subject to certain restrictions. A function-identifier that references an integer or numeric function may be referenced anywhere an arithmetic expression may be specified.

Group Item. A named contiguous set of elementary or group items.

High Order End. The leftmost character of a string of character.

I-O-CONTROL. The name of an ENVIRONMENT DIVISION paragraph in which object program requirements for specific input-output techniques, rerun points, sharing of same areas by several data files, and multiple file storage on a single input-output device are specified.

I-O-CONTROL Entry. An entry in the I-O-CONTROL paragraph of the ENVIRONMENT DIVISION which contains clauses which provide information required for the transmission and handling of data on named files during the execution of a program.

I-O Mode. The state of a file after execution of an OPEN statement with the I-O phrase specified, for that file and before the execution of a CLOSE statement for that file.

I-O Status. A conceptual entity which contains the two-character value indicating the resulting status of an input-output operation. This value is made available to the program through the use of the FILE STATUS clause in the file control entry for the file.

Identifier. A data name, followed as required by the syntactically correct combination of qualifiers, subscripts, indices, and reference-modifiers necessary to make unique reference to a data item. When referencing a data item that is a function, a function-identifier is used. The rules for identifier associated with general formats may, however, specifically prohibit reference to functions, qualification, subscription, or reference modification.

Imperative Statement. A statement that begins with an imperative and specifies an unconditional action to be taken. An imperative-statement may consist of a sequence of imperative statements.

Implementor-Name. A system-name that refers to a particular feature available on that implementor's computing system.

Implicit Scope Terminator. A separator period which terminates the scope of any preceding unterminated statement, or a phrase of a statement, which by its occurrence, indicates the end of the scope of any statement contained with the preceding phrase.

Index. A computer storage position or register, the contents of represent the identification of a particular element in a table.

Index Data Item. A data item in which the value associated with an index name can be stored in a form specified by the implementor.

Index Name. A user-defined word that names an index associated a specific table.

Indexed Data Name. An identifier that is composed of a data name followed by one or more index names enclosed in parentheses.

Indexed File. A file with indexed organization.

Indexed Organization. The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

Initial Program. A program that is placed into an state every time the program is called in a run unit.

Initial State. The state of a program when it is first called in a run unit.

Input File. A file that is opened in the input mode.

Input Mode. The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement for that file.

Input-Output File. A file that is opened in the I-O mode.

INPUT-OUTPUT SECTION. The section of the ENVIRONMENT DIVISION that names the files and the external media required by an object program and which provides information required for transmission and handling of data during execution of the object program.

Input-Output Statement. A statement that causes files to be processed by performing operations upon individual records or upon the file as a unit. The input-output statements are: ACCEPT (with the identifier phrase), CLOSE, DELETE, DISABLE, DISPLAY, ENABLE, OPEN, PURGE, READ, RECEIVE, REWRITE, SEND, SET (with the TO ON or TO OFF phrase), START, and WRITE.

Input Procedure. A set of statements that is executed each time a record is released to the sort-file.

Integer. A numeric literal or a numeric data item that does not include any character positions to the right of the assumed decimal-point. Or, a numeric function whose definition provides that all digits to the right of the decimal point are zero in the returned value for any possible evaluation of the function. Where the term 'integer' appears in general formats, integer must not be a numeric data item, must not be signed, nor must not be zero unless explicitly allowed by the rules of the format.

Integer Function. A function of the category numeric whose definition provides that all digits to the right of the decimal point are zero in the returned value for any possible evaluation of the function.

Intra-Record Data Structure. The entire collection of groups and elementary data items from a logical record which is defined by a contiguous subset of the data description entries which describe that record. These data description entries include all entries whose level number is greater than the level number of the first data description entry describing the intra-record data structure.

Invalid Key Condition. A condition, at object time, caused when specific value of the key associated with an indexed or relative file is determined to be invalid.

Key. A data item which identifies the location of a record, or of data items which serve to identify the ordering of data.

Key of Reference. The key, either prime or alternate, currently being used to access records within an indexed file.

Keyword. A reserved word or function-name whose presence is required when the format in which the word appears is used in a source program.

Language Name. A system name that specifies a particular programming language.

Letter. A character belonging to one of the following two sets: (1) uppercase letters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z; (2) lowercase letters: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z.

77 Level Description Entry. A data description entry that describes a noncontiguous data item with the level number 77.

Level Indicator. Two alphabetic characters that identify a specific file type or a position in hierarchy.

Level Number. A user-defined word which indicates the position data item in the hierarchical structure of a logical record or which indicates special properties of a data description entry. A level number is expressed as a one 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 number 66, 77, and 88 identify special properties of a data description entry.

Library Name. A user-defined word that names a COBOL library that is to be used by the compiler for a given source program compilation.

Library Text. A sequence of character strings and/or separators COBOL library.

LINAGE-COUNTER. A special register whose value points to the current position within the page body.

Line. (See Report Line) Not used in HP COBOL II.

Line Number. An integer that denotes the vertical position of a line on a page. Not used for HP COBOL II.

LINKAGE SECTION. The section in the DATA DIVISION of the called program that describes data items available from the calling program. These data items may be referred to by both the calling and called program.

Literal. A character string whose value is implied by the order set of characters comprising the string.

Logical Operator. One of the reserved words AND, OR, or NOT. I formation of a condition, both or either of AND and OR can be used as logical connectives. NOT can be used for logical negation.

Logical Page. A conceptual entity consisting of the top margin the page body, and the bottom margin.

Logical Record. The most inclusive data item. The level number a record is 01. (See Report Writer Logical Record)

Low Order End. The rightmost character of a string of characters.

Mass Storage. A storage medium on which data may be organized a maintained in both a sequential and nonsequential manner.

Mass Storage Control System (MSCS). An input-output control system that directs, or controls, the processing of mass storage files.

Mass Storage File. A collection of records that is assigned to mass storage medium.

Merge File. A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

Message. Data associated with an end of message indicator or an of group indicator. Not used in HP COBOL II. (See Message Indicators)

Mnemonic Name. A user-defined word that is associated in the ENVIRONMENT DIVISION with a specified implementor name.

MSCS. (See Mass Storage Control System)

Native Character Set. The ASCII character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

Native Collating Sequence. The ASCII collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

Negated Combined Condition. The 'NOT' logical operator immediately followed by a parenthesized combined condition.

Negated Simple Condition. The 'NOT' logical operator immediately followed by a simple condition.

Nested Programs. A COBOL program that contains another COBOL program. In the following example, TEST1 is a nested program. TEST contains TEST1.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TEST.
     ⋮
IDENTIFICATION DIVISION.
PROGRAM-ID. TEST1.
END PROGRAM TEST1.
     ⋮
END PROGRAM TEST.
```

See "Containing and Contained Programs" in this glossary.

Next Executable Sentence. The next sentence to which control will be transferred after execution of the current statement is complete.

Next Executable Statement. The next statement to which control will be transferred after execution of the current statement is complete.

Next Record. The record which logically follows the current record a file.

Noncontiguous Items. Elementary data items, in the WORKING-STORAGE and LINKAGE SECTIONs, which bear no hierarchic relationship to other data items.

Nonnumeric Item. A data item whose description permits its contents to be composed of any combination of characters taken from the computer's character set. Certain categories of nonnumeric items may be formed from more restricted character sets.

Nonnumeric Literal. A character-string bounded by quotation marks. The string of characters may include any character in the computer's character set. To represent a single quotation mark character within a nonnumeric literal, two contiguous quotation marks must be used.

Numeric Character. A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Numeric Function. A function of the class and category numeric but which for some possible evaluation does not satisfy the requirements of an integer function.

Numeric Item. A data item whose description restricts its contents a value represented by characters chosen from the digits '0' through '9'; if signed, the item may also contains a '+', '-', or an operational sign.

Numeric Literal. A literal composed of one or more numeric characters that also may contain either a decimal point, or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

OBJECT-COMPUTER. The name of an ENVIRONMENT DIVISION paragraph the computer environment, within which the object program is executed, is described.

Object Computer Entry. An entry in the OBJECT-COMPUTER paragraph of the ENVIRONMENT DIVISION which contains clauses which describe the computer environment in which the object program is to be executed.

Object of Entry. A set of operands and reserved words, within a DIVISION entry, that immediately follows the subject of the entry.

Object Program. A set or group of executable machine language instructions and other material designed to interact with data to provide problem solution. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program. Where there is no danger of ambiguity, the word 'program' alone may be used in place of the phrase 'object program'.

Object Time. The time at which an object program is executed. The term is synonymous with execution time.

Obsolete Element. A COBOL language element in the 1985 revision of ANSI standard COBOL that is to be deleted from the next revision of standard COBOL.

Octal Literal. A literal composed of a "%" followed by between 11 octal digits with a maximum value of 37777777777. No algebraic sign nor decimal point may appear in the literal. Octal Literals may appear as numeric literals and in VALUE clauses.

Open Mode. The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O or EXTEND.

Operand. Whereas the general definition of operand is 'that component which is operated upon', for the purposes of this publication, any lowercase word (or words) that appears in a statement or entry format may be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

Operational Sign. An algebraic sign, associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

Optional File. A file which is declared as being not necessarily present each time the object program is executed. The object program causes an interrogation for the presence or absence of the file.

Optional Word. A reserved word that is included in a specific format only to improve the readability of the language and whose presence is optional when the format in which the word appears is used in a source program.

Output File. A file that is opened in either the output mode or extend mode.

Output Mode. The state of a file after execution of an OPEN statement with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement for that file.

Output Procedure. A set of statements to which control is given during execution of a SORT statement after the function is completed, or during execution of a MERGE statement after the merge function has selected the next record in merged order.

Page. A vertical division of a report representing a physical separation of report data, the separation being based on internal reporting requirements and/or external characteristics of the reporting medium.

Page Body. That part of the logical page in which lines can be written and/or spaced.

Page Footing. A report group that is presented at the end of a report page as determined by the Report Writer Control System.

Page Heading. A report group that is presented at the beginning a report page and determined by the Report Writer Control System.

Paragraph. In the PROCEDURE DIVISION, a paragraph name followed by a period and a space and by zero, one, or more sentences. In the IDENTIFICATION and ENVIRONMENT divisions, 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 divisions. The permissible paragraph headers are:

In the IDENTIFICATION DIVISION:

```
PROGRAM-ID.
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.
```

In the ENVIRONMENT DIVISION:

```
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
FILE-CONTROL.
I-O CONTROL.
```

Paragraph Name. A user-defined word that identifies and begins paragraph in the PROCEDURE DIVISION.

Parameter. An identifier, a literal, or an arithmetic expression that specifies a value to be used in the evaluation of a function.

Phrase. A phrase is an ordered set of one or more consecutive COBOL character strings that form a portion of a COBOL procedural statement or of a COBOL clause.

Physical Page. A device dependent concept defined by the implementor.

Physical Record. (See Block)

Prime Record Key. A key whose contents uniquely identify a record within an indexed file.

Printable Group. A report group that contains at least one print. Not used in HP COBOL II.

Printable Item. A data item, the extent and contents of which a specified by an elementary report entry. This elementary report entry contains a COLUMN NUMBER clause, a PICTURE clause, and a SOURCE, SUM or VALUE clause.

Procedure. A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the PROCEDURE DIVISION.

Procedure Branching Statement. A statement that causes the explicit transfer of control to a statement other than the next executable statement in the sequence in which the statements are written in the source program. The procedure branching statements are: ALTER, CALL, EXIT, EXIT PROGRAM, GO TO, MERGE (with the OUTPUT PROCEDURE phrase), PERFORM and SORT (with the INPUT PROCEDURE or OUTPUT PROCEDURE phrase).

Procedure Name. A user-defined word which is used to name a paragraph or section in the PROCEDURE DIVISION. It consists of a paragraph name (which may be qualified), or a section name.

Program Name. A user-defined word that identifies a COBOL source program.

Program Name Entry. An entry in the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION which contains clauses that specify the program name and assign selected program attributes to the program.

Pseudo-Text. A sequence of character strings and/or separators bounded by, but not including, pseudo-text delimiters.

Pseudo-Text Delimiter. Two contiguous equal sign (=) characters to delimit pseudo-text.

Punctuation Character. A character that belongs to the following set:

Character	Meaning
,	comma
;	semicolon
.	period
"	quotation mark
(	left parenthesis
)	right parenthesis
	space
=	equal sign
:	colon

Qualified Data Name. An identifier that is composed of a data name followed by one or more sets of either of the connectives OF and IN, followed by a data name qualifier.

Qualifier.

1. A data name which is used in a reference together with another data name at a lower level in the same hierarchy.
2. A section name which is used in a reference together with a paragraph name specified in that section.
3. A library name which is used in a reference together with a text name associated with that library.

Queue. A logical collection of messages awaiting transmission or processing. Not used in HP COBOL II.

Queue Name. A symbolic name that indicates to the MCS the logic path by which a message or a portion of a completed message may be accessible in a queue. Not used in HP COBOL II.

Random access. An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from or placed into a relative or indexed file.

Record. (See Logical Record)

Record Area. A storage area allocated for the purpose of processing the record described in a record description entry in the FILE SECTION.

Record Description. (See Record Description Entry)

Record Description Entry. The total set of data description entries associated with a particular record.

Record Key. A key, either the prime record key or an alternate record key, whose contents identify a record within an indexed file.

Record Name. A user-defined word that names a record described in a record description entry in the DATA DIVISION.

Record Number. The ordinal number of a record in the file whose organization is sequential.

Reel. A discrete portion of a storage medium, the dimensions of which are determined by each implementor, that contains part of a file, all of a file, or any number of files. The term is synonymous with unit and volume.

Reference Format. A format that provides a standard method for describing COBOL source programs.

`Reference-Modifier.` A syntactically correct combination of character-strings and separators that defines a unique data item. It includes a delimiting left parenthesis separator, the leftmost character position, a colon separator, optionally a length, and a delimiting right parenthesis separator.

Relation. (See Relational Operator)

Relation Character. A character that belongs to the following set:

Character	Meaning
>	greater than
<	less than
=	equal to

Relation Condition. The proposition, for which a truth value can be determined, that the value of an arithmetic expression or data item has a specific relationship to the value of another arithmetic expression or data item. (See Relational Operator)

Relational Operator. A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meaning are:

Relational Operators	Meaning
IS (NOT) GREATER THAN IS (NOT) >	Greater than or not greater than
IS (NOT) LESS THAN IS (NOT) <	Less than or not less than
IS (NOT) EQUAL TO IS (NOT) =	Equal to or not equal to
`IS GREATER THAN OR EQUAL TO` `IS >=`	Greater than or equal to
`IS LESS THAN OR EQUAL TO` `IS <=`	Less than or equal to

Relative File. A file with relative organization.

Relative Key. A key whose contents identify a logical record in relative file.

Relative Organization. The permanent logical file structure in each record is uniquely identified by an integer value greater than zero, which specifies the record's logical ordinal position in the file.

Relative Record Number. The ordinal number of a record in a file whose organization is relative. This number is treated as a numeric literal which is an integer.

Reserved Word. A COBOL word specified in the list of words that may be used in COBOL source programs, but must not appear in the programs as user-defined words or system names.

Routine Name. A user-defined word that identifies a procedure written in a language other than COBOL.

Run Unit. A set of one or more object programs which function, object time, as a unit to provide problem solutions.

Static Storage. Static storage is storage that is allocated and initialized once at the beginning of the run unit. It remains fixed throughout the run unit. Working-storage data of subroutines with static storage retain their value from call to call.

Section. A set of zero, one, or more paragraphs or entries, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

Section Header. A combination of words followed by a period and a space that indicates the beginning of a section in the ENVIRONMENT, DATA and PROCEDURE divisions.

In the ENVIRONMENT and DATA divisions, a section header is composed of reserved words followed by a period and a space. The permissible section headers are:

In the ENVIRONMENT DIVISION:

```
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
```

In the DATA DIVISION:

```
FILE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
COMMUNICATION SECTION.
REPORT SECTION.
```

In the PROCEDURE DIVISION, a section header is composed of a section name, followed by reserved word SECTION, followed by a segment number (optional) followed by a period and a space.

Section Name. A user-defined word that names a section in the PROCEDURE DIVISION.

Segment Number. A user-defined word that classifies sections in the PROCEDURE DIVISION for purposes of segmentation. Segment numbers may contain only the characters '0', '1', ... , '9'. A segment number may be expressed either as a one or two digit number.

Sentence. A sequence of one or more statement, the last of which is terminated by a period followed by a space.

Separately Compiled Program. A program which, together with its contained programs, is compiled separately from all other programs.

Separator. A punctuation character used to delimit character strings.

Sequential Access. An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

Sequential File. A file with sequential organization.

Sequential Organization. The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

Sign Condition. The proposition, for which a truth value can be determined, that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

Simple Condition. Any single condition chosen from the set:

```
relation condition
class condition
condition name condition
switch-status condition
sign condition
(simple-condition)
```

Sort File. A collection of records to be sorted by a SORT state. The sort-file 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.

Source. The symbolic identification of the originator of a transmission to a queue.

SOURCE-COMPUTER. The name of an ENVIRONMENT DIVISION paragraph which the computer environment, within which the source program is compiled, is described.

Source Computer Entry. An entry in the SOURCE-COMPUTER paragraph of the ENVIRONMENT DIVISION which contains clauses which describe the computer environment in which the source program is to be compiled.

Source Item. An identifier designated by a SOURCE clause that provides the value of a printable item.

Source Program. Although it is recognized that a source program may be represented by other forms and symbols, in this document it always refers to a syntactically correct set of COBOL statements beginning with the IDENTIFICATION DIVISION. In contexts where there is no danger of ambiguity, the word 'program' alone may be used in place of the phrase ' source program'.

Special Character. A character that belong to the following set:

Character	Meaning
+	plus sign
-	minus sign
*	asterisk
/	stroke (virgule, slash)
=	equal sign
$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point)
”	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol
:	colon

Special Character Word. A reserved word which is an arithmetic operator or a relation character.

SPECIAL-NAMES. The name of an ENVIRONMENT DIVISION paragraph in which implementor names are related to user-specified mnemonic names.

Special Names Entry. An entry in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION which provides means for specifying the currency sign; choosing the decimal point; specifying symbolic characters; relating implementor names to user-specified mnemonic names; relating alphabet names to character sets or collating sequences; and relating class-names to sets of characters.

Special Registers. Compiler generated storage areas whose primary use is to store information produced in conjunction with the use of specific COBOL features.

Standard Data Format. The concept used in describing the characteristics of data in a COBOL DATA DIVISION under which the characteristics or properties of the data are expressed in a form oriented to the appearance of the data on a printed page of infinite length and breadth, rather than a form oriented to the manner in which the data is stored internally in the computer, or on a particular external medium.

Statement. A syntactically valid combination of words and symbols written in the PROCEDURE DIVISION beginning with a verb.

Subject of Entry. An operand or reserved word that appears immediately following the level indicator or the level number in a DATA DIVISION entry.

Subprogram. There are three types of subprograms, dynamic, nondynamic, or combined.

Subscript. An integer, or a data-name optionally followed by an integer with the operator + or -, or an index-name optionally followed by an integer with the operator + or -, that identifies a particular element in a table. A subscript may be the word ALL when the subscripted identifier is used as a function argument.

Subscripted Data Name. An identifier that is composed of a data name followed by one or more subscripts enclosed in parentheses.

Switch-Status Condition. The proposition, for which a truth value can be determined, that an implementor defined switch, capable of being set to an 'on' or 'off' status, has been set to a specific status.

Symbolic-Character. A user-defined word that specifies a user defined figurative constant.

System-Name. A COBOL word which is used to communicate with the operating environment.

Table. A set of logically consecutive items of data that are defined in the DATA DIVISION by means of the OCCURS clause.

Table Element. A data item that belongs to the set of repeated items comprising a table.

Terminal. The originator of a transmission to a queue, or the receiver of a transmission from a queue. Not used in HP COBOL II.

Text Name. A user-defined word which identifies library text.

Text-Word. A character or a sequence of contiguous characters within area A and/or area B in a COBOL library, source program, or in pseudo-text that is:

- A separator (except for: a space, a pseudo-text delimiter, the opening and closing delimiters for nonnumeric literals.) The right and left parentheses are always considered text-words.
- A literal (including, in the case of nonnumeric literals, the opening quotation mark and the closing quotation mark that bound the literal.)
- Any other sequence of contiguous COBOL characters except comment lines and the word 'COPY', bounded by separators.

Truth Value. The representation of the result of the evaluation condition in terms of one of two values:

```
true
false
```

Unary Operator. A plus (+) or a minus (-) sign, which precedes variable or left parenthesis in an arithmetic expression and which has the effect of multiplying the expression by +1 or -1 respectively.

Unit. A module of mass storage the dimensions of which are determined by each implementor.

Unsuccessful Execution. The attempted execution of a statement that does not result in the execution of all the operations specified by that statement. The unsuccessful execution of a statement does not affect any data referenced by that statement, but may affect status indicators.

User-Defined Word. A COBOL word that must be supplied by you to satisfy the format of a clause or statement.

Variable. A data item whose value may be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

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 which 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.

Verb. A word that expresses an action to be taken by a COBOL compiler or object program.

Volume. A discrete portion of a storage medium, the dimensions of which are determined by each implementor, that contains part of a file, all of a file, or any number of files. The term is synonymous with reel and unit.

Word. A character-string of not more than 30 characters that forms a user-defined word, a system name, a reserved word, or a function-name.

WORKING-STORAGE SECTION. The section of the DATA DIVISION that describes working-storage data items, composed of noncontiguous items or working-storage records or of both.

F

COBOL Reserved Word List

This appendix lists all the reserved words in COBOL.

Table F-1. COBOL Reserved Words

ACCEPT
ACCESS
ACTUAL
ADD
ADVANCING
AFTER
ALL
ALPHABET
ALPHABETIC
ALPHABETIC-LOWER
ALPHABETIC-UPPER
ALPHANUMERIC
ALPHANUMERIC-EDITED
ALSO
ALTER
ALTERNATE
AND
ANY
ARE
AREA
AREAS
ASCENDING
ASSIGN
AT
AUTHOR

BEFORE
BEGINNING
BINARY
BLANK
BLOCK
BOTTOM
BY

CALL
CANCEL
CD
CF
CH
CHARACTER
CHARACTERS
CLASS
CLOCK-UNITS
CLOSE
COBOL
CODE
CODE-SET
COLLATING
COLUMN
COMMA
COMMON
COMMUNICATION
COMP
COMP-3
COMPUTATIONAL
COMPUTATIONAL-3
COMPUTE
CONDITIONALLY
CONFIGURATION
CONTAINS
CONTENT
CONTINUE
CONTROL
CONTROLS
CONVERTING
COPY
CORR
CORRESPONDING
COUNT
CURRENCY
CURRENT-DATE

DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DAY-OF-WEEK
DE
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DETAIL
DISABLE
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC

EGI
ELSE
EMI
ENABLE
END
END-ACCEPT
END-ADD
END-CALL
END-COMPUTE
END-DELETE
END-DIVIDE
END-EVALUATE
END-IF
END-MULTIPLY
END-OF-PAGE
END-PERFORM
END-READ
END-RECEIVE
END-RETURN
END-REWRITE
END-START
END-STRING
END-SUBTRACT
END-UNSTRING
END-WRITE
ENDING
ENTER
ENTRY
ENVIRONMENT
EOP
EQUAL
ERROR
ESI
EVALUATE
EVERY
EXAMINE
EXCEPTION
EXCLUSIVE
EXIT
EXTEND
EXTERNAL

Table F-1. COBOL Reserved Words (continued)

FALSE	INSTALLATION	NEXT	PROGRAM
FD	INTO	NO	PROGRAM-ID
FILE	INTRINSIC	NOLIST	PURGE
FILE-CONTROL	INVALID	NOT	
FILE-LIMIT	IS	NUMBER	QUEUE
FILE-LIMITS		NUMERIC	QUOTE
FILLER	JUST	NUMERIC-EDITED	QUOTES
FINAL	JUSTIFIED		
FIRST		OBJECT-COMPUTER	RANDOM
FOOTING	KEY	OCCURS	RD
FOR		OF	READ
FREE	LABEL	OFF	RECEIVE
FROM	LAST	OMITTED	RECORD
FUNCTION[1]	LEADING	ON	RECORDING
	LEFT	OPEN	RECORDS
GENERATE	LENGTH	OPTIONAL	REDEFINES
GIVING	LESS	OR	REEL
GLOBAL	LIMIT	ORDER	REFERENCE
GO	LIMITS	ORGANIZATION	REFERENCES
GOBACK	LINAGE	OTHER	RELATIVE
GREATER	LINAGE-COUNTER	OUTPUT	RELEASE
GROUP	LINE	OVERFLOW	REMAINDER
	LINE-COUNTER		REMOVAL
HEADING	LINES	PACKED-DECIMAL	RENAMES
HIGH-VALUE	LINKAGE	PADDING	REPLACE
HIGH-VALUES	LOCK	PAGE	REPLACING
	LOW-VALUE	PAGE-COUNTER	REPORT
I-O	LOW-VALUES	PERFORM	REPORTING
I-O-CONTROL		PF	REPORTS
IDENTIFICATION	MEMORY	PH	RERUN
IF	MERGE	PIC	RESERVE
IN	MESSAGE	PICTURE	RESET
INDEX	MODE	PLUS	RETURN
INDEXED	MODULES	POINTER	REVERSED
INDICATE	MORE-LABELS	POSITION	REWIND
INITIAL	MOVE	POSITIVE	REWRITE
INITIALIZE	MULTIPLE	PRINTING	RF
INITIATE	MULTIPLY	PROCEDURE	RH
INPUT		PROCEDURES	RIGHT
INPUT-OUTPUT	NATIVE	PROCEED	ROUNDED
INSPECT	NEGATIVE	PROCESSING	RUN

1 FUNCTION is a reserved word when the POST85 control option is specified.

Table F-1. COBOL Reserved Words (continued)

SAME	STANDARD-1	THEN	
SD	STANDARD-2	THROUGH	WHEN
SEARCH	START	THRU	WHEN-COMPILED
SECTION	STATUS	TIME	WITH
SECURITY	STOP	TIME-OF-DAY	WORDS
SEEK	STRING	TIMES	WORKING-STORAGE
SEGMENT	SUB-QUEUE-1	TO	WRITE
SEGMENT-LIMIT	SUB-QUEUE-2	TOP	
SELECT	SUB-QUEUE-3	TRAILING	ZERO
SEND	SUBTRACT	TRUE	ZEROES
SENTENCE	SUM	TYPE	ZEROS
SEPARATE	SUPPRESS		
SEQUENCE	SYMBOLIC	UN-EXCLUSIVE	+
SEQUENTIAL	SYNC	UNIT	-
SET	SYNCHRONIZED	UNSTRING	*
SIGN		UNTIL	/
SIZE	TABLE	UP	**
SORT	TALLY	UPON	>
SORT-MERGE	TALLYING	USAGE	>=
SOURCE	TAPE	USE	<
SOURCE-COMPUTER	TERMINAL	USING	<=
SPACE	TERMINATE		<>
SPACES	TEST	VALUE	=
SPECIAL-NAMES	TEXT	VALUES	:
STANDARD	THAN	VARYING	

G

COBEDIT Program and COPY Libraries

This appendix explains the COBEDIT program that allows you to copy source statements from COBOL COPY libraries. With COBEDIT you can develop and maintain COBOL COPY libraries. The EDIT, EXIT, HELP, and LIBRARY commands are discussed as features in using the COPY Library.

The COBEDIT Program

The COBEDIT program resides in the PUB group of the SYS account. This program allows you to create, modify, and look at a COBOL COPY Library file.

As part of the create process, you have the option to copy any ASCII file into the newly created library. Since the records of the file are eventually used in a COBOL program, they must be valid COBOL statements or preprocessor commands. Because of the restriction placed on modules by the COPY command, there should not be any COPY statements as part of the records in a library.

The commands available in the COBEDIT program are `HELP, BUILD, COPY, LIST, EDIT, SHOW, PURGE, KEEP,` and `LIBRARY`. Enter one of these commands when the COBEDIT prompt, a greater-than sign (>), appears. Several MPE commands may also be issued from within the COBEDIT program by typing a colon (:) followed by the command that is to be executed. Any programmatically executable command may be used, that is, those commands that are executed by the COMMAND intrinsic.

To enter the COBEDIT program, issue the MPE RUN command as shown here:

```
:RUN COBEDIT.PUB.SYS
```

When this command is executed, the COBEDIT program displays a header, including the current date and time, and a "HELP" message followed by the greater-than prompt.

```
HP32233A.02.00 COPYLIB EDITOR - COBEDIT MON, MAR 26, 1991, 10:03 AM
(C) HEWLETT-PACKARD CO. 1986
TYPE "HELP" FOR A LIST OF COMMANDS.
>
```

Only one user *per logon group* can be editing any library file. Other users in the same logon group must wait until the current user has finished editing. And, only one user at a time can be editing any particular library file. COBEDIT opens the library file exclusively.

COPY Libraries

The COPY statement is a COBOL feature that enables you to copy COBOL source statements into your program from COPY Libraries. In this way, multiple programs can use a single paragraph, file description or record description.

On an HP computer system, COBEDIT copy libraries are KSAM files of ASCII records which contain one or more modules. These modules are the sets of text which can be copied into your program. You can have more than one library and, by using multiple COPY statements, can copy text from several libraries.

A module in a copy library is distinguished from other modules by a string of alphanumeric characters in columns 73 through 80 of each record in that module. Since this is the only way to distinguish one module from another, the string of characters must be unique with respect to every other module in the same library.

The following illustrates the use of the COBEDIT program to list three modules within a library named MYLIB:

```
:RUN COBEDIT.PUB.SYS

HP32233A.02.00 COPYLIB EDITOR - COBEDIT MON, MAR 26, 1991, 10:08 AM
(C) HEWLETT-PACKARD CO. 1986
TYPE "HELP" FOR A LIST OF COMMANDS.
>LIBRARY MYLIB
>LIST ALL

Text-name MODULE1

001000$CONTROL SUBPROGRAM
001100  IDENTIFICATION DIVISION.
001200  PROGRAM-ID.  DUMMY-SUB.

Text-name MODULE2

005100  WORKING-STORAGE SECTION.
005200  01  UNIV-TOTALER  PIC 9(8) COMP-3.
005300  01  UNIV-ACCUM    PIC 9(8) COMP-3.

Text-name MODULE3

008000      PERFORM TEST-IT.
008100      IF RESULTANT IS LESS THAN 2
008200       PERFORM TEST-FAILED;
008300      ELSE NEXT SENTENCE.
```

The three modules are MODULE1, MODULE2, and MODULE3. There is no restriction on a module name. The names above were chosen to help clarify the example.

COBEDIT Commands

There are 10 commands in the COBEDIT program. Each is listed in Table G-1 and discussed on the following pages. Note that user input is underlined in each example of the commands.

Table G-1. COBEDIT Commands

Command	Meaning
BUILD	Build a COPYLIB file.
COPY	Copy modules into the library as in the BUILD command.
EDIT	Create or edit a module to add to a COPYLIB file.
EXIT	Leave the COBEDIT program.
HELP	List all COBEDIT commands.
KEEP	Add a module to the currently active COPYLIB file.
LIBRARY	Activate an already existing COPYLIB file.
LIST	List text-names or one or more modules of the currently active COPYLIB file.
PURGE	Purge a module of the currently active library or purge the library itself.
SHOW	Show the name of the current library, its key file and the latest module to be accessed.

BUILD Command

The BUILD command allows you to build a new KSAM file to be used as a library file. Once this library file is built, it remains open and available for use until you exit the COBEDIT program, or specify a new library by issuing another BUILD command or a LIBRARY command.

Syntax

BUILD [*file-name*] [,*maxrecs*]

Parameters

file-name — any name you wish to give your new library file, subject to the naming conventions for any MPE file. The *file-name* may be from one to eight alphanumeric characters, the first of which must be alphabetic.

maxrecs — if specified, must be greater than 0. It specifies the maximum number of records that may be placed in the file being built. If no value is specified for *maxrecs*, the default is 2500.

Description

If a file name is not specified, COBEDIT prompts you for one. After you are prompted for a name, a second chance to provide a file name is given if RETURN is pressed. If RETURN is pressed again, the BUILD command is terminated and no library file is created.

If you name a file in the BUILD command, or if a name is specified when COBEDIT prompts you for one, you are next prompted for a name to be used as the key file for the library file being created.

The restrictions on the key file name are the same as for *file-name*.

If RETURN is pressed, an MPE file system error message is listed, followed by an error message from COBEDIT. Then, the BUILD command is terminated.

Once a library file and an associated key file have been named, the COBEDIT program attempts to create a KSAM file using the specified names. If this attempt fails, an MPE error message is generated. Otherwise, you are given the opportunity to copy a file into your newly created library file. When the file name prompt is given and, if you respond with a carriage return, the BUILD command is terminated. To copy a file into the library file, the name of the file must be typed in response to the prompt. This name can be fully qualified and specified in the form *filename.group.account*.

You must have the capability to access files in a group or account other than your own. One of the ways this is accomplished is by using the MPE RELEASE command. See the *MPE XL Commands Reference Manual* for details. Also, see the *Account Structure and Security Reference Manual* for details on file security and access.

If you do not have access to the specified file, the following message is returned:

```
SECURITY VIOLATION (FSERR 93)
BUILD TERMINATED
```

After the file has been specified, you are asked if the file is in COPYLIB format. This is equivalent to asking you if the file to be copied has text-names in columns 73 through 80.

If you respond with Y for YES, COBEDIT attempts to copy the requested file. Note that if the text-name is blank, the COBEDIT program copies the records into your library and assigns a default text-name, BOO-BOO.

If a negative response is given, COBEDIT asks you for a text-name to be used for the copied records. This text-name must be from one to eight characters long.

After a file has been copied into your library, you are asked if there are more files to be copied. A negative response terminates the BUILD command. A positive response causes the COBEDIT program to repeat the questions and actions described in the preceding three paragraphs.

Note If the file to be copied is in copylib format and has duplicate copies of one or more modules, COBEDIT gives an error message.

Examples

To illustrate the BUILD command, user input is underlined:

```
>BUILD
What is the name of your library file?   MYLIB
Name a key file to be used with MYLIB:   KMYLIB
To copy a file into MYLIB now, enter the file name.
File name?   COBCOPY
Is the file in copylib format?   NO
Text-name for library module?   MODULE1
5 records copied to library file.
Do you wish to copy more files?
Respond YES or NO:   NO
Library file created; requested file(s) copied.
>
>BUILD MYLIB
Name a key file to be used with MYLIB:   MYLIBKEY
Unable to create KSAM file
DUPLICATE PERMANENT FILE NAME   (FSERR 100)
>
```

Note that if you name a file to be copied into your library file and the library file does not have a sufficient amount of free space to contain the records of the file being copied, no records are copied and the BUILD command is terminated. A library file that is too small to contain the records from a specified file is used to illustrate this.

```
>BUILD ATLAS, 3
Name a key file to be used with ATLAS:   KEYATLAS
To copy a file into ATLAS now, enter the file name.
File name?   COBCOPY
NOT ENOUGH ROOM FOR FILE COBCOPY
0 records copied to library file.
BUILD TERMINATED
>
```

If you are building a KSAM/XL COPY library, the name of the key file is ignored because KSAM/XL files do not use a key file. When the BUILD command prompts you for a key file name, press the RETURN key.

Example

To create a KSAM/XL copy file:

```
:FILE MYLIB;KSAMXL
:COBEDIT
>BUILD *MYLIB
Name a key file to be used with *MYLIB: <RETURN>

⋮

Library file created.
```

COPY Command

The COPY command allows you to copy additional modules into a library that was created previously using the BUILD command. To use COPY, the library must be the current library or it must be activated by using the LIBRARY command. COPY prompts and executes in a way similar to the BUILD command.

Syntax

COPY

Example

```
:RUN COBEDIT.PUB.SYS

HP32233A.02.00 COPYLIB EDITOR - COBEDIT MON, MAR 26, 1991, 10:12 AM
(C) HEWLETT-PACKARD CO. 1986
TYPE "HELP" FOR A LIST OF COMMANDS.
>COPY
No library is open.
>LIB MYLIB
>COPY
To copy a file into MYLIB now, enter the file name.
File name?  COBCOPY
Is the file in copylib format?  NO
Text-name for Library module?   MOD2
13 records copied to library file.
Do you wish to copy more files?
Respond YES or NO:   NO
Requested file(s) copied.
>EXIT

END OF PROGRAM
```

EDIT Command

The EDIT command calls the EDIT/3000 subsystem, and optionally allows you to name a module from the currently active library to be edited.

Syntax

EDIT [*text-name*]

Parameters

text-name the name of a module in the currently active library.

Description

EDTXT is created by COBEDIT as a permanent file when required for edit operations. EDTXT is purged by COBEDIT when no longer required. Thus, only one user per logon group can be editing any library file.

EDTXT is the name of the temporary text file used as the interface between COBEDIT and EDIT/3000. If you name a module to be edited, a copy of the module, excluding the text-name in columns 73 through 80, is moved into EDTXT.

If a module is not named, a single blank record with a record number of .001 is moved into EDTXT. This blank record is placed in EDTXT in order to place the EDIT/3000 work file in COBOL format. If you do not want to use the blank record, delete it.

Once you have entered the EDIT/3000 subsystem, any of its features, except two can be used to perform any editing task.

The two features you cannot use are the TEXT and KEEP commands.

The TEXT command cannot be used since EDTXT is automatically used as the TEXT file when you enter the EDIT command. However, you can use the JOIN command to append ASCII files to EDTXT.

The KEEP command cannot be used for the same reason. An automatic KEEP is issued, naming EDTXT as the KEEP file.

Example 1

```
>EDIT

HP32201A.07.20 EDIT/3000  TUE, MAR 26, 1991, 10:15 AM
(C) HEWLETT-PACKARD CO.  1990
NOTE: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRONT
/L ALL
 .001
/D
 .001
*** WARNING *** WORK FILE IS EMPTY.
/A
     1     $CONTROL SUBPROGRAM
     1.1    PROGRAM-ID.  FRESHTEST.
     1.2    AUTHOR.  JAMES FISH.
     1.3    //
...
/KEEP MINE
INVALID COMMAND
/E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD?Y
>
```

In the example above, note the error message, `INVALID COMMAND`, which follows a KEEP command attempted while in the EDIT/3000 text editor. You can *not* use the EDIT/3000 KEEP command to keep the file `MINE` because an automatic KEEP is issued, naming EDTXT as the KEEP file.

However, to keep the data entered in EDTXT in your copylib, you *must* use the COBEDIT `KEEP` command. The COBEDIT `KEEP` command is fully explained later.

Example 2

```
>EDIT MODULE1
Previous Edit text was not saved.
OK to clear? (Y/N) Y
HP32201A.07.20 EDIT/3000  TUE, MAR 26, 1991, 10:18 AM
(C) HEWLETT-PACKARD CO.  1990
NOTE: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRONT
/L ALL
      1.   SORT-PARA.
      1.1         SORT SORTFL ON ASCENDING KEY FIRST-KEY
      1.2          INPUT PROCEDURE IS INP-SECTION
      1.3          OUTPUT PROCEDURE IS OUTP-SECTION
      1.4                 THROUGH OUTP-END-SECTION.

/ADD
      1.5   INP_SECTION.
      1.6             OPEN INPUT FILE-IN
      1.7             IF IN-REC IS ALPHABETIC
      1.8                THEN RELEASE IN-REC
      1.9             ELSE NEXT SENTENCE.
      2.0             CLOSE FILE-IN.
      2.1   OUTP-SECTION.
      2.2             OPEN OUTPUT FILE-OUT.
      2.3             IF SORT-REC IS NOT NUMERIC
      2.4                THEN RETURN SORTFL RECORD INTO FOR-WRITE
      2.5                WRITE REC-OUT FROM FOR-WRITE;
      2.6             ELSE NEXT SENTENCE.
      2.7   OUTP-END-SECTION.
      2.8             CLOSE FILE-OUT.
      2.9             //
...
/E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD?Y
>
```

In the example above, when the EDIT command is issued, the module named `MODULE1` is specified. Note the message immediately following the EDIT command above. This message is issued because the data stored in EDTXT was not kept to the library file before the EDIT command was issued. Since the response to the CLEAR question is Y (yes), EDTXT is cleared, and the records of `MODULE1` are copied into it.

Also, although records have been copied from `MODULE1`, the records of `MODULE1` are still in the library file. These are kept in the library file by issuing a KEEP command for the records in EDTXT, using a different text-name, or the same name.

Example 3

As a final illustration of using the EDIT command, a file created outside of the COBEDIT program is joined to the work space associated with EDTXT.

```
>EDIT
Previous Edit text was not saved.
OK to clear? (Y/N) N
>
>KEEP MODULE4
>EDIT

HP32201A.07.20 EDIT/3000  TUE, MAR 26, 1991, 10:21 AM
(C) HEWLETT-PACKARD CO.  1990
NOTE: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRONT
/L ALL
 .001
/M
MODIFY .001
R* THIS MODULE IS CREATED BY JOINING THE FILE, FROMEDIT,
 * THIS MODULE IS CREATED BY JOINING THE FILE, FROMEDIT

/A
       .101* TO THE CURRENT WORK FILE.
       .201//
...
/JOINQ FROMEDIT
NUMBER OF LINES JOINED =2
/L ALL
       .001* THIS MODULE IS CREATED BY JOINING THE FILE, FROMEDIT,
       .101* TO THE CURRENT WORKFILE.
       .201* THIS LINE AND THE FOLLOWING WERE JOINED TO THE WORK
       .301* FILE FROM THE FILE, FROMEDIT.
/E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD? Y
>
```

EXIT Command

The EXIT command is used to exit the COBEDIT program.

Syntax

E[XIT]

Description

To exit COBEDIT, type EXIT or E.

If you have used the EDIT command, and no KEEP command was issued before the EXIT command is executed, the following message is displayed:

```
Edit text not empty.  OK to clear?
```

If you respond with anything except Y or YES, the EXIT command terminates, and COBEDIT remains active. A Y or YES response causes COBEDIT to clear EDTXT, close the currently active library, and cease execution.

If a KEEP command has been performed for the current contents of EDTXT, or if the EDIT command was not used during the current execution of the COBEDIT program, then when the EXIT command is executed, COBEDIT ceases execution with no warning message.

Example 1

```
>EDIT

HP32201A.07.20 EDIT/3000  TUE, MAR 26, 1991, 10:25 AM
(C) HEWLETT-PACKARD CO.  1990
NOTE: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRONT
/L ALL
       .001
/E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD? Y
>EXIT
Edit text not empty.  OK to clear? Y

END OF PROGRAM
:
```

Example 2

```
:RUN COBEDIT.PUB.SYS

HP32233A.02.00 COPYLIB EDITOR - COBEDIT MON, MAR 26, 1991, 10:26 AM
(C) HEWLETT-PACKARD CO. 1986
TYPE "HELP" FOR A LIST OF COMMANDS.
>E

END OF PROGRAM
:
```

Example 3

```
:RUN COBEDIT.PUB.SYS

HP32233A.02.00 COPYLIB EDITOR - COBEDIT MON, MAR 26, 1991, 10:27 AM
(C) HEWLETT-PACKARD CO. 1986
TYPE "HELP" FOR A LIST OF COMMANDS.
>LIB MYLIB
>LIST ALL

Text-name MODULE1

001000$CONTROL SUBPROGRAM
001100 IDENTIFICATION DIVISION.
001200 PROGRAM-ID.  DUMMY-SUB.

Text-name MODULE2

005100 WORKING-STORAGE SECTION.
005200 01  UNIV-TOTALER   PIC 9(8) COMP-3.
005300 01  UNIV-ACCUM     PIC 9(8) COMP-3.

Text-name MODULE3

008000      PERFORM TEST-IT.
008100      IF RESULTANT IS LESS THAN 2
008200       PERFORM TEST-FAILED;
008300      ELSE NEXT SENTENCE.

>EDIT
HP32201A.07.20 EDIT/3000  TUE, MAR 26, 1991, 10:28 AM
(C) HEWLETT-PACKARD CO.  1990
NOTE: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRONT
/L ALL
       .001
/A
       .101* THIS IS TO SHOW WHAT HAPPENS WHEN A KEEP COMMAND
       .201* IS ISSUED BEFORE THE EXIT COMMAND IS USED.
       .301//
...
/E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD? Y
>KEEP MOD4
>E
END OF PROGRAM
:
```

HELP Command

The HELP command lists and gives a brief description of all commands available in the COBEDIT program.

Syntax

HELP

Example

```
> HELP

The following is a list of COBEDIT commands:

BUILD library-name [ , filesize ]
Create a new library file with name "library-name".

COPY
Copy modules into library, as in Build command.

EDIT [ text-name ]
Activate EDIT/3000 and text in that module of the current
library which contains "text-name" in the id field.

EXIT Exit the COBEDIT program.

HELP
Display a list of COBEDIT commands.

KEEP [ text-name ]
Insert an (edited) module in the current library.

LIBRARY library-name
Designate "library-name" as the current library.

LIST  [ text-name ]
      [   ALL   ]
Display all or part of the current library on $STDLIST.
With no parameter, will list the text-names of the current library.

PURGE { text-name }
      {   ALL   }
Delete a module from the current library.  The ALL
option will purge the entire library.

SHOW
Display an information block for the current library.

:{ MPE Command }
Certain MPE commands may be executed from COBEDIT.
```

KEEP Command

The KEEP command allows you to add a module to the currently active library, or replace an already existing module.

Syntax

KEEP [*text-name*]

Parameters

text-name is the name to be used for the module being kept.

Description

If the module to be kept is one that already exists on the file, and you named that module in a previous EDIT command, you do not have to specify a text-name in the KEEP command. In this case, you are asked if you want to replace the module in the library.

Example 1

```
>LIST MYLIB
>LIST ALL

Text-name MODULE1

001000$CONTROL SUBPROGRAM
001100 IDENTIFICATION DIVISION.
001200 PROGRAM-ID.  DUMMY-SUB.

Text-name MODULE2

005100 WORKING-STORAGE SECTION.
005200 01  UNIV-TOTALER    PIC 9(8) COMP-3.
005300 01  UNIV-ACCUM      PIC 9(8) COMP-3.

Text-name MODULE3

008000      PERFORM TEST-IT.
008100      IF RESULTANT IS LESS THAN 2
008200       PERFORM TEST-FAILED;
008300      ELSE NEXT SENTENCE.

Text-name MOD4

000101* THIS IS TO SHOW WHAT HAPPENS WHEN A KEEP COMMAND
000201* IS ISSUED BEFORE THE EXIT COMMAND IS USED.
>EDIT MODULE1

HP32201A.07.20 EDIT/3000  TUE, MAR 26, 1991, 10:32 AM
(C) HEWLETT-PACKARD CO.  1990
NOTE: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRONT
/L ALL
```

```
  1.   $CONTROL SUBPROGRAM
  1.1   IDENTIFICATION DIVISION.
  1.2   PROGRAM-ID.  DUMMY-SUB.
/M 1.2
MODIFY   1.2
PROGRAM-ID.  DUMMY-SUB.
             RTEST-KEEP.
PROGRAM-ID.  TEST-KEEP.

/A
  1.3  AUTHOR. MYSELF.
  1.4//
...
/E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD? Y
>KEEP
"MODULE1 " already exists on Library MYLIB.
OK to clear? Y

>LIST ALL

Text-name MODULE1

001000$CONTROL SUBPROGRAM
001100 IDENTIFICATION DIVISION.
001200 PROGRAM-ID.  TEST-KEEP.
001300 AUTHOR. MYSELF.

Text-name MODULE2

005100 WORKING-STORAGE SECTION.
005200 01  UNIV-TOTALER    PIC 9(8) COMP-3.
005300 01  UNIV-ACCUM      PIC 9(8) COMP-3.

Text-name MODULE3

008000      PERFORM TEST-IT.
008100      IF RESULTANT IS LESS THAN 2
008200       PERFORM TEST-FAILED;
008300      ELSE NEXT SENTENCE.

Text-name MOD4

000101* THIS IS TO SHOW WHAT HAPPENS WHEN A KEEP COMMAND
000201* IS ISSUED BEFORE THE EXIT COMMAND IS USED.
```

Example 2

```
>EDIT

HP32201A.07.20 EDIT/3000  TUE, MAR 26, 1991, 10:36 AM
(C) HEWLETT-PACKARD CO.  1990
NOTE: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRONT
/MODIFY
MODIFY    .001

I*THIS MODULE WILL BE ADDED TO MYLIB BY
*THIS MODULE WILL BE ADDED TO MYLIB BY

/A
  .002*USING A TEXT NAME IN THE KEEP COMMAND
  .003//
...
/E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD?Y
>KEEP MOD5
>LIST ALL

Text-name MODULE1

001000$CONTROL SUBPROGRAM
001100 IDENTIFICATION DIVISION.
001200 PROGRAM-ID.  TEST-KEEP.
001300 AUTHOR. MYSELF.

Text-name MODULE2

005100 WORKING-STORAGE SECTION.
005200 01  UNIV-TOTALER    PIC 9(8) COMP-3.
005300 01  UNIV-ACCUM      PIC 9(8) COMP-3.

Text-name MODULE3

008000      PERFORM TEST-IT.
008100      IF RESULTANT IS LESS THAN 2
008200       PERFORM TEST-FAILED;
008300      ELSE NEXT SENTENCE.
```

```
Text-name MOD4

000101* THIS IS TO SHOW WHAT HAPPENS WHEN A KEEP COMMAND
000201* IS ISSUED BEFORE THE EXIT COMMAND IS USED.

Text-name MOD5

000101* THIS MODULE WILL BE ADDED TO MYLIB BY
000201* USING A TEXT NAME IN THE KEEP COMMAND.
```

Note that if you use the KEEP command without a text-name, and the data in EDTXT was not entered by using text from an already existing module, the message `Invalid text-name` is returned. Also, if the KEEP command is issued, and you have issued no EDIT command since, then when another KEEP command is issued, the message `Edit file is empty` is returned.

LIBRARY Command

The LIBRARY command allows you to select the library that you wish to access. When you issue this command, the currently active library is closed, and the specified library is opened and made available.

Syntax

LIBRARY *library-name*

Parameters

library-name is the name of the library file you want to access.

Description

It can be in any group and account. Note that only one user at a time can be editing a particular library file. COBEDIT opens the library file exclusively.

The fully qualified form of a library name is the same as for all MPE files.

Note that you can specify the name of the currently active library, even though it is already open. This has no effect on the COBEDIT program.

If no library name is specified in the LIBRARY command, COBEDIT prompts you for one.

When the LIBRARY command executes, it checks to make sure that the file named is a valid library file. If it is not, an appropriate error message is generated by the MPE file system, and a COBEDIT error message occurs in the following two cases.

The two cases are: when no file of the specified name exists, and when an error occurs while trying to open the file.

Example

```
>LIB
Library name?  MYLIB
>SHOW
************************************************
Library file:  MYLIB.MANAGERS.USERS
Text-name:
Key file: KMYLIB
************************************************
>LIB COPYLIB
>SHOW
************************************************
Library file:  COPYLIB.MANAGERS.USERS
Text-name:
Key file: KCOPYLIB
************************************************
>LIB CLIB.PUB.USERS
SECURITY VIOLATION  (FSERR 93)
FILE CLIB.PUB.USERS NOT OPENED.
>:TELL WENDY.USERS; PLEASE RELEASE FILE CLIB FOR UPDATE
FROM /S21 WENDY.USERS/ IT'S RELEASED NOW
>:TELL WENDY.USERS; THANKS
>LIB CLIB.PUB.USERS
>SHOW
************************************************
Library file; CLIB.PUB.USERS
Text-name:
Key file: CLIBKEY.KING.USERS
************************************************
```

In the previous examples, the LIBRARY command is used to obtain access to three different files. The first use of the command specified no library. COBEDIT therefore prompted for one.

The third attempt to use the LIBRARY command failed, since the desired library, CLIB, resides in a group other than the logon group, MANAGERS. The availability of MPE commands in COBEDIT make it easy to request that the file be released. Once the file is released, obtaining access to it presents no problem.

The SHOW command is used to show which library file is currently open and available.

As a final example of the LIBRARY command, an attempt to open a non-existent file is made:

```
>LIBRARY FROTH
NONEXISTENT PERMANENT FILE  (FSERR 52)
FILE FROTH NOT OPENED.
```

LIST Command

The LIST command allows you to list information about your currently active library.

The information available is a list of all module names within the library, or a list of all or one of the modules in the library. A control Y terminates the listing.

If no library is open (you have not built one, or used the LIBRARY command to name one, or purged the latest one, and have not opened another), the response to executing a LIST command with or without parameters is `No library file is open.`

Syntax

LIST [*text-name*]

[ALL]

Parameters

text-name	is the name of a module in the currently active library.
ALL	indicates that all modules in the library are to be listed, beginning with the first module on the file, and proceeding to the last.

Description

If neither *text-name* nor the word ALL is used in the LIST command, only the names of the modules in the library are returned.

Note The listing is directed to the $STDLIST device. If you wish to obtain a "hard copy" listing, the $STDLIST device can be redirected by using the :RUN command or by executing COBEDIT in a batch job.

COBEDIT LIST Command

The following examples illustrate the use of redirection and batch jobs.

Example of Redirection

```
:FILE PRINT; DEV=LP
:RUN COBEDIT.PUB.SYS;STDLIST=*PRINT

HP32233A.02.00 COPYLIB EDITOR - COBEDIT MON, MAR 26, 1991, 10:43 AM
(C) HEWLETT-PACKARD CO. 1986
TYPE "HELP" FOR A LIST OF COMMANDS.
>LIB COPYLIB
>SHOW
>LIST ALL
>EXIT
```

Example of Batch Job

```
:JOB LIBPRINT,USER.ACCOUNT
:RUN COBEDIT.PUB.SYS
LIB COPYLIB
SHOW
LIST ALL
EXIT
:EOJ
```

Example

```
>LIBRARY MYLIB
>LIST
Text-names of modules in MYLIB:
    MODULE1
    MODULE2
    MODULE3
    MOD4
    MOD5
>LIST MODULE2

Text-name MODULE2

005100 WORKING-STORAGE SECTION.
005200 01  UNIV-TOTALER   PIC 9(8) COMP-3.
005300 01  UNIV-ACCUM     PIC 9(8) COMP-3.

>LIST ALL

Text-name MODULE1

001000$CONTROL SUBPROGRAM
001100 IDENTIFICATION DIVISION.
001200 PROGRAM-ID.  TEST-KEEP.
001300 AUTHOR. MYSELF.

Text-name MODULE2

005100 WORKING-STORAGE SECTION.
005200 01  UNIV-TOTALER    PIC 9(8) COMP-3.
005300 01  UNIV-ACCUM      PIC 9(8) COMP-3.

Text-name MODULE3

008000      PERFORM TEST-IT.
008100      IF RESULTANT IS LESS THAN 2
008200       PERFORM TEST-FAILED;
008300      ELSE NEXT SENTENCE.

Text-name MOD4

000101* THIS IS TO SHOW WHAT HAPPENS WHEN A KEEP COMMAND
000201* IS ISSUED BEFORE THE EXIT COMMAND IS USED.

Text-name MOD5

000101* THIS MODULE WILL BE ADDED TO MYLIB BY
000201* USING A TEXT NAME IN THE KEEP COMMAND.
```

PURGE Command

The PURGE command allows you to purge either a single module from your currently active library, or the entire library.

If you choose to purge the entire library, it no longer exists after successful execution of the PURGE command.

Syntax

PURGE {*text-name*}

{ALL }

Parameters

text-name	is the name of a module to be purged from the currently active library. This is the module to be purged.
ALL	indicates that you want the entire library, including its key file, to be purged.

Description

If you specify ALL in the PURGE command, COBEDIT double checks to be sure that you want the entire library file purged. COBEDIT displays the following message where *library-name* is the name of your currently active library:

```
Is it OK to purge library library-name?
```

If the response is not Y or YES, purging does not occur and the current library file remains active. If an affirmative response is not given, the following message is returned:

```
COBOL library file library-name purged.
```

Example

To illustrate the PURGE command, a file called MESSEDUP is used. This library contains only two modules. The first is a module copied into it at the time MESSEDUP was created. This module has no text-name associated with it. Thus, it is accessed with the default text-name assigned by COBEDIT that is B00-B00.

```
>LIB MESSEDUP
>LIST ALL

Text-name B00-B00

000101* THESE RECORDS WERE COPIED INTO MESSEDUP FROM AN ASCII
000201* FILE, AND SINCE THE COBEDIT PROGRAM THOUGHT IT WAS IN
000301* COPYLIB FORMAT, IT ASSIGNED THE DEFAULT TEXT-NAME, B00-B00.

Text-name M1

001000$CONTROL USLINIT
002000 IDENTIFICATION DIVISION.
003000 DATA DIVISION.
004000 PROCEDURE DIVISION.
>PURGE M1
>LIST ALL

Text-name B00-B00

000101* THESE RECORDS WERE COPIED INTO MESSEDUP FROM AN ASCII
000201* FILE, AND SINCE THE COBEDIT PROGRAM THOUGHT IT WAS IN
000301* COPYLIB FORMAT, IT ASSIGNED THE DEFAULT TEXT-NAME, B00-B00.
>PURGE ALL
Is it OK to purge library MESSEDUP?  YES
COBOL Library file MESSEDUP purged.
>LIST ALL
No library file is open.
```

SHOW Command

The SHOW command is used to find out the name of the currently active library, its key file, and the name of the module that was most recently accessed by COBEDIT.

If no library is open, the message `No library is open` occurs.

Syntax

SHOW

Example

```
:RUN COBEDIT.PUB.SYS

HP32233A.02.00 COPYLIB EDITOR - COBEDIT MON, MAR 26, 1991, 11:03 AM
(C) HEWLETT-PACKARD CO. 1986
TYPE "HELP" FOR A LIST OF COMMANDS.
>SHOW
No library is open.
>LIB MYLIB
>SHOW
**********************************************
Library name:  MYLIB.USERS.MANAGERS
Text-name:
Key file:  KMYLIB
**********************************************
>EDIT MOD5
HP32201A.07.20 EDIT/3000  TUE, MAR 26, 1991, 11:04 AM
(C) HEWLETT-PACKARD CO.  1990
NOTE: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRONT
/L ALL
  .101*THIS MODULE WILL BE ADDED TO MYLIB BY
  .201*USING A TEXT NAME IN THE KEEP COMMAND
/A
  .301*THIS LINE IS ADDED TO SHOW THE EFFECT OF USING
  .401*THE SHOW COMMAND WHEN A MODULE HAS BEEN ACCESSED.
  .501//
...
/E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD? Y
>SHOW
```

```
*********************************************
Library name:  MYLIB.USERS.MANAGERS  Text-name:  MODS  Key file:  KMYLIB
*********************************************
>KEEP MOD6
>EXIT

END OF PROGRAM
:
```

H

MPE XL System Dependencies

This appendix gives system-specific information about the HP COBOL II/XL programming language on the MPE XL operating system. It does not explain every feature of HP COBOL II/XL. For more information, refer to the main body of this manual.

Introduction

HP COBOL II/XL is Hewlett-Packard's implementation of the 1985 ANSI COBOL standard (X3.23-1985) and the 1974 ANSI COBOL standard (X3.23-1974), the COBOL programming languages that meet the 1985 and 1974 standards set by the American National Standards Institute (ANSI).

The HP COBOL II/XL compiler compiles COBOL'74 programs as well as COBOL'85 programs. When you invoke it through its ANSI74 entry point (using the COB74XL command file), it accepts only syntax that conforms to COBOL'74. When you invoke it through its ANSI85 entry point (using the COB85XL command file), it accepts the syntax of COBOL'85 plus the intrinsic functions that were defined in 1989 by Addendum 1 of the ANSI COBOL'85 standard. The ANSI85 entry point is the default.

Figure H-1 shows the relationships between the two entry points of the HP COBOL II/XL compiler, and the two revisions of the ANSI standard, COBOL'85, and COBOL'74.

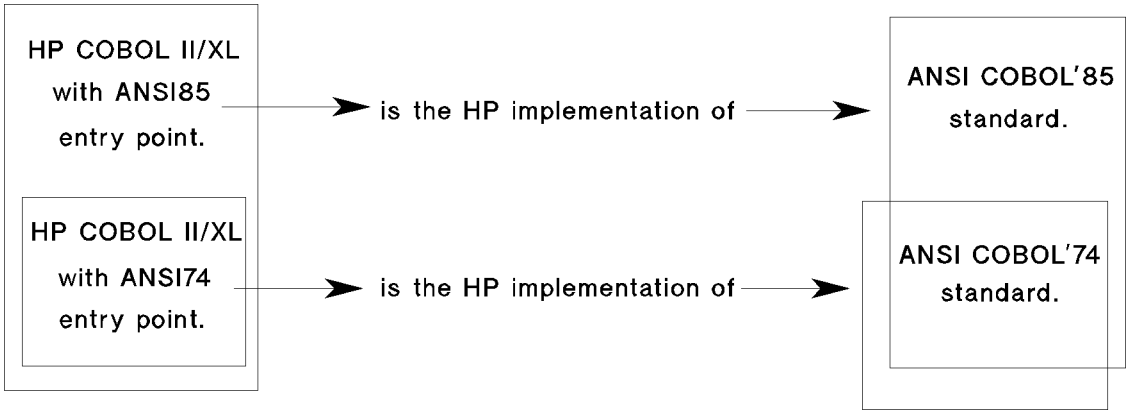


Figure H-1.
Relationships between HP COBOL II/XL and the ANSI Standards COBOL'74 and COBOL'85

MPE XL System Dependencies

The HP COBOL II/XL product consists of a compiler, which translates HP COBOL II/XL programs into machine object files, and a run-time library. The object code that the compiler generates contains calls to routines in the run-time library.

HP COBOL II/XL runs on the MPE XL operating system. You can use the debuggers that run on MPE XL to debug your HP COBOL II/XL programs. They are DEBUG (the MPE XL System Debugger), HP Symbolic Debugger/XL, and HP TOOLSET/XL.

Table H-1 lists HP subsystems with which HP COBOL II/XL can interface.

Table H-1. Subsystems that Interface with HP COBOL II/XL

Subsystem	Description	Where to Look for Details
DEBUG	MPE XL System Debugger.	*System Debug Reference Manual*
HP Symbolic Debugger/XL	A full-featured symbolic debugger that is interactive at the source level.	*HP Symbolic Debugger/XL Reference Manual*
HP TOOLSET/XL	A programming environment for developing COBOL programs. It provides source management, a symbolic debugger, and an editor that is specifically for COBOL.	*HP TOOLSET/XL Reference Manual*
TurboIMAGE/XL	A network database management system. Your COBOL program accesses TurboIMAGE/XL routines with intrinsic calls.	*TurboIMAGE/XL Reference Manual*
HPSQL	A relational database management system whose COBOL preprocessor has macros that generate calls to HPSQL.	*HPSQL/XL COBOL Application Programming Guide*
HP System Dictionary/XL	A dictionary of MPE XL data elements.	*HP System Dictionary/XL General Reference Manual*

Compiling, Linking, and Executing Programs

To make your HP COBOL II source program a valid MPE XL program file, you must compile, link, and execute it. There are two ways to perform these tasks:

- With command files.
- With the RUN command and the Link Editor.

This section describes compilation, linking, and execution and explains the different ways of performing them.

Overview

The HP COBOL II compiler compiles the source program, which is created in a text file. The compiler translates the source code into binary form and stores it in an object file.

The MPE XL Link Editor links the object file into a program file by binding the procedures in the object files together and defining the initial requirements of the user data space.

The MPE XL operating system allocates the space for the program, binds its external procedures to it, and runs it. (The external procedures are in executable libraries.)

Figure H-2 shows how a source program becomes an executing program.

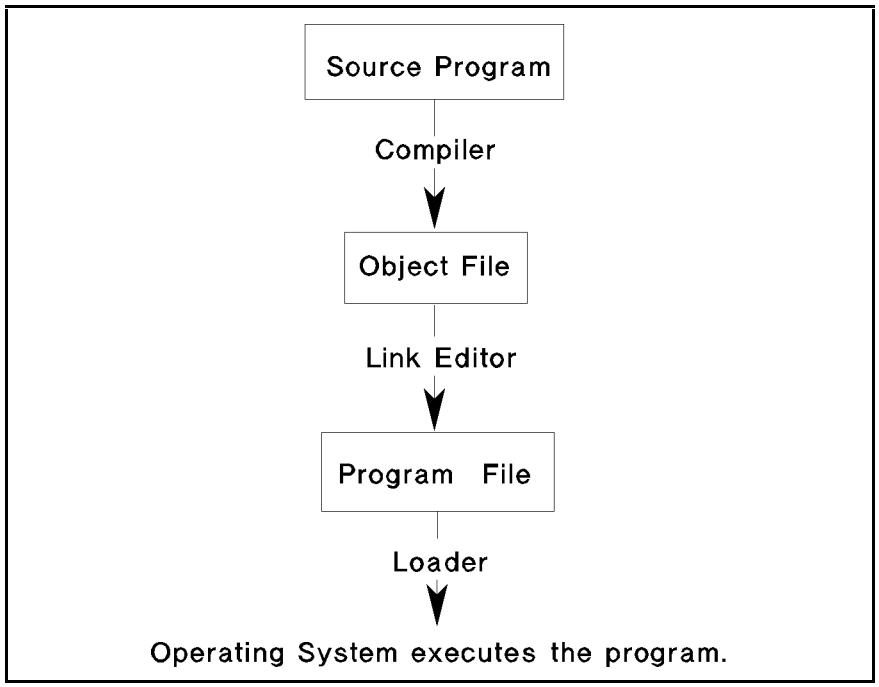


Figure H-2. How a Source Program Becomes an Executing Program

Command Files

Table H-2 lists the MPE XL system-wide command files that you can use to compile, link, and execute HP COBOL II programs. You can enter these commands as part of the input stream in job or batch mode or from your terminal in a session.

The first three command files invoke the HP COBOL II compiler through the entry point that conforms to the 1985 ANSI COBOL standard. The next three command files invoke the HP COBOL II compiler through the entry point that conforms to the 1974 ANSI COBOL standard. You can look at these command files by using the MPE XL PRINT command. For example, to display the COB85XL command file, enter `PRINT COB85XL.PUB.SYS`. Refer to the *Link Editor/XL Reference Manual* for information on how to link object files.

Table H-2. Command Files

Command	Description
`COB85XL`	Invokes the COBOL compiler using the 1985 ANSI standard entry point and creates an object file.
`COB85XLK`	Invokes the COBOL compiler using the 1985 ANSI standard entry point, links the object file, and creates a program file.
`COB85XLG`	Invokes the COBOL compiler using the 1985 ANSI standard entry point, and creates and runs a program file in $NEWPASS.
`COB74XL`	Invokes the COBOL compiler using the 1974 ANSI standard entry point and creates an object file.
`COB74XLK`	Invokes the COBOL compiler using the 1974 ANSI standard entry point, links the object file, and creates a program file.
`COB74XLG`	Invokes the COBOL compiler using the 1974 ANSI standard entry point, and creates and runs a program file in $NEWPASS.

Syntax

```
COB85XL [textfile][,[objectfile][,[listfile][,[masterfile] [,newfile]]]]
        [;INFO="info"][;WKSP=workspacename] [;XDB=xdbfile]

COB85XLK [textfile][,[progfile][,[listfile][,[masterfile] [,newfile]]]]
         [;INFO="info"][;WKSP=workspacename] [;XDB=xdbfile]

COB85XLG [textfile][,[listfile][,[masterfile][,newfile]]]
         [;INFO="info"][;WKSP=workspacename] [;XDB=xdbfile]

COB74XL [textfile][,[objectfile][,[listfile][,[masterfile] [,newfile]]]]
        [;INFO="info"][;WKSP=workspacename] [;XDB=xdbfile]

COB74XLK [textfile][,[progfile][,[listfile][,[masterfile] [,newfile]]]]
         [;INFO="info"][;WKSP=workspacename] [;XDB=xdbfile]

COB74XLG [textfile][,[listfile][,[masterfile][,newfile]]]
         [;INFO="info"][;WKSP=workspacename] [;XDB=xdbfile]
```

Parameters

textfile	MPE or TSAM file containing your source program. This file can be compiled. The default is $STDIN.
objectfile	Relocatable object code file. This file can be linked. The default is $NEWPASS or $OLDPASS. The object file code can be NMOBJ or NMRL. The compiler will take the appropriate actions for existing files. Refer to RLINIT or RLFILE in the section on "Control Options" for new files.
progfile	Executable program file. This file can be executed. The default is $NEWPASS.
listfile	File on which your source code will be listed. The default is $STDLIST.
masterfile	MPE or TSAM file to be merged with *textfile* to produce a composite source program. If *masterfile* is omitted, the entire source is from *textfile*.
newfile	MPE file into which the merged *textfile* and *masterfile* is written. For details, refer to the *HP COBOL II Reference Manual*. If *newfile* is omitted, no new file is written.

info

A string whose value is a command list of the form:

"$*compiler_command*[$*compiler_command*] . . ."

where no *compiler_command* contains the character $.

If the number of commands is long enough, you can use an ampersand (&) to continue the *info* string. The length limit for a compiler command is the same as the length limit for a source program line.

In the listing file, the string "INFO=" appears where the sequence numbers normally appear.

The *info* string is processed before any source, including compiler commands in the source. Therefore, you may not want to use the default settings of these commands in the source file. You should only include commands such as SUBPROGRAM, which are required for proper compilation, in the source file. This allows you to specify commands like NOLIST, MAP, BOUNDS, or CROSSREF uniquely within the *info* string for each compilation.

workspacename

Work space in which HP TOOLSET/XL can manage versions of the source program.

xdbfile

MPE XL file into which a listing of the source code is written. *xdbfile* is used to view the source code in the HP Symbolic Debugger/XL.

Compiling Your Program With the RUN Command

The MPE XL RUN command runs the HP COBOL II compiler, which compiles your source program. You can invoke the HP COBOL II compiler and compile your HP COBOL II program with either the RUN command or a command file.

Syntax

```
RUN { COBOL   }.PUB.SYS [ ,{ ANSI85 } ];PARM=parm;INFO=info
    { COBOLII }            { ANSI74 }
```

Parameters

`COBOL` The compiler is invoked in Native Mode and generates object code especially designed for the MPE XL operating system. The default entry point is ANSI85.

`COBOLII` The compiler is invoked in Compatibility Mode and generates object code especially designed for the MPE V operating system. The code runs on the MPE XL operating system, but not as efficiently as Native Mode code does. The default entry point is ANSI74.

`ANSI85` The compiler is invoked through its ANSI85 entry point and conforms to the COBOL'85 standard.

`ANSI74` The compiler is invoked through its ANSI74 entry point and conforms to the COBOL'74 standard.

parm Tells the RUN command which of the following files have been redefined by FILE commands and are not to be defaulted. See Table H-3 for the values *parm* can have and what they mean.

- *newfile* MPE file into which the merged *textfile* and *masterfile* is written. The formal designator is COBNEW. If *newfile* is omitted, no new file is written.
- *masterfile* MPE or TSAM file to be merged with *textfile* to produce a composite source program. The formal designator is COBMAST. If *masterfile* is omitted, the entire source is from *textfile*.
- *objectfile* Relocatable object code file (Native Mode only). The formal designator is COBOBJ. The default is $NEWPASS or $OLDPASS.
- *uslfile* Relocatable object code file (Compatibility Mode only). The formal designator is COBUSL. The default is $NEWPASS or $OLDPASS.
- *listfile* MPE file into which the source listing and errors are written. The formal designator is COBLIST. The default is $STDLIST.
- *textfile* MPE or TSAM file containing your source program. The formal designator is COBTEXT. The default is $STDIN.

info A string whose value is a command list of the form:

"$*compiler_command*[$*compiler_command*] . . ."

where no *compiler_command* contains the character $ (even if it is within quotes). Refer to Appendix B for more information on compiler commands.

If the number of commands is long enough, you can use an ampersand (&) to continue the *info* string. The length limit for a compiler command is the same as the length limit for a source program line.

In the listing file, the string "INFO=" appears where the sequence numbers normally appear.

The *info* string is processed before any source, including compiler commands in the source. Therefore, you may not want to use the default settings of these commands in the source file. You should only include commands such as SUBPROGRAM, which are required for proper compilation, in the source file. This allows you to specify commands like NOLIST, MAP, BOUNDS, or CROSSREF uniquely within the *info* string for each compilation.

Table H-3. PARM Values and Their Meanings

PARM Value	√ means "Do not use the default for this file:"				
	newfile	*masterfile*	*objectfile*	*listfile*	*textfile*
0					
1					√
2				√	
3				√	√
4			√		
5			√		√
6			√	√	
7			√	√	√
8		√			
9		√			√
10		√		√	
11		√		√	√
12		√	√		
13		√	√		√
14		√	√	√	
15		√	√	√	√

Table H-3. PARM Values and Their Meanings (continued)

PARM Value	√ means "Do not use the default for this file:"				
	newfile	*masterfile*	*objectfile*	*listfile*	*textfile*
16	√				
17	√				√
18	√			√	
19	√			√	√
20	√		√		
21	√		√		√
22	√		√	√	
23	√		√	√	√
24	√	√			
25	√	√			√
26	√	√		√	
27	√	√		√	√
28	√	√	√		
29	√	√	√		√
30	√	√	√	√	
31	√	√	√	√	√

The following is an example of a command file. It can be used as a template to create a command file that merges master and text files to create new files. Given the file named COMPCOB with the following contents:

```
PARM text=$stdin,obj=$newpass,list=$stdlist,mast=$null,new=$null,INFO=""
FILE COBTEXT=!TEXT
FILE COBOBJ=!OBJ
FILE COBLIST=!LIST
FILE COBMAST=!MAST
FILE COBNEW=!NEW
RUN COBOL.PUB.SYS,ANSI85;PARM=31;INFO="!INFO"
```

You can invoke the command file as follows:

```
:COMPCOB PRGCHNGS,,,PRGMAST,PRGNEW
```

Linking Your Program

To link your main program and a subprogram and create a single program file, use the HP Link Editor/XL LINK command. If the main program object code file is *mainobj*, the subprogram object code file is *subobj*, and the program file to be created is *progfile*, then the command is:

```
LINK FROM = mainobj,subobj; TO = progfile
```

This is one of the simplest cases. For more examples, refer to the *HP COBOL II/XL Programmer's Guide*. For complete information on the LINK command, refer to the *HP Link Editor/XL Reference Manual*.

Executing Your Program with the RUN Command

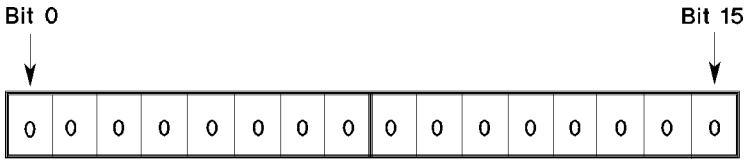
The RUN command also executes the program file. The PARM parameter of the RUN command has two functions:

- Setting software switches.
- Setting the object-time debug module switch.

Setting Software Switches

You must use the PARM=*parameter* keyword of the RUN command to set software switches. The parameter is an octal value corresponding to a 16-bit word. Each bit is the actual switch corresponding to one of the software switches, depending on its position in the word. To set one or more switches to "on", assign the appropriate value to *parameter*. For example, to set switches SW0, SW3, and SW5 to "on", assign the value %112000 to PARM. The example below illustrates this.

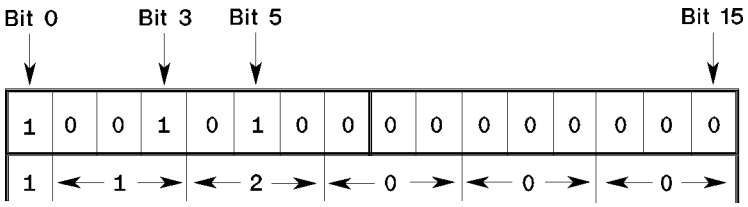
The following shows the switch word initially, where all switches are "off":



After executing the following command:

```
RUN MYPROG;PARM=%112000
```

The switch word looks like the following:



Hexadecimal constants can also be used for the PARM value.

Setting the Object-Time Debug Module Switch

Bit 15 of the PARM parameter is the object-time debug module switch. If bit 15 of the PARM parameter in the RUN command has a value of 1, the object-time DEBUG mode is active. Otherwise, it is inactive. For more information on the object-time debug module switch, see Chapter 13.

Note The object-time debug module switch is equivalent to SW15.

The minimum and maximum values you can use for the PARM parameter word are %0 (all off), and %177777 (all switches, from SW0 to SW15, "on"). If bit 15 is "on", the run-time debugging code is enabled.

Control Options

Control options are used with the $CONTROL command. Control options fall into these three categories:

- MPE XL specific control options that depend on the MPE XL operating system. These options work for HP COBOL II/XL but not other COBOL versions.
- Control options that work for both HP COBOL II/XL and other COBOL versions, but they work differently.
- Obsolete control options, which HP COBOL II/XL no longer supports.

MPE XL-Specific Control Options

This section explains MPE XL-specific control options that depend on the MPE XL operating system and therefore work for HP COBOL II/XL but not for other COBOL versions. They are:

- CALLINTRINSIC
- CMCALL
- INDEX16
- INDEX32
- NLS
- OPTFEATURES
- OPTIMIZE
- POST85
- RLFILE
- RLINIT
- SYMDEBUG=XDB
- VALIDATE
- NOVALIDATE

CALLINTRINSIC

The CALLINTRINSIC option is an aid for migration of HP COBOL II/V programs containing intrinsic calls into HP COBOL II/XL programs. The option causes the compiler to check all called subprograms to determine whether or not they are intrinsics. A warning message is generated each time the compiler locates intrinsics that are called using CALL statements lacking the INTRINSIC parameter. In addition, the compiler generates code in each of these cases, and assumes that the call was to an intrinsic (not to a user program). Use this option for migration only because it extends compilation time.

CMCALL

The CMCALL option is provided as a tool for migration from MPE V to MPE XL based systems. This option affects all external names except those generated by a CALL identifier statement. An external name is generated according to the following rules:

- Hyphens within names are removed.
- Uppercase characters are converted to lowercase.
- Names are truncated to 15 characters.

If CMCALL is specified, the HP COBOL II program can only call or be called by the following kinds of programs:

- COBOL programs compiled with the CMCALL option.
- Programs run in compatibility mode.
- Programs written in other languages that depend on the above rules holding true.

If the CMCALL option is not specified, external names are generated according to the default naming conventions. Refer to "External Names" in the section on "Interprogram Communication" later in this appendix.

INDEX16 and INDEX32

These parameters are used to allocate storage for index data items. (Refer to the section "USAGE Clause" in Chapter 7 for information about index data items.)

32 bits (4 bytes) of storage are allocated for each index data item. 32-bit index data items are fully functional as described in the Chapter 7, "Data Division."

You cannot use index data items having 16 bits of storage allocated to them that come from non-COBOL II/XL files. Specify this option only when reading a record that contains an index data item that was created on a computer having 16-bit architecture. The option causes the byte offsets of the other fields in the record to remain the same as those on the computer on which it was created.

NLS

The `$CONTROL NLS` (Native Language Support) compiler option provides support for international (multi-byte or non-ASCII) characters in certain character operations. For more information on NLS, refer to the *Native Language Programmer's Guide*. Th NLS option makes string comparisons sensitive to international character sets and allows input and output of international characters.

Syntax.

```
$CONTROL NLS= { ON       }
              { OFF      }
              { LITERALS }
              { COMPARE  }
```

`ON` Enables NLS support for both string literals and comparisons. This provides the same service that both the `LITERALS` value and the `COMPARE` value provide.

`OFF` Disables NLS support. This is the default value.

`LITERALS` Enables handling of international characters in string literals during compilation of an HP COBOL II program. `LITERALS` and `COMPARE` are mutually exclusive. Use `NLS ON` to enable `LITERALS` and `COMPARE` at the same time.

`COMPARE` Enables relation condition comparison of non-numeric operands to be sensitive to the character set (and associated collating sequence) that you select. `LITERALS` and `COMPARE` are mutually exclusive. Use `NLS ON` to enable `LITERALS` and `COMPARE` at the same time.

Editing inserts the appropriate single-byte `DECIMAL-POINT`, comma, and single-byte `CURRENCY-SIGN`.

`ACCEPT ... FREE` requires the appropriate `DECIMAL-POINT` for numeric data.

Location. This compiler option can only appear once in your program: on the first line or in the INFO string.

Default. The default for this option is `OFF`.

Limitations. The environment variable `NLDATALANG` must be set at both compile time and run time. The values set at compile time and at run time can be different. For example, the following sets `NLDATALANG` to Norwegian:

```
:SETJCW NLDATALANG 10
```

Some of the other values for `NLDATALANG` are:

Table H-4. Values for NLDATALANG Environment Variable

Language	NLDATALANG Value
Native-3000 (Default)	**00**
American	01
Canadian-French	02
English (British)	05
French	07
German	08
Italian	09
Japanese	221
Norwegian	10
Spanish	12
Swedish	13

For more values and more information on NLDATALANG, refer to the *Native Language Support Reference Manual*.

Using $CONTROL NLS decreases both compile-time and run-time performance in some cases and always reduces the backwards compatibility of your programs.

Only single-byte CURRENCY-SIGNs and single-byte DECIMAL-POINTs are supported. Their values are overridden by NLS, not by the program-collating sequence.

Comparisons by indexed sequential files are done in the collating sequence specified during their creation in KSAMUTIL. By default, HP COBOL II/XL creates KSAM files with an ASCII (binary) collating sequence.

International characters are not supported in macros or in preprocessor commands, such as PAGE, TITLE, VERSION, and COPYRIGHT.

International characters are also not supported in COPY REPLACING or REPLACE. These characters cannot be on any line containing embedded COPY or REPLACE character strings or in COPYLIB with REPLACING if any tokens on that line are replaced. International characters are supported in the following COBOL functions: MAX, MIN, ORD-MAX, and ORD-MIN. The compiler does not support international characters in the following COBOL functions: CHAR, LOWER-CASE, ORD, and UPPER-CASE. These functions use the current COBOL program collating sequence, which defaults to the ASCII character set.

Operations on data items that contain international characters are completed in the same manner as if the characters are ASCII. These include the following:

- Reference modification on MOVEs and compares.
- INSPECT (TALLYING, REPLACING, CONVERTING).
- EXAMINE.
- STRING and UNSTRING.
- Class conditions for user-defined classes, except "alphabetic" clauses.
- Relation conditions with figurative constants.
- Display international characters.

Example 1. This example displays international characters. Substitute # signs with NLS characters to make this example work.

```
001000$CONTROL NLS=ON
001100 IDENTIFICATION DIVISION.
001200 PROGRAM-ID. EXAMPLE.
001700 DATA DIVISION.
002300 WORKING-STORAGE SECTION.
002400 01 NLS-FIELD PIC X(08) VALUE  "G#K#".
002600 PROCEDURE DIVISION.
002700 NLSEXAMPLE.
002800     DISPLAY "NLS-FIELD is initialized to ", NLS-FIELD.
002900     DISPLAY "NLS literals can also be used in COBOL "
003000             " programs as follows :".
003100     DISPLAY "   The following DISPLAY statement uses NLS string."
003200     DISPLAY "        A#K#B".
003300     DISPLAY "   Characters between A and B are NLS characters.".
003400     STOP RUN.
```

Example 2. This example compares two NLS strings. Substitute **#** signs with NLS characters to make this example work.

```
001000$CONTROL NLS=ON
001100 IDENTIFICATION DIVISION.
001200 PROGRAM-ID. EXAMPLE1.
001700 DATA DIVISION.
002300 WORKING-STORAGE SECTION.
002400 01 NLS-FIELD PIC X(08) VALUE  "G#K#".
002410 01 NLS-FIELD-1 PIC X(06) VALUE  "G#K#".
002500 01 NLS-FIELD-2 PIC X(6) VALUE  "G#K#".
002600 PROCEDURE DIVISION.
002700 NLSONEXAMPLE.
004700 IF NLS-FIELD <> NLS-FIELD-2
004800    DISPLAY "Fields of different lengths are not identical"
004801 ELSE
004802    DISPLAY "Fields of different lengths are identical"
004803 END-IF.
004804
004810 IF NLS-FIELD-1 <> NLS-FIELD-2
004820    DISPLAY "Same length fields are not identical"
004830 ELSE
004840    DISPLAY "Same length fields are identical"
004850 END-IF.
004860
004900 MOVE "aa" TO NLS-FIELD NLS-FIELD-2.
005000 IF NLS-FIELD = NLS-FIELD-2
005100    DISPLAY "strings are identical"
005200 ELSE
005300    DISPLAY "strings with ascii characters are not identical"
005400 END-IF.
005500 STOP RUN.
```

OPTFEATURES

The OPTFEATURES option for the $CONTROL command allows you to make your programs run faster by generating more efficient code.

Syntax.

```
$CONTROL OPTFEATURES = [CALLALIGNED[16]] [LINKALIGNED[16]]
```

If both CALLALIGNED and LINKALIGNED are specified, you must separate them with a space.

By default, the compiler expects parameters passed to subprograms *by reference* to be byte-aligned. The compiler generates extra code in the called program to move byte-aligned data to temporary storage that is 32-bit aligned (or 16-bit aligned) when required by arithmetic expressions.

When LINKALIGNED is specified, the compiler generates code that expects parameters in the LINKAGE SECTION (01s and 77s) to be 32-bit aligned. If LINKALIGNED16 is specified, the compiler generates code for 16-bit alignment. The LINKALIGNED options are not meaningful for main programs. The compiler generates more efficient code for accessing formal parameters when you specify one of these options. If you do not specify a LINKALIGNED option, the compiler generates code that assumes the parameters are on byte boundaries (see "BOUNDS" later in this section for details).

When CALLALIGNED is specified, the compiler checks the alignment of all identifiers in the USING phrase of CALL statements. An error message is issued for parameters not on a 32-bit boundary. If CALLALIGNED16 is specified, the compiler checks for 16-bit boundaries. Once you have changed the alignment of the flagged parameters, you can remove the CALLALIGNED option. Subprograms written in languages other than HP COBOL II may require parameters to be aligned on 32- or 16-bit boundaries. The CALLALIGNED or CALLALIGNED16 option can be used to flag any parameters that are not properly aligned for the subprogram.

Note CALLALIGNED does not apply to intrinsic calls. Alignment requirements for intrinsics are always checked when you use the CALL INTRINSIC format of the CALL statement.

Example.

```
$CONTROL BOUNDS,OPTFEATURES=CALLALIGNED LINKALIGNED, OPTIMIZE=1
```

OPTIMIZE

The OPTIMIZE option specifies the level of object code optimization you want. If your program does not contain the OPTIMIZE option, the object code is not optimized. This is equivalent to specifying optimization level zero. OPTIMIZE must appear before the IDENTIFICATION DIVISION.

Syntax.

```
$CONTROL OPTIMIZE [=0]
                  [=1]
```

The following table summarizes the levels of optimization you can request.

Table H-5. $CONTROL OPTIMIZE Parameters

If you specify:	You get:
Nothing	No optimization. This is the default.
`$CONTROL OPTIMIZE=0`	No optimization. This is the default.
`$CONTROL OPTIMIZE`	Level one optimization.
`$CONTROL OPTIMIZE=1`	Level one optimization.

For more information on the OPTIMIZE option, see the *HP COBOL II/XL Programmer's Guide*.

POST85

The POST85 option enables the COBOL functions and makes the word FUNCTION a reserved word. If you use the word FUNCTION as an identifier in your program, you must change it to another word before you can use this option. Otherwise, you get a compile-time error.

The COBOL functions were added to the ANSI standard after the 1985 ANSI standard was published. For more information, see Chapter 10, "COBOL Functions."

RLFILE and RLINIT

When you compile a program on an MPE XL system, the compiler produces a relocatable object file (file code NMOBJ) for the program. You can then use the HP Link Editor/XL to place the relocatable object file in a relocatable library (file code NMRL).

Alternatively, you can direct the compiler to place the relocatable program in a relocatable library directly. The RLFILE option of the $CONTROL command lets you do this. The RLFILE option lets you closely simulate the placing of object modules in a USL file on MPE V systems. When you use RLFILE, the compiler assumes that the formal file designator name of the relocatable library is COBOBJ and that it has a file code of NMRL. The compiler also treats each procedure in the source file as a separate compilation unit and places each of the compilation units into the library as separate relocatable modules. When you use the RLFILE option, you can erase modules in an existing library before placing new modules into it by using the RLINIT option.

If you use the RLFILE option, be sure that the output file (if it already exists) is a relocatable library with file code NMRL. If a previous compilation created COBOBJ as a relocatable object file, you must purge it before compiling.

For details about RLFILE and RLINIT, see the *HP COBOL II/XL Programmer's Guide*.

SYMDEBUG=XDB

The SYMDEBUG=XDB option causes the compiler to put symbolic debug information into the object file for use with HP Symbolic Debugger/XL. The main program should include this option if any subprogram includes the option.

This option must appear before the IDENTIFICATION DIVISION. If the SYMDEBUG option is used without the "= XDB" phrase, the symbolic debug information is formatted for use with HP TOOLSET/XL instead of with HP Symbolic Debugger/XL.

VALIDATE and NOVALIDATE

The VALIDATE and NOVALIDATE options allow you to specify whether or not you want the program to check the validity of the data. VALIDATE causes the compiler to generate code for each arithmetic operation or MOVE and to check for valid decimal digits and signs in the data associated with all identifiers having USAGE IS DISPLAY, PACKED-DECIMAL, or COMPUTATIONAL-3 clauses.

When a program that was compiled with the VALIDATE option encounters an illegal ASCII or decimal digit, the program aborts and one of the following messages is displayed:

```
Illegal ASCII digit
```

or:

```
Illegal decimal digit
```

Refer to the section "Run-Time Trap Handling" later in this appendix for information about how to handle these errors when they occur.

To prevent these errors, use a de-edited MOVE to check that no illegal characters or blanks are placed in a numeric field when moving from an edited field (see the "MOVE Statement" in Chapter 9 for information on de-edited moves). Also, use the NUMERIC class condition to detect invalid digits.

NOVALIDATE is the default on MPE XL systems. If you specify NOVALIDATE or do not specify VALIDATE, no validation code is generated. This results in more efficient code.

Control Options that Work Differently

This section describes control options that work for both HP COBOL II/XL and other COBOL versions, but work differently. They are:

- ANSISUB
- BOUNDS
- CODE
- USLINIT

ANSISUB

ANSISUB specifies that the source code is a subprogram that strictly conforms to the ANSI standard (refer to "$CONTROL Command" in Appendix B for more information on ANSISUB). In HP COBOL II/XL, there is no run-time performance penalty for specifying ANSISUB. The DATA DIVISION is not written to a file between calls in HP COBOL II/XL as it is in HP COBOL II/V. The calling program can CANCEL a subprogram that is compiled with this option, if necessary. The program file of a program compiled with ANSISUB is larger because it contains code to reinitialize the DATA DIVISION if the subprogram is cancelled.

BOUNDS

In addition to its normal functionality (described in Appendix B), the control options BOUNDS and OPTFEATURES=LINKALIGNED (or LINKALIGNED16) provide run-time checking of the LINKAGE SECTION. When you specify both the BOUNDS and LINKALIGNED options in a $CONTROL command, the compiler generates code that checks whether or not all the parameters passed to a subprogram are aligned on 32-bit or 16-bit boundaries. If a parameter is not aligned on a 32-bit or 16-bit boundary, a run-time trap occurs. (See the section "Run-Time Trap Handling" for more information.)

In cases when a program contains illegal PERFORM statements, the compiler issues an error message if these statements would cause the program to abort during execution. Illegal PERFORM statements cause a program to abort in the following situations:

- When the program contains too many indirectly recursive PERFORM statements.
- When the program uses too many GO TO statements to get out of PERFORMed paragraphs.
- When the program has too many PERFORM statements with common exit points.

See the section "Language Dependencies" later in this appendix for more information on illegal PERFORM constructs.

CODE

The CODE parameter requests a copy of the assembly language listing of the object code written to the temporary file, COBASSM. This object code is a listing of the machine code generated by the compiler.

USLINIT

USLINIT is ignored in HP COBOL II/XL programs. Object files are always initialized by the HP COBOL II/XL compiler. See the section "RLFILE and RLINIT" for relocatable library files.

Obsolete Control Options

Obsolete control options are no longer supported by HP COBOL II/XL. They are:

- ANSIPARM
- BIGSTACK
- SORTSPACE

The compiler issues a questionable error message when it encounters these control options and ignores them. Delete these options from your HP COBOL II source files.

Data Alignment and Limits on MPE XL

All addresses on MPE XL systems are byte addresses that are aligned in some way. Addresses divisible by 2 are half-word aligned. Addresses divisible by 4 are word aligned. A word aligned address is also half-word aligned and byte aligned. Refer to the *HP COBOL II/XL Programmer's Guide* for more information on addressing.

Alignment

By default, HP COBOL II/XL data is aligned in the following way on MPE XL:

- In the WORKING-STORAGE and FILE sections, level 01 and 77, COMP SYNCHRONIZED or BINARY SYNCHRONIZED, and index data items are 32-bit-aligned.
- The first item in WORKING-STORAGE is 64 bit-aligned.
- In the LINKAGE SECTION, all level 01 and 77 items are 8-bit-aligned (byte-aligned) even if the SYNCHRONIZED clause is specified. The SYNCHRONIZED clause adds slack bytes as if the 01 record began on a 32-bit boundary.

The SYNC32 control option does not affect the above, because it specifies the default.

The SYNC16 control option affects the WORKING-STORAGE and FILE sections as follows:

- Level 01, 77, and indexed data items are 32-bit-aligned.
- COMP or BINARY SYNCHRONIZED data items are 16-bit-aligned.

The control options OPTFEATURES=LINKALIGNED and OPTFEATURES=LINKALIGNED16 affect only the LINKAGE SECTION. When you use one of these options, the compiler assumes that levels 01 and 77 and index data items (depending on SYNC16/SYNC32) are on 32- and 16-bit boundaries, respectively.

Limits on Data Items

The following are limits on data items:

- The maximum number of data items in any one USING clause of the CALL statement, the ENTRY statement, or the PROCEDURE DIVISION header is 255. Or, the number of data items in these USING clauses cannot be greater than the total number of 01 and 77 level items defined in the LINKAGE SECTION.
- The maximum number of EXTERNAL data items and files is as follows:
  - The sum of the following items must be less than or equal to 4000:
    - The number of EXTERNAL files, multiplied by two.
    - The number of EXTERNAL records.
- The maximum number of 01 and 77 level entries permitted in the LINKAGE SECTION of a subprogram, excluding redefinitions, is 255.

HP COBOL II/XL Language Dependencies

HP COBOL II/XL language dependencies are features of HP COBOL II/XL that depend on the MPE XL operating system. Often they are tailored to the architecture of the computer on which MPE XL runs. This section groups these language dependencies by what they affect—the program division or interprogram communication.

IDENTIFICATION DIVISION

In the IDENTIFICATION DIVISION, the compiler uses the name specified in the PROGRAM-ID paragraph to generate an external name according to the following conventions:

- Long names are truncated to 30 characters.
- All uppercase characters are converted to lowercase unless the first character of the name is a backslash (\).
- If a name begins with a backslash, it is interpreted literally as beginning with the character after the backslash. No conversion to lowercase or uppercase is performed. As with other non-numeric literals, the backslash and name must be enclosed in quotation marks.
- Hyphens are converted to underscores (unless $CONTROL CMCALL is specified).

ENVIRONMENT DIVISION

The following are dependencies in the ENVIRONMENT DIVISION:

- The RESERVE clause of the SELECT statement is ignored in HP COBOL II/XL. This is because the clause is used to allocate input/output buffers and buffers are not used on MPE XL systems.
- In the ASSIGN clause, an operating system file designator is of the following form:

  file/lockword.group.account

  This is the fully qualified form of a file name. If the file exists in your log-on group, then you need to specify only the *file* parameter and any lockword assigned to it. If the file exists in a group other than your log-on group, you must specify the file (and lockword, if any), and the group where the file resides. Finally, if the file exists in a group and account other than your log-on, you must use the fully qualified file name. If the file resides in a group or account other than your own, you must have access to the file.
- In the ASSIGN clause, the *device* parameter allows you to select a file system defined device type (such as DISC, TAPE, or LP) or a private volume. If you omit this parameter, the default device type, DISC, is assumed. If this parameter is followed by a left parenthesis, the CCTL (carriage control) option is assumed. CCTL is described in Chapter 6, "Environment Division."
- In the ASSIGN clause, the *file-size* parameter can be a maximum of nine digits long. If omitted, a file size of 10,000 records is assigned by default.

DATA DIVISION

The following are dependencies in the DATA DIVISION:

- **Word size**—The word size on MPE XL systems is 32 bits or 4 bytes.
- **Index names**—Index names are 4 bytes long.
- **Subscripts**—The performance when referencing table elements using subscripts defined as `PIC S9(9) COMP SYNC` is the same as referencing table elements using index names.
- **Synchronization**—The default synchronization is on 32-bit boundaries.
- **LINAGE-COUNTER**—The LINAGE-COUNTER is a 9-digit unsigned integer.
- **CODE-SET clause of the FD entry**—The CODE-SET clause specifies the character code convention used to represent the data in the file (ASCII, EBCDIC, EBCDIK, STANDARD-1, STANDARD-2, or NATIVE). The default is ASCII.
- **Optimal data types**—Chapter 3 of the *HP COBOL II/XL Programmer's Guide* provides a discussion of optimal data types.

PROCEDURE DIVISION

The following are dependencies in the PROCEDURE DIVISION:

- **Segmentation**—The MPE Segmenter is not available on MPE XL systems. The HP COBOL II compiler ignores segment numbers after section names (except where ALTER and alterable GO TO statements exist). No performance gain is achieved by including segment numbers after section names. The function formerly accomplished by segmenting an HP COBOL II program using section numbers is provided by the memory manager of the MPE XL operating system.
- **ACCEPT Statement**—An ACCEPT operation prematurely terminated by an `:EOD` or `:EOJ` (in job mode) causes a read error condition and abort of the program.
- **DISPLAY Statement**—The format of output when displaying unedited signed numeric data items is improved in HP COBOL II. For signed items, the sign overpunch is removed and a leading minus or plus sign is printed. A decimal point is inserted where applicable. For example, if the data item `A` is declared as `PIC S999V99` and has the value 1.3, and the following statement is executed:

  ```
  DISPLAY A
  ```

  The result would be:

  ```
  +001.30
  ```

  The console device when you use UPON CONSOLE is assumed to be 50 characters long. The SYSOUT device when you use UPON SYSOUT is assumed to be 132 characters long. If the data you display is longer than these, it will be displayed on multiple lines.

  Escape sequences are stripped out of the data when displayed UPON CONSOLE.

- **Error Handling**—Decimal data is not checked for validity unless you specify the control option VALIDATE. When you specify this option, a program aborts on the following conditions:
  - Illegal ASCII digit.
  - Illegal decimal digit.
  - Illegal signs.

  Your program may abort with the following conditions:
  - Division by zero.
  - An overflow condition occurs without an ON SIZE ERROR phrase.

  Using a common point to exit from multiple PERFORM statements may also cause errors in HP COBOL II. If too many common exit points from PERFORM statements are found, an error message is issued at run time. Using the control option BOUNDS causes the compiler to generate code to see if the paragraph stack overflows.

  In addition, illegal GO TO statements (those that branch out of PERFORMed paragraphs) cause a program to abort in HP COBOL II. Using control option BOUNDS causes the program to trap instead.

  You can control the run-time trap handling environment by setting the MPE XL variable COBRUNTIME (see the section "Run-Time Trap Handling" and the *HP COBOL II/XL Programmer's Guide* for more information).

- **EXCLUSIVE/UN-EXCLUSIVE Statement**—The following applies to the use of the EXCLUSIVE statement:

  Programs that are updating or adding to the file must utilize a run-time file equation with the "SHR" option specified to permit other programs to share the file.

  Programs that want READ ONLY access to the file may improve performance when the program is not concerned with data currency, but is currently sharing the file with other programs that are concerned with data currency. The program must specify either the "L" option within the SELECT statement of the file, or the "LOCK" option on a run-time file equation without utilizing the EXCLUSIVE/UN-EXCLUSIVE statement in the PROCEDURE DIVISION. This enables HP COBOL II to open the file with dynamic locking specified, but to refrain from actually locking it for input operations.

  The COBOLLOCK and COBOLUNLOCK procedures are not available in HP COBOL II/XL. Use the EXCLUSIVE and UN-EXCLUSIVE statements instead.

Note Use of an EXCLUSIVE statement for a file causes any OPEN of the same file to be executed with the dynamic locking facility enabled.

- **OPEN Statement**—INDEX organization files are implemented as MPE KSAM files. Compatibility Mode KSAM files use two files, a data file and a key file. The key file name for an INDEX file is the data file name plus the letter "K". If the file name is already eight characters long, the last character is replaced by "K". Native Mode INDEX files use one Native Mode KSAM file. When only ANSI85 is specified, files with U or V type records will not be checked for record length mismatch. For more information, see the *HP COBOL II/XL Programmer's Guide*.
- **PERFORM Statement**—The following are illegal PERFORM constructs on HP COBOL II/XL:
  - PERFORM statements with a common exit point.
  - PERFORM statements where the exit point of one PERFORM statement is contained within the range of another, subsequently called PERFORM statement.

These two illegal cases are illustrated in Figure H-3 below.

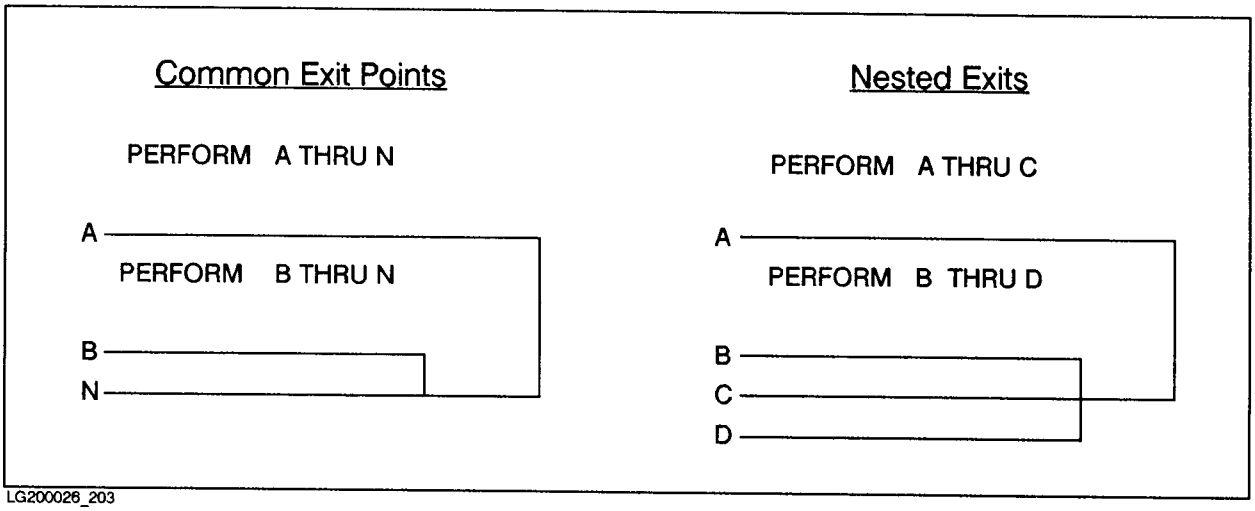


Figure H-3. Invalid PERFORM Constructs

Note The behavior of illegal PERFORM constucts is not always consistent. The compiler does not flag these constructs as errors.

Interprogram Communication

This section explains these aspects of interprogram communication:

- External names.
- Subprogram types.
- Intrinsics.
- Parameter alignment.

External Names

An external name is one that is visible to other programs. The compiler generates external names for the PROGRAM-ID name (*programname*), the name in the ENTRY statement (*literal-1*), and the program name in the CALL statement (*literal-1*) according to the following conventions (unless $CONTROL CMCALL is specified):

- Long names are truncated to 30 characters.
- Uppercase characters are converted to lowercase unless the first character of the name is a backslash (\).
- If a name begins with a backslash, it is interpreted literally as beginning with the character after the backslash.
- Hyphens are converted to underscores.

When $CONTROL CMCALL is specified, the external names are generated according to the following conventions:

- Hyphens within names are removed.
- Uppercase characters are converted to lowercase.
- Names are truncated to 15 characters.

Subprogram Types

Subprogram types are discussed in Chapter 11, "Interprogram Communication," and in the *HP COBOL II/XL Programmer's Guide*. There are no restrictions as to which type of subprogram (ANSISUB, DYNAMIC, or SUBPROGRAM) can be in a relocatable library or an executable library in HP COBOL II/XL.

When an ON EXCEPTION phrase, an ON OVERFLOW phrase, or *identifier-1* is specified in the CALL statement, the following restrictions apply:

- *identifier-1* cannot be numeric.
- The subprogram called must reside in an executable library or in the program file.

Calling Intrinsics

You must include the INTRINSIC phrase to call all MPE operating system intrinsics. Using the INTRINSIC phrase is also recommended for calling subsystem intrinsics to provide for more thorough error checking and greater portability of programs.

Parameter passing to intrinsics is described in detail in the *HP COBOL II/XL Programmer's Guide*.

The system intrinsic file is SYSINTR.PUB.SYS.

.LOC. Pseudo-Intrinsic

Addresses on MPE XL systems are either 32 or 64 bits long. To get a 32-bit address, you must define the result of the .LOC. pseudo-intrinsic as:

```
PIC S9(9) USAGE IS BINARY.
```

To get a 64-bit address, you must define the result as:

```
PIC S9(18) USAGE IS BINARY.
```

Caution Pseudo-intrinsics are highly machine-dependent and should not be used in programs that may be run on different machines and architectures now or in the future.

Parameter Alignment

The compiler passes information to the Link Editor about the actual alignment of each parameter of the CALL statement.

If you use either CALLALIGNED or CALLALIGNED16 with the control option OPTFEATURES, an error message is issued for each parameter of a CALL statement that is not on a 32- or 16-bit boundary, respectively.

If you specify either LINKALIGNED or LINKALIGNED16 with the control option OPTFEATURES for a subprogram, the subprogram expects all parameters in the USING phrase of the PROCEDURE DIVISION header to be aligned on 32-bit or 16-bit boundaries, respectively. Without this option, the compiler assumes that all parameters are on byte boundaries.

On MPE V systems, the @ sign preceding a parameter in the USING phrase of the CALL statement indicates that the byte address of that parameter should be passed to the subprogram being called. Since all addresses are byte addresses on MPE XL systems, the @ sign on parameters is ignored.

Run-Time Trap Handling

The HP COBOL II compiler handles run-time traps for a variety of run-time error conditions. You can control your program's response to these conditions. This section:

- Describes the traps that the HP COBOL II compiler supports.
- Explains how to specify what actions should be taken when run-time errors occur.
- Explains when and how to enable the trap mechanism.

Supported Traps

The HP COBOL II compiler supports the following traps:

- **Illegal ASCII digit (Error 711)**—This error occurs if the program is compiled with the $CONTROL VALIDATE option and an illegal ASCII digit is encountered. It also occurs when an unsigned number is detected in a signed numeric field or vice-versa.
- **Illegal decimal digit (Error 710)**—This error occurs if the program is compiled with the $CONTROL VALIDATE option and an illegal decimal digit is encountered. It also occurs when an unsigned number is detected in a signed numeric field or vice-versa.
- **Range Error (Error 751)**—This error occurs if the program is compiled with the $CONTROL BOUNDS option and one of the following occurs:
  - The identifier named in an OCCURS DEPENDING ON clause is out of bounds.
  - A subscript or index is out of bounds.
  - A reference modification is out of bounds.
  - For some COBOL functions, parameter is out of range.
- **No Size Error Phrase (Error 747)**—This error occurs:
  - If a division by zero or other size error occurs without an ON SIZE ERROR phrase.
  - For COBOL functions, IEEE traps can occur for invalid parameters.
  - For COBOL functions, XLIBTRAP traps can occur for invalid parameters.
- **Invalid GO TO (Error 754)**—This error occurs for an alterable GO TO that was never altered. That is, it never specified the target of the GO TO statement.
- **Address Alignment (Error 753)**—This error occurs if the program is compiled with the $CONTROL BOUNDS option and either the control option OPTFEATURES=LINKALIGNED or LINKALIGNED16 and a parameter is passed that is not on a 32-bit or 16-bit boundary, respectively.
- **Paragraph Stack Overflow (Error 748)**—This error occurs when a program is compiled with the $CONTROL BOUNDS option and one of the following occurs:
  - A recursive PERFORM.
  - Too many nested PERFORM statements have the same common exit point.
  - Too many illegal GO TO statements are used to jump out of PERFORMed paragraphs.

Handling Run-Time Errors with COBRUNTIME

The default action when one of these errors occurs is to print an error message and abort the program. To specify an action other than the default, you need to do both of the following:

- Compile your program with $CONTROL VALIDATE and $CONTROL BOUNDS.
- Set a global variable, called COBRUNTIME, to a set of characters before running the program.

A global variable is similar to a job control word. Each character position in COBRUNTIME corresponds to a particular error condition. The letter in each character position of COBRUNTIME instructs the compiler how to handle that particular error, as shown in Table H-6.

Table H-6. Run-Time Error Handling Options

Option	Meaning
A or blank	Print the error message and abort (default).
C	Print the error message and continue.
D	Print the error message and enter debug mode.
I	Ignore the error. (Continue without printing an error message).
M	Print the error message, change the illegal digit to some legal digit, and continue. This option is only valid for illegal decimal or ASCII digit errors. (See character position 1 in Table H-7.) When used for other errors, M is treated as a blank.
N	Change the illegal digit to a legal digit and continue without printing an error message. This option is only valid for illegal decimal or ASCII digit errors in positions 1, 7, and 8. See the description of character positions 1, 7, and 8 in Table H-7 for details. When used in other positions, N is treated as a blank.

Note The M and N options alter the offending source fields unless the source field is defined as PIC X and the target is PIC 9 DISPLAY.

Setting COBRUNTIME

You set the run-time environment using the MPE XL SETVAR command with the variable COBRUNTIME. For example:

```
SETVAR COBRUNTIME "string"
```

In the above example "string" is a string of nine either uppercase or lowercase characters representing the run-time options A, C, D, I, M, N, or blank, as shown in Table H-6. A blank in the string is interpreted as "A", or "Abort," the default. Each character position in the string represents a specific trap that you can request, as shown in Table H-7.

Table H-7. Character Position in Specific Traps

Character Position	Trap Type
1	Illegal ASCII or decimal digit.
2	Range error (OCCURS DEPENDING ON identifier, subscript, index, or reference modification out of bounds).
3	No SIZE ERROR phrase.
4	Invalid GO TO.
5	Address Alignment.
6	Paragraph stack overflow (recursive PERFORMs or too many PERFORMs with a common exit point).
7	Leading blanks in a numeric field. If this position contains I, leading blanks in a numeric field are ignored. If this position contains N, leading blanks are changed to zeros. If this position contains a value other than N or I, the action entered in character position 1 is used.
8	Unsigned number in signed numeric field or signed number in unsigned numeric field. If this position contains I, the invalid sign is ignored. If this position contains N, the invalid sign is corrected. If this position contains a value other than N or I, the action entered in character position 1 is used.
9	Only affects a NUMERIC class condition with a PACKED-DECIMAL identifier. If this field contains the character I, then the following conditions do *not* make a NUMERIC test false: ■ A signed value in an unsigned PACKED-DECIMAL field. ■ An unsigned value in a signed PACKED-DECIMAL field. ■ Any invalid sign nibble (half-byte). If this field contains anything other than I, the above conditions make the NUMERIC class condition false.

M can only appear in the first character position, and N can only appear in character positions 1, 7, and 8. This is because the action taken by M and N only applies to an illegal ASCII or decimal digit errors. If either letter appears in any other character position, it is treated as a blank. If character positions 7 or 8 are blank or are not equal to N or I, the action specified in character position 1 is used.

Setting COBRUNTIME to the following will closely simulate HP COBOL II/V actions:

```
SETVAR COBRUNTIME "MCCAAANNI"
```

Example

For example, for a program compiled with the control options VALIDATE and BOUNDS, the following MPE XL command sets COBRUNTIME:

```
SETVAR COBRUNTIME "M␣IDCANNI"
```

The above SETVAR command has the following effects when you run the program:

- Fixes any invalid digits that are found, prints an error message, and continues running (M in position 1).
- Aborts if a trap on an OCCURS DEPENDING ON item, a subscript, an index, or a reference modification goes out of bounds (blank in position 2).
- Ignores any traps that occur on size errors, division by zero, or illegal intrinsic function parameters, if these are used without an ON SIZE ERROR clause (I in position 3).
- Prints an error message and places you in debug mode if an invalid GO TO error occurs (D in position 4).
- Prints an error message and continues on an address alignment trap (C in position 5).
- Aborts on illegal PERFORMs or illegal GO TOs out of performed paragraphs, or on paragraph stack overflow (A in position 6).
- Changes leading blanks in numeric fields to zeros without reporting an error (N in position 7).
- Fixes illegal signs in numeric fields without reporting an error (N in position 8).
- Does not return a false NUMERIC class condition on a PACKED-DECIMAL data item if the sign of the data item is illegal (I in position 9).

Refer to the *HP COBOL II/XL Programmer's Guide* for more discussion and examples of handling errors.

The COBOL Trap Mechanism and Other Languages

In order to work, the COBOL trap mechanism must be armed for the ON SIZE ERROR phrase, $CONTROL VALIDATE command, and $CONTROL BOUNDS command. (The traps must actually be enabled *and* armed. However, in this section, the term *armed* implies enabled and armed. For more details, see the *Trap Handling Programmer's Guide*.)

Besides the HP COBOL II run-time library, the COBOL trap mechanism uses the DEBUG macro file COBMAC.PUB.SYS.

The procedure COBOLTRAP is provided in the HP COBOL II run-time library to arm the COBOL trap mechanism. For an HP COBOL II *main* program, the compiler automatically calls COBOLTRAP. When the COBOL trap mechanism is armed and an error occurs that activates a trap, the HP COBOL II run-time library gains control.

For operating efficiency and compatibility with other HP programming languages, the HP COBOL II/XL compiler does not call COBOLTRAP for COBOL *subprograms*. If an HP COBOL II subprogram is called by a program or subprogram written in a language *other than* COBOL, it is your responsibility to arm the COBOL trap mechanism before the call.

If the COBOL trap mechanism is armed and if a subprogram in another language is called, problems may occur if trap handling is not set for that language. In particular, these problems may occur:

- FORTRAN ON statements may not work.
- Pascal TRY/RECOVER may not work for RANGE errors.
- Business Basic ON statements may not work.

The COBOL trap mechanism will produce a COBOL run-time error instead of the appropriate language error. For example, a Pascal run-time error may be reported as "NO SIZE ERROR PHRASE (COBERR 747)".

The procedure COBOLTRAP has no parameters. To call it, simply code:

```
CALL "COBOLTRAP".
```

The following are recommended programming steps for three possible trap-handling scenarios:

- HP COBOL II programs and subprograms calling an HP COBOL II subprogram:

  Because the COBOL trap mechanism is armed in the HP COBOL II main program, there is no need to call COBOLTRAP in the subprogram.

- HP COBOL II programs and subprograms calling a subprogram written in another language:

  Step 1: Arm or disarm the trap mechanism for the other language.

  Step 2: Issue a CALL statement calling the other language subprogram or procedure.

  Step 3: Arm the COBOL trap mechanism by calling the COBOLTRAP procedure.

- Programs and subprograms written in other languages calling an HP COBOL II subprogram:

  Step 1: Arm the COBOL trap mechanism by calling the COBOLTRAP procedure.

  Step 2: Call the HP COBOL II subprogram.

  Step 3: Arm or disarm the trap mechanism for the other language.

Example 1

This example shows how an HP COBOL II/XL program calling a Pascal procedure can arm the software traps for Pascal and then rearm the traps for HP COBOL II/XL. This example sets the trap mechanism to the default for languages such as Pascal, FORTRAN, and HP C.

```
001000 IDENTIFICATION DIVISION.
001100 PROGRAM-ID. COBOLPROF.
001300 DATA DIVISION.
001400 WORKING-STORAGE SECTION.
001500 77 PARM1    PIC S99V99 COMP.
001600 77 PARM2    PIC X(18).
001700 PROCEDURE DIVISION.
001710 P1.
001800* Step 1: call the procedure pastrap to arm the software trap
001900* for pascal
002000     CALL "PASTRAP".
002100* Step 2: call the pascal procedure pasprog
002200     CALL "PASPROG" USING PARM1 PARM2.
002300* Step 3: arm COBOL traps again
002400     CALL "COBOLTRAP".
002500     STOP RUN.
```

The procedure *Pastrap* is coded as follows:

```
$subprogram$
program example_1;
procedure xaritrap; intrinsic;
procedure xlibtrap; intrinsic;
procedure hpenbltrap; intrinsic;
procedure pastrap;
var oldmask, oldplabel : integer;
begin
xaritrap(0,0,oldmask,oldplabel);       { disarm arithmetic traps }
xlibtrap(0,oldplabel);                 { disarm library traps    }
{ enable all traps but IEEE }
hpenbltrap(hex('fff83fff'),oldmask);   { set to Pascal default   }
end;
begin end.
```

Example 2

This example shows how an HP COBOL II/XL program calling a FORTRAN procedure can arm the software traps for FORTRAN and then rearm the traps for HP COBOL II/XL. This example saves and restores the state of the trap mechanism before and after the call to the non-COBOL subprogram. This method must be used if the called subprogram changes the default trap mechanism. For example, if a FORTRAN subprogram uses an ON statement, this method retains the state of the subprogram's trap mechanism.

```
001000 IDENTIFICATION DIVISION.
001100 PROGRAM-ID. IO888E.
001200 DATA DIVISION.
001300 WORKING-STORAGE SECTION.
001400 1 buffer.
001500   5 c                 pic s9(9) comp value 0.
001600 1 trap-stuff.
001700   5 old-mask          pic s9(9) comp value 0.
001800   5 old-plabel        pic s9(9) comp value 0.
001900*  enable all but IEEE traps
002000   5 old-enblemask     pic s9(9) comp value %37776037777.
002100   5 old-libplabel     pic s9(9) comp value 0.
002200   5 dummy-var         pic s9(9) comp.
002300 procedure division.  house.
002400     perform 2 times
002500        perform restore-fortran-state
002600        call "fortransub" using buffer
002700        perform save-fortran-state
002800        display c
002900        end-perform
003000     stop run.
003100
003200 save-fortran-state.
003300     call intrinsic "XARITRAP" using 0 0 old-mask old-plabel
003400     call intrinsic "HPENBLTRAP" using 0 old-enblemask
003500     call intrinsic "XLIBTRAP" using 0 old-libplabel
003600     call "coboltrap".
003700 restore-fortran-state.
003800     call intrinsic "XARITRAP" using old-mask old-plabel
003900                                     dummy-var dummy-var
004000     call intrinsic "HPENBLTRAP" using old-enblemask dummy-var
004100     call intrinsic "XLIBTRAP" using old-libplabel dummy-var.
```

Example 3

This example shows how a Pascal program calling an HP COBOL II/XL subprogram can arm the software traps for HP COBOL II/XL and then rearm the traps for Pascal.

```
program pasprog;
module cobol_trap_handling;
export
procedure savepascaltraps;
procedure restorepascaltraps;
procedure coboltrap;
implement
{ private save variables for trap handling }
var o_a_mask, o_a_plabel,
    o_l_mask, o_l_plabel,
    o_e_mask : integer;
procedure xaritrap; intrinsic;
procedure xlibtrap; intrinsic;
procedure hpenbltrap; intrinsic;
procedure coboltrap; external;
procedure savepascaltraps;
begin
   { save old values }
   xaritrap(0,0,o_a_mask,o_a_plabel);
   xlibtrap(0,o_l_plabel);
   hpenbltrap(0,o_e_mask);
end;
procedure restorepascaltraps;
var dummy : integer;
begin
   { restore old pascal values }
   xaritrap(o_a_mask,o_a_plabel,dummy,dummy);
   xlibtrap(o_l_plabel,dummy);
   hpenbltrap(o_e_mask,dummy);
end;
end;
import cobol_trap_handling;
var i : integer;
procedure cobolsubprog(var parm1: integer);
   external cobol;
begin
   i:=-33;
   savepascaltraps;
   coboltrap;
   cobolsubprog(i);
   restorepascaltraps;
end.
```

```
001000$CONTROL SUBPROGRAM,OPTFEATURES=LINKALIGNED
001100 IDENTIFICATION DIVISION.
001200 PROGRAM-ID. COBOLSUBPROG.
001300 DATA DIVISION.
001400 WORKING-STORAGE SECTION.
001500 01  J    PIC ----,---,---.
001600 LINKAGE SECTION.
001700 01  I    PIC S9(9)   BINARY.
001800 PROCEDURE DIVISION USING I.
001900 P1.
002000     MOVE I TO J.
002100     DISPLAY "INPUT PARM WAS" J.
```

Example HP COBOL II/XL Program

This section contains a complete COBOL program listing from the HP COBOL II/XL compiler.

```
PAGE 0001         COBOL II/XL   HP31500A.04.03  [85]    FRI, JUN  7, 1991,  4:02
                 PM     Copyright Hewlett-Packard CO. 1987

 00001          001000$CONTROL MAP,SOURCE,CROSSREF,VERBS
 00002          002000 IDENTIFICATION DIVISION.
 00003          003000 PROGRAM-ID.                      EXAMPLE.
 00004          004000 AUTHOR.                          HEWLETT-PACKARD.
 00005          005000 DATE-COMPILED.  FRI, JUN  7, 1991,  4:02 PM
 00006          006000*********************************************************
 00007          007000*   BRIEF PROGRAM DESCRIPTION
 00008          008000*
 00009          009000*    THIS IS AN EXAMPLE OF THE USE OF COBOL85.  THIS IS A
 00010          010000*  SEQUENTIAL UPDATE PROGRAM USING STRUCTURED PROGRAMMING
 00011          011000*  TECHNIQUES.  THE TRANSACTION FILE USED BY THE UPDATE
 00012          012000*  PROGRAM HAS ALREADY BEEN EDITED AND SORTED INTO THE
 00013          013000*  PROPER SEQUENCE FOR UPDATE PROCESSING.
 00014          014000*********************************************************
 00015          015000*   FILE REQUIREMENTS
 00016          016000*
 00017          017000*   COPY FILES: NONE REQUIRED IN THIS PROGRAM
 00018          018000*
 00019          019000*   DATA FILES--INPUT FILES:  OLD INVENTORY MASTER
 00020          020000*                             UPDATE TRANSACTIONS
 00021          021000*               OUTPUT FILES: NEW INVENTORY MASTER
 00022          022000*                             TRANSACTION ERROR FILE
 00023          023000*                             PRINTED REPORT OF UPDATE
 00024          024000*               I-O    FILES: NONE
 00025          025000*
 00026          026000*********************************************************
 00027          027000 ENVIRONMENT DIVISION.
 00028          028000 INPUT-OUTPUT SECTION.
 00029          029000 FILE-CONTROL.
 00030          030000     SELECT OLD-INV-MAST
 00031          031000         ASSIGN TO "OLDMAST".
 00032          032000     SELECT NEW-INV-MAST
 00033          033000         ASSIGN TO "NEWMAST".
 00034          034000     SELECT TRAN-FILE
 00035          035000         ASSIGN TO "TRANFILE".
 00036          036000     SELECT ERROR-FILE
 00037          037000         ASSIGN TO "TRANERR".
 00038          038000     SELECT PRINT-FILE
 00039          039000         ASSIGN TO "PRINT,UR".
```

```
PAGE 0002/COBTEXT  EXAMPLE
 00040          040000/ ******  D A T A   D I V I S I O N  ******
 00041          041000 DATA DIVISION.
 00042          042000 FILE SECTION.
 00043          043000*
 00044          044000 FD  OLD-INV-MAST.
 00045          045000*
 00046          046000 01  OLD-INV-MAST-REC.
 00047          047000     03  OM-PART-NBR         PIC  X(05).
 00048          048000     03  FILLER              PIC  X(35).
 00049          049000*
 00050          050000 FD  NEW-INV-MAST.
 00051          051000*
 00052          052000 01  NEW-INV-MAST-REC        PIC  X(40).
 00053          053000*
 00054          054000 FD  TRAN-FILE.
 00055          055000*
 00056          056000 01  TRAN-REC.
 00057          057000     03  TR-UPDATE-CODE          PIC  X(01).
 00058          058000         88  TR-ADD-CODE                  VALUE "A".
 00059          059000         88  TR-CHANGE-CODE               VALUE "C".
 00060          060000         88  TR-DELETE-CODE               VALUE "D".
 00061          061000     03  TR-PART-NBR             PIC  X(05).
 00062          062000     03  TR-DESCRIPTION          PIC  X(25).
 00063          063000     03  TR-PART-COST-FLD.
 00064          064000         05  TR-PART-COST        PIC  9(07)V99.
 00065          065000     03  TR-PART-PRICE-FLD.
 00066          066000         05  TR-PART-PRICE       PIC  9(05)V99.
 00067          067000     03  TR-PART-QUANTITY-FLD.
 00068          068000         05  TR-PART-QUANTITY    PIC  9(04).
 00069          069000*
 00070          070000 FD  ERROR-FILE.
 00071          071000 01  ERROR-REC               PIC  X(51).
 00072          072000*
 00073          073000 FD  PRINT-FILE.
 00074          074000 01  PRINT-REC               PIC  X(132).
```

```
PAGE 0003/COBTEXT  EXAMPLE
 00075          075000/ ******  W O R K I N G   S T O R A G E  ******
 00076          076000 WORKING-STORAGE SECTION.
 00077          077000*
 00078          078000*
 00079          079000 01  WS-PRINT-CONTROL.
 00080          080000     03  WS-LINE-CTR         PIC S9(03) BINARY VALUE 999.
 00081          081000     03  WS-PAGE-CTR         PIC S9(03) BINARY VALUE 0.
 00082          082000     03  WS-SPACING          PIC S9(01) BINARY VALUE 1.
 00083          083000     03  WS-LINE-LMT         PIC S9(03) BINARY VALUE 45.
 00084          084000*
 00085          085000 01  WS-ACCUMULATORS.
 00086          086000     03  WS-CHANGES-CTR      PIC S9(05) BINARY.
 00087          087000     03  WS-ADDITIONS-CTR    PIC S9(05) BINARY.
 00088          088000     03  WS-DELETES-CTR      PIC S9(05) BINARY.
 00089          089000     03  WS-TOTAL-CTR        PIC S9(05) BINARY.
 00090          090000     03  WS-ERRORS           PIC S9(05) BINARY.
 00091          091000     03  WS-TRANS-READ       PIC S9(05) BINARY.
 00092          092000*
 00093          093000 01  WS-UPDT-MESSAGES.
 00094          094000     03  WS-CHANGE-MSG       PIC  X(10) VALUE "CHANGED".
 00095          095000     03  WS-ADDITION-MSG     PIC  X(10) VALUE "ADDED".
 00096          096000     03  WS-DELETE-MSG       PIC  X(10) VALUE "DELETED".
 00097          097000*
 00098          098000 01  WS-MASTER-REC.
 00099          099000     03  WS-MR-PART-NBR      PIC  X(05).
 00100          100000     03  WS-MR-DESCRIPTION   PIC  X(25).
 00101          101000     03  WS-MR-PART-COST     PIC S9(07)V99 BINARY SYNC.
 00102          102000     03  WS-MR-PART-PRICE    PIC S9(05)V99 BINARY SYNC.
 00103          103000     03  WS-MR-PART-QUANTITY PIC S9(04)    BINARY SYNC.
 00104          104000*
 00105          105000 01  HDG-1.
 00106          106000     03  HDG1-DATE          PIC  X(08).
 00107          107000     03                     PIC  X(22) VALUE SPACES.
 00108          108000     03  HDG1-REPORT-NAME   PIC  X(24)
 00109          109000         VALUE "INVENTORY UPDATE LISTING".
 00110          110000     03                     PIC  X(20) VALUE SPACES.
 00111          111000     03                     PIC  X(06) VALUE "PAGE  ".
 00112          112000     03  HDG1-PAGE-NBR      PIC  ZZ9.
 00113          113000*
 00114          114000 01  HDG-2.
 00115          115000     03                     PIC  X(20)
 00116          116000         VALUE "PART     PART      ".
 00117          117000     03                     PIC  X(20)
 00118          118000         VALUE "                   ".
 00119          119000     03                     PIC  X(20)
 00120          120000         VALUE "      PART        PA".
 00121          121000     03                     PIC  X(20)
 00122          122000         VALUE "RT       PART    UPD".
 00123          123000     03                     PIC  X(20)
 00124          124000         VALUE "ATE                ".
 00125          125000     03                     PIC  X(20)
 00126          126000         VALUE "                   ".
 00127          127000     03                     PIC  X(13)
 00128          128000         VALUE "             ".
 00129          129000*
 00130          130000 01  HDG-3.
 00131          131000     03                     PIC  X(20)
```

```
PAGE 0004/COBTEXT  EXAMPLE
 00132          132000          VALUE "NUMBER   DESCRIPTION".
 00133          133000     03                      PIC  X(20)
 00134          134000          VALUE "                  ".
 00135          135000     03                      PIC  X(20)
 00136          136000          VALUE "      COST        PR".
 00137          137000     03                      PIC  X(20)
 00138          138000          VALUE "ICE   QUANTITY   MES".
 00139          139000     03                      PIC  X(20)
 00140          140000          VALUE "SAGE              ".
 00141          141000     03                      PIC  X(20)
 00142          142000          VALUE "                  ".
 00143          143000     03                      PIC  X(13)
 00144          144000          VALUE "            ".
 00145          145000*
 00146          146000 01  TOTALS-HDG-1.
 00147          147000     03                      PIC  X(20)
 00148          148000          VALUE "TOTALS FOR INVENTORY".
 00149          149000     03                      PIC  X(20)
 00150          150000          VALUE " UPDATE RUN OF  -   ".
 00151          151000     03  TOT1-HDG-DATE       PIC  X(08).
 00152          152000*
 00153          153000 01  TOTALS-HDG-2.
 00154          154000     03                      PIC  X(20)
 00155          155000          VALUE "CHANGES           ".
 00156          156000     03  TOT2-CHANGES        PIC  ZZ,ZZ9.
 00157          157000*
 00158          158000 01  TOTALS-HDG-3.
 00159          159000     03                      PIC  X(20)
 00160          160000          VALUE "ADDITIONS         ".
 00161          161000     03  TOT3-ADDITIONS      PIC  ZZ,ZZ9.
 00162          162000*
 00163          163000 01  TOTALS-HDG-4.
 00164          164000     03                      PIC  X(20)
 00165          165000          VALUE "DELETIONS         ".
 00166          166000     03  TOT4-DELETIONS      PIC  ZZ,ZZ9.
 00167          167000*
 00168          168000 01  TOTALS-HDG-5.
 00169          169000     03                      PIC  X(20)
 00170          170000          VALUE "TOTAL UPDATES     ".
 00171          171000     03  TOT5-UPDATES        PIC  ZZ,ZZ9.
 00172          172000*
 00173          173000 01  TOTALS-HDG-6.
 00174          174000     03                      PIC  X(20)
 00175          175000          VALUE "ERRORS            ".
 00176          176000     03  TOT6-ERRORS         PIC  ZZ,ZZ9.
 00177          177000*
 00178          178000 01  TOTALS-HDG-7.
 00179          179000     03                      PIC  X(20)
 00180          180000          VALUE "TOTAL TRANSACTIONS  ".
 00181          181000     03  TOT7-TRANS-READ     PIC ZZ,ZZ9.
 00182          182000*
 00183          183000 01  WS-UPDATE-LINE.
 00184          184000     03  WS-UP-PART-NBR       PIC  X(05).
 00185          185000     03                       PIC  X(04) VALUE SPACES.
 00186          186000     03  WS-UP-DESCRIPTION    PIC  X(25).
 00187          187000     03                       PIC  X(04) VALUE SPACES.
 00188          188000     03  WS-UP-PART-COST      PIC  Z,ZZZ,ZZZ.99-.
```

```
PAGE 0005/COBTEXT  EXAMPLE
 00189          189000     03                      PIC  X(03) VALUE SPACES.
 00190          190000     03  WS-UP-PART-PRICE    PIC  ZZ,ZZZ.99-.
 00191          191000     03                      PIC  X(05) VALUE SPACES.
 00192          192000     03  WS-UP-PART-QUANTITY PIC  ZZZ9.
 00193          193000     03                      PIC  X(04) VALUE SPACES.
 00194          194000     03  WS-UP-UPDT-MESSAGE  PIC  X(10).
 00195          195000*

PAGE 0006/COBTEXT  EXAMPLE
 00196          196000/  ******  P R O C E D U R E   D I V I S I O N  *****
 00197          197000 PROCEDURE DIVISION.
 00198          198000 100-MAIN-PROGRAM.
 00199          199000     OPEN INPUT  OLD-INV-MAST
 00200          200000                 TRAN-FILE
 00201          201000          OUTPUT NEW-INV-MAST
 00202          202000                 ERROR-FILE
 00203          203000                 PRINT-FILE
 00204          204000*
 00205          205000     MOVE SPACES       TO PRINT-REC
 00206          206000     MOVE CURRENT-DATE TO HDG1-DATE
 00207          207000                          TOT1-HDG-DATE
 00208          208000     INITIALIZE WS-ACCUMULATORS WS-UPDATE-LINE
 00209          209000     PERFORM 300-GET-TRANSACTION
 00210          210000     PERFORM 310-GET-OLD-MASTER
 00211          211000*
 00212          212000*  M A I N   P R O G R A M    D R I V E R
 00213          213000*
 00214          214000     PERFORM UNTIL WS-MR-PART-NBR EQUAL ALL "9" AND
 00215          215000                   TR-PART-NBR    EQUAL ALL "9"
 00216          216000
 00217          217000         EVALUATE TRUE
 00218          218000            WHEN WS-MR-PART-NBR GREATER TR-PART-NBR
 00219          219000                PERFORM 210-MASTER-COMPARED-HIGH
 00220          220000            WHEN WS-MR-PART-NBR LESS    TR-PART-NBR
 00221          221000                PERFORM 240-MASTER-COMPARED-LOW
 00222          222000            WHEN WS-MR-PART-NBR = ALL "9"
 00223          223000                CONTINUE
 00224          224000            WHEN OTHER
 00225          225000                PERFORM 250-MASTER-AND-TRAN-EQUAL
 00226          226000         END-EVALUATE
 00227          227000     END-PERFORM
 00228          228000*
 00229          229000* PRINT TOTALS AND QUIT
 00230          230000*
 00231          231000     PERFORM 420-PRINT-TOTALS
 00232          232000     CLOSE OLD-INV-MAST
 00233          233000           NEW-INV-MAST
 00234          234000           TRAN-FILE
 00235          235000           ERROR-FILE
 00236          236000           PRINT-FILE
 00237          237000     STOP RUN.
 00238          238000*
```

```
PAGE 0007/COBTEXT  EXAMPLE
 00239          239000/
 00240          240000 210-MASTER-COMPARED-HIGH.
 00241          241000*
 00242          242000     IF  TR-ADD-CODE
 00243          243000         PERFORM 220-ADD-TO-MASTER
 00244          244000     ELSE
 00245          245000         PERFORM 230-TRAN-IN-ERROR.
 00246          246000*
 00247          247000 220-ADD-TO-MASTER.
 00248          248000     MOVE TR-PART-NBR      TO WS-MR-PART-NBR
 00249          249000     MOVE TR-DESCRIPTION   TO WS-MR-DESCRIPTION
 00250          250000     MOVE TR-PART-COST     TO WS-MR-PART-COST
 00251          251000     MOVE TR-PART-PRICE    TO WS-MR-PART-PRICE
 00252          252000     MOVE TR-PART-QUANTITY TO WS-MR-PART-QUANTITY
 00253          253000     MOVE WS-MASTER-REC    TO NEW-INV-MAST-REC
 00254          254000     MOVE TR-PART-NBR      TO WS-UP-PART-NBR
 00255          255000     MOVE TR-DESCRIPTION   TO WS-UP-DESCRIPTION
 00256          256000     MOVE TR-PART-COST     TO WS-UP-PART-COST
 00257          257000     MOVE TR-PART-PRICE    TO WS-UP-PART-PRICE
 00258          258000     MOVE TR-PART-QUANTITY TO WS-UP-PART-QUANTITY
 00259          259000     MOVE WS-ADDITION-MSG  TO WS-UP-UPDT-MESSAGE
 00260          260000     PERFORM 300-GET-TRANSACTION
 00261          261000     PERFORM 330-WRITE-NEW-MASTER
 00262          262000     PERFORM 320-PRINT-UPDATE
 00263          263000     MOVE OLD-INV-MAST-REC TO WS-MASTER-REC
 00264          264000     ADD 1                 TO WS-ADDITIONS-CTR.
 00265          265000*
 00266          266000 230-TRAN-IN-ERROR.
 00267          267000     MOVE TRAN-REC TO ERROR-REC
 00268          268000     WRITE ERROR-REC
 00269          269000     PERFORM 300-GET-TRANSACTION
 00270          270000     ADD 1 TO WS-ERRORS.
 00271          271000*
 00272          272000 240-MASTER-COMPARED-LOW.
 00273          273000     MOVE WS-MASTER-REC TO NEW-INV-MAST-REC
 00274          274000     PERFORM 330-WRITE-NEW-MASTER
 00275          275000     PERFORM 310-GET-OLD-MASTER.
 00276          276000*
 00277          277000 250-MASTER-AND-TRAN-EQUAL.
 00278          278000     EVALUATE TRUE
 00279          279000         WHEN TR-DELETE-CODE
 00280          280000             PERFORM 260-DELETE-MASTER
 00281          281000         WHEN TR-CHANGE-CODE
 00282          282000             PERFORM 270-CHANGE-MASTER
 00283          283000         WHEN OTHER
 00284          284000             PERFORM 230-TRAN-IN-ERROR
 00285          285000     END-EVALUATE.
 00286          286000*
 00287          287000 260-DELETE-MASTER.
 00288          288000     MOVE WS-MR-PART-NBR      TO WS-UP-PART-NBR
 00289          289000     MOVE WS-MR-DESCRIPTION   TO WS-UP-DESCRIPTION
 00290          290000     MOVE WS-MR-PART-COST     TO WS-UP-PART-COST
 00291          291000     MOVE WS-MR-PART-PRICE    TO WS-UP-PART-PRICE
 00292          292000     MOVE WS-MR-PART-QUANTITY TO WS-UP-PART-QUANTITY
 00293          293000     MOVE WS-DELETE-MSG       TO WS-UP-UPDT-MESSAGE
 00294          294000     PERFORM 320-PRINT-UPDATE
 00295          295000     PERFORM 310-GET-OLD-MASTER
```

```
PAGE 0008/COBTEXT  EXAMPLE
 00296          296000     PERFORM 300-GET-TRANSACTION
 00297          297000     ADD 1                    TO WS-DELETES-CTR.
 00298          298000*
 00299          299000 270-CHANGE-MASTER.
 00300          300000*
 00301          301000     IF  TR-DESCRIPTION  SPACES
 00302          302000         MOVE TR-DESCRIPTION   TO WS-MR-DESCRIPTION
 00303          303000         END-IF
 00304          304000     IF  TR-PART-COST-FLD  SPACES
 00305          305000         MOVE TR-PART-COST     TO WS-MR-PART-COST
 00306          306000         END-IF
 00307          307000     IF  TR-PART-PRICE-FLD  SPACES
 00308          308000         MOVE TR-PART-PRICE    TO WS-MR-PART-PRICE
 00309          309000         END-IF
 00310          310000     IF  TR-PART-QUANTITY-FLD  SPACES
 00311          311000         MOVE TR-PART-QUANTITY TO WS-MR-PART-QUANTITY
 00312          312000         END-IF
 00313          313000*
 00314          314000     MOVE WS-MR-PART-NBR      TO WS-UP-PART-NBR
 00315          315000     MOVE WS-MR-DESCRIPTION   TO WS-UP-DESCRIPTION
 00316          316000     MOVE WS-MR-PART-COST     TO WS-UP-PART-COST
 00317          317000     MOVE WS-MR-PART-PRICE    TO WS-UP-PART-PRICE
 00318          318000     MOVE WS-MR-PART-QUANTITY TO WS-UP-PART-QUANTITY
 00319          319000     MOVE WS-CHANGE-MSG       TO WS-UP-UPDT-MESSAGE
 00320          320000     PERFORM 320-PRINT-UPDATE
 00321          321000     PERFORM 300-GET-TRANSACTION
 00322          322000     ADD 1                    TO WS-CHANGES-CTR.
 00323          323000*
 00324          324000 300-GET-TRANSACTION.
 00325          325000     READ TRAN-FILE
 00326          326000       AT END
 00327          327000         MOVE ALL "9" TO TR-PART-NBR
 00328          328000       NOT AT END
 00329          329000         ADD 1 TO WS-TRANS-READ
 00330          330000     END-READ.
 00331          331000*
 00332          332000 310-GET-OLD-MASTER.
 00333          333000     READ OLD-INV-MAST
 00334          334000       AT END
 00335          335000         MOVE ALL "9" TO OM-PART-NBR
 00336          336000     END-READ
 00337          337000*
 00338          338000     MOVE OLD-INV-MAST-REC TO WS-MASTER-REC.
 00339          339000*
 00340          340000 320-PRINT-UPDATE.
 00341          341000*
 00342          342000     IF  WS-LINE-CTR GREATER WS-LINE-LMT
 00343          343000         PERFORM 410-PRINT-HEADING
 00344          344000         END-IF
 00345          345000*
 00346          346000     MOVE WS-UPDATE-LINE TO PRINT-REC
 00347          347000     MOVE 1 TO WS-SPACING
 00348          348000     PERFORM 400-WRITE-PRINT-LINE
 00349          349000     ADD 1 TO WS-LINE-CTR.
 00350          350000*
 00351          351000 330-WRITE-NEW-MASTER.
 00352          352000     WRITE NEW-INV-MAST-REC.
```

```
PAGE 0009/COBTEXT  EXAMPLE
 00353          353000*
 00354          354000 400-WRITE-PRINT-LINE.
 00355          355000     WRITE PRINT-REC BEFORE ADVANCING WS-SPACING
 00356          356000     MOVE SPACES    TO PRINT-REC
 00357          357000     ADD WS-SPACING TO WS-LINE-CTR.
 00358          358000*
 00359          359000 410-PRINT-HEADING.
 00360          360000     WRITE PRINT-REC BEFORE ADVANCING PAGE
 00361          361000     MOVE ZEROES      TO WS-LINE-CTR
 00362          362000     ADD 1            TO WS-PAGE-CTR
 00363          363000     MOVE WS-PAGE-CTR TO HDG1-PAGE-NBR
 00364          364000     MOVE 2           TO WS-SPACING
 00365          365000     MOVE HDG-1       TO PRINT-REC
 00366          366000     PERFORM 400-WRITE-PRINT-LINE
 00367          367000     MOVE 1           TO WS-SPACING
 00368          368000     MOVE HDG-2       TO PRINT-REC
 00369          369000     PERFORM 400-WRITE-PRINT-LINE
 00370          370000     MOVE 2           TO WS-SPACING
 00371          371000     MOVE HDG-3       TO PRINT-REC
 00372          372000     PERFORM 400-WRITE-PRINT-LINE.
 00373          373000*
 00374          374000 420-PRINT-TOTALS.
 00375          375000     ADD WS-CHANGES-CTR WS-ADDITIONS-CTR WS-DELETES-CTR
 00376          376000         GIVING WS-TOTAL-CTR
 00377          377000*
 00378          378000     MOVE 1                TO WS-SPACING
 00379          379000     MOVE WS-CHANGES-CTR   TO TOT2-CHANGES
 00380          380000     MOVE WS-ADDITIONS-CTR TO TOT3-ADDITIONS
 00381          381000     MOVE WS-DELETES-CTR   TO TOT4-DELETIONS
 00382          382000     MOVE WS-TOTAL-CTR     TO TOT5-UPDATES
 00383          383000     MOVE WS-ERRORS        TO TOT6-ERRORS
 00384          384000     MOVE WS-TRANS-READ    TO TOT7-TRANS-READ
 00385          385000     WRITE PRINT-REC BEFORE ADVANCING PAGE
 00386          386000*
 00387          387000     MOVE TOTALS-HDG-1 TO PRINT-REC
 00388          388000     PERFORM 400-WRITE-PRINT-LINE
 00389          389000     MOVE TOTALS-HDG-2 TO PRINT-REC
 00390          390000     PERFORM 400-WRITE-PRINT-LINE
 00391          391000     MOVE TOTALS-HDG-3 TO PRINT-REC
 00392          392000     PERFORM 400-WRITE-PRINT-LINE
 00393          393000     MOVE TOTALS-HDG-4 TO PRINT-REC
 00394          394000     PERFORM 400-WRITE-PRINT-LINE
 00395          395000     MOVE TOTALS-HDG-5 TO PRINT-REC
 00396          396000     PERFORM 400-WRITE-PRINT-LINE
 00397          397000     MOVE TOTALS-HDG-6 TO PRINT-REC
 00398          398000     PERFORM 400-WRITE-PRINT-LINE
 00399          399000     MOVE TOTALS-HDG-7 TO PRINT-REC
 00400          400000     PERFORM 400-WRITE-PRINT-LINE.
 00401          401000*
 00402          402000
```

```
PAGE 0010/COBTEXT  EXAMPLE         SYMBOL TABLE MAP
 LINE# LVL SOURCE NAME                    BASE  OFFSET     SIZE   USAGE     CATE
               GORY R O J BZ

 FILE SECTION

 00030 FD  OLD-INV-MAST                   DP+        24        C0 SEQUENTIAL
 00046 01    OLD-INV-MAST-REC             DP+       504        28 DISP      AN
 00047 03    OM-PART-NBR                  DP+       504         5 DISP      AN
 00048 03    FILLER                       DP+       509        23 DISP      AN
 00032 FD  NEW-INV-MAST                   DP+        E4        C0 SEQUENTIAL
 00052 01    NEW-INV-MAST-REC             DP+       4DC        28 DISP      AN
 00034 FD  TRAN-FILE                      DP+       1A4        C0 SEQUENTIAL
 00056 01    TRAN-REC                     DP+       4A8        33 DISP      AN
 00057 03    TR-UPDATE-CODE               DP+       4A8         1 DISP      AN
 00058 88    TR-ADD-CODE
 00059 88    TR-CHANGE-CODE
 00060 88    TR-DELETE-CODE
 00061 03    TR-PART-NBR                  DP+       4A9         5 DISP      AN
 00062 03    TR-DESCRIPTION               DP+       4AE        19 DISP      AN
 00063 03    TR-PART-COST-FLD             DP+       4C7         9 DISP      AN
 00064 05    TR-PART-COST                 DP+       4C7         9 DISP      N
 00065 03    TR-PART-PRICE-FLD            DP+       4D0         7 DISP      AN
 00066 05    TR-PART-PRICE                DP+       4D0         7 DISP      N
 00067 03    TR-PART-QUANTITY-FLD         DP+       4D7         4 DISP      AN
 00068 05    TR-PART-QUANTITY             DP+       4D7         4 DISP      N
 00036 FD  ERROR-FILE                     DP+       264        C0 SEQUENTIAL
 00071 01    ERROR-REC                    DP+       474        33 DISP      AN
 00038 FD  PRINT-FILE                     DP+       324        C0 SEQUENTIAL
 00074 01    PRINT-REC                    DP+       3F0        84 DISP      AN

 WORKING-STORAGE SECTION

 00079 01  WS-PRINT-CONTROL              DP+       530         8 DISP      AN
 00080 03    WS-LINE-CTR                  DP+       530         2 COMP      NS
 00081 03    WS-PAGE-CTR                  DP+       532         2 COMP      NS
 00082 03    WS-SPACING                   DP+       534         2 COMP      NS
 00083 03    WS-LINE-LMT                  DP+       536         2 COMP      NS
 00085 01  WS-ACCUMULATORS               DP+       538        18 DISP      AN
 00086 03    WS-CHANGES-CTR               DP+       538         4 COMP      NS
 00087 03    WS-ADDITIONS-CTR             DP+       53C         4 COMP      NS
 00088 03    WS-DELETES-CTR               DP+       540         4 COMP      NS
 00089 03    WS-TOTAL-CTR                 DP+       544         4 COMP      NS
 00090 03    WS-ERRORS                    DP+       548         4 COMP      NS
 00091 03    WS-TRANS-READ                DP+       54C         4 COMP      NS
 00093 01  WS-UPDT-MESSAGES              DP+       550        1E DISP      AN
 00094 03    WS-CHANGE-MSG                DP+       550         A DISP      AN
 00095 03    WS-ADDITION-MSG              DP+       55A         A DISP      AN
 00096 03    WS-DELETE-MSG                DP+       564         A DISP      AN
 00098 01  WS-MASTER-REC                 DP+       570        2A DISP      AN
 00099 03    WS-MR-PART-NBR               DP+       570         5 DISP      AN
 00100 03    WS-MR-DESCRIPTION            DP+       575        19 DISP      AN
 00101 03    WS-MR-PART-COST              DP+       590         4 COMP-SYNC NS
 00102 03    WS-MR-PART-PRICE             DP+       594         4 COMP-SYNC NS
 00103 03    WS-MR-PART-QUANTITY          DP+       598         2 COMP-SYNC NS
 00105 01  HDG-1                         DP+       59C        53 DISP      AN
 00106 03    HDG1-DATE                    DP+       59C         8 DISP      AN
 00107 03    FILLER                       DP+       5A4        16 DISP      AN
```

```
PAGE 0011/COBTEXT  EXAMPLE         SYMBOL TABLE MAP
 LINE# LVL SOURCE NAME                    BASE  OFFSET     SIZE   USAGE     CATE
               GORY R O J BZ

 00108 03   HDG1-REPORT-NAME              DP+       5BA        18 DISP      AN
 00110 03   FILLER                        DP+       5D2        14 DISP      AN
 00111 03   FILLER                        DP+       5E6         6 DISP      AN
 00112 03   HDG1-PAGE-NBR                 DP+       5EC         3 DISP      NE
 00114 01  HDG-2                          DP+       5F0        85 DISP      AN
 00115 03   FILLER                        DP+       5F0        14 DISP      AN
 00117 03   FILLER                        DP+       604        14 DISP      AN
 00119 03   FILLER                        DP+       618        14 DISP      AN
 00121 03   FILLER                        DP+       62C        14 DISP      AN
 00123 03   FILLER                        DP+       640        14 DISP      AN
 00125 03   FILLER                        DP+       654        14 DISP      AN
 00127 03   FILLER                        DP+       668         D DISP      AN
 00130 01  HDG-3                          DP+       678        85 DISP      AN
 00131 03   FILLER                        DP+       678        14 DISP      AN
 00133 03   FILLER                        DP+       68C        14 DISP      AN
 00135 03   FILLER                        DP+       6A0        14 DISP      AN
 00137 03   FILLER                        DP+       6B4        14 DISP      AN
 00139 03   FILLER                        DP+       6C8        14 DISP      AN
 00141 03   FILLER                        DP+       6DC        14 DISP      AN
 00143 03   FILLER                        DP+       6F0         D DISP      AN
 00146 01  TOTALS-HDG-1                   DP+       700        30 DISP      AN
 00147 03   FILLER                        DP+       700        14 DISP      AN
 00149 03   FILLER                        DP+       714        14 DISP      AN
 00151 03   TOT1-HDG-DATE                 DP+       728         8 DISP      AN
 00153 01  TOTALS-HDG-2                   DP+       730        1A DISP      AN
 00154 03   FILLER                        DP+       730        14 DISP      AN
 00156 03   TOT2-CHANGES                  DP+       744         6 DISP      NE
 00158 01  TOTALS-HDG-3                   DP+       74C        1A DISP      AN
 00159 03   FILLER                        DP+       74C        14 DISP      AN
 00161 03   TOT3-ADDITIONS                DP+       760         6 DISP      NE
 00163 01  TOTALS-HDG-4                   DP+       768        1A DISP      AN
 00164 03   FILLER                        DP+       768        14 DISP      AN
 00166 03   TOT4-DELETIONS                DP+       77C         6 DISP      NE
 00168 01  TOTALS-HDG-5                   DP+       784        1A DISP      AN
 00169 03   FILLER                        DP+       784        14 DISP      AN
 00171 03   TOT5-UPDATES                  DP+       798         6 DISP      NE
 00173 01  TOTALS-HDG-6                   DP+       7A0        1A DISP      AN
 00174 03   FILLER                        DP+       7A0        14 DISP      AN
 00176 03   TOT6-ERRORS                   DP+       7B4         6 DISP      NE
 00178 01  TOTALS-HDG-7                   DP+       7BC        1A DISP      AN
 00179 03   FILLER                        DP+       7BC        14 DISP      AN
 00181 03   TOT7-TRANS-READ               DP+       7D0         6 DISP      NE
 00183 01  WS-UPDATE-LINE                 DP+       7D8        57 DISP      AN
 00184 03   WS-UP-PART-NBR                DP+       7D8         5 DISP      AN
 00185 03   FILLER                        DP+       7DD         4 DISP      AN
 00186 03   WS-UP-DESCRIPTION             DP+       7E1        19 DISP      AN
 00187 03   FILLER                        DP+       7FA         4 DISP      AN
 00188 03   WS-UP-PART-COST               DP+       7FE         D DISP      NE
 00189 03   FILLER                        DP+       80B         3 DISP      AN
 00190 03   WS-UP-PART-PRICE              DP+       80E         A DISP      NE
 00191 03   FILLER                        DP+       818         5 DISP      AN
 00192 03   WS-UP-PART-QUANTITY           DP+       81D         4 DISP      NE
 00193 03   FILLER                        DP+       821         4 DISP      AN
 00194 03   WS-UP-UPDT-MESSAGE            DP+       825         A DISP      AN
```

```
PAGE 0012/COBTEXT  EXAMPLE         SYMBOL TABLE MAP
 LINE# LVL SOURCE NAME                    BASE  OFFSET     SIZE   USAGE     CATE
               GORY R O J BZ

 STORAGE LAYOUT               (#ENTRYS)

           FIRST TIME FLAG, etc.          DP+         8         4
           Global USE area                DP+        10         C
           RUN TIME $ . ,                 DP+        1C         4
           SORT/MERGE PLABEL              DP+        20         4
           FILE TABLE            (5)      DP+        24       3C0
           TALLY                          DP+       3E8         4
           USER STORAGE                   DP+       3E8       447
           TEMPCELL pool                  SP        -48         C
           Constant pool     C$           CODE        0        40
           Literal pool ~    S$           CODE        0        6C

PAGE 0013/COBTEXT  EXAMPLE         CROSS REFERENCE LISTING
                                         IDENTIFIERS
ERROR-FILE                         00036   00070   00202   00235
ERROR-REC                          00071   00267   00268
HDG-1                              00105   00365
HDG-2                              00114   00368
HDG-3                              00130   00371
HDG1-DATE                          00106   00206
HDG1-PAGE-NBR                      00112   00363
HDG1-REPORT-NAME                   00108
NEW-INV-MAST                       00032   00050   00201   00233
NEW-INV-MAST-REC                   00052   00253   00273   00352
OLD-INV-MAST                       00030   00044   00199   00232   00333
OLD-INV-MAST-REC                   00046   00263   00338
OM-PART-NBR                        00047   00335
PRINT-FILE                         00038   00073   00203   00236
PRINT-REC                          00074   00205   00346   00355   00356   00360
                  00365   00368   00371   00385   00387   00389
   00391   00393   00395   00397   00399
TALLY                              00000
TOT1-HDG-DATE                      00151   00207
TOT2-CHANGES                       00156   00379
TOT3-ADDITIONS                     00161   00380
TOT4-DELETIONS                     00166   00381
TOT5-UPDATES                       00171   00382
TOT6-ERRORS                        00176   00383
TOT7-TRANS-READ                    00181   00384
TOTALS-HDG-1                       00146   00387
TOTALS-HDG-2                       00153   00389
TOTALS-HDG-3                       00158   00391
TOTALS-HDG-4                       00163   00393
TOTALS-HDG-5                       00168   00395
TOTALS-HDG-6                       00173   00397
TOTALS-HDG-7                       00178   00399
TR-ADD-CODE                        00058   00242
TR-CHANGE-CODE                     00059   00281
TR-DELETE-CODE                     00060   00279
TR-DESCRIPTION                     00062   00249   00255   00301   00302
TR-PART-COST                       00064   00250   00256   00305
TR-PART-COST-FLD                   00063   00304
TR-PART-NBR                        00061   00215   00218   00220   00248   00254
                  00327
TR-PART-PRICE                      00066   00251   00257   00308
TR-PART-PRICE-FLD                  00065   00307
TR-PART-QUANTITY                   00068   00252   00258   00311
TR-PART-QUANTITY-FLD               00067   00310
TR-UPDATE-CODE                     00057
TRAN-FILE                          00034   00054   00200   00234   00325
TRAN-REC                           00056   00267
```

```
WS-ACCUMULATORS                  00085   00208
WS-ADDITION-MSG                  00095   00259
WS-ADDITIONS-CTR                 00087   00264   00375   00380
WS-CHANGE-MSG                    00094   00319
WS-CHANGES-CTR                   00086   00322   00375   00379
WS-DELETE-MSG                    00096   00293
WS-DELETES-CTR                   00088   00297   00375   00381
WS-ERRORS                        00090   00270   00383
WS-LINE-CTR                      00080   00342   00349   00357   00361
WS-LINE-LMT                      00083   00342

PAGE 0014/COBTEXT  EXAMPLE       CROSS REFERENCE LISTING
                                       IDENTIFIERS
WS-MASTER-REC                    00098   00253   00263   00273   00338
WS-MR-DESCRIPTION                00100   00249   00289   00302   00315
WS-MR-PART-COST                  00101   00250   00290   00305   00316
WS-MR-PART-NBR                   00099   00214   00218   00220   00222   00248
                  00288   00314
WS-MR-PART-PRICE                 00102   00251   00291   00308   00317
WS-MR-PART-QUANTITY              00103   00252   00292   00311   00318
WS-PAGE-CTR                      00081   00362   00363
WS-PRINT-CONTROL                 00079
WS-SPACING                       00082   00347   00355   00357   00364   00367
                  00370   00378
WS-TOTAL-CTR                     00089   00376   00382
WS-TRANS-READ                    00091   00329   00384
WS-UP-DESCRIPTION                00186   00255   00289   00315
WS-UP-PART-COST                  00188   00256   00290   00316
WS-UP-PART-NBR                   00184   00254   00288   00314
WS-UP-PART-PRICE                 00190   00257   00291   00317
WS-UP-PART-QUANTITY              00192   00258   00292   00318
WS-UP-UPDT-MESSAGE               00194   00259   00293   00319
WS-UPDATE-LINE                   00183   00208   00346
WS-UPDT-MESSAGES                 00093
```

```
PAGE 0015/COBTEXT  EXAMPLE          CROSS REFERENCE LISTING
                                          PROCEDURES and PROGRAMS
100-MAIN-PROGRAM                    00198
210-MASTER-COMPARED-HIGH            00240   00219
220-ADD-TO-MASTER                   00247   00243
230-TRAN-IN-ERROR                   00266   00245   00284
240-MASTER-COMPARED-LOW             00272   00221
250-MASTER-AND-TRAN-EQUAL           00277   00225
260-DELETE-MASTER                   00287   00280
270-CHANGE-MASTER                   00299   00282
300-GET-TRANSACTION                 00324   00209   00260   00269   00296   00321
310-GET-OLD-MASTER                  00332   00210   00275   00295
320-PRINT-UPDATE                    00340   00262   00294   00320
330-WRITE-NEW-MASTER                00351   00261   00274
400-WRITE-PRINT-LINE                00354   00348   00366   00369   00372   00388
                  00390   00392     00394   00396   00398   00400
410-PRINT-HEADING                   00359   00343
420-PRINT-TOTALS                    00374   00231
EXAMPLE                             00003
```

```
PAGE 0016/COBTEXT  EXAMPLE          STATEMENT OFFSETS
                                    Entry = example
     STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   ...

      198       58    259      76C    304      C14    360     11D0
      203       58    260      798    305      C38    361     11F4
      205      110    261      7AC    307      CC0    362     11FC
      207      124    262      7C0    308      CE4    363     120C
      208      154    263      7D4    310      D50    364     1274
      209      224    264      7F0    311      D74    365     1280
      210      238    266      814    314      DBC    366     12A0
      214      250    267      814    315      DD0    367     12B4
      215      274    268      82C    316      E14    368     12C0
      217      2A0    269      848    317      E90    369     12D4
      218      2A4    270      85C    318      F04    370     12E8
      219      2D4    272      880    319      F6C    371     12F4
      220      2E8    273      880    320      F98    372     1308
      221      31C    274      898    321      FAC    374     1330
      222      330    275      8AC    322      FC0    375     1330
      224      360    277      8D4    324      FE4    378     134C
      225      360    278      8D4    326      FE4    379     1358
      227      374    279      8D8    328     1004    380     13C4
      231      37C    280      8E8    329     1020    381     1430
      236      390    281      8FC    330     1030    382     149C
      237      3FC    282      910    332     1048    383     1508
      240      414    283      924    334     1048    384     1574
      242      414    284      92C    335     1064    385     15E0
      243      424    285      940    338     1078    387     1604
      244      438    287      958    340     10A8    388     1624
      245      440    288      958    342     10A8    389     1638
      247      468    289      96C    343     10C0    390     1680
      248      468    290      9B0    346     10D4    391     1694
      249      488    291      A2C    347     10F4    392     16DC
      250      4E4    292      AA0    348     1100    393     16F0
      251      56C    293      B08    349     1114    394     1738
      252      5D8    294      B34    351     1138    395     174C
      253      620    295      B48    352     1138    396     1794
      254      638    296      B5C    354     1168    397     17A8
      255      658    297      B70    355     1168    398     17F0
      256      6B4    299      B94    356     1194    399     1804
      257      6FC    301      B94    357     11A8    400     184C
      258      73C    302      BB8    359     11D0

0 ERROR(s), 0 QUESTIONABLE, 0 WARNING(s)

    DATA AREA IS      834 BYTES.
    CPU TIME = 0:00:03.  WALL TIME = 0:00:05.
```

Index

D

E

F

G

M

N

O

P

Q

R

S

T

U

V

W

Z