UNISYS

Application Development Solutions

COBOL ANSI-74
Programming Reference Manual

Volume 1:
Basic Implementation

ClearPath MCP 15.0

April 2013 8600 0296–208

Contents

Section 3. File and Task Concepts

Section 4. IDENTIFICATION DIVISION

Section 5. ENVIRONMENT DIVISION

Section 6. Data Concepts

Section 7. DATA DIVISION

Section 8. PROCEDURE DIVISION Concepts

Section 9. PROCEDURE DIVISION Statements

Section 13. ANSI Inter-Program Communication (IPC)

Section 14. COMMUNICATION SECTION

Section 15. Libraries

Section 16. Internationalization

Section 17. Control of the Compilation Process

Figures

Tables

Examples

Section 1
Program Structure

Purpose

This two-volume set provides a complete description of the implementation of COBOL74. To use this manual, you should be familiar with the general concepts of COBOL programming, a programming language based on the English language and, as such, composed of paragraphs, sentences, clauses, and words. The language is designed so that your source program is self-explanatory to someone who does not understand COBOL74.

This volume provides the syntax and rules specific to COBOL74. Volume 2 contains the syntax for using COBOL74 to write application programs that interface with the following products:

- Advanced Data Dictionary System (ADDS)
- Enterprise Database Server for ClearPath MCP
- Transaction Server
- Enterprise Database Server Transaction Processing System (TPS)
- Screen Design Facility (SDF)

Audience

The primary audience for this manual includes programmers and system analysts who are experienced in developing, maintaining, and reading COBOL programs. The secondary audience consists of technical support personnel and information systems management. A possible tertiary audience includes programmers who are learning COBOL, but the manual is not designed for this audience.

To use this manual, you should be familiar with the general concepts of COBOL programming; a programming language based on the English language and, as such is composed of paragraphs, sentences, clauses, and words. The language is designed so that your source program is self-explanatory to someone who does not understand COBOL74.

Conventions

Throughout this manual, *Volume 2* refers to the *COBOL ANSI-74 Programming Reference Manual, Volume 2: Product Interfaces,* unless a different manual title is specified.

This implementation of COBOL74 is based on, and is fully compatible with, the American National Standard programming language COBOL, X3.23-1974. Throughout this manual, extensions to the American National Standard for Programming Language COBOL, ANSI X3.23-1974 are identified as such with the phrase *Extension to ANSI X3.23-1974 COBOL*.

The elements of the language are described in the order in which they are coded in a program. The IDENTIFICATION DIVISION is described first; the PROCEDURE DIVISION is described last. For a statement or a clause, individual elements are described in the order in which they are listed in the format.

COBOL modules including the Report Writer, Debugging, Communication, Segmentation, and Inter-Program Communication modules are described in separate sections.

The GENERALSUPPORT system library must be installed before you can use the features of the COBOL74 compiler described in this manual.

Acknowledgment

Any organization wishing to reproduce this COBOL manual in whole or in part 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 this section as part of the introduction to the manual. Those using a short passage, as in a book, need not quote this entire preface.

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 COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contractor or by the committee in connection therewith.

Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Language.

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. These authors or copyright holders are the following:

- FLOW-MATIC (Trademark of Sperry Rand Corporation), Programming for the Univac R I and II, Data Automation Systems, copyrighted 1958, 1959 by Sperry Rand Corporation
- IBM Commercial Translator, Form No. F28-8013, copyrighted 1959 by International Business Machines Corporation
- FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis–Honeywell

Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Documentation Updates

This document contains all the information that was available at the time of publication. Changes identified after release of this document are included in problem list entry (PLE) 18896354. To obtain a copy of the PLE, contact your Unisys representative or access the current PLE from the Unisys Product Support Web site:

http://www.support.unisys.com/all/ple/18896354

Note: *If you are not logged into the Product Support site, you will be asked to do so.*

What's New?

New or Revised Information	Location
Updated information about the sequence area.	Section 1: Columns 1–6: Sequence Area

Source Program Components

A source program is code written in COBOL74 that the compiler accepts as input. When you compile the source program, the compiler verifies that your code follows the rules presented in this manual and translates the source program into an object program. The object program directs the computer to operate on the data. If the compiler indicates that your source program needs corrections, you can make the appropriate changes and then recompile it. The object program always reflects the source program you create.

Program Divisions

A COBOL74 source program is divided into four parts called divisions. The divisions must appear in the following order:

1. IDENTIFICATION DIVISION
2. ENVIRONMENT DIVISION
3. DATA DIVISION
4. PROCEDURE DIVISION

A division header consists of the name of a division, the word DIVISION, and a period.

Table 1–1 describes the purpose of each division.

Table 1–1. Purpose of COBOL74 Divisions

Division Name	Purpose of Division
IDENTIFICATION	Identifies and provides documentation about the source program.
ENVIRONMENT	Specifies the computer system and associates files with I/O devices.
DATA	Describes the structure of the data, the constants to be used, the intermediate storage areas, and any external data.
PROCEDURE	Instructs the computer to perform the steps necessary to solve the problem addressed by the source program. This division uses the data described in the DATA DIVISION.

Sections

The ENVIRONMENT, DATA, and PROCEDURE DIVISIONs can be subdivided into sections. A section further identifies the purpose of a division.

A section begins with a name that identifies the section. COBOL74 specifies the names of sections in the ENVIRONMENT and DATA DIVISIONs. You specify names of sections in the PROCEDURE DIVISION.

A section header consists of the name of the section, the word SECTION and a period. A section continues with one or more successive paragraphs that follow the period. A section ends immediately before the next section, at the end of the division, or at the keywords END DECLARATIVES in the DECLARATIVES SECTION of the PROCEDURE DIVISION.

Table 1–2 describes the sections associated with each division

Table 1–2. Purpose of Sections

Division Name	Section Name	Purpose of Section
IDENTIFICATION	None	Not applicable
ENVIRONMENT	CONFIGURATION	Specifies computer equipment

Table 1–2. Purpose of Sections

Division Name	Section Name	Purpose of Section
	INPUT-OUTPUT	Associates files with specific devices
DATA	FILE	Describes the record structure of files
	DATA-BASE	Describes one or more databases that can be used by the COBOL program
	WORKING-STORAGE	Describes intermediate data items
	LOCAL-STORAGE	Describes data to be either passed or received as parameters to an external procedure
	LINKAGE	Describes data items to be referenced by the calling program and the called program
	COMMUNICATION	Describes the data items in the source program that serve as the interface between the data communications interface (DCI) library and the program
	REPORT	Describes the contents and format of generated reports for the Report Writer
PROCEDURE	User defined	Groups paragraphs into sections that you define

Note: *DATA-BASE and LOCAL-STORAGE are extensions to ANSI X3.23-1974 COBOL.*

Paragraphs

In the IDENTIFICATION and ENVIRONMENT DIVISIONs, a paragraph begins with a header that identifies the paragraph. A paragraph header consists of a reserved word followed by a period. The paragraph continues with one or more successive entries.

In the PROCEDURE DIVISION, a paragraph begins with a user-defined word called a paragraph-name. It is followed by a period and optionally one or more entries.

A paragraph ends immediately before the next paragraph header or section name, at the end of the division, or at the keywords END DECLARATIVES in the DECLARATIVES SECTION of the PROCEDURE DIVISION.

Example 1–1 shows examples of paragraphs in the IDENTIFICATION, ENVIRONMENT, and PROCEDURE DIVISIONs.

```
010000 IDENTIFICATION DIVISION.
010050*The PROGRAM-ID paragraph header is a reserved word.
010100 PROGRAM-ID. GUEST-CREDIT-AUTHORIZATION.
```

```
100000 ENVIRONMENT DIVISION.
100050 CONFIGURATION SECTION.
100150*The SOURCE-COMPUTER paragraph header is a reserved word.
100200 SOURCE-COMPUTER. MICROA.
100250 INPUT-OUTPUT SECTION.
100300*The FILE-CONTROL paragraph header is a reserved word.
100350 FILE-CONTROL.
            :
200000 PROCEDURE DIVISION.
200050* You define paragraph-names in the PROCEDURE DIVISION.
200100 MAIN-PARAGRAPH.
            :
```

Example 1–1. Coding Paragraphs

Sentences

A sentence consists of one or more statements. You must end a sentence with a period.

A sentence can be one of the following three types:

- A compiler-directing sentence that directs the compiler to take a specific action during the compilation process.
- A conditional sentence that tests a truth value and specifies an action depending on the result of the test.
- An imperative sentence that specifies an unconditional action to be taken.

Example

The following is an example of a conditional sentence:

```
IF BALANCE LESS THAN ZERO
   PERFORM PROCESS-DEBIT
ELSE
   NEXT SENTENCE.
```

For More Information

For more information about types of sentences, refer to Section 8, "PROCEDURE DIVISION Concepts."

Statements

A statement is a syntactically valid combination of words and symbols beginning with a COBOL74 verb. A statement ends whenever the compiler detects a new verb or a period.

Clauses, Phrases, and Options

A clause is a set of consecutive COBOL74 words that represents a valid portion of a statement or entry. A phrase is a set of consecutive COBOL74 words that represents a valid portion of a statement, entry, or clause. In this manual, optional phrase and clauses are often referred to as options.

Words

A word is a string of characters that form a valid COBOL74 word for creating valid phrases, clauses, statement, sentences, paragraphs, sections, and divisions.

For More Information

For a complete description of the rules for forming words, refer to Section 2, "Language Elements."

Line Layout

The compiler expects the components of your source program to appear in specific areas along a line of code. Each line has 80 positions, which are grouped into five areas that make up the line format (also called the reference format), as shown in Figure 1–1.

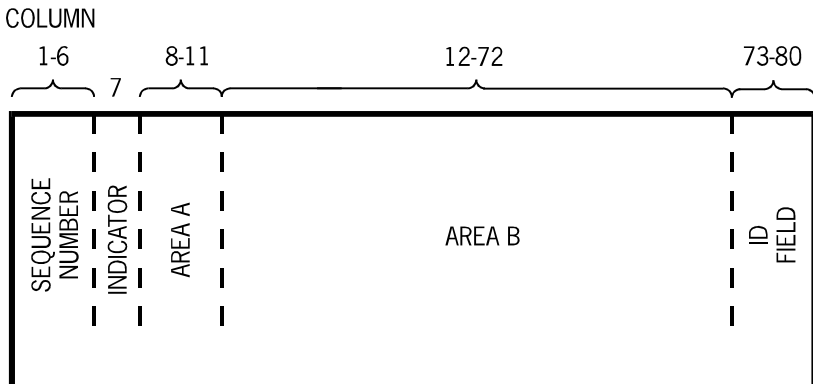


Figure 1–1. Line Format

Table 1–3 shows the columns associated with each area, the name of the area, and the program components that can begin in that area.

Table 1–3. Placement of Source Program Component within Areas

Columns	Name	Acceptable Entries
1–6	Sequence area	Sequence number
7	Indicator area	Asterisk (*), slash (/), D, hyphen (–), or dollar sign ($)

Table 1–3. Placement of Source Program Component within Areas

Columns	Name	Acceptable Entries
8–11	Area A	• Division header • Section header • Paragraph header • File description (FD) entry • Sort merge description (SD) entry • Level numbers • DECLARATIVES keyword • END DECLARATIVES keyword
12–72	Area B	• Sentences • Level numbers other than 01 or 77
73–80	Identification area	Comment entries

For More Information

For information on using the FREE compiler option to remove the margin restrictions, refer to "FREE" in Section 17, "Control of the Compilation Process."

Columns 1–6: Sequence Area

You can add information in the sequence number area to label or identify a source program line. The compiler accepts any combination of characters in any order, in the sequence number area. The sequence number area fields of any two records do not have any particular relationship to the order in which they are processed by the compiler.

Unisys strongly recommends that you enter a numeric representation of the relative position of the source image within the source unit in the sequence number field (that is, a sequence number). This is necessary as some Unisys extensions to COBOL ANSI-74 and some system software require that the sequence number field of a source record contain a numeric value.

If you add non-numeric information in the sequence field, then you might be unable to use certain Unisys features within the compiler, and you might limit your ability to use the source files with certain Unisys utility software.

Column 7: Indicator Area

The indicator area is blank or contains one of five characters that indicate a special purpose for the line. Leave column 7 blank, unless there is a programmatic need to use one of the following listed characters in that location.

- An asterisk (*) indicates the line is a comment line.
- A slash (/) indicates the line is a comment line and causes the printer to eject a page and print the comment at the top of the source listing.

- The letter D indicates the line is a debugging line.
- A hyphen (–) indicates the line is a continuation line.
- A dollar sign ($) indicates the line is a compiler control record.
- A space ( ) indicates the line is a normal COBOL74 source image.

When the FREE compiler control option is

- Set, any character in column 7 other than an asterisk, slash, hyphen, dollar sign, or space is treated as part of the source image.
- Reset, any character in column 7 other than an asterisk, slash, hyphen, or dollar sign is treated as a space.

For More Information

- For information about the types of lines, see "Special-Purpose Lines" later in this section.
- For information on the compiler control option, refer to "FREE" in Section 17, "Control of the Compilation Process."

Columns 8–11: Area A

Division, section, and paragraph headers, level indicators, the level numbers 01 and 77, and the keywords for declaratives must begin in area A. You can begin your entry anywhere within area A. It is acceptable for your entry to extend into area B.

The rules for forming division and section headers are as follows:

- Enter the division or section name in area A.
- Follow the name by at least one space.
- Enter DIVISION or SECTION, whichever is appropriate.
- End the entry with a period.

The rules for forming paragraph headers are as follows:

- Enter the name of the paragraph.
- Follow the name with a period.

The rules for forming the indicators of a file description (FD) or sort-merge description (SD) entry or a level-number 01 or 77 entry are as follows:

- Enter FD, SD, 01, or 77, whichever is appropriate.
- Follow the entry with the appropriate associated name and descriptive information.

The rules for forming declarative keywords are as follows:

- Enter *DECLARATIVES.* on a line by itself at the beginning of the DECLARATIVES SECTION of the PROCEDURE division.

- Enter *END DECLARATIVES.* on a line by itself at the end of the DECLARATIVES SECTION.

Example

Example 1–2 shows coding of area A entries.

```
000100 IDENTIFICATION DIVISION.
000200 ENVIRONMENT DIVISION.
000300 CONFIGURATION SECTION.
000400 SOURCE-COMPUTER. MICROA.
000500 INPUT-OUTPUT SECTION.
000600 FILE-CONTROL.
       :
001000 DATA DIVISION.
001100 FILE SECTION.
001200 FD  GUEST-FILE
       :
001400 01 NAME-RECORD.
       :
001800 WORKING-STORAGE SECTION.
001900 77 EOF-FLAG.
       :
003000 PROCEDURE DIVISION.
003100 DECLARATIVES.
       :
003300 END DECLARATIVES.
003400 MAIN SECTION.
       :
```

Example 1–2. Coding Area A Entries

For More Information

- For information about level numbers, refer to Section 7, "DATA DIVISION."
- For information about declaratives, refer to Section 8,"PROCEDURE DIVISION Concepts."

Columns 12–72: Area B

The bulk of your code, including sentences, statements, clauses, and many level-number entries, begin in area B. Your entry can begin anywhere within area B.

In the DATA DIVISION, all level numbers other than 01 and 77 can begin anywhere in area A or B.

The first sentence or entry in a paragraph begins either on the same line as the paragraph name or in area B of the next nonblank line that is not a comment line. Additional sentences begin after the previous sentence in area B of the same line or in area B of the next nonblank line that is not a comment line.

For readability, begin all level numbers other than 01 or 77 in area B and indent each successively higher level number two to four positions. In addition, define level numbers with an increment value greater than one so that it is possible to insert new levels between two levels, if the need arises.

Example

Example 1–3 shows the use of indentation to make your coding more readable. Subordinate entries are indented four spaces. For example, MONTH, DAY, and YEAR are all elements of DATER.

```
010000* Indentation improves the readability of your program.
010050*
010100 01  INPUT-RECORD.
100350     03 DATER
100400         05  MONTH             PIC 99.
100450         05  DAY               PIC 99.
100500         05  YEAR               PIC 99.
100550     03 FILLER                 PIC X(33).
100600 66  IN-DATE RENAMES MONTH THRU YEAR.
```

Example 1–3. Coding Area B Entries for Readability

Columns 73–80: Identification Area

This area is an optional area. You can use it for documentation purposes.

Special-Purpose Lines

Special purpose lines are lines that have a special character in the indicator area (column 7) or are blank. These include comment lines, continuation lines, debugging lines, compiler control options, and blank lines.

Comment Lines

A comment line is any line with either an asterisk (*) or a slash (/) in the indicator area (column 7) of the line. A comment line can appear anywhere in a source program and can use any combination of characters from the character set of the computer, not just the characters in the COBOL74 character set. You can use successive comment lines.

If you use the slash character to indicate a comment, the printer ejects a page before printing the comment on the output listing of the compiler.

Example

Example 1–4 shows the use of comment lines.

```
100100 IDENTIFICATION DIVISION.
100200* The purpose of this program is to...
100300* Notice that comments can use lowercase letters.
```

```
100400 PROGRAM-ID. GUEST-CREDIT-AUTHORIZATION.
100500/ This comment is printed at the top of a new page.
100600 ENVIRONMENT DIVISION.
```

Example 1–4. Coding Comment Lines

Continuation Lines

A continuation line is a line that is continued from a previous line. You indicate a continuation line with a hyphen (–) in the indicator area (column 7). The hyphen means that the first nonblank character in area B (columns 12–72) of the continuation line follows the last nonblank character of the preceding line, with no spaces inserted. You can use successive continuation lines. Area A (columns 8–11) of a continuation line must be blank.

The compiler determines the length of the continuation line as follows:

- For nonnumeric literals, the first nonblank character in Area B to the end of Area B
- For other strings, the first nonblank character in Area B to the last nonblank character in Area B

If no hyphen is present in the indicator area of a line, the compiler assumes that the last character in the preceding line is followed by a space.

The rules for using a continuation line with a nonnumeric literal (which must begin with a quotation mark) or an undigit literal (which must begin with a commercial at sign) are as follows:

- Use all 72 positions on the line to be continued. All spaces at the end of the line are considered to be part of the literal.
- Do not close the literal in column 72 with a quotation mark for the nonnumeric literal or a commercial at sign (@) for the undigit literal because doing so will delimit the literal and prevent it from being continued.
- Enter as the first nonblank character in area B a quotation mark for the nonnumeric literal or a commercial at sign for the undigit literal. The literal continues with the character immediately after the quotation mark or a commercial at sign.
- Limit the length of a nonnumeric literal to 160 characters. If the length is greater than 160 characters, but less than 262 characters, a syntax error is generated. If the length exceeds 261 characters, the compilation process might terminate abnormally without providing a syntax error.

Example

Example 1–5 shows coding of continuation lines.

```
100150*Line 100200 continues the clause begun on the preceding line.
100100      SELECT GUEST-FILE ASSIGN TO DISK OR
100200-     GANIZATION IS SEQUENTIAL.
              .
              .
```

```
100700* Line 100900 continues a nonnumeric literal.
100800 77 HEADER-LINE     PIC X(60)  VALUE IS "    JANUARY    FEBRUARY
100900-    "   MARCH    APRIL    MAY    JUNE    ".
101000
101500*Line 102000 continues an undigit literal.
101000 77  HEX-LITERAL     PIC X(30)  VALUE IS @AAAAAAAAAABBBBBBBBBBCCC
102000-    @CCCCCC@.
102300
102500* The first quote continues the nonnumeric literal; the second
102550* quote ends the literal.
102600 01  WARNING MESSAGE PIC X(24)  VALUE IS "WRONG ENTRY FOR THIS KEY
102700-    "".
```

Example 1–5. Coding Continuation Lines

Debugging Lines

A debugging line is any line with a D in the indicator area (column 7) of the line. A debugging line with spaces in columns 8 through 72 is considered to be the same as a blank line. You can enter a debugging line anywhere after the OBJECT-COMPUTER paragraph.

The debugging module is activated when you specify the WITH DEBUGGING MODE clause in the SOURCE-COMPUTER paragraph. If you do not activate the debugging module, the compiler treats a debugging line like a comment line. Therefore, you should make sure that your program is syntactically correct when the debugging lines are considered to be comment lines.

You can use successive debugging lines and can continue debugging lines; however, each continuation line must contain a D in the indicator area, and character strings cannot be continued across multiple lines.

Example

Example 1–6 shows the use of debugging lines.

```
010000 IDENTIFICATION DIVISION.
100000 ENVIRONMENT DIVISION.
100100 SOURCE-COMPUTER. MICROA WITH DEBUGGING MODE
             .
             .
100600 WORKING STORAGE SECTION.
100700D77  PERFORMANCE-COUNT   PIC 9(4).
100800D77  BAD-RECORDS         PIC 9(4).
100900D77  RATIO               PIC 9(4).99.
             .
             .
101000 PROCEDURE DIVISION.
102000 OPEN-IT.
102100     OPEN INPUT GUEST-FILE.
```

```
103000D    MOVE ZEROS TO PERFORMANCE-COUNT, BAD-RECORDS, RATIO.
104000 READ-IT.
104100     READ GUEST-FILE AT END GO TO FINISH-IT.
105000D    ADD 1 TO PERFORMANCE-COUNT.
106000D    IF IN-KEY NOT NUMERIC ADD 1 TO BAD-RECORDS.
              .
              .
107000     GO TO READ-IT.
108000 FINISH-IT.
108100     CLOSE GUEST-FILE.
109000D    DIVIDE PERFORMANCE-COUNT BY BAD-RECORDS GIVING RATIO.
              .
              .
```

Example 1–6. Coding Debugging Lines

For More Information

- For information about the WITH DEBUGGING MODE clause, refer to Section 4, "IDENTIFICATION DIVISION".
- For information about the debug module, refer to Section 11, "Debugging."

Compiler Control Options

A compiler control option is any line with a dollar sign ($) in the indicator area (column 7) of the line. This line specifies compiler control options to use during the compilation process.

For More Information

For a discussion of the available compiler control options and their uses, refer to Section 17, "Control of the Compilation Process".

Blank Lines

A blank line is a line that has no entries in the indicator area, area A, or area B. You can use a blank line anywhere in the source program except immediately preceding a continuation line.

Section 2
Language Elements

This section describes the following elements used to form the components of a source program:

- Character set
- Separators
- Character strings

Character Set

The most basic and indivisible unit of the COBOL74 language is the character. The set of characters you use to write COBOL74 programs includes the letters of the alphabet, digits, and special characters. The character set consists of 52 characters, including the 51 characters specified in the ANSI-74 standard plus the commercial at sign (@), which is an extension to ANSI X3.23-1974 COBOL. You link these individual characters together to form character strings and separators and you link character strings and separators to form a source program.

Table 2–1 shows the COBOL74 character set.

Table 2–1. Character Set

Character	Meaning	Character	Meaning
0 through 9	Digit	A through Z	Letter
	Space or blank	+	Plus sign
–	Minus sign or hyphen	*	Asterisk
/	Virgule, or slash	=	Equal sign
$	Dollar sign	,	Comma or decimal point
;	Semicolon	.	Period or decimal point
"	Quotation mark	(	Left parenthesis
)	Right parenthesis	>	Greater than sign
<	Less than sign	@	Commercial at sign

Separators

A separator is a string of one or more punctuation characters. You can link a separator with another separator or with a character string. A character-string must be linked to a separator. Table 2–2 lists all the separators and the rules governing them.

Table 2–2. Valid Separators

Separator	Explanation
(space)	Spaces can precede or follow all other separators. Special rules for spaces are as follows: • A space is required before the opening pseudotext delimiter. • A space that precedes the ending quotation mark of a nonnumeric literal is considered to be part of the literal. • A space that follows the opening quotation mark of a nonnumeric literal is considered to be part of the literal.
.	Periods mark the end of a COBOL74 entry. A period is always treated as a separator when it is followed by a space. In an extension to ANSI X3.23-1974 COBOL, a period does not need to be followed by a space in most cases. However, the period should always be followed by a space when it is used as a separator character. This practice prevents problems in those environments in which the space is required but not encountered.
, ;	Commas and semicolons delimit clauses. A comma or a semicolon is always treated as a separator when it is followed by a space. In an extension to ANSI X3.23-1974 COBOL, they do not need to be followed by a space in most cases. However, the comma and semicolon should always be followed by a space when they are used as a separator character. This practice prevents problems in those environments in which the space is required but not encountered.
( )	Left and right parentheses delimit subscripts, indexes, arithmetic expressions, or conditions. They must appear in balanced pairs.
" "	Quotation marks delimit nonnumeric literals. They must appear in balanced pairs. If the literal is continued on another line, you need to enter another quotation mark as the first nonblank character in area B An opening quotation mark must be preceded immediately by a space, left parenthesis, (comma, or semicolon). A closing quotation mark must be followed immediately by a space, comma, semicolon, period, or right parenthesis.

Table 2–2. Valid Separators

Separator	Explanation
==	Pseudotext delimiters set off pseudotext. They must appear in balanced pairs. An opening pseudotext delimiter must be preceded by a space. A closing pseudotext delimiter must be followed by a space, comma, semicolon, or period.
@	Commercial at signs delimit undigit literals. An opening @ character must be preceded immediately by a space, comma, semicolon, or left parenthesis; a closing @ character must be followed immediately by a space, comma, semicolon, period, or right parenthesis.
NC"	NC" characters precede a Kanji literal. A quotation mark follows the Kanji literal.

Notes:

The following are Unisys extensions to ANSI X3.23-1974 COBOL:

- *Undigit literals and the use of the character to delimit them.*
- *National literals and the use of the character sequence N" to introduce them.*
- *The use of the semicolon only (without a space character immediately following the semicolon) as a valid separator.*
- *The use of the comma only (without a space character immediately following the comma) as a valid separator.*

Any punctuation character that you use in a PICTURE character string or numeric literal is considered to be part of the PICTURE character string or numeric literal rather than a punctuation character. You delimit PICTURE character strings by spaces, commas, semicolons, or periods.

The rules established for the formation of separators do not apply to the characters in nonnumeric literals, comment-entries, or comment lines. When you are coding nonnumeric literals, comment-entries, or comment lines, you can use the complete character set of the computer, not just the COBOL74 character set.

Character Strings

A character string is a set of one or more characters delimited by separators that form one of the following:

- Word
- Literal
- PICTURE character string
- Comment-entries

The maximum size of a syntactically valid nonnumeric literal character string in COBOL74 is 160 characters.

A character string containing between 161 and 261 characters causes the compiler to issue syntax errors. Exceeding the 261-character limit might cause the compilation process to terminate abnormally without syntax errors, thereby causing unpredictable results.

For More Information

- For information about the various types of COBOL words, refer to "Word Types" later in this section.
- For definitions and examples of the various types of literals you can use in a COBOL program, refer to "Literals" later in this section.
- For information about PICTURE character strings in the PICTURE clause, refer to Section 7, "DATA DIVISION."
- Refer to Section 1, "Program Structure," for information about comment-entries.

Word Types

A COBOL word is a character string that forms one of the following:

- Reserved word
- Context-sensitive keyword
- Application-specific keyword
- System-name
- User-defined word

You can use a given COBOL word in your program as both a system-name and a user-defined word, or as both a system-name and a reserved word. You cannot use a reserved word as a user-defined word.

Reserved Words

A reserved word is a COBOL74 word that has a specific meaning to the compiler. For example, MOVE is a reserved word that directs the compiler to perform a move operation.

A reserved word is a word that is reserved by the compiler and that you cannot use as a user-defined word anywhere in the source program. No exceptions exist for specific divisions, sections, or statements. Reserved words are used in the following six ways:

- As connectives that qualify data, link two or more operands in a series, or link logical operators to form conditions
- As figurative constants that associate names to values that you commonly use in a source program
- As keywords that are verbs or other required pieces of a syntax
- As optional words that increase the readability of your program

- As special registers that are compiler-generated, read-only storage areas that provide you with access to specific COBOL74 features
- As special-character words that indicate arithmetic or relational operations

For More Information

For a complete list of reserved words in COBOL74, refer to Appendix B, "Reserved Words and Keywords."

Connectives

Connectives are reserved words that you can use in any of the following two ways:

- As a qualifier to associate a data-name, a condition-name, a text-name, or a paragraph-name with its qualifier. Examples of qualifier connectives are OF or IN.
- As logical connectives to form conditions. AND and OR are logical connectives.

Figurative Constants

Figurative constants are reserved words that act as literals for values you might commonly use in a source program. These reserved words make your programming task easier by relieving you of the burden of assigning names to specified constant values. You can use a figurative constant wherever a literal can be used. For example, a *MOVE SPACES TO data-name* statement fills the data-name with spaces. The only figurative constant that acts as a numeric literal is the ZERO (ZEROS, ZEROES) figurative constant, and then only when you use it in a context that requires a numeric literal.

You can use figurative constants in the following types of statements:

- In MOVE and IF statements for moving and comparing data items
- In DISPLAY and STOP statements for displaying one character
- In STRING and UNSTRING statements for manipulating one character

Figurative constants increase the readability of your program. The singular and plural forms of figurative constants are equivalent and interchangeable. Each figurative constant is a distinct word, except for the *ALL literal* constant, which is two distinct words.

The individual figurative constants are described in Table 2–3.

Table 2–3. Figurative Constants

Figurative Constant	Explanation
ZERO, ZEROS, ZEROES	Represents numeric 0 or one or more of the alphanumeric character 0, depending on context.
SPACE, SPACES	Represents one or more of the alphabetic character space from the character set of the computer.

Table 2–3. Figurative Constants

Figurative Constant	Explanation
HIGH-VALUE, HIGH-VALUES	Represents the alphanumeric character or characters that occupy the last position in your program's collating sequence. If you specify a collating sequence in the SPECIAL-NAMES paragraph, the HIGH-VALUE figurative constant represents the character or characters that occupy the last position in the collating sequence that you specify. This figurative constant may produce unexpected results when used with the system default ccsversion in the SPECIAL-NAMES paragraph. For more information, see "SPECIAL-NAMES" in Section 5, "ENVIRONMENT DIVISION."
LOW-VALUE, LOW-VALUES	Represents the alphanumeric character or characters that occupy the first position in your program's collating sequence. If you specify a collating sequence in the SPECIAL-NAMES paragraph, the LOW-VALUE figurative constant represents the character or characters that occupy the first position in the collating sequence that you specify. This figurative constant may produce unexpected results when used with the system default ccsversion in the SPECIAL-NAMES paragraph. For more information, see "SPECIAL-NAMES" in Section 5, "ENVIRONMENT DIVISION."
QUOTE, QUOTES	Represents one or more alphanumeric quotation mark characters. You can use this figurative constant to avoid using a quotation mark within a literal. For example, *MOVE QUOTE TO OUT-LINE* causes a quotation mark to be printed. You cannot use QUOTE or QUOTES to bound a nonnumeric literal. For example, QUOTE XYZ QUOTE is incorrect as a way of stating the nonnumeric literal "XYZ".
ALL literal	Represents a continuous sequence of any alphanumeric literal. The literal part of the figurative constant must be a nonnumeric literal or a figurative constants other than ALL. When you use a figurative constant other than ALL as the literal, the word ALL is redundant. For example, MOVE ALL SPACES is equivalent to MOVE SPACES. You might want to retain the word ALL to improve the readability of your program.

When a figurative constant represents a string of one or more characters, the compiler determines the length of the string from context according to the following rules.

- When a figurative constant is moved to or compared with another data item, the compiler repeats the string of characters specified by the figurative constant, character by character, until the receiving string has as many characters as the associated data item. The compiler independently completes the character repetition before it applies any JUSTIFIED clause that is associated with the data item.

- When you compare the alphanumeric figurative constant HIGH-VALUE, LOW-VALUE, or QUOTE with a numeric data item in a relation condition, the compiler uses the rules for nonnumeric comparison.
- When you move an alphanumeric figurative constant HIGH-VALUE, LOW-VALUE, or QUOTE to a numeric or numeric-edited data item, the compiler uses the rules for moving an alphanumeric item to a numeric or numeric-edited item; that is, the results are the same as if an alphanumeric data item contained the figurative constant value in all its character positions. The compiler moves the data as if the figurative constant was an unsigned numeric integer; therefore, it converts nonnumeric characters into numeric characters. For example, it converts the LOW-VALUE EBCDIC character to EBCDIC 0, the HIGH-VALUE EBCDIC character to EBCDIC 9, and the EBCDIC QUOTE character to EBCDIC 9.

  If you specify a CCSVERSION collating sequence in the SPECIAL-NAMES paragraph, and if you move the alphanumeric figurative constant HIGH-VALUE, LOW-VALUE, or QUOTE to a numeric or numeric-edited item, the compiler converts the QUOTE figurative constant to EBCDIC 9; however, the conversion of the HIGH-VALUE or the LOW-VALUE figurative constant into numeric characters might yield unexpected results.
- When the figurative constants ZERO, SPACE, HIGH-VALUE, or LOW-VALUE are moved to or compared with a Kanji data item, the compiler represents the actual character associated with each figurative constant as one or more of the Kanji characters.

Examples

The following example initializes a value in the WORKING-STORAGE SECTION to three zeros.

```
77  NUMBER-OF-GUESTS  PIC 9(3)  VALUE ZEROS
```

The next example uses LOW-VALUE and HIGH-VALUE to process an End-Of-File condition.

```
77 EOF-FLAG           PIC X     VALUE LOW-VALUE
         :
     READ GUEST-FILE AT END
       MOVE HIGH-VALUE TO EOF-FLAG.
     IF EOF-FLAG EQUAL TO HIGH-VALUE...
```

The next example uses ZERO and SPACES for COMPARE and MOVE operations.

```
IF NUMBER-OF-GUESTS EQUAL TO ZERO
 MOVE SPACES TO GUEST-LAST-NAME.
```

The next example uses the ALL literal figurative constant in a *MOVE ALL literal TO data-item* statement. The first column shows the value of the literal, the second column shows the size in characters of the data item designated as the receiving field, and the third column shows the value of that data item after the MOVE statement completes.

ALL Literal	Receiving Field Size	Receiving Field Contents
ALL "ABC"	7 characters	ABCABCA
ALL "3"	5 characters	33333
ALL "HI-LO"	12 characters	HI-LOHI-LOHI
ALL "LIMIT"	4 characters	LIMI

The next example shows that the figurative constant ALL is redundant when used with a figurative constant.

Figurative Constant	Receiving Field Size	Receiving Field Contents
QUOTES	3 characters	"""
ALL QUOTES	3 characters	"""

For More Information

- For information about specifying a collating sequence, refer to "OBJECT-COMPUTER" and "SPECIAL-NAMES" in Section 5, "ENVIRONMENT DIVISION."
- Refer to "MOVE" in Section 9, "PROCEDURE DIVISION Statements," for information about rules for moving data.

Keywords and Optional Words

A keyword is a word that is required by the context in which it appears. In the format notation, keywords are uppercased and underlined. There are the following three kinds of keywords:

- Verbs, such as ADD, READ, and MOVE
- Functional words, such as NEGATIVE and SECTION
- Other words that appear in statement and entry formats

Optional words are reserved words that increase the readability of your program. They do not affect the execution of your program. In the format notation, they appear as uppercase words that are not underlined.

Example

In the following example, RECORD is a keyword, CONTAINS and CHARACTERS are optional words, TO is a keyword required when the integer-1 option is used, and DEPENDING and ON are keywords required when the DEPENDING ON option is used. Data-name is a user-defined word rather than a keyword, but if the DEPENDING ON option is present, the data-name user-defined word must appear too. Likewise, if data-name is present, DEPENDING ON must appear too.

```
RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS

[DEPENDING ON data-name]
```

Refer to Appendix A, "General Format Notation," for a full explanation of the format notation.

Special Registers

Special registers are compiler-generated, read-only storage areas that primarily give access to information produced with the use of specific COBOL74 features.

Table 2–4 explains each of the special registers.

Table 2–4. Special-Register Definitions

Register	Explanation
DATE	Contains the system date formatted as the year of century, month of year, and day of month. DATE is an unsigned, 6-digit, elementary numeric integer. For example, July 1, 1990 is expressed as 900701. Refer to Section 15, "Libraries," for an example of accessing a 4-digit year by using the CURRENT_DATE procedure.
DATE YYYYMMDD (extension to ANSI X3.23-1974 COBOL)	Contains the system date formatted as the 4-digit year, month of year, and day of month. DATE is an unsigned, 8-digit, elementary numeric integer. For example, July 1, 1990 is expressed as 19900701. Refer to Section 15, "Libraries," for an example of accessing a 4-digit year by using the CURRENT_DATE procedure.
DAY	Contains the system date formatted as the year of century followed by the number of days since the beginning of the year. DAY is an unsigned, 5-digit, elementary numeric integer. For example, July 1, 1990 is expressed as 90182.

Table 2–4. Special-Register Definitions

Register	Explanation
DAY YYYYDDD (extension to ANSI X3.23-1974 COBOL)	Contains the system date formatted as the 4-digit year followed by the number of days since the beginning of the year. DAY is an unsigned, 7-digit, elementary numeric integer. For example, July 1, 1990 is expressed as 1990182.
DEBUG-ITEM	Provides information about the conditions that caused execution of a debugging section. Each execution of a debugging section has the special register DEBUG-ITEM associated with it.
TIME	Contains the elapsed time after midnight based on a 24-hour clock in hours, minutes, seconds, and hundredths of a second. TIME is an unsigned, 8-digit, elementary numeric integer. For example, 2:41 p.m. is expressed as 14410000. The maximum value of TIME is 23595999.
TIMER (extension to ANSI X3.23-1974 COBOL)	Represents the number of 2.4-microsecond intervals since midnight. TIMER is a single, unsigned 11-digit, numeric integer. It is composed of the current value of the computer's interval timer.
TODAYS-DATE (extension to ANSI X3.23-1974 COBOL)	Represents the date as the month of the year, followed by the day of the month, followed by the year of the century. TODAYS-DATE is a 6-digit, unsigned, elementary numeric integer. For example, July 1, 1990 is expressed as 070190.
TODAYS-DATE MMDDYYYY (extension to ANSI X3.23-1974 COBOL)	Represents the date as the month of the year, followed by the day of the month, followed by the 4-digit year. TODAYS-DATE is a 8-digit, unsigned, elementary numeric integer. For example, July 1, 1990 is expressed as 07011990.
TODAYS-NAME (extension to ANSI X3.23-1974 COBOL)	Provides the current day of the week. TODAYS-NAME is an elementary, 9-character, alphanumeric item. If the day of the week is less than 9 characters long, it is left-justified in the 9-character area provided, with space-fill on the right.
LINAGE-COUNTER	Contains at any time the number of lines advanced within a printed page. LINAGE-COUNTER is a fixed data-name for a line counter suitable for computation. It is generated by the presence of a LINAGE clause in a file description (FD). The implicit class of a LINAGE-COUNTER is numeric. No data item is referenced; it is treated as a LINENUMBER attribute for purposes of retrieval. The compiler automatically supplies one LINAGE-COUNTER for each file in the FILE SECTION that has a LINAGE clause in its FD entry.

Table 2–4. Special-Register Definitions

Register	Explanation
LINE-COUNTER	Provides the vertical position in a report. LINE-COUNTER is a fixed data-name for a line counter suitable for computation. It is generated for each report description (RD) in the REPORT SECTION. The compiler automatically provides one LINE-COUNTER register for each report in the RD entry.
PAGE-COUNTER	Provides page numbers within a report group. PAGE-COUNTER is a fixed data-name for a page counter suitable for computation. It is generated for each report-description (RD) entry in the REPORT SECTION. The compiler automatically supplies one PAGE-COUNTER for each report that has the word PAGE-COUNTER as a source data item in a RD entry.

Special-Character Words

Special characters are reserved words used to indicate the arithmetic operations of addition, subtraction, multiplication, division, and exponentiation and the relational operations of comparing less than, greater than, and equal to conditions.

Table 2–5 shows the meaning of each special-character word.

Table 2–5. Special-Character Words

	Character	Meaning
Arithmetic Operator	+	Plus sign
	–	Minus sign
	*	Multiplication
	/	Division
	**	Exponentiation
Relation Character	<	Less than sign
	>	Greater than sign
	=	Equals sign

Context-Sensitive Keywords

A context-sensitive keyword is a word that the compiler recognizes as reserved when it is used in a compiler-defined syntax. If you want to use that word as a user-defined word in another place, the compiler recognizes it as a user-defined word in that context.

For More Information

For a list of context-sensitive keywords, refer to Appendix B, "Reserved Words and Keywords."

Application-Specific Keywords

An application-specific keyword is a word that is reserved by the compiler for the extent of the program. Application-specific keywords are used for applications such as internationalization and port files. You must specify that you are using application-specific keywords by using the RESERVE clause of the SPECIAL-NAMES paragraph in the ENVIRONMENT DIVISION.

For More Information

- For information about using the RESERVE clause (an extension to ANSI X3.23-1974 COBOL) to indicate that the compiler should treat certain keywords as application-specific, refer to "SPECIAL-NAMES" in Section 5, "ENVIRONMENT DIVISION."
- For the list of application-specific keywords, refer to Appendix B, "Reserved Words and Keywords."

System-Name

A system-name is a word that you use to communicate with the operating environment. A system-name can be one of the following two types:

- A computer-name, such as MICROA and A17, that identifies the computer on which the program is to be compiled or run. Computer-name is treated as a comment-entry.
- An implementor-name, such as ODT and SW1, that refers to a particular feature available with your system.

The rules for forming a system-name are as follows:

- Make the system-name no more than 30 characters long.
- Select each character from the set of characters A through Z, 0 through 9, and the hyphen (–).
- Do not use the hyphen as the first or last character of a system-name.

User-Defined Words

A user-defined word is a word that you define to complete the format of a clause or statement. The rules for forming a user-defined word are as follows:

- Make the user-defined word up to 30 characters long.
- Select each character from the set of characters A through Z, 0 through 9, and the hyphen (–).
- Do not use the hyphen as the first or last character of a word.
- Do not use a reserved word.

- Make sure that all user-defined words, except for level-numbers and segment-numbers, are unique. You can use qualification to ensure that a word is unique. Level-numbers and segment-numbers do not need to be unique. A given level-number or segment-number can be identical to a paragraph-name or a section-name.
- Include at least one alphabetic character in all user-defined words except paragraph-names, section-names, level-numbers, text-names, library-names, family-names, and segment-numbers.

Table 2–6 shows some of the user-defined words and explains how they are used in your program.

Table 2–6. User-Defined Words

User-Defined Word	Explanation
Alphabet-name	Assigns a name to a specific character set and collating sequence in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.
CD-name	Assigns a name to a communication description (CD).
Condition-name	Assigns a name to a specific value, set of values, or range of values within a complete set of values that a conditional variable can have. A conditional variable is a data item that can assume more than one value. The values that it can assume have condition-names assigned to them. A condition-name can also assign a name to a switch or device. Condition-names can be defined in the DATA DIVISION or in the SPECIAL-NAMES paragraph within the ENVIRONMENT DIVISION. You can use a condition-name as an abbreviation for a relation condition. A relation condition states that the associated conditional variable is equal to one of the set of values to which that condition-name is assigned.
Data-name	Names a data item described in a data-description entry.
Family-name	Identifies a family of disks on which a file resides.
File-name	Names a file described in a file-description entry or a sort-merge file description (FD) entry in the FILE SECTION of the DATA DIVISION.
Index-name	Names an index associated with a specific table.
Level-number	Assigns a one- or two-digit number that shows the hierarchical position of a data item or a special property of a data-description entry.
Library-name	Names a COBOL library that is to be used in conjunction with the COPY statement by the compiler for a given source program compilation.
Mnemonic-name	Assigns a user-defined word to an implementor-name. These associations are established in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

Table 2–6. User-Defined Words

User-Defined Word	Explanation
Paragraph-name	Identifies and begins a paragraph in the PROCEDURE DIVISION Paragraph-names are equivalent only if composed of the same sequence and number of digits and/or characters.
Program-name	Identifies a source program in the IDENTIFICATION DIVISION.
Record-name	Names a record described in a record-description (RD) entry in the DATA DIVISION.
Report-name	Names a Report Writer report described in a report-description (RD) entry in the DATA DIVISION.
Routine-name	Identifies a procedure written in a language other than COBOL74.
Section-name	Names a section in the PROCEDURE DIVISION. Section-names are equivalent only if composed of the same sequence and number of digits and/or characters.
Segment-number	Groups sections in the PROCEDURE DIVISION for the purposes of segmentation.
Text-name	Specifies the external identification of a file in the COBOL library.

Note: *Family-name and Routine-name are extensions to ANSI X3.23-1974 COBOL.*

This manual also uses user-defined words that are not identified in Table 2–6 in order to clarify the meaning of a format notation. For example, an identifier for an event is called an event-identifier. You can determine that a word is user defined if it appears in lowercase letters in the format notation. Some of the user-defined words described in Volume 2 are form-name, formlibrary-name, and group-list-name.

For information about qualification, refer to Section 6, "Data Concepts."

Literals

A literal is a string of characters whose value is either the ordered set of characters of which the literal is composed or a reserved word that refers to a figurative constant.

Every literal is one of the following five types:

- Nonnumeric
- Numeric
- Floating point
- Undigit
- Kanji

Nonnumeric

A nonnumeric literal is a string of characters delimited on both sides by quotation marks. An example is "Month Year". You can use any allowable character in the character set of the computer to form a nonnumeric literal. Nonnumeric literals are in the alphanumeric category.

The rules for the formation of nonnumeric literals are as follows:

- Nonnumeric literals can be between 1 and 160 characters long.
- A single quotation mark is delimited by two contiguous quotation marks within a nonnumeric literal. Each embedded pair of contiguous quotation marks represents a single quotation mark character.
- Delimiting quotation marks are excluded from the value of the nonnumeric literal.
- Except delimiting quotation marks, all other punctuation characters within the literal are considered to be part of the nonnumeric literal.
- Any literals used for arithmetic computation must not be enclosed in quotes as nonnumeric literals. The literal "7.7" is a nonnumeric literal and is stored differently from the numeric literal 7.7 (not enclosed in quotes).

Examples

The following are examples of nonnumeric literals. The string on the left shows the literal as it appears in your source program. The string on the right shows the literal as it is stored by the compiler.

Literal in Source Program	Literal Stored by Compiler
"ANNUAL DUES"	ANNUAL DUES
"(MILES/GALLON)"	(MILES/GALLON)
"–123.456"	–123.456
"A""B"	A"B
"""LIMITATIONS"""	"LIMITATIONS"

Numeric

A numeric literal is a character string selected from the digits 0 through 9, the plus sign (+), the minus sign (–), and the decimal point.

The rules for the formation of numeric literals are as follows:

- A literal can be between 1 and 23 digits long.
- A literal must contain at least one digit.
- A literal must not contain more than one sign character. If a sign is used, it must appear as the leftmost character of the literal. If the literal is unsigned, it is positive.

- A literal must not contain more than one decimal point. The decimal point is treated as an assumed decimal point and can appear anywhere within the literal except as the rightmost character. If the literal contains no decimal point, the literal is an integer. A literal that conforms to the rules for the formation of numeric literals, but is also enclosed in quotation marks, is a nonnumeric literal and is treated as such by the compiler.
- The value of a numeric literal is the algebraic quantity represented by the characters in the numeric literal. Every numeric literal is in the numeric category. The size of a numeric literal in standard data-format characters is equal to the number of digits that you specify.

Examples

The following are examples of numeric literals:

```
12345
.005
+1.008
-.0965
7842.1
```

Floating Point

A floating-point literal is a string of characters that uses two numbers to represent one original number. The first number is called the mantissa. It has a value between 0 (zero) and nine. The second number is called the exponent. It represents the power of ten by which the first number is multiplied to obtain the original number. The format of a floating-point literal is as follows:

```
mantissa E exponent
```

For example, 8,765,432.1 is 8.7654321E6 in floating-point notation.

The advantage of floating-point notation is that you can handle very small and very large numbers easily. You can use floating-point literals as alternatives to coding numeric literals. You should consider using floating-point literals with REAL and DOUBLE data items.

The rules for the formation of floating-point literals are as follows:

- The mantissa can be signed and must have one decimal point.
- The exponent can be signed and must be an integer.

Table 2–7 shows the smallest and largest permitted values for single-precision and double-precision data items using floating-point literals.

Table 2–7. Range of Values Permitted for Floating-Point Literals

Type	Smallest Permitted Value	Largest Permitted Value
Single	8.75811540203E–47	4.31359146674E68
Double	1.938545857137585833556 40E–29581	1.948829382050280791244 69E29603

Examples

The following are examples of floating-point literals.

```
1.E-40
-.0023E29
+.0012345E-5
+1.2E9500
2.E40
+123.45678901234E20
```

Undigit (Extension to ANSI X3.23-1974 COBOL)

An undigit literal is a string of characters that represents the hexadecimal equivalent of an EBCDIC character.

Rules for Forming an Undigit Literal

- Delimit both ends of the literal with the commercial at sign (@) character.
- Select the characters from the hexadecimal digits 0 (zero) through 9 and the characters A through F.

Rules for Using an Undigit Literal

You can use an undigit literal only in numeric and alphanumeric contexts. In both contexts, it is recommended that the sizes of the undigit literal and the associated data item match. The results of using sizes that do not match can be unpredictable.

The compiler interprets the undigit literal as either a 4-bit numeric literal or an 8-bit alphanumeric literal. You determine the interpretation of the undigit literal by specifying the type of the data item to which the undigit literal is associated.

Alphanumeric Contexts

An undigit literal is alphanumeric when the category of the associated data item is alphanumeric.

An undigit literal is interpreted as alphanumeric in the following cases:

- In the VALUE clause associated with an alphanumeric, alphabetic, or group data item, or in the VALUE clause of condition-names associated with such items
- In the MOVE statement, where the category of the receiving field is either alphanumeric or alphabetic
- In the conditional expression of an IF, PERFORM, or SEARCH statement, where the category of the other relational operand is either alphanumeric or alphabetic
- In an INSPECT, STRING, UNSTRING, DISPLAY, STOP, DISABLE, or ENABLE statement
- When used with ALL, and the item associated with the undigit literal is an alphanumeric, alphabetic, or group data item.

Numeric Contexts

An undigit literal is numeric if it appears in the VALUE clause associated with a COMPUTATIONAL item, whether or not the undigit literal appears with ALL.

You can use undigit literals for numeric destinations in the MOVE statement and in the VALUE clauses associated with numeric data items, when a program meets all of the following criteria:

- The usage of the destination is COMP.
- The picture string for the destination does not contain the symbols S, V, or P.
- There is neither a SIGN clause nor a BLANK WHEN ZERO clause associated with the data item associated with the undigit literal.

Examples

The following are examples of undigit literals and their EBCDIC equivalents. A program might include an undigit literal to send control sequence messages to a remote terminal.

Undigit literal	EBCDIC Equivalent
@0D@	CR (carriage return)
@25@	LF (line feed)

Kanji (Extension to ANSI X3.23-1974 COBOL)

A Kanji (National Character) literal is intended to be used with KANJI data items as an alternative to using standard nonnumeric literals. The general format for a Kanji literal is the following:

```
NC"character-string"
```

The rules for the formation of a Kanji literal are as follows:

- A Kanji literal is bounded on the left by the separator *NC"* and on the right by a quotation mark (").
- The character string contains a string of Kanji characters between the blank space after the first quotation mark and the blank space preceding the end quotation mark. The compiler recognizes the start and the end of a Kanji character string by means of the two blank spaces within the quotation marks.
- Spaces are not allowed within the character string of Kanji characters.
- A Kanji literal can be from 1 to 80 characters long.

A Kanji literal occupies twice as much storage space as a literal that is not Kanji.

Section 3
File and Task Concepts

To develop successful COBOL74 programs, you need to understand some concepts that underlie files and tasks. For example, to use files efficiently you need to be knowledge of file attributes, file organization, and access mode. An understanding of the tasking concept includes knowledge of task attributes.

Physical Aspects of a File

File information describes both the physical aspects of the file and the logical characteristics of the data in the file.

The physical characteristics of a file describe the data as it appears on the input or output medium. This description refers to the grouping of the logical records within the physical limitations of the file medium.

A physical record is a physical unit of data with a size and recording mode convenient for storing data on an input or output device of a particular computer. The size of a physical record is hardware dependent and has no direct relationship to the size of the information file contained on a device.

The distinction between a physical record and a logical record, which is described next, is important.

Logical Aspects of a File

The conceptual characteristics of a file are the explicit definitions of each logical entity in the file. In a COBOL74 program, the input or output statements refer to one logical record.

A COBOL logical record is related information that is uniquely identifiable and treated as a unit.

One or more logical records can be contained in a single physical record. In a mass storage file, however, one logical record could require more than one physical record. In this manual, references to records mean logical records unless the term *physical record* is specified.

The concept of a logical record is not restricted to file data. A logical record can also apply to the definition of working-storage. Thus, working-storage can be grouped into logical records and defined by a series of record-description entries.

For More Information

Refer to "Levels" in Section 6, "Data Concepts," for more information about records.

Assigning a File to a Device

The logical file mechanism supports access to remote and port files as well as to other devices. The devices to which a file can be assigned are specified in the SELECT clause of the ENVIRONMENT DIVISION.

Remote Files

Assignment of a file to a remote device enables the use of the logical file mechanism to access a family of terminal or station devices. This mechanism uses traditional file-handling methods rather than the specialized, data-communications handling methods of the communication module.

Port Files

User processes communicate across a network through the standard I/O file mechanism using a special kind of file called a port file. The program opens and closes port files just like other files. A user can communicate with a process by performing read and write operations to a port file. A port file is composed of one or more port subfiles, each of which can be connected to a different process. Communication between processes on the same host or system is affected by using port files without going through a network. In addition, there is a service associated with each port file. This service can be assigned with the SERVICE file attribute. For example, a user can set the SERVICE file attribute to BNANATIVESERVICE.

A subfile provides a two-way, point-to-point, logical communication path between two programs. To establish this path, each program must describe the desired connection. This process is called matching. Each program describes its matching properties by using file attributes.

In the SELECT clause of the ENVIRONMENT DIVISION, the ACTUAL KEY clause specifies the subfile index of a port file. If the ACTUAL KEY value is 0 (zero), the OPEN statement opens all subfiles, the READ statement performs a nonselective read operation, the WRITE statement performs a broadcast write operation, and the CLOSE statement closes all opened subfiles associated with the port file.

If no ACTUAL KEY clause is specified, the file must contain a single subfile, which is assumed to be the subfile in associated I/O statements.

For More Information

The *I/O Subsystem Programming Guide* provides more information about and an example of coding a port file application.

File Attributes

File attributes provide the capability for defining, monitoring, or changing file properties or attributes.

Note: *File attributes provide you with access to functionality not otherwise available within the language. File attributes can also be used to declare and access files. When both a file attribute and standard COBOL74 syntax are available to accomplish a desired function, it is always preferable to use the standard COBOL74 syntax. Changing the attribute can lead to unexpected results in cases when the attribute is also used or altered by the compiler.*

For More Information

- Refer to the *I/O Subsystem Programming Guide* for information about how to use file attributes.
- Refer to the *File Attributes Programming Reference Manual* for the details about a specific attribute.

File-Attribute Identifiers

File-attribute identifiers provide the ability to monitor, manipulate, define, or dynamically change any specific file attribute.

The general format of the file-attribute identifier follows:

```
ATTRIBUTE attribute-name { [ ]    } file-name [(arithmetic-expression)]
                         { OF }
                         { IN }
```

Explanation of Format

The attribute-name is defined by the system. Examples of attribute names include FILETYPE, TITLE, and MAXRECSIZE. For more information on attribute names, see the *I/O Subsystem Programming Guide.*

Port Files

A subfile index is required for accessing or changing attributes of a subfile of a port file. A subfile index must be an arithmetic expression.

The arithmetic-expression option can be used only with a port file. The value of the expression specifies the subfile of the file that is affected. If the arithmetic-expression is not specified, the attribute of the port is accessed. If the arithmetic-expression is specified and its value is not 0 (zero), it specifies a subfile index and causes the attribute of the subfile to be accessed. If the arithmetic-expression is specified and its value is 0 (zero), then the arithmetic-expression causes the attribute of all subfiles to be accessed.

File-Attribute Categories

A file attribute belongs to one of five categories, depending on the type of attribute-name specified in the file-attribute identifier. The five file-attribute categories are described in the following paragraphs.

Alphanumeric File-Attribute Identifier

Where allowed in syntax, an alphanumeric file-attribute identifier is similar to an elementary alphanumeric DISPLAY data item that has a size equal to the maximum size allowed for the specified attribute. The contents of the alphanumeric file-attribute identifier are left-justified with space-fill. Alphanumeric file-attribute identifiers are allowed as operands in relation conditions and as sending operands in Format 1 MOVE statements.

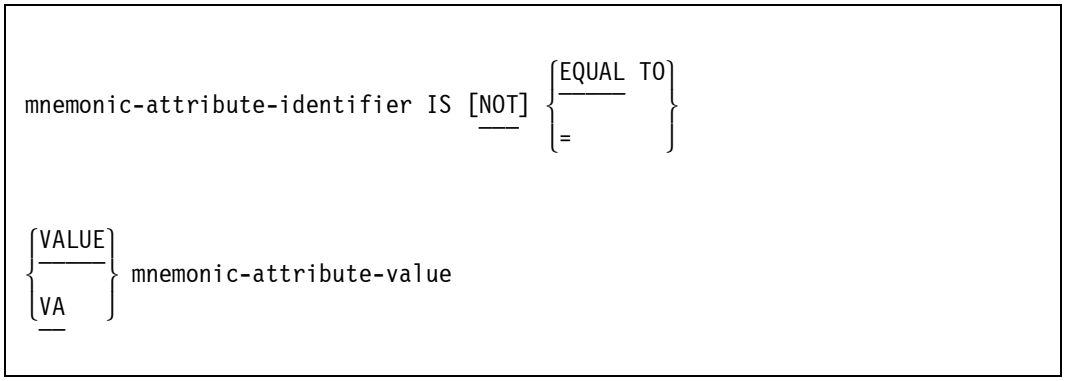
Numeric File-Attribute Identifier

Where allowed in syntax, a numeric file-attribute identifier is treated as if it were a data item declared as PICTURE S9(11) USAGE BINARY EXTENDED. Numeric file-attribute identifiers are allowed as operands in arithmetic expressions and as sending operands in Format 1 MOVE statements. Some numeric file attributes represent information about the number of areas, blocks, records, and so forth in the file. These attributes are “one relative” in that their value specifies the exact number of areas, blocks, records, and so forth in the file.

Mnemonic File-Attribute Identifier

Certain file attributes are associated with values that are best expressed as mnemonic-names because the magnitude of the actual value is unrelated to its meaning. Mnemonic file-attribute identifiers can appear as the subject of a mnemonic-attribute relation condition, with the name for one of the values associated with the specified attribute used as the object. The name for the attribute value must follow the reserved word VALUE as shown in the next example.

Mnemonic-attribute relation conditions are allowed in any conditional expression. The general format of a mnemonic-attribute relation condition follows:



Mnemonic-attribute relation conditions cannot be abbreviated. The names for the mnemonic-attribute values are system-names and are not necessarily reserved words. Boolean file attributes are considered mnemonic attributes in COBOL and are associated with the mnemonic-attribute values TRUE and FALSE.

Boolean File-Attribute Identifier

These attributes are referenced in the same manner as numeric file-attribute identifiers. These attributes return the value 1 for TRUE and 0 for FALSE.

Event File-Attribute Identifier

The file attributes of the type EVENT are the same as the variables of the USAGE EVENT identifier. They can be used whenever an event-identifier is allowed.

File Organization and Access Methods

The organization of a file determines the access mode of that file. The organization can be sequential, relative, or indexed.

Sequential Organization

Sequential files are organized so that each record in the file except the first has a unique predecessor record, and each record except the last has a unique successor record. These predecessor/successor relationships are established by the order of the WRITE statements when the file is created. Once established, the predecessor/successor relationships do not change except when records are added to the end of the file.

Records in a file with sequential organization can be accessed in the sequence established when the records were written to the file. A sequential mass storage file can be used for input and output at the same time. This feature enables a record to be read, updated, and returned with modifications to its original position for purposes of file maintenance.

A file with sequential organization enables you to specify records in rerun points and share memory areas among files.

Relative Organization

Relative I/O enables you to access file records in either a random or a sequential manner. Each record in a relative file is uniquely identified by an integer value greater than 0 (zero). The value is called the relative record number. It specifies the logical, ordinal position of the file record.

Records are read from, and written to, the file based on the relative record number. For example, the tenth record is the record addressed by relative record number 10 and occupies the tenth record area, whether or not record areas 1 through 9 have been written.

Indexed Organization

Indexed I/O enables you to access file records in either a random or a sequential manner. Each record in an indexed file is uniquely identified by the value of one or more keys within that record.

The record description of an indexed file includes one or more key data items, each associated with an index. The index provides a logical path to the data records, based on the contents of the record keys in each record.

Current-Record Pointer

The current-record pointer is a conceptual entity. It indicates to the program the next record to be accessed within a given file.

For a file opened in the output mode, the current-record pointer concept has no meaning.

For sequential files, the current-record pointer indicates the next record for OPEN and READ statements.

For relative and indexed files, the current-record pointer indicates the next record for OPEN, READ, SEEK, and START statements.

Task Attributes

A task attribute is any one of a number of items that describe and control various aspects of the execution of a process. The program can access a task attribute by using a task identifier. The task identifier is a data item declared with task usage in the USAGE clause.

A program can assign or change the value of a task attribute by using the CHANGE statement or the MOVE statement.

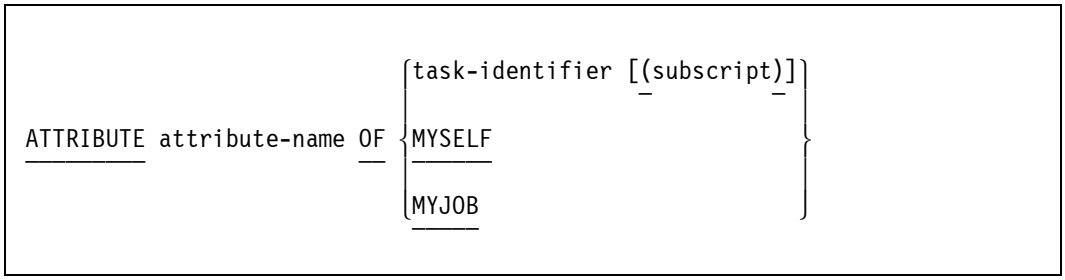
The syntax for setting task attributes is documented under the CHANGE statement in Section 9, "PROCEDURE DIVISION Statements."

More information about tasking with COBOL74 is provided in the *Task Management Programming Guide.*

Task-Attribute Identifiers (Extension to ANSI X3.23-1974 COBOL)

Task-attribute identifiers are used to change or interrogate the task attributes of related processes in a synchronous or asynchronous processing environment. You should be familiar with the concepts of tasking, the task attributes, and their possible variations.

The general format of the task-attribute identifier follows:



Explanation of Format

task-identifier

A task-identifier can be attached to a program. For example,

```
CHANGE ATTRIBUTE NAME OF PROG2 TO "OBJECT/TASK.".
```

subscript

The optional subscript is used when the task item is declared with an OCCURS clause. A maximum of one subscript is permitted. For example,

```
CHANGE ATTRIBUTE DECLAREDPRIORITY OF PROG1 (1) TO 1.
```

MYSELF

The reserved word MYSELF is a compiler-supplied task item that enables a program to access its own task attributes. Thus, any attribute of a given task can be referenced within that task as *ATTRIBUTE attribute-name OF MYSELF*. For example,

```
CHANGE ATTRIBUTE DECLAREDPRIORITY OF MYSELF TO 90.

CHANGE ATTRIBUTE DECLAREDPRIORITY OF ATTRIBUTE PARTNER
OF MYSELF TO 65.
```

The second example illustrates another task running with a task that you are running. The PARTNER attribute refers to the other task and the example changes the DECLAREDPRIORITY of the other task.

MYJOB

The reserved word MYJOB is a compiler-supplied task item that enables a program to access the task attributes of its job. Thus, any attribute of a job can be referenced in any task of that job as *ATTRIBUTE attribute-name OF MYJOB*. For example,

```
CHANGE ATTRIBUTE RESTART OF MYJOB TO 5.
```

Task-Attribute Types

Task attributes of type EVENT can be used in place of any valid event-identifier (USAGE EVENT).

Task attributes of type TASK are themselves task-identifiers of some other associated task. This type of attribute can be employed to access or manipulate the task attributes of the associated task.

Task attributes of type POINTER accept or return an alphanumeric DISPLAY item.

All other task attributes accept or return a numeric identifier, literal, arithmetic expression, or the value associated with a mnemonic. If the value is not in the permissible range for the attribute specified, an error occurs at compile time or at execution time.

A task-attribute-mnemonic is a name associated with a constant value for an attribute that has a set number of predetermined possible values.

The attribute names and their mnemonics are not treated as COBOL reserved words. They are reserved only within the context in which they are used and can be also used as data-names or procedure-names if they are not regular reserved words. Therefore, if a data-name has the same name as the system attribute mnemonic, the value assigned to the attribute by a CHANGE statement is determined by the use of the optional word VALUE. If the word VALUE is present, the attribute is set to the value of the system mnemonic. If the word VALUE is omitted, the attribute is set to the current value of data-name.

Interrogating Task Attributes

You can interrogate a task attribute in any of the following ways:

- By specifying a task attribute in the sending field of a MOVE statement. The following is an example:

```
MOVE ATTRIBUTE PROCESSTIME OF PROG3 TO PRINT-P-TIME.
```

  When an attribute is moved into an area by a MOVE statement, the use of the receiving field must be consistent with the type of the attribute. Boolean attributes (those attributes having mnemonic values of TRUE or FALSE) return the number 0 if FALSE or the number 1 if TRUE. Boolean or INTEGER attributes should be moved to a numeric receiving field. Type POINTER attributes should be moved to a nonnumeric receiving field.

- By making the task attribute the subject or object of a condition. The following is an example:

```
IF ATTRIBUTE LOCKED OF PROG1 (1) = TRUE
   CHANGE ATTRIBUTE TASKVALUE OF PROG1 (1) TO -1.

IF ATTRIBUTE NAME OF PROG2 = "X/Y/Z."
PERFORM PRINT-ROUTINE
UNTIL ATTRIBUTE STATUS OF PROG1 (2) = VALUE SUSPENDED.
```

- By using attributes with an implicit numeric class in DISPLAY statements. The following is an example:

```
DISPLAY ATTRIBUTE STATUS OF PROG2
        ATTRIBUTE PROCESSTIME OF PROG2.
```

- By using attributes with an implicit numeric class in place of any identifier in an arithmetic statement, except the receiving-field identifier.

Task attributes can be tested against their associated attribute mnemonics.

The program fragment of Example 3–1 sets the BDBASE option of the OPTION task attribute. Accessing specific options of the type OPTION task attribute requires use of mnemonic-attribute identifiers. The mnemonic-attribute identifiers represent specific bits in the type OPTION task attribute word. One way to access these bits is to use the Format 3 MOVE statement.

```
11000 WORKING-STORAGE SECTION.
11100 01  OPTION-WORD  PIC 9(11)  BINARY.
11200 01  VALUE-ONE    PIC 9(11)  BINARY  VALUE 1.
11300 PROCEDURE DIVISION.
11400 P-1.
11500     MOVE ATTRIBUTE OPTION OF MYSELF TO OPTION-WORD.
11600     MOVE VALUE-ONE TO OPTION-WORD [0:VALUE BDBASE:1].
11700     CHANGE ATTRIBUTE OPTION OF MYSELF TO OPTION-WORD.
```

Example 3–1. Setting the BDBASE Option

For More Information

For a description of the rules that govern move operations, refer to “MOVE” in Section 9, “PROCEDURE DIVISION Statements.”

Section 4
IDENTIFICATION DIVISION

The first division of the source program, the IDENTIFICATION DIVISION, provides identifying information about the source program such as the name of the program, the creation date, the compilation date, and other documentation information.

With the exception of the DATE-COMPILED paragraph, the entire IDENTIFICATION DIVISION is copied from the input source program and listed on the output listing.

The general format of the IDENTIFICATION DIVISION is as follows:

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

[ PROGRAM-ID. program-name.]

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

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

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

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

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

Note: *Because the AUTHOR, INSTALLATION, DATE-WRITTEN, and SECURITY paragraph headers have associated text consisting only of comment-entries, they are not further documented.*

Explanation of Format

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

ID DIVISION is a synonym for IDENTIFICATION DIVISION. (The reserved word ID is an extension to ANSI X3.23-1974 COBOL.)

The comment-entry can be any combination of characters from the character set of the computer. The continuation of the comment-entry by the use of the hyphen in the indicator area is not permitted; however, the comment-entry can be contained in one or more lines.

Example 4–1 shows coding of the IDENTIFICATION DIVISION.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.         GENERAL-UPDATE.
AUTHOR.             JOHN SMITH.
INSTALLATION.       MISSION VIEJO.
DATE-WRITTEN.       SEPTEMBER 10, 1984.
DATE-COMPILED.      SEPTEMBER 15, 1984.
SECURITY.           FOREVER.
```

Example 4–1. Coding the IDENTIFICATION DIVISION

PROGRAM-ID Paragraph

The PROGRAM-ID paragraph gives the name by which a program is identified. The general format is as follows:

```
PROGRAM-ID. program-name.
```

Explanation of Format

The program-name identifies the source program and all listings pertaining to a particular program. The program-name must conform to the rules for formation of a user-defined word.

The following describes the different uses of the program-name:

Value of FEDLEVEL Compiler Option	Use of Program-name
<5	The program-name is treated as a comment.

Value of FEDLEVEL Compiler Option	Use of Program-name
=5	The program-name is the entry-point name when the program is used as a library. If a library program does not use the PROGRAM-ID paragraph to designate an entry-point name, the entry-point name is PROCEDUREDIVISION.

For More Information

Refer to Section 15, "Libraries" for more information about creating a library.

DATE-COMPILED Paragraph

The DATE-COMPILED paragraph provides the compilation date in the IDENTIFICATION DIVISION source program listing.

The general format of this paragraph is as follows:

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

Explanation of Format

The DATE-COMPILED paragraph causes the current date to be inserted during compilation. If a DATE-COMPILED paragraph is present, it is replaced during compilation with a paragraph of the following form:

```
DATE-COMPILED. current-date.
```

The current-date represents the date and time at which the compilation of the source program started.

The comment-entry can be any combination of characters from the character set of the computer. The continuation of the comment-entry buy use of the hyphen in the indicator area is not permitted; however, the comment-entry can be contained in one or more lines.

Section 5
ENVIRONMENT DIVISION

The second division of a source program, the ENVIRONMENT DIVISION, specifies a standard method of expressing aspects that depend on the physical characteristics of a specific computer. This division enables you to specify the compiling computer, the object computer, the files handled by the object program, and the I/O procedures to be used.

The ENVIRONMENT DIVISION must be included in every COBOL source program and must begin with the reserved words ENVIRONMENT DIVISION followed by a period and a space.

The ENVIRONMENT DIVISION consists of two sections: the CONFIGURATION SECTION and the INPUT-OUTPUT SECTION.

The CONFIGURATION SECTION explains the characteristics of the source computer and the object computer.

The INPUT-OUTPUT SECTION provides the information needed to control transmission and handling of data between external media and the object program.

The following general format shows the overall syntax for the ENVIRONMENT DIVISION. The individual sections and paragraphs are further defined later in this section.

The general format of the ENVIRONMENT DIVISION is as follows:

```
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]
```

CONFIGURATION SECTION

The CONFIGURATION SECTION lists the characteristics of the source computer and the object computer. This section is divided into the following three paragraphs:

- The SOURCE-COMPUTER paragraph, which describes the computer configuration on which the source program is compiled
- The OBJECT-COMPUTER paragraph, which describes the computer configuration on which the object program produced by the compiler is to be run
- The SPECIAL-NAMES paragraph, which relates hardware names used by the COBOL compiler to the mnemonic-names in the source program

SOURCE-COMPUTER

The SOURCE-COMPUTER paragraph identifies the computer on which the program is to be compiled.

The general format of this paragraph is as follows:

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

Explanation of Format

The computer-name can be any single COBOL word. It is handled as a comment entry that describes the computer on which the source program is to be compiled. The computer-name is for documentation only.

For information on the effects of specifying DEBUGGING MODE, refer to Section 11, “Debugging.”

OBJECT-COMPUTER

The OBJECT-COMPUTER paragraph identifies the computer on which the program is to be executed.

The general format of this paragraph is as follows:

```
OBJECT-COMPUTER. computer-name

 ┌                                   ┌            ┐ ┐
 │                                   │ WORDS      │ │
 │ , MEMORY SIZE IS integer-1        < CHARACTERS > │
 │                                   │ MODULES    │ │
 └                                   └            ┘ ┘

 ┌                                 ┌         ┐ ┐
 │ , DISK SIZE IS integer-2        < WORDS   > │
 │                                 │ MODULES │ │
 └                                 └         ┘ ┘

[ , STACK SIZE IS integer-3 ]

[ , CODE SEGMENT-LIMIT IS integer-4 [ WORDS ] ]

[ , SEGMENT-LIMIT IS segment-number ]

[ , PROGRAM COLLATING SEQUENCE IS alphabet-name ].
```

Explanation of Format

computer-name

The computer-name is a system-name that identifies the hardware for which object code is to be generated. A valid OBJECT-COMPUTER system-name can be any single COBOL74 word. It is treated as a comment.

MEMORY SIZE

The MEMORY SIZE clause is used only in conjunction with the SORT statement. The SORT statement can also specify MEMORY SIZE and takes precedence over the OBJECT-COMPUTER paragraph. When MEMORY SIZE is not specified in either the SORT statement or the OBJECT-COMPUTER paragraph, a default memory size of 12,000 words is assumed. If this option is used and a SORT statement does not appear in the program, the option is ignored. One module of memory is equivalent to 16,384 words of memory.

DISK SIZE (Extension to ANSI X3.23-1974 COBOL)

The DISK SIZE clause is used only in conjunction with the SORT statement. If this option is omitted in a sort program, DISK SIZE is assumed to be 900,000 words. If this option is used and a SORT statement does not appear in the program, the option is ignored. One module of disk is equivalent to 1.8 million words of disk.

STACK SIZE (Extension to ANSI X3.23-1974 COBOL)

The STACK SIZE clause is for documentation purposes only.

CODE SEGMENT-LIMIT (Extension to ANSI X3.23-1974 COBOL)

The CODE SEGMENT-LIMIT clause specifies the value for the size of an object-code segment in words. During the code-generation process, when the compiler completes the code for a paragraph or section, it ends the current segment and starts a new segment if the size of the current segment exceeds the target value.

Integer-4 must be in the range 256 through 7000. If the CODE SEGMENT-LIMIT clause is not specified, the default segment size is 1500 words.

For information about the SEGMENT-LIMIT clause, refer to Section 10, “Segmentation.”

PROGRAM COLLATING SEQUENCE

If the PROGRAM COLLATING SEQUENCE clause is specified, the collating sequence associated with alphabet-name is used to determine the truth value of any nonnumeric comparisons that are explicitly specified in relation or condition-name conditions or implicitly specified by the presence of a CONTROL clause in a report-description entry.

For localization purposes, the program can specify the PROGRAM COLLATING SEQUENCE clause and a CCSVERSION collating sequence associated with an alphabet-name. In this case, the truth value of the alphabetic characters that are explicitly specified in the class condition do not always consist entirely of the characters A through Z and the space. The class of alphabetic characters is determined based on the system collating sequence when the CCSVERSION collating sequence is specified.

If the PROGRAM COLLATING SEQUENCE clause is not specified, the EBCDIC collating sequence is used. If the PROGRAM COLLATING SEQUENCE clause is specified, the program-collating sequence is the collating sequence associated with the alphabet-name specified in that clause.

The PROGRAM COLLATING SEQUENCE clause is also applied to any nonnumeric merge or sort keys, unless the COLLATING SEQUENCE phrase of the respective MERGE or SORT statement is specified.

For information about specifying a collating sequence using the internationalization features, refer to Section 16, “Internationalization.”

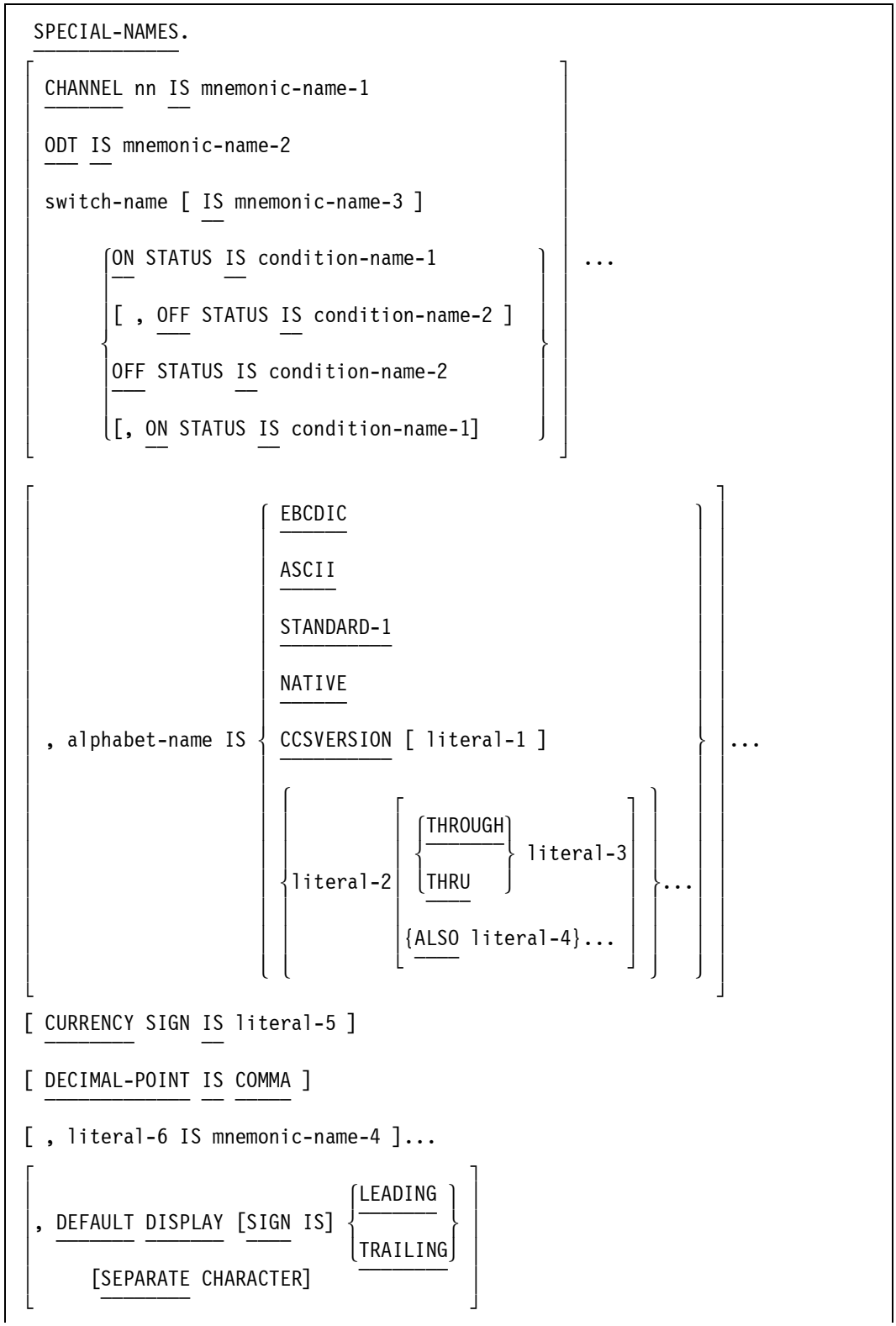
SPECIAL-NAMES

The SPECIAL-NAMES paragraph enables you to do the following:

- Supply a name for a channel number, a switch, or the Operator Display Terminal (ODT)
- Supply a name for a character code set or collating sequence
- Specify a default sign position (Extension to ANSI X3.23-1974 COBOL)
- Designate a set of words to be recognized as reserved words for a specified kind of application (Extension to ANSI X3.23-1974 COBOL)
- Define a currency sign in edited numeric data
- Specify the role of the comma and period in edited numeric data
- Rename a file title for binding purposes

Volume 2 of this manual describes the SPECIAL-NAMES paragraph extensions for specific product interfaces.

The general format for the SPECIAL-NAMES paragraph is as follows:



```
[                                                                    ]
[                 {COMPUTATIONAL}                                    ]
[                 {------------}                                     ]
[  , DEFAULT      {COMP        } [ SIGN IS ] {LEADING }              ]
[                 {----        }             {-------}              ]
[                 {CMP         }             {TRAILING}             ]
[                 {---         }             {--------}             ]
[                                                                    ]
[     [ SEPARATE CHARACTER ]                                         ]
[                                                                    ]
[ RESERVE WORD LIST IS NETWORK CAPABLE ]
```

Explanation of Format

CHANNEL nn IS mnemonic-name

The *CHANNEL nn IS mnemonic-name* clause relates a mnemonic-name to a channel number, where nn is an integer from 01 to 11. You can use the mnemonic-name in a WRITE or SEND statement.

ODT IS mnemonic-name

The *ODT IS mnemonic-name* clause associates a user-defined word with the operator display terminal (ODT). You can use the mnemonic name in an ACCEPT or DISPLAY statement.

switch-name

The switch-name can be SW1, SW2, SW3, SW4, SW5, SW6, SW7, or SW8. The program uses switches to communicate with the external environment. A switch has a value of either ON or OFF. You can define a condition-name for each value of the switch. You can then check the status of the switch by testing the condition-name. You can set the switch at program initiation time or through Work Flow Language (WFL) using the task attributes SW1, SW2, SW3, SW4, SW5, SW6, SW7, and SW8.

The *IS mnemonic-name-3* clause associates a user-defined word with a switch-name.

The ON STATUS IS phrase associates a condition-name with the ON status of a switch. The condition-name is TRUE when the switch is set and FALSE when the switch is not set.

The OFF STATUS IS phrase associates a condition-name with the OFF status of a switch. The condition-name is TRUE when the switch is not set, and FALSE when the switch is set.

A condition-name designates a value for either the ON or OFF value of a switch. You can associate one condition-name value with the ON status and another with the OFF status. You define the condition-name as a level-number 88 data item in the DATA DIVISION.

alphabet-name IS

The *alphabet-name IS* clause relates an alphabet-name to a collating sequence or character set. The alphabet-name refers to a collating sequence when you use it in the PROGRAM COLLATING clause of the OBJECT-COMPUTER paragraph or the COLLATING SEQUENCE phrase of a MERGE or a SORT statement. The alphabet-name refers to a character code set when you use it in a CODE-SET clause of a file-description entry.

The ASCII or STANDARD-1 phrase identifies alphabet-name as the collating sequence or the character code set defined in the American National Standard Code for Information Interchange, X3.4-1968.

The NATIVE phrase or EBCDIC phrase identifies alphabet-name as the native character code set or native collating sequence. The native character code set is the character code set associated with USAGE IS DISPLAY, EBCDIC.

The translation tables for EBCDIC-to-ASCII and ASCII-to-EBCDIC translation determine the correspondence between characters of the ASCII character code set and characters of the EBCDIC character code set.

If the CCSVERSION option is specified, then the character code set and the collating sequence identified with the alphabet-name is the system collating sequence. If the CCSVERSION phrase is specified without literal-1, the collating sequence identified with the alphabet-name is the internationalized system default collating sequence. If the CCSVERSION phrase is specified with literal-1, the collating sequence is the one identified by literal-1, provided that literal-1 is valid. The alphabet-name cannot be referred to in a CODE-SET clause.

If you do not specify the CCSVERSION option, the default value that will be used for CCSVERSION is always ASERIESNATIVE, regardless of the default system value for CCSVERSION when the program is compiled or run.

Example 5–1 lists a program in which the CCSVERSION option is not specified.

```
000100 IDENTIFICATION DIVISION.
000200 ENVIRONMENT DIVISION.
000900 INPUT-OUTPUT SECTION.
001000 FILE-CONTROL.
001100    SELECT TEST ASSIGN TO DISK.
001200 DATA DIVISION.
001300 FILE SECTION.
001400    FD TEST.
001500    01 TEST-DATA PIC X(10).
001600 PROCEDURE DIVISION.
001700 BEGIN.
001800    IF ATTRIBUTE TITLE OF TEST = "TEST."
001900       DISPLAY "TEST".
001920    MOVE HIGH-VALUES TO TEST-DATA.
001940    IF TEST-DATA = HIGH VALUES
```

```
001960        DISPLAY "OK".
002000 STOP RUN.
```

Example 5–1. CCSVERSION Not Specified

If you specify "<alphabet-name> is CCSVERSION" with no <literal-1> option under the SPECIAL-NAMES section, then the systemwide value that is defined for CCSVERSION is used when the program is compiled or run.

Example 5–2 lists a program that uses the systemwide value for CCSVERSION that is defined when the program is run.

```
000100 IDENTIFICATION DIVISION.
000200 ENVIRONMENT DIVISION.
000300 CONFIGURATION SECTION.
000400 OBJECT-COMPUTER.
000500     A6,
000600     PROGRAM COLLATING SEQUENCE IS CCS.
000700 SPECIAL-NAMES.
000800      CCS IS CCSVERSION.
000900 INPUT-OUTPUT SECTION.
001000 FILE-CONTROL.
001100    SELECT TEST ASSIGN TO DISK.
001200 DATA DIVISION.
001300 FILE SECTION.
001400    FD TEST.
001500    01 TEST-DATA PIC X(10).
001600 PROCEDURE DIVISION.
001700 BEGIN.
001800    IF ATTRIBUTE TITLE OF TEST = "TEST."
001900       DISPLAY "TEST".
001920    MOVE HIGH-VALUES TO TEST-DATA.
001940    IF TEST-DATA = HIGH VALUES
001960        DISPLAY "OK".
002000 STOP RUN.
```

Example 5–2. CCSVERSION Is Defined at RUN Time

If you specify "<alphabet-name> is CCSVERSION <literal-1>", then you get a CCSVERSION other than the default for the system at run time. Regardless of the default value of CCSVERSION when the program is compiled or run, the CCSVERSION identified by <literal-1> is obtained at execution time and used. A run-time error occurs if the literal is not valid at execution time.

Example 5–3 illustrates the CCSVERSION specified by <literal-1>. In this case, CZECHOSLOVAKIA is obtained and used at execution time.

```
000100 IDENTIFICATION DIVISION.
000200 ENVIRONMENT DIVISION.
000300 CONFIGURATION SECTION.
000400 OBJECT-COMPUTER.
000500     A6,
000600     PROGRAM COLLATING SEQUENCE IS CCS.
000700 SPECIAL-NAMES.
000800      CCS IS CCSVERSION "CZECHOSLOVAKIA"
000900 INPUT-OUTPUT SECTION.
001000 FILE-CONTROL.
001100    SELECT TEST ASSIGN TO DISK.
001200 DATA DIVISION.
001300 FILE SECTION.
001400    FD TEST.
001500    01 TEST-DATA PIC X(10).
001600 PROCEDURE DIVISION.
001700 BEGIN.
001800    IF ATTRIBUTE TITLE OF TEST = "TEST."
001900       DISPLAY "TEST".
001920    MOVE HIGH-VALUES TO TEST-DATA.
001940    IF TEST-DATA = HIGH VALUES
001960        DISPLAY "OK".
002000 STOP RUN.
```

Example 5–3. CCSVERSION Specified by <literal-1>

The CCSVERSION phrase can only be specified once. Only one CCSVERSION can be specified in a program.

Literal-2 specifies the positional value of the character in the program collating sequence. A given character can be specified only once as a literal in an alphabet-name clause. The value of each literal specifies both of the following characteristics:

- The ordinal number of a character within the native character set, if the literal is numeric. Numeric literals must be unsigned integers and must have values in the range 1 through 256.
- The actual character within the native character set, if the literal is nonnumeric. If the value of the nonnumeric literal contains multiple characters, each character in the literal, starting with the leftmost character, is assigned a successive ascending position in the collating sequence being specified.

The order in which the literals appear in the alphabet-name clause specifies, in ascending sequence, the ordinal number of the character within the collating sequence being specified.

Any characters in the native collating sequence that you do not specify in the literal phrase assume a position greater than any of the characters that you do specify in the collating sequence being specified. The relative order within the set of these unspecified characters is unchanged from the native collating sequence.

The character that has the highest ordinal position in the program-collating sequence specified is associated with the figurative constant HIGH-VALUE. If more than one character has the highest position in the program-collating sequence, the last character specified is associated with the figurative constant HIGH-VALUE.

The character that has the lowest ordinal position in the program-collating sequence specified is associated with the figurative constant LOW-VALUE. If more than one character has the lowest position in the program-collating sequence, the first character specified is associated with the figurative constant LOW-VALUE.

Note: *Using the internationalized system default ccsversion can produce unexpected results for the HIGH-VALUE and LOW-VALUE figurative constants. These unexpected results can occur when the program is run on a host with a system default ccsversion that differs from the ccsversion compiled into the program. In this case, the HIGH-VALUE and LOW-VALUE figurative constants will contain the values that are correct for the ccsversion compiled into the program.*

For example, if the program was compiled on a host with a system default ccsversion of SPANISH, but the program is run on a host with a default ccsversion of FRANCE, the HIGH-VALUE and LOW-VALUE constants define their values from the SPANISH ccsversion at compile-time, rather than from the FRANCE ccsversion.

The *THROUGH literal-3* phrase assigns successive ascending positions to the set of contiguous characters in the native character set, beginning with the character specified by the value of literal-1 and ending with the character specified by the value of literal-3. The set of contiguous characters can specify characters of the native character set in either ascending or descending sequence. The words THROUGH and THRU are equivalent. Each literal must be one character in length.

The *ALSO literal-4* phrase assigns literal-4 to the same position in the collating sequence as literal-1.

CURRENCY SIGN IS literal-5

The *CURRENCY SIGN IS literal-5* clause assigns the symbol used to represent the currency symbol in the PICTURE clause. If your program does not specify a currency symbol, the program uses the dollar sign ($) as the currency symbol in the PICTURE clause. The literal must be a single character. The currency symbol cannot be any of the following characters:

- Digits 0 through 9
- Alphabetic characters A, B, C, D, L, P, R, S, V, X, Z, and the space
- The following special characters:

* (asterisk)	+ (plus sign)
– (minus sign)	. (period)
; (semicolon)	((left parenthesis)
) (right parenthesis)	" (quotation mark)
, (comma)	/ (stroke)
= (equal sign)	

DECIMAL-POINT IS COMMA

The *DECIMAL-POINT IS COMMA* clause causes the comma to act as the decimal point and the period to represent the separator for thousands in the character string of the PICTURE clause and in numeric literals. For example, 1,000.00 changes to 1.000,00 with this option specified.

literal-6 IS mnemonic-name-4 (Extension to ANSI X3.23-1974 COBOL)

The *literal-6 IS mnemonic-name-4* clause associates a mnemonic-name with a valid program name. Literal-6 can be of the form AAA/BBB/CCC..., where each group of characters between two slashes is one directory of the program-name. A directory can have a maximum of 17 characters, and a file title can have a maximum of 14 directories. This clause is used for binding or tasking.

DEFAULT DISPLAY SIGN (Extension to ANSI X3.23-1974 COBOL)

The *DEFAULT DISPLAY SIGN* clause specifies a default sign position for all signed DISPLAY data items. If you declare a signed data item in the DATA DIVISION and do not use the optional SIGN clause, the program uses the default sign for that type of data item. The use of the optional SIGN clause in the DATA DIVISION overrides the default sign specification in the SPECIAL-NAMES paragraph.

DEFAULT COMPUTATIONAL SIGN (Extension to ANSI X3.23-1974 COBOL)

The *DEFAULT COMPUTATIONAL SIGN* clause specifies a default sign position for all signed COMPUTATIONAL data items. If you declare a signed data item in the DATA DIVISION and do not use the optional SIGN clause, the program uses the default sign for that type of data item. The use of the optional SIGN clause in the DATA DIVISION overrides the default sign specification in the SPECIAL-NAMES paragraph.

RESERVE NETWORK (Extension to ANSI X3.23-1974 COBOL)

The *RESERVE NETWORK* clause tells the compiler to handle the network class of application-specific keywords as reserved words for the extent of the program. If the program does not include the RESERVE NETWORK option, then the program can use the network keywords as normal identifiers. If you need to use port files with OSINATIVE service, then you should include the RESERVE NETWORK option in your program in order for the compiler to recognize keywords like the RESPOND verb as reserved words.

For More Information

- For information about identifying a collating sequence, refer to "OBJECT-COMPUTER" in this section.
- For more information about specifying a sign position, refer to "SIGN Clause" and "PICTURE Clause" in Section 7, "DATA DIVISION," and explanations of the LEADING and TRAILING options.
- For more information on binding, refer to the *MCP/AS Binder Programming Reference Manual*.
- For definition of port files, refer to "Port Files" in Section 3, "File and Task Concepts."
- For information on localizing a COBOL74 application, refer to Section 16, "Internationalization."
- For information on the RESERVE SEMANTIC clause and the DICTIONARY clause of the SPECIAL-NAMES paragraph, refer to Volume 2 of this manual.

INPUT-OUTPUT SECTION

The INPUT-OUTPUT SECTION contains the information needed to control transmission and handling of data between external media and the object program. If included, this section must begin with the reserved words INPUT-OUTPUT SECTION, followed by a period and a space. The INPUT-OUTPUT SECTION is divided into two paragraphs:

- The FILE-CONTROL paragraph, which names and associates the file with external media
- The I-O-CONTROL paragraph, which defines special control techniques to be used in the object program

FILE-CONTROL Paragraph

The FILE-CONTROL paragraph enables you to do the following.

- Name each file.
- Identify the file medium.
- Specify hardware.
- Specify alternate I/O areas.
- Specify the organization of the file.

The FILE-CONTROL paragraph is required in the INPUT-OUTPUT SECTION. You must include the reserved words FILE-CONTROL, followed by a period, a space, and the file-control entries.

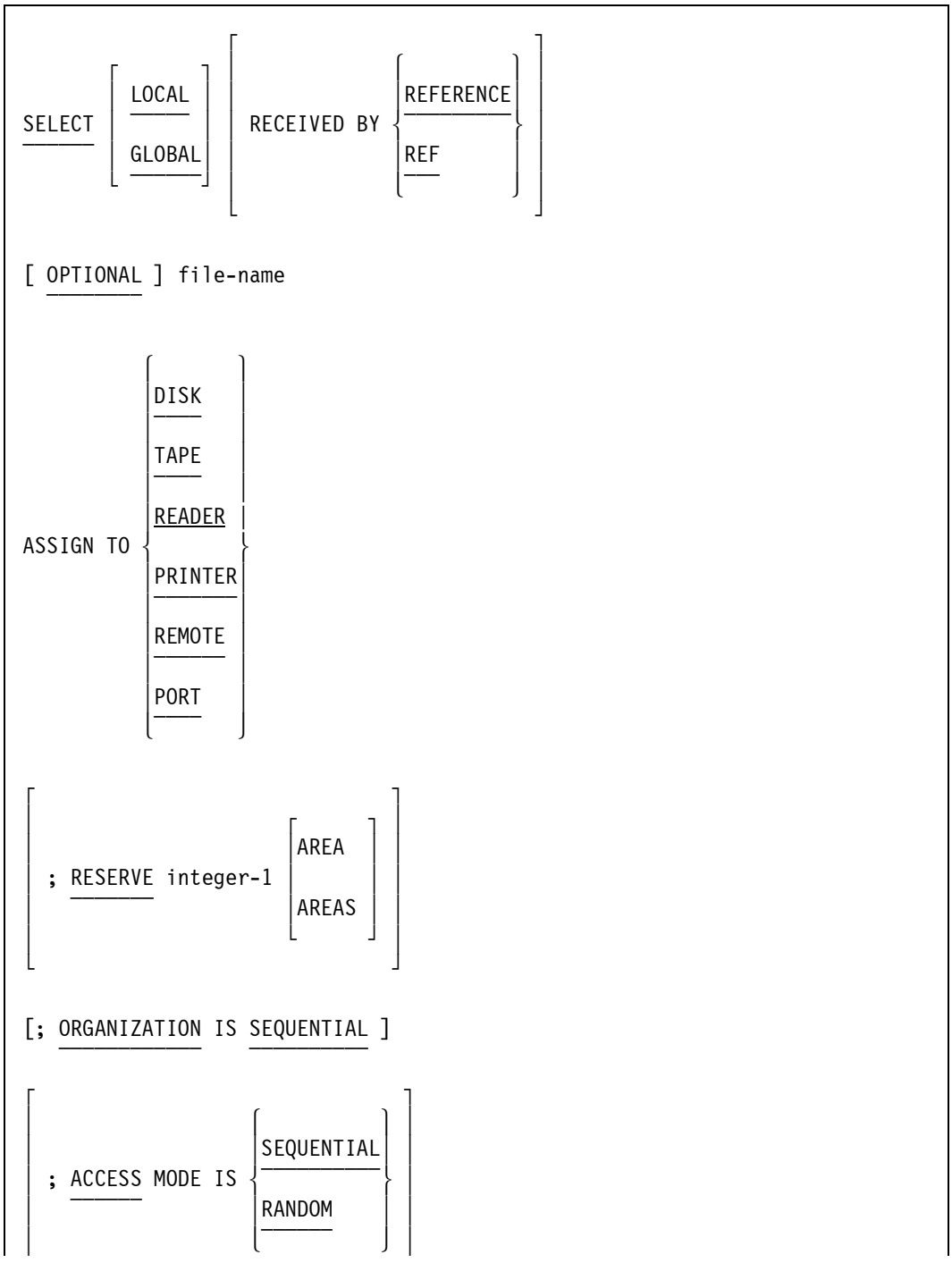
The formats for the file-control entries are used in the following ways:

Format	This format is used for . . .
1	Sequential files.
2	Relative files.
3	Indexed files.
4	Sort and merge files.
5	Data base files. See Volume 2 for information on the DICTIONARY-REFERENCE clause.

Sequential I/O

Your program must use Format 1 of the FILE-CONTROL paragraph if it is doing sequential I/O.

Format 1: Sequential I/O



```
└                                ┘

[ ; ACTUAL KEY IS data-name-1 ]

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

Explanation of Format 1

SELECT

The SELECT clause declares each file described in the DATA DIVISION. Each file described in the DATA DIVISION must be named once as a file-name in the SELECT clause. Each file specified in the SELECT clause must have a file-description (FD) entry in the DATA DIVISION. The SELECT clause must be the first clause in the FILE-CONTROL paragraph. The clauses that follow the SELECT clause can appear in any order.

LOCAL (Extension to ANSI X3.23-1974 COBOL)

The LOCAL option is meaningful only for programs being compiled as procedures. The LOCAL option specifies that the file is a formal parameter for a procedure and can be named only in WITH and USING clauses in the declarative USE statement associated with the procedure.

The LEVEL compiler option must be greater than 2 to use the LOCAL option.

GLOBAL (Extension to ANSI X3.23-1974 COBOL)

The GLOBAL option is meaningful only for programs being compiled as procedures. The GLOBAL option specifies that the first record description must match, by name and array type, a similar record description for the file in the host. For example,

```
SELECT GLOBAL GFILE ASSIGN TO DISK.
```

The GLOBAL compiler option has no effect on ENVIRONMENT DIVISION or FILE SECTION entries. The LEVEL compiler option must be greater than 2 to use the GLOBAL option.

RECEIVED BY REFERENCE or RECEIVED BY REF (Extension to ANSI X3.23-1974 COBOL)

The RECEIVED BY REFERENCE option enables two or more programs to use the file with which this option appears. Either program can perform I/O to the file. The default is RECEIVED BY REFERENCE.

This option is meaningful only if the file-name appears in the USING clause of the PROCEDURE DIVISION header. If the program does not have a RECEIVED BY REFERENCE clause, the compiler issues a warning when it encounters the file-name in the USING clause.

The compiler issues a syntax error if the LOCAL and the RECEIVED BY REFERENCE clauses appear in the same file.

RECEIVED BY REF is a synonym for RECEIVED BY REFERENCE.

OPTIONAL

The OPTIONAL phrase specifies an input file that is optional. You can use ODT commands to indicate to the system that an optional file does not exist. In that case, the first read operation on the file returns an end-of-file condition.

If the file is present on its assigned media, then no action is required; the file is opened. If the file is not present, the program is suspended. You can resume the suspended program by either supplying the file, or by using ODT commands as described in the preceding paragraph.

ASSIGN

The ASSIGN clause associates the named file with a storage medium. (ASSIGN TO REMOTE and ASSIGN TO PORT are extensions to ANSI X3.23-1974 COBOL.) DISK specifies that mass storage is the storage medium of the file. You can more precisely specify the storage medium by using the file attribute mechanism (that is, the VALUE OF clause in the file-description entry) or by using a file equation.

RESERVE

The RESERVE clause enables you to specify the number of I/O areas to be allocated. If the RESERVE clause is specified, the number of I/O areas allocated is equal to the value of integer-1. Two areas are automatically supplied when the RESERVE clause is omitted.

ORGANIZATION

The ORGANIZATION clause specifies the logical structure of a file. The file organization is established when a file is created and cannot subsequently be changed. The default organization for a file is SEQUENTIAL.

ACCESS MODE

The ACCESS MODE clause specifies whether records in a sequentially organized file are to be accessed sequentially or randomly. You can specify random access for mass-storage files only. (RANDOM access is an extension to ANSI X3.23-1974 COBOL.)

The default mode of access is sequential.

ACTUAL KEY (Extension to ANSI X3.23-1974 COBOL)

The ACTUAL KEY clause can be specified only for mass-storage, port, and remote files. Data-name-1 must be defined in the DATA DIVISION as an elementary numeric item that describes an unsigned integer.

If the ACTUAL KEY clause is specified, the following rules apply:

- For mass-storage files specifying an ACTUAL KEY, the value of the ACTUAL KEY data item specifies the logical ordinal position of the record in the file.
- For port files, the value of the ACTUAL KEY data item specifies the subfile index of the port file.
- For remote files, the value of the ACTUAL KEY data item specifies the ordinal number of the station within the station list of the remote file. A zero value specifies all stations within the station list of the remote file.

The ACTUAL KEY clause must be specified for a port file that contains more than one subfile.

FILE STATUS

When the FILE STATUS clause is specified, the system moves a value into the data item specified by data-name-2 after execution of every statement that explicitly or implicitly references that file. This value indicates the status of execution of the statement. Data-name-2 must be defined in the DATA DIVISION as a two-character, alphanumeric data item and must not be defined in the FILE SECTION, the REPORT SECTION, or the COMMUNICATION SECTION. Data-name-2 can be qualified.

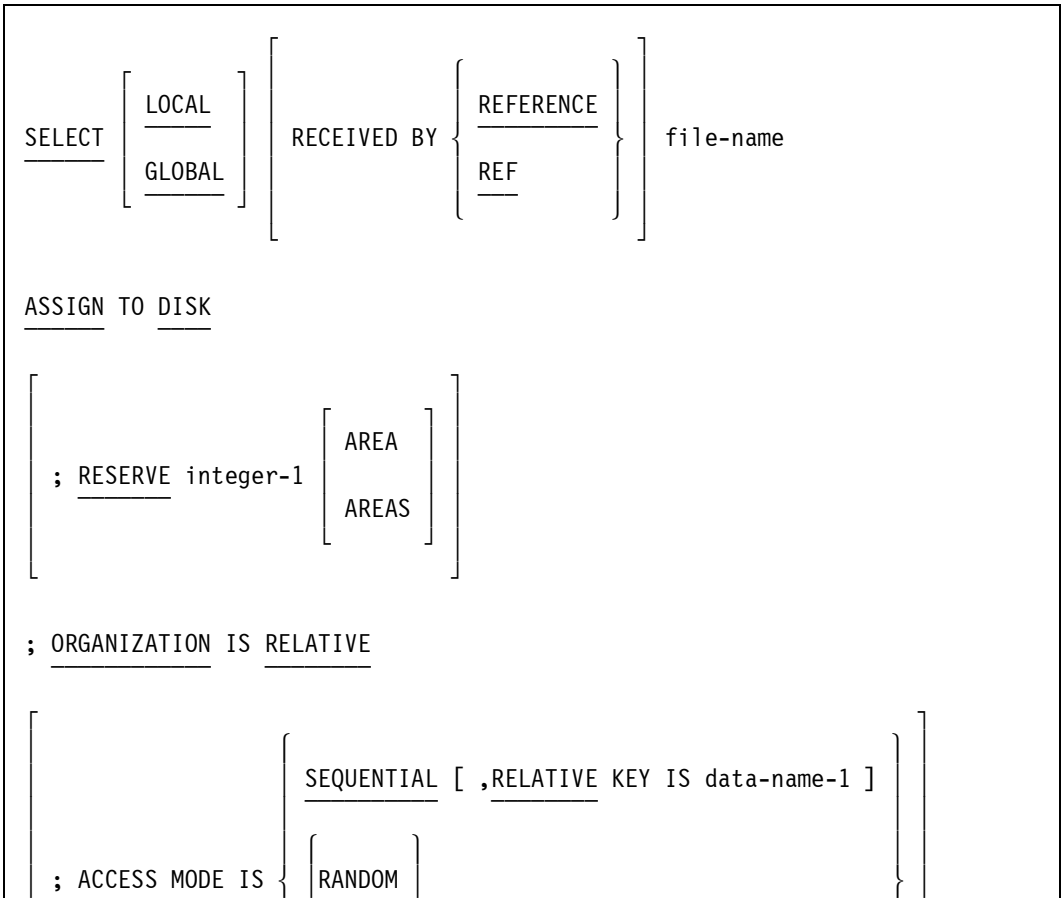
For More Information

- For information about status values, refer to "I/O Status" later in this section.
- For more information about sequential I/O, refer to "File Organization and Access Methods" in Section 3, "File and Task Concepts."

Relative I/O

Your program must use Format 2 when it is performing relative I/O.

Format 2: Relative I/O



```
[ [ SEQUENTIAL ]                [ SEQUENTIAL ]                              ]
[                               [            ] , RELATIVE KEY IS data-name-1 ]
[                               [ DYNAMIC    ]                               ]
[                               [            ]                               ]

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

Explanation of Format 2

SELECT

The SELECT clause declares each file described in the DATA DIVISION. Each file described in the DATA DIVISION must be named once as a file-name in the SELECT clause. Each file specified in the SELECT clause must have a file-description (FD) entry in the DATA DIVISION. The SELECT clause must be the first clause in the FILE-CONTROL paragraph. The clauses that follow the SELECT clause can appear in any order, except that the RELATIVE KEY clause must follow the ACCESS MODE clause.

LOCAL (Extension to ANSI X3.23-1974 COBOL)

This clause is useful only for programs being compiled as procedures. The LOCAL clause specifies that the file is a formal parameter for a procedure and can be named only in WITH and USING clauses in the declarative USE statement associated with the procedure.

The LEVEL compiler option must be greater than 2 to use the LOCAL clause.

GLOBAL (Extension to ANSI X3.23-1974 COBOL)

The GLOBAL clause is meaningful only for programs being compiled as procedures. The GLOBAL clause specifies that the first record description must match, by name and array type, a similar record description for the file in the host. For example,

```
SELECT GLOBAL GFILE ASSIGN TO DISK.
```

The GLOBAL compiler option has no effect on ENVIRONMENT DIVISION or FILE SECTION entries. The LEVEL compiler option must be greater than 2 to use the GLOBAL option.

RECEIVED BY REFERENCE or RECEIVED BY REF (Extension to ANSI X3.23-1974 COBOL)

The RECEIVED BY REFERENCE phrase enables two or more programs to use the file with which this option appears. Either program can perform I/O to the file. The default is RECEIVED BY REFERENCE.

This option is meaningful only if the file-name appears in the USING clause of the PROCEDURE DIVISION header. If the program does not have a RECEIVED BY REFERENCE clause, the compiler issues a warning when it encounters the file-name in the USING clause.

The compiler issues a syntax error if the LOCAL and the RECEIVED BY REFERENCE clauses appear in the same file.

RECEIVED BY REF is a synonym for RECEIVED BY REFERENCE.

ASSIGN

The ASSIGN clause associates the named file with a storage medium. DISK specifies that mass storage is the storage medium of the file. You can more precisely specify the storage medium by using the file attribute mechanism (the VALUE OF clause in the file-description entry) or through file equation.

RESERVE

The RESERVE clause enables specification of the number of I/O areas allocated. If the RESERVE clause is specified, the number of I/O areas allocated is equal to the value of integer-1. Two areas are allocated when the RESERVE clause is omitted.

ORGANIZATION

The ORGANIZATION clause specifies the logical structure of a file. The file organization is established when a file is created and cannot subsequently be changed. The default file organization is sequential.

ACCESS MODE

When the ACCESS MODE is SEQUENTIAL, records in the file are accessed in the sequence dictated by the file organization. This sequence is the order of ascending relative record numbers of existing records in the file. If the access mode is sequential, the system maintains the value of the relative key on all I/O operations. The default mode of access is sequential.

All records stored in a relative file are uniquely identified by relative record numbers. The relative record number of a given record specifies the logical ordinal position of the record in the file. The first logical record has a relative number of 1, and subsequent logical records have relative record numbers of 2, 3, 4, and so forth.

The RELATIVE KEY phrase is required when the access mode is dynamic or random. When the access mode is dynamic, records in the file can be accessed sequentially, randomly, or both, depending on the verbs used in the PROCEDURE DIVISION. Dynamic access is random by relative key except in the case of the READ NEXT statement, in which case the system updates the value of the relative key.

If the access mode is random, the value of the RELATIVE KEY data item indicates the record to be accessed.

If a relative file is referenced by a START statement, the RELATIVE KEY phrase must be specified for that file. Data-name-1 must not be defined in a record-description (RD) entry associated with that file-name. Data-name-1 can be qualified.

FILE STATUS

When the FILE STATUS clause is specified, the system moves a value into the data item specified by data-name-2 after execution of every statement that explicitly or implicitly references that file. This value indicates the status of execution of the statement. Data-name-2 must be defined in the DATA DIVISION as a two-character, alphanumeric data item and must not be defined in the FILE SECTION, the REPORT SECTION, or the COMMUNICATION SECTION. Data-name-2 can be qualified.

For information about status values, refer to "I/O Status" later in this section.

Indexed I/O

Your program must use Format 3 if it is performing indexed I/O.

Format 3: Indexed I/O

```
SELECT [ LOCAL  ] [ RECEIVED BY { REFERENCE } ] file-name
       [ GLOBAL ] [             { REF       } ]

ASSIGN TO DISK

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

; ORGANIZATION IS INDEXED

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

; RECORD KEY IS data-name-1

[                  {            } ]
```

```
                    BINARY
  ; COMPARISON IS { LOGICAL    }
                    EQUIVALENT

[ ; KEY-LENGTH IS integer-2 ]

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

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

Explanation of Format 3

SELECT

The SELECT clause declares each file described in the DATA DIVISION. Each file described in the DATA DIVISION must be named once as a file-name in the SELECT clause. Each file specified in the SELECT clause must have a file-description (FD) entry in the DATA DIVISION. The SELECT clause must be the first clause in the FILE-CONTROL paragraph. The clauses that follow the SELECT clause can appear in any order.

LOCAL (Extension to ANSI X3.23-1974 COBOL)

This clause is useful only for programs being compiled as procedures. The LOCAL clause indicates that the file is a formal parameter for a procedure and can be named only in WITH and USING clauses in the declarative USE statement associated with the procedure.

The LEVEL compiler option must be greater than 2 to use the LOCAL option.

GLOBAL (Extension to ANSI X3.23-1974 COBOL)

The GLOBAL clause is meaningful only for program being compiled as procedures. The GLOBAL clause specifies that the first record description must match, by name and array type, a similar record description for the file in the host. For example,

```
SELECT GLOBAL GFILE ASSIGN TO DISK.
```

The GLOBAL compiler option has no effect on ENVIRONMENT DIVISION or FILE SECTION entries. The LEVEL compiler option must be greater than 2 to use the GLOBAL option.

RECEIVED BY REFERENCE or RECEIVED BY REF

These phrases, extensions to ANSI X3.23-1974 COBOL, enable two or more programs to use the file with which the option appears. Either program can perform I/O to the file. The default is RECEIVED BY REFERENCE.

This option is meaningful only if the file-name appears in the USING clause of the PROCEDURE DIVISION header. If the program does not have a RECEIVED BY REFERENCE clause, the compiler issues a warning when it encounters the file-name in the USING clause.

The compiler issues a syntax error if the LOCAL and the RECEIVED BY REFERENCE clauses appear in the same file.

RECEIVED BY REF is a synonym for RECEIVED BY REFERENCE.

ASSIGN

The ASSIGN clause associates the named file with a storage medium. DISK specifies that mass storage is the storage medium of the file. You can more precisely specify the storage medium by using the file attribute mechanism (the VALUE OF clause in the file-description entry) or through file equation.

RESERVE

The RESERVE clause enables you to specify the number of I/O areas allocated. If the RESERVE clause is specified, the number of I/O areas allocated equals the value of integer-1. Two areas are allocated when the RESERVE clause is omitted.

ORGANIZATION

The ORGANIZATION clause specifies the logical structure of a file. The file organization is established when a file is created and cannot be changed later.

The default file organization is sequential.

ACCESS MODE

When the access mode is sequential, records in the file are accessed in the sequence dictated by the file organization. For indexed files, this sequence is the ascending order of record values within a given key of reference.

If the access mode is random, the value of the record key data item indicates the record to be accessed.

When the access mode is dynamic, records in the file can be accessed sequentially, randomly, or both.

The default mode of access is sequential.

RECORD KEY

The RECORD KEY clause specifies the prime record key for the file. The values of the prime record key must be unique among the file records. This prime record key provides an access path to records in an indexed file. This clause is required for indexed files.

COMPARISON (Extension to ANSI X3.23-1974 COBOL)

The COMPARISON clause specifies the type of comparison to be performed when searching for the key. This clause is used only for internationalization purposes. A binary comparison uses the binary value of the key. If the program does not specify the type of comparison, it performs a binary comparison.

A logical comparison uses the collating sequence value of the key. The collating sequence is the arrangement of members of a character set according to the ordering sequence values (OSVs) and the priority sequence values (PSVs). Elements occupy different code positions, with elements having the same OSV differentiated by the PSV assigned to the code position.

An equivalent comparison uses the ordering value of the key. The ordering sequence is the arrangement of members of a character set according to a predetermined scheme. Different elements can have the same ordering attribute. For example, you might want the equivalent uppercase and lowercase characters (a and A, b and B, and so on) to have the same ordering values.

KEY-LENGTH (Extension to ANSI X3.23-1974 COBOL)

The KEY-LENGTH option specifies the number of 8-bit characters the system uses when it stores a translated key value. The clause is used for internationalization only. Translated display data can require a larger storage area when including ordering and collating information. The KEY-LENGTH clause can be used with the prime record key or the alternate record key when the internationalization features are used.

If a key length is provided, the number of 8-bit characters used to store a translated key equals the value of integer-2. Otherwise, the length of the key item is used. Truncation of the stored key occurs if the key length value is too small. If the program specifies a system default ccsversion by coding the *alphabet-name IS CCSVERSION* clause without the literal-1 option in the SPECIAL-NAMES paragraph, that same ccsversion must be the system default ccsversion at run time. A run-time error occurs when opening the indexed file for output if the run-time ccsversion does not match the compiled ccsversion.

ALTERNATE RECORD KEY

An ALTERNATE RECORD KEY clause specifies an alternate record key for the file. This alternate record key provides an alternate access path to records in an indexed file.

The data items referenced by data-name-1 and data-name-2 must each be defined as data items of the category alphanumeric or numeric within a record-description entry associated with that file-name. Neither data-name-1 nor data-name-2 can describe a variable-size item. Data-name-1 and data-name-2 can be qualified.

Data-name-2 cannot reference an item with the leftmost character position corresponding to the leftmost character position of an item referenced by data-name-1 or by any other data-name-2 associated with this file.

The data descriptions of data-name-1 and data-name-2 and their relative locations within a record must be the same as those used when the file was created. The number of alternate keys for the file must also be the same as the number used when the file was created.

The DUPLICATES phrase specifies that the value of the associated alternate record key can be duplicated within any of the records in the file. If the DUPLICATES phrase is not specified, the value of the associated alternate record key must not be duplicated among any of the records in the file.

FILE STATUS

When the FILE STATUS clause is specified, a value is moved by the operating system into the data item specified by data-name-3 after execution of every statement that explicitly or implicitly references that file. This value indicates the status of execution of the statement.

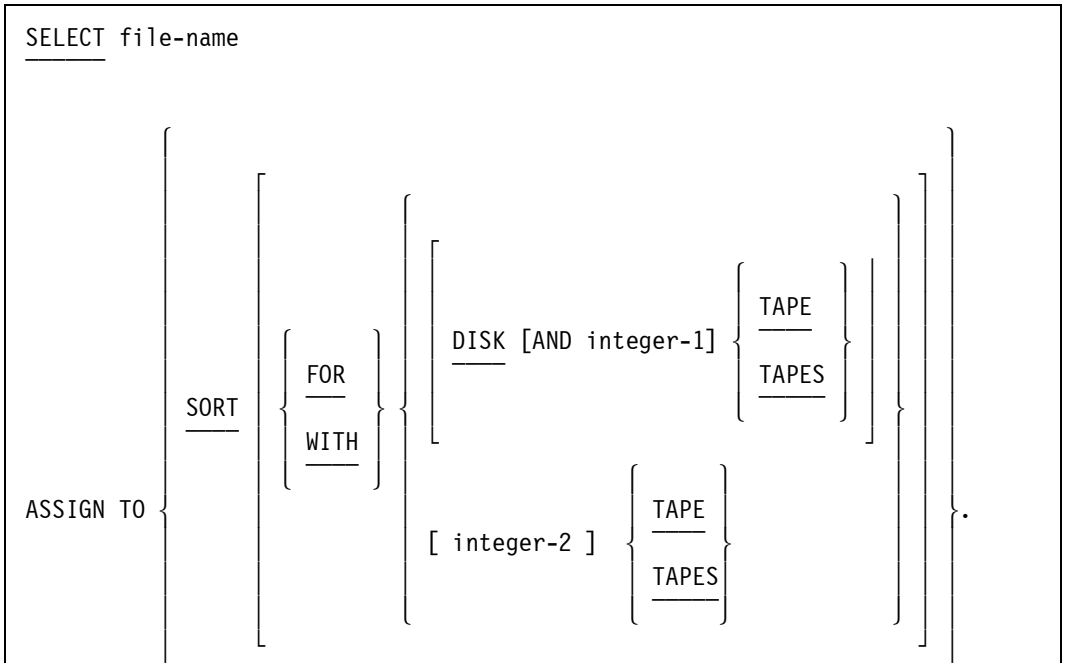
Data-name-3 must be defined in the DATA DIVISION as a two-character, alphanumeric data item and must not be defined in the FILE SECTION, the REPORT SECTION, or the COMMUNICATION SECTION. Data-name-3 can be qualified.

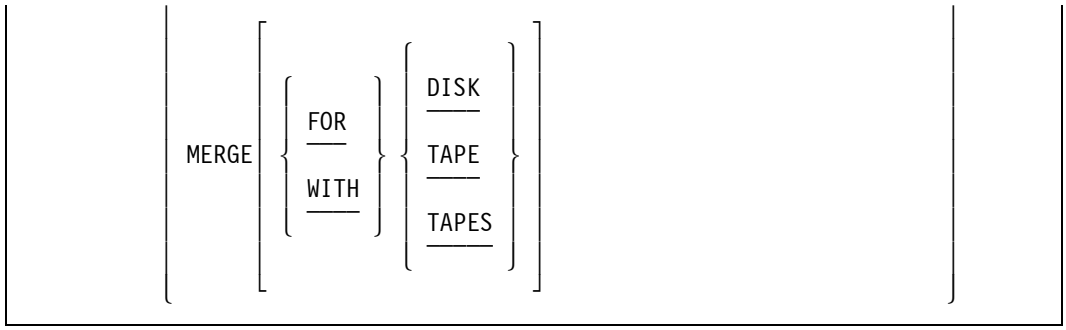
For information about status values, refer to "I/O Status" later in this section.

Sort-Merge

Your program must use Format 4 for files that are to be sorted or merged.

Format 4: Sort-Merge





Explanation of Format 4

Each sort or merge file described in the DATA DIVISION must be named once, and only once, as a file-name in the FILE-CONTROL paragraph. Each sort or merge file specified in the file-control entry must have a sort-merge file-description entry in the DATA DIVISION.

Because the file-name represents a sort or merge file, only the ASSIGN clause is permitted to follow the file-name in the SELECT statement. The ASSIGN clause associates the named sort or merge file to a storage medium.

The options following the SORT and MERGE clauses in Format 4 are extensions to ANSI X3.23-1974 COBOL.

If TAPE or TAPES is specified, the primary work medium is still DISK. TAPE or TAPES can be specified to contain any overflow. If integer-2 is not specified, the default number of tapes is three. Integer-1 and integer-2 must have values in the range 3 through 8.

When DISK is specified, mass storage is the primary work medium. TAPE or TAPES can be specified to contain any overflow. If integer-1 is not specified, three tapes are assumed.

I-O-CONTROL

The I-O-CONTROL paragraph specifies the following:

- The points at which rerun is to be established
- The memory area to be shared by different files
- The location of files on a multiple-file reel

The general format of this paragraph is as follows:

```
I-O-CONTROL.

[ ; RERUN ON DISK EVERY integer-1 RECORDS OF file-name-2 ]...
```

```
[ ; SAME [RECORD] AREA FOR file-name-3 {, file-name-4}... ]...

[                                                                  ]
[ ; MULTIPLE FILE TAPE CONTAINS file-name-5 [POSITION integer-3]   ]
[                                                                  ]...
[    [, file-name-6 [POSITION integer-4]]...                       ]
[                                                                  ]
```

Explanation of Format

The I-O-CONTROL paragraph is optional.

RERUN clause

The RERUN clause causes rerun information to be recorded whenever integer-1 RECORDS of file-name-2 have been processed.

File-name-2 can be an input file or an output file, with any organization or access. File-name-2 cannot be specified in more than one RERUN clause.

The compiler causes the object program to generate RERUN information only if the RERUN clause is exactly as shown in the syntax diagram. The compiler might fail to issue syntax errors for any RERUN clause that does not conform to these rules. In order for the RERUN clause to be effective, you must ensure that all RERUN clauses are in conformance with the syntax diagram. The results of specifying a RERUN clause that is not in conformance are unpredictable.

SAME

The SAME AREA and SAME RECORD AREA clauses are used as follows:

- A program can include more than one SAME clause if following rules are obeyed:
  - A file-name must not appear in more than one SAME clause.
  - If one or more file-names of a SAME AREA clause appear in a SAME RECORD AREA clause, all of the file-names in that SAME AREA clause must appear in the SAME RECORD AREA clause. However, additional file-names that do not appear in the SAME AREA clause can also appear in the SAME RECORD AREA clause. The rule that only one of the files mentioned in a SAME AREA clause can be open at any time takes precedence over the rule that all files mentioned in a SAME RECORD AREA clause can be open at any time.
- The files referenced in the SAME AREA or SAME RECORD AREA clause need not all have the same organization or access.

The SAME AREA clause specifies that two or more files that do not represent SORT or MERGE files are to use the same memory area during processing. The area being shared includes all storage area assigned to the files specified; thus, only one file can be open at a time.

The SAME RECORD AREA clause specifies that two or more files are to use the same memory area for processing of the current logical record. All the files can be open at the same time. A logical record in the SAME RECORD AREA is considered a logical record of each opened output file and the most recently read input file, which all have file-names appearing in this SAME RECORD AREA clause. This is equivalent to an implicit redefinition of the area; that is, records are aligned at the leftmost character position.

MULTIPLE FILE

The MULTIPLE FILE clause, which can be used only with sequential I/O, specifies the locations of files on a multiple-file reel. This clause is required when more than one file shares the same physical reel of tape. Regardless of the number of files on a single reel, only those files used in the object program need to be specified.

The POSITION phrase is ignored by the compiler. You must specify the files in consecutive order. No more than one file on the same tape reel can be open at one time.

One MULTIPLE FILE clause is used for each multiple-file tape. The titles of all the files listed in a given clause must have a common volume-identifier: otherwise, the files appear on different tapes. The volume-identifier is a nonnumeric literal, 1 to 17 characters in length, that cannot contain any special characters or spaces.

After each file is read or written to, the file is closed without being rewound (CLOSE file-name WITH NO REWIND) and the next file is opened without being rewound. If the volume-id is correct, the files are written to or read from the same reel.

For more information about multiple-file tapes and device dependencies, refer to the *I/O Subsystem Programming Guide.*

I/O Status

The system can indicate to the COBOL74 program the status of I/O operations during their execution if the program specifies a FILE STATUS clause in the file-control entry. The FILE STATUS clause designates a two-character data item into which the system places a value that indicates the status of the I/O operation. For example, a value of 00 means that the operation completed successfully.

Depending on the organization of the file, the execution of an OPEN, CLOSE, READ, SEEK, WRITE, REWRITE, DELETE, or START statement causes the system to update the status of the I/O operation. The status value is updated before the program executes any applicable USE procedure.

Successful Completion

00 Successful I/O with No Errors or Warnings

The I/O statement completed successfully with no errors or advisory indications.

09 Successful I/O with URGENT Indicator

The I/O statement for a file assigned to PORT completed successfully. The file contained data marked URGENT (extension to ANSI X3.23-1974 COBOL).

At End

10 At End-of-file

The execution of a Format 1 READ (sequential) statement was unsuccessful because of an attempt to read a record when there was no next logical record in the file.

Invalid Key

21 Sequence Error on Input

This value can be returned in the following circumstances:

- For a sequential file with random access that is not assigned to a port file, this file status value means the value of the ACTUAL KEY data item associated with the file was less than 1 or greater than the ordinal number of the last record previously written to the file when this Format 2 READ statement was encountered (extension to ANSI X3.23-1974 COBOL).
- For an indexed file, the file status value means that either the ascending sequence requirements of successive record key values were violated on this WRITE statement, or the prime record key value was changed between the successful execution of a READ statement and this REWRITE statement.

22 Duplicate Key on Input

This value can be returned in the following circumstances:

- For a relative or indexed file, an attempt was made to write a record into a position in the file that was already occupied by a nondeleted record.
- The Format 2 WRITE statement for a sequential file with random access that is not a port file cannot be completed. The value of the ACTUAL KEY data item associated with the file is less than 1 or greater than the ordinal number of the last record that could be written to the file (given the specifications for the file) (extension to ANSI X3.23-1974 COBOL).

23 No Record Found

This value can be returned in the following circumstances:

- For a relative or indexed file, an attempt was made to access a record identified by a key, but no record with a matching key exists in the file.
- A READ statement on a file declared ORGANIZATION SEQUENTIAL, ACCESS RANDOM was executed when the ACTUAL KEY value was less than 1 or greater than the ordinal position of the last record previously written to the file.
- A WRITE statement on a file declared ORGANIZATION SEQUENTIAL, ACCESS RANDOM was executed when the ACTUAL KEY value was less than 1 or greater than the highest value that could physically be written to the file.

24 Invalid Subport

For a file assigned to PORT, the ACTUAL KEY data item contained a value less than 0 (zero) or greater than the number of subfiles declared for the file (extension to ANSI X3.23-1974 COBOL).

25 Insufficient Medium Space

This value is returned if the NORESOURCEWAIT attribute for the file has been set to TRUE, and the MCP is unable to allocate space on the medium for this I/O statement. Since the default setting for the NORESOURCEWAIT attribute is FALSE, this error condition does not occur under ordinary circumstances (extension to ANSI X3.23-1974 COBOL).

Permanent Error

30 Permanent Error

This value can be returned in the following circumstances:

- The execution of the I/O statement was unsuccessful because of an I/O error, such as a data check, a parity error, or a transmission error.
- For ORGANIZATION SEQUENTIAL files only, an attempt was made to write beyond the externally defined physical boundaries of the file.
- An attempt was made to perform a Format 1 WRITE (sequential) statement against a file that prohibits sequential access (extension to ANSI X3.23-1974 COBOL).
- An attempt was made to add a record through a WRITE statement or to delete a record, but the file prohibits the addition or deletion of records (extension to ANSI X3.23-1974 COBOL).
- A WRITE statement to a port file had an option inappropriate to the value of the SERVICE attribute for the file (extension to ANSI X3.23-1974 COBOL).
- A permanent error occurred during the accessing of a file located on a remote system (extension to ANSI X3.23-1974 COBOL).

34 Boundary Violation

The I/O operation was unsuccessful as the result of an attempt to exceed an inflexible limit on the size of a file.

Unisys Defined (Extension to ANSI X3.23-1974 COBOL)

81 File Not Open

Execution of an OPEN AVAILABLE or OPEN OFFER statement for a file assigned to PORT was not successful. The program continues executing even though the OPEN statement was not successful.

In general, this value is returned when the operating system returns control to the program, rather than terminating the program, after an unsuccessful open operation.

82 Form Not Found

A READ FORM or WRITE FORM statement to a file that is designated ASSIGN TO REMOTE requested a form that does not reside in the formlibrary or requested a form for which the compile-time version does not equal the run-time version.

Unisys Defined (Extension to ANSI X3.23-1974 COBOL)

83 Indexed File Recovery Required

An I/O statement on an indexed file encountered an I/O error in the software supporting the indexed file, and the software is attempting to recover. For the recovery to be complete, the file must be closed.

91 Short Block

A physical block shorter than the physical blocksize declared for the file was received from the hardware device. Note that the operation is considered to have been completed successfully; an I/O status value of "91" is to be considered a warning rather than an error.

92 Data Error

This value can be returned in the following circumstances:

- When logical records are declared variable in length, and the logical record length is supplied by the programmer (by the RECORD CONTAINS...TO... clause), a data error occurs if the length supplied in the I/O statement is less than the minimum record size or greater than the maximum record size declared for the file.
- For files declared ORGANIZATION INDEXED only, the record size of the physical file does not match the record size supplied by the program for this I/O statement.

93 Broadcast Write Failed

For a file assigned to PORT with an ACTUAL KEY clause present in the SELECT statement, a broadcast WRITE statement was attempted (indicated by setting the ACTUAL KEY data item to the number 0), but the WRITE statement was unsuccessful for one or more subfiles.

94 No Data

A READ...WITH NO WAIT statement was executed against a file assigned to PORT, and there was no data to be read.

95 Insufficient Buffers

A WRITE...WITH NO WAIT statement was executed against a file assigned to PORT, and not enough buffers were available for the file to complete the requested data transfer. A "No buffers" condition in which no data was transferred can be distinguished from a condition in which some, but not all, of the data was transferred by examining the STATE attribute value when the error is first discovered. Further information on the STATE attribute can be found in the *File Attributes Reference Manual*.

96 Timeout

A time limit elapsed before the transfer of data to or from the hardware device could be completed.

97 Break on Output

The physical device to which the data is being written is equipped with a break mechanism that can halt the transfer of data to the device. That break mechanism was activated, halting the transfer of data before the transfer was complete.

98 A deadly embrace condition or a timeout occurred when two or more programs attempted to lock a record during updates to a file whose organization is INDEXED or RELATIVE.

99 Unexpected Error

An error other than those listed previously in this table might have occurred during the I/O operation. The nature of the error cannot be determined.

For information about file status values for the communication module, refer to Section 14, "COMMUNICATION SECTION."

ENVIRONMENT DIVISION Program Sample

Example 5–4 shows coding of the ENVIRONMENT DIVISION. The example includes both a CONFIGURATION SECTION and INPUT-OUTPUT SECTION in its ENVIRONMENT DIVISION. The SELECT clause in the FILE-CONTROL paragraph assigns two sequential files, INFIL and OUTFIL, to the storage medium TAPE. The SAME clause in the I-O-CONTROL paragraph (Format 1) specifies that INFIL and OUTFIL share the same memory area during processing. However, only one of these files can be open at a time.

```
006000 ENVIRONMENT DIVISION.
008000 CONFIGURATION SECTION.
010000 SOURCE-COMPUTER. A17.
012000 OBJECT-COMPUTER. A17.
014000 SPECIAL-NAMES.
016000     SW5 ON STATUS IS SW5-ON
018000         OFF STATUS IS SW5-OFF;
020000     CURRENCY SIGN IS "E";
022000     DECIMAL-POINT IS COMMA.
024000 INPUT-OUTPUT SECTION.
026000 FILE-CONTROL.
028000     SELECT INFIL ASSIGN TO TAPE.
030000     SELECT OUTFIL ASSIGN TO TAPE.
032000 I-O-CONTROL.
034000     SAME AREA FOR INFIL OUTFIL.
```

Example 5–4. Coding the ENVIRONMENT DIVISION

Section 6
Data Concepts

To understand the DATA DIVISION, you need to understand some of the concepts that pertain to the data in your program. The record concept encompasses structure, record level, and data items within the record. The data concept includes categorizing of data, aligning data, and referring to data. The table concept for handling sets of data includes subscripting and indexing. The edit concept for formatting data includes using symbols to format data, insertion editing, and zero-suppression and replacement editing.

Records

The most inclusive data item is the logical record. The record is identified by a 01-level entry. One or more related data items are defined in the record.

A data-description entry is an entry in the DATA DIVISION that describes a data item. Each data-description entry consists of a level-number followed by a data-name, if required, followed by a series of independent clauses, as required.

A record description is a set of data-description entries that describe the characteristics of the particular record. You can specify the following in a record description:

- The items that are included in the record
- The order in which the items appear in the record
- The way the items are related to each other in the record

The size of a record description is the sum of the maximum sizes of all items subordinate to a 01-level item. The maximum size of a record description is restricted by the explicit or implicit usage of the 01-level item as shown in Table 6–1.

Table 6–1. Usage and Maximum Size of a Record Description

Usage	Maximum Size
BINARY or DISPLAY	65,535 characters
COMP	65,535 digits
TASK, EVENT, INDEX, LOCK, or REAL	65,535 words

Levels

Logical records have subdivisions for data reference. The items in these subdivisions are organized with a system of levels. Once a subdivision has been specified, it can be further subdivided to permit more detailed data referral.

Understanding Elementary and Group Items

The smallest element of a data description is called an elementary item. Notice that elementary items cannot have subordinate levels. A record consists of a sequence of elementary items or of one elementary item.

To refer to elementary items as a set, you can combine the elementary items into groups. Each group consists of a named sequence of one or more elementary items. You can combine groups, in turn, into groups of two or more groups. Thus, an elementary item can belong to more than one group of one or more elementary items. You can combine groups, in turn, only of a level-number and a data-name, optionally followed by a VALUE, USAGE, or REDEFINES clause, followed by a period.

A group includes all group and elementary items following it until a level-number occurs that is less than or equal to the level-number of that group. You must describe all items immediately subordinate to a given group item by using identical level-numbers greater than the level-number used to describe that group item.

COBOL74 defines all group items to be alphanumeric.

Alignment on Byte Boundary

If a group item or an elementary EBCDIC item is not going to begin on a byte boundary, the compiler adds a 4-bit filler to align the item on a byte boundary. When such alignment takes place, the compiler includes a “FILLER ADDED” message in the output listing.

The following example shows that filler is added when an EBCDIC item, either elementary or group, does not begin on a byte boundary.

```
 01 X
  03 A PIC 9(3) COMP.
*                    <-----FILLER ADDED
  03 B PIC X.
  03 C PIC 9 (5) COMP.
*                    <-----FILLER ADDED
  03 GROUP.
```

Limits

No elementary data item can start more than 65,535 bytes (131,070 COMPUTATIONAL digits) from the beginning of the 01-level record in which the data item is contained.

No elementary data item whose USAGE IS COMPUTATIONAL can start more than 32,767 bytes (more precisely, 65,535 COMPUTATIONAL digits) from the beginning of the 01-level record in which the data item is contained.

Example

Example 6–1 shows the coding of elementary and group items. A group is composed of all group and elementary items described under it. A group item ends when a level-number less than or equal to the numeric value of the group item itself is encountered.

```
030000*  The name of the record is PRODUCTION-RECORD.
032000*
034000 01  PRODUCTION-RECORD.
036000*
038000*   ITEM-NO and LOT-NO are elementary items.
040000*
042000     03  ITEM-NO              PIC 9(5).
044000     03  LOT-NO               PIC 9(6).
046000*
048000*   The first group item is ITEM-DATE. The elementary items MONTH,
048900*   DAYS, and YEAR are subordinate to ITEM-DATE. STANDARD-COST
048950*   is an elementary item.
050000*
052000     03  ITEM-DATE.
060000         05  MONTH            PIC 99.
062000         05  DAYS             PIC 99.
064000         05  YEAR             PIC 99.
072000     03  STANDARD-COST        PIC 9(5)V99.
066000*
068000*   Both PRODUCTION-CODE and MACHINE-SHOP are group items.
068900*   The items MILLING and FINISHING are elementary items
068910*   subordinate to the MACHINE-SHOP group item.  The items
068920*   ASSEMBLY, INSPECTION, and WARRANTY-CODE are elementary items.
068930*   The group item PRODUCTION-CODE ends with the INSPECTION data item.
070000*
074000     03  PRODUCTION-CODE.
076000         05  MACHINE-SHOP.
084000             07  MILLING      PIC 999.
086000             07  FINISHING    PIC 99.
088000         05 ASSEMBLY          PIC 9(4).
090000         05  INSPECTION       PIC X(5).
092000     03  WARRANTY-CODE        PIC XX.
```

Example 6–1. Coding Elementary and Group Items

Organizing Data with Level-Numbers

A system of level-numbers shows the organization of elementary items and group items. Because records are the most inclusive data items, level-numbers of records start at 01. You should assign less inclusive data items higher (but not necessarily successive) level-numbers that are not greater in value than 49. The program writes separate entries for each level-number used. The special level-numbers 66, 77, and 88 represent entries for which no true concept of level exists.

Constructing a Record

Table 6–2 shows the type of entry that can be assigned to each type of level-number.

Table 6–2. Assigning Level-Numbers

Level-Number	Related Entry
01	The first entry in each record description. Multiple 01-level entries subordinate to a level indicator other than RD represent implicit redefinitions of the same area.
02 through 49	The hierarchy of data in a logical record.
66	RENAMES clause entries for regrouping data items.
77	WORKING-STORAGE or LINKAGE SECTION items that are not organized into a hierarchy.
88	Condition-names that specify values of conditional variables.

A level-number is required as the first element in each data-description entry. Table 6–3 shows the level-numbers that can be used with each type of data-description entry: file description (FD), sort merge description (SD), communications description (CD), report description (RD) and the data-description entries in WORKING STORAGE and LINKAGE SECTIONs.

Table 6–3. Level-Numbers Associated with Data-Description Entries

Data-Description Entry	Level-Numbers Allowed
FD, SD, or CD	Level-numbers 01 through 49, 66, or 88
RD	Level-numbers 01 through 49
Data-description entries in WORKING-STORAGE and LINKAGE SECTIONs	Level-numbers 01 through 49, 66, 77, or 88

Example

Example 6–2 illustrates a record description in COBOL74 for a record called EMPLOYEE-INFO. The PICTURE clause is associated with each elementary item.

In the example, the following information applies:

- The EMP-PAY-DATA group includes all items until the EMP-LAST-REVIEW group, which has an equal level-number.
- The EMP-DEDUCTIONS group includes all items until the EMP-LAST-REVIEW group, which has a lower level-number than EMP-DEDUCTIONS.
- The EMP-DEDUCTIONS group is a part of the EMP-PAY-DATA group.
- The EMP-INSURANCE group is a part of the EMP-DEDUCTIONS group.
- EMP-HOSPITAL is a part of the EMP-INSURANCE group.

```
01  EMPLOYEE-INFO.
    03        EMP-NO                PIC 9(5).
    03        EMP-COST-CNTR         PIC 99.
    03        EMP-NAME.
      05      EMP-LAST-NAME         PIC X(13).
      05      EMP-FIRST-INITIAL     PIC X.
      05      EMP-M-INITIAL         PIC X.
    03        EMP-ANNUAL-SALARY     PIC 9(6)V99.
    03        EMP-DT-HIRED.
      05      EMP-H-MONTH           PIC 99.
      05      EMP-H-DAY             PIC 99.
      05      EMP-H-YEAR            PIC 99.
    03        EMP-PAY-DATA.
      05      EMP--GROSS            PIC 9(6)V99.
      05      EMP-DEDUCTIONS.
        07    EMP-INSURANCE.
          09  EMP-HOSPITAL          PIC 9(4)V99.
          09  EMP-LIFE              PIC 9(4)V99.
        07    EMP-TAXES.
          09  EMP-FICA              PIC 9(4)V99.
          09  EMP-STATE-TAX         PIC 9(4)V99.
          09  EMP-WITHHOLDING       PIC 9(4)V99.
   03         EMP-LAST-REVIEW.
     05       EMP-R-MONTH           PIC 99.
     05       EMP-R-DAY             PIC 99.
```

Example 6–2. Level-Number Construction for a Record

Data

Data handling in COBOL includes classifying data into categories and classes, qualifying data to ensure uniqueness, and positioning data in a receiving field.

Categories

COBOL74 supports the following seven categories of data items:

- Alphabetic
- Numeric
- Numeric-edited
- Alphanumeric
- Alphanumeric-edited
- Kanji
- Kanji-edited

Classes

These seven categories of data items are grouped into the following three classes:

- Alphabetic
- Numeric
- Alphanumeric

For alphabetic and numeric data items, the classes and categories are synonymous. The alphanumeric class includes the categories of alphanumeric-edited, numeric-edited, alphanumeric (without editing), Kanji, and Kanji-edited. Every elementary item, except an index data item, belongs to one of the classes and also to one of the categories. The class of a group item is treated at execution time as alphanumeric regardless of the class of the elementary items subordinate to that group item.

Relationship Between Categories and Classes

Table 6–4 depicts the relationship of the classes and categories of data items.

Table 6–4. Classes and Categories of Data Items

Level of Item	Class	Category
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Numeric-edited
		Alphanumeric
		Alphanumeric-edited
		Kanji
		Kanji-edited
Group	Alphanumeric	Alphabetic
		Numeric
		Numeric-edited
		Alphanumeric
		Alphanumeric-edited
		Kanji
		Kanji-edited

Qualifying Data to Ensure Uniqueness

Every user-specified identifier that defines an element in a COBOL source program must be unique. The user-specified identifier can be unique if it meets one of the following two conditions:

- No other name has the identical spelling and hyphenation.
- The name exists within a hierarchy of names. You can make a name unique by mentioning one or more of the higher levels of the hierarchy when you refer to the name.

The higher-level names in the hierarchy are called qualifiers. The process that ensures uniqueness is called qualification. You must provide enough qualification to make the name unique. Unless it is necessary to make the name unique, you do not need to specify all the levels of the hierarchy.

In the DATA DIVISION, you must associate all data-names used for qualification with a level indicator or a level-number. Therefore, two identical data-names must not appear as entries subordinate to a group item unless they can be made unique through qualification.

In the hierarchy of qualification, names associated with a level indicator, such as a file description (FD), are the most significant. The next most significant names are those associated with level-number 01, then level-number 02, and so on through 49.

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

You can make subscripted or indexed data-names, conditional variables, procedure-names, and data-names unique by qualification. Regardless of the available qualification, no name can be both a data-name and a procedure-name.

You can qualify a data-name, a condition-name, a paragraph-name, or a text-name by specifying one or more phrases composed of an IN or an OF keyword followed by a qualifier. The keywords IN and OF are logically equivalent.

Each qualifier must be of a successively higher level and must be within the same hierarchy as the name it qualifies.

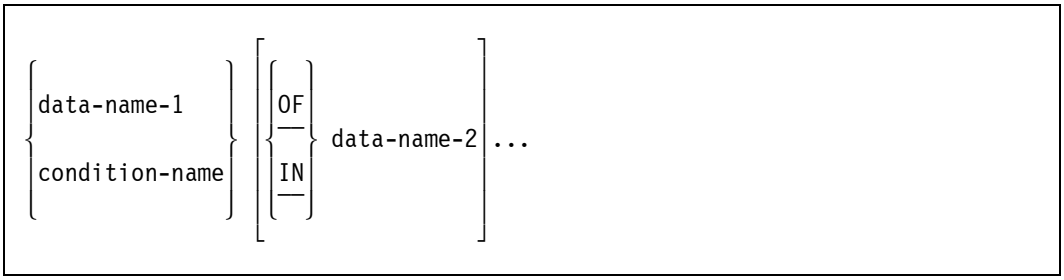
The same name must not appear at two levels in a hierarchy.

Qualification has three formats. These formats are used as follows:

Format	Explanation
1	Qualifies a data-name or a condition-name.
2	Qualifies a paragraph-name.
3	Qualifies a COPY library-name.

Format 1

The Format 1 qualification is as follows:



Explanation of Format 1

The rules for qualifying a data-name or condition-name are as follows:

- You can use the name of a conditional variable as a qualifier for any of the condition-names of the variable.
- If a data-name or a condition-name is assigned to more than one data item in a source program, you must qualify the data-name or a condition-name each time it is referenced in the PROCEDURE, ENVIRONMENT, and DATA DIVISIONs (except in the REDEFINES clause where qualification is unnecessary and must not be used).

- A data-name cannot be subscripted when it is being used as a qualifier.
- A name can be qualified even if it does not require qualification. If more than one combination of qualifiers ensures uniqueness, any partial set of qualifiers for another set of qualifiers for a data-name must not be the same as any partial set of qualifiers for another data-name. Qualified data-names can have up to 49 qualifiers, inclusive.

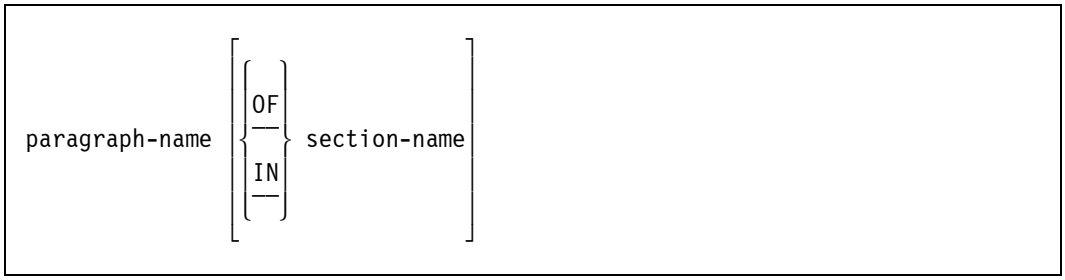
  The following example illustrates qualified names:

```
01  EMP-FULL-NAME
    03  LAST-NAME
    03  FIRST-NAME
    03  MIDDLE-NAME
01  EMP-NAME
    03  LAST-NAME
    03  FIRST-NAME
    03  MIDDLE-INITIAL
```

In this example, because LAST-NAME is not unique, it must be used with a qualifier to specify either LAST-NAME OF EMP-FULL-NAME or LAST-NAME OF EMP-NAME. Because MIDDLE-INITIAL is unique, it can be used without qualification. However, qualifying MIDDLE-INITIAL as MIDDLE-INITIAL OF EMP-NAME is also correct.

Format 2

The Format 2 qualification is as follows:



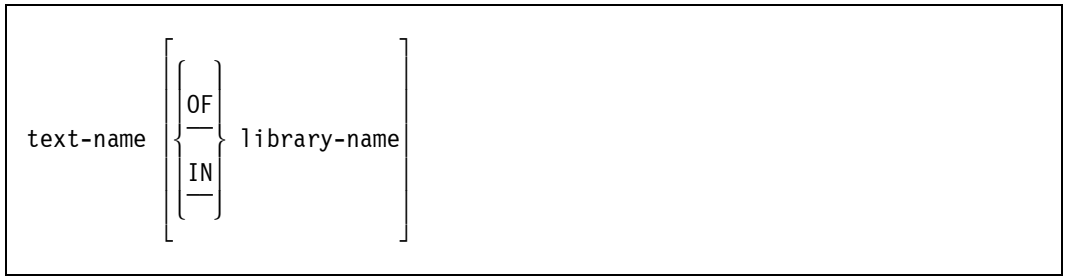
Explanation of Format 2

A section-name is the highest and only qualifier available for a paragraph-name.

You must not duplicate a paragraph-name in a section. When you qualify a paragraph-name with a section-name, do not include the word SECTION. You do not have to qualify a paragraph-name if you refer to it within the same section.

Format 3

The Format 3 qualification is as follows:



Explanation of Format 3

If text-name is unique, it defaults to the library it is in. Otherwise, you must qualify the text-name each time you refer to it. You must also qualify the text-name if more than one COBOL library is available to the compiler during compilation.

Aligning Data

The positioning of data within an elementary item by using a move operation depends on the data category of the receiving item. Table 6–5 describes the alignment rules that apply for each category of data item.

Table 6–5. Alignment Rules for Move Operation by Data Categories

Data Category	Alignment Rules
Numeric	Aligns the data by decimal point and moves it to the receiving character positions with zero-fill or truncation on either end, as required. If you do not specify an assumed decimal point, the compiler treats the data item as if it had an assumed decimal point immediately following its rightmost character and aligns it as previously described.
Numeric-edited	Aligns the data that was moved to the edited data item; the data is aligned by decimal point with zero-fill or truncation at either end, as required, within the receiving character positions of the data item, except where editing requirements cause replacement of the leading zeros.
Alphanumeric (other than a numeric-edited data item), alphanumeric-edited, or alphabetic	Moves the sending data to the receiving character positions and aligns it at the leftmost character position in the data item with space-fill or truncation to the right, as required.
Kanji or Kanji-edited	Moves the sending data to the receiving character positions and aligns it at the leftmost character position in the data item. Fills the receiving data item on the right with Kanji space characters or truncates it from the right to fully occupy the field.

For information on aligning data in the receiving field of an alphanumeric move operation, refer to "JUSTIFIED Clause" in Section 7, "DATA DIVISION."

Tables

A table is a set of logically consecutive data items that you specify in the DATA DIVISION with an OCCURS clause. By using a table, you can do the following with your data:

- Define contiguous data items.
- Access an item by its position in the table.
- Specify the number of repetitions for an item.
- Identify each item with a subscript or an index.
- Access items in multidimensional, variable-length tables.
- Specify ascending or descending keys.
- Search a dimension of a table for an item that satisfies a specified condition.

You can define a table composed of contiguous data items by including the OCCURS clause in a data-description entry. The OCCURS clause specifies that the item is to be repeated the number of times you designate.

The item is considered a table element, and the name and description of the item apply to each repetition or occurrence. Because each table element does not have a unique data-name, you must refer to a desired occurrence by specifying the table element data-name and the desired occurrence number. You can specify an occurrence number with subscripting or indexing.

For information on defining the number of items in a table, refer to "OCCURS Clause" in Section 7, "DATA DIVISION."

Defining Tables

To define a one-dimensional table, you must use an OCCURS clause as part of the data description of the table element. The OCCURS clause must not appear in the description of group items that contain the table element.

Example 6–3 shows a one-dimensional table definition.

```
01  TABLE-1.
    03  TABLE-ELEMENT   OCCURS 20 TIMES
        05 NAME         PIC X(30)
        05 SSAN         PIC 3(6).
```

Example 6–3. Defining a One-Dimensional Table

To create a two-dimensional table, you need to define a one-dimensional table within each occurrence of an element of another one-dimensional table. To accomplish this, include an OCCURS clause in the data description of the element of the table and in the description of only one group item that contains that table element. To define a three-dimensional table, you must include the OCCURS clause in the data description of the element of the table and in the description of two group items that contain that table element. In COBOL, you can define tables of up to 48 dimensions.

Example

Example 6–4 is a table with the following dimensions:

- One dimension for CONTINENT-NAME. The CONTINENT-NAME occurs eight times within the table.
- Two dimensions for COUNTRY-NAME. The COUNTRY-NAME occurs 15 times within each CONTINENT-NAME.
- Three dimensions for CITY-NAME and CITY-POPULATION. The CITY-NAME and the CITY-POPULATION occur 20 times within each COUNTRY-NAME.

The table includes 4,928 data items grouped as follows:

Number of Items	Dimension
8 data items	CONTINENT-NAME
120 data items	COUNTRY-NAME
2,400 data items	CITY-NAME
2,400 data items	CITY-POPULATION

Example 6–4 provides data-description entries for a three-dimensional table. CITY-NAME(3,7,19) refers to the nineteenth city of the seventh country of the third continent.

```
01  CENSUS-TABLE.
    05  CONTINENT-TABLE                    OCCURS 8 TIMES.
        10  CONTINENT-NAME          PIC X(16).
        10  COUNTRY-TABLE                  OCCURS 15 TIMES.
            15  COUNTRY-NAME       PIC X(18).
            15  CITY-TABLE                 OCCURS 20 TIMES.
                20  CITY-NAME      PIC X(10)
                20  CITY-POPULATION  PIC X (12)
```

Example 6–4. Defining a Three-Dimensional Table

Accessing Tables

Whenever a program refers to a table element, the reference must indicate the intended occurrence of the element. For access to a one-dimensional table, the occurrence number of the desired element provides complete information. For access to tables of more than one dimension, supply an occurrence number for each dimension of the table accessed.

In Example 6–4, shown earlier in this section, a reference to the fourth CONTINENT-NAME is complete, but a reference to the fourth COUNTRY-NAME is not. To refer to the dimension COUNTRY-NAME, which is an element of a two-dimensional table, the fourth COUNTRY-NAME within a particular CONTINENT-NAME must be referenced.

Subscripting

A subscript is an integer or a data-name whose value refers to an individual element in a list or table of like elements that have not been assigned individual data-names.

To use a subscript, you must first provide an OCCURS clause to define multiple occurrences of a data item. The data items to be repeated must all have the same format.

The format for subscripting follows:

```
{data-name     }
{              } (subscript-1 [,subscript-2 [,subscript-3]...])
{condition-name}
```

Explanation of Format

The subscript identifies the table element that is to be accessed.

After the table element data-name, you must delimit the subscript or the set of subscripts that identifies the table element by enclosing the subscripts in parentheses. The table element data-name with its subscript is called a subscripted data-name or an identifier. When more than one subscript is required, write the subscripts in descending order of inclusive dimensions of the data organization.

A PERFORM statement can include a subscript. In the VARYING option, the compiler initializes, increments, and tests the subscript. In the UNTIL or TIMES option, you must provide the code to initialize and increment the subscript.

You can represent the subscript either by a numeric literal that is an integer or by a data-name that is a numeric elementary item representing an integer. When you represent the subscript with a data-name, you can qualify the data-name but not subscript it.

In the REPORT SECTION of Report Writer, you cannot use a sum counter or the special registers LINE-COUNTER and PAGE-COUNTER as subscripts.

The subscript can be signed; if signed, it must be positive. The lowest valid subscript value is 1, which points to the first element of the table. Subscript values of 2, 3, and so on point to subsequent sequential elements of the table. The highest permitted subscript value is the maximum number of occurrences of the item, as specified in the OCCURS clause.

When the program executes a statement that refers to an indexed table element, the value of each direct or relative index should *not* be as follows:

- Less than a value that corresponds to the beginning of the first occurrence of the table element
- Greater than a value that corresponds to the beginning of the last occurrence of the table element (as determined by the OCCURS clause)

The index value need not precisely address the beginning of a table element. For example, you can set an index-name to the value of an index data item that has been set to the value of another index-name. (Such assignments are made without conversion.)

Subscripts may be used even when the table is declared as INDEXED BY. (This is an extension to ANSI X3.23-1974 COBOL). See the discussion on subscripting earlier in this section for the applicable conventions.

An invalid index occurs at run time if the program attempts to access an item beyond the end of the 01-level record. No other range checking is performed on index-names, either during the execution of the SET statement that modifies the value of the index-name, or during references to the table.

Thus, you must make sure that the contents of an index-name are valid for the table that it refers to at execution time.

Failing to observe the table limits can produce unexpected results in those cases where the values of the index-names are incorrect, but do not cause access outside the bounds of the 01-level record. Using inappropriate index values to access parts of an 01-level record outside the bounds of the table is strongly discouraged.

For More Information

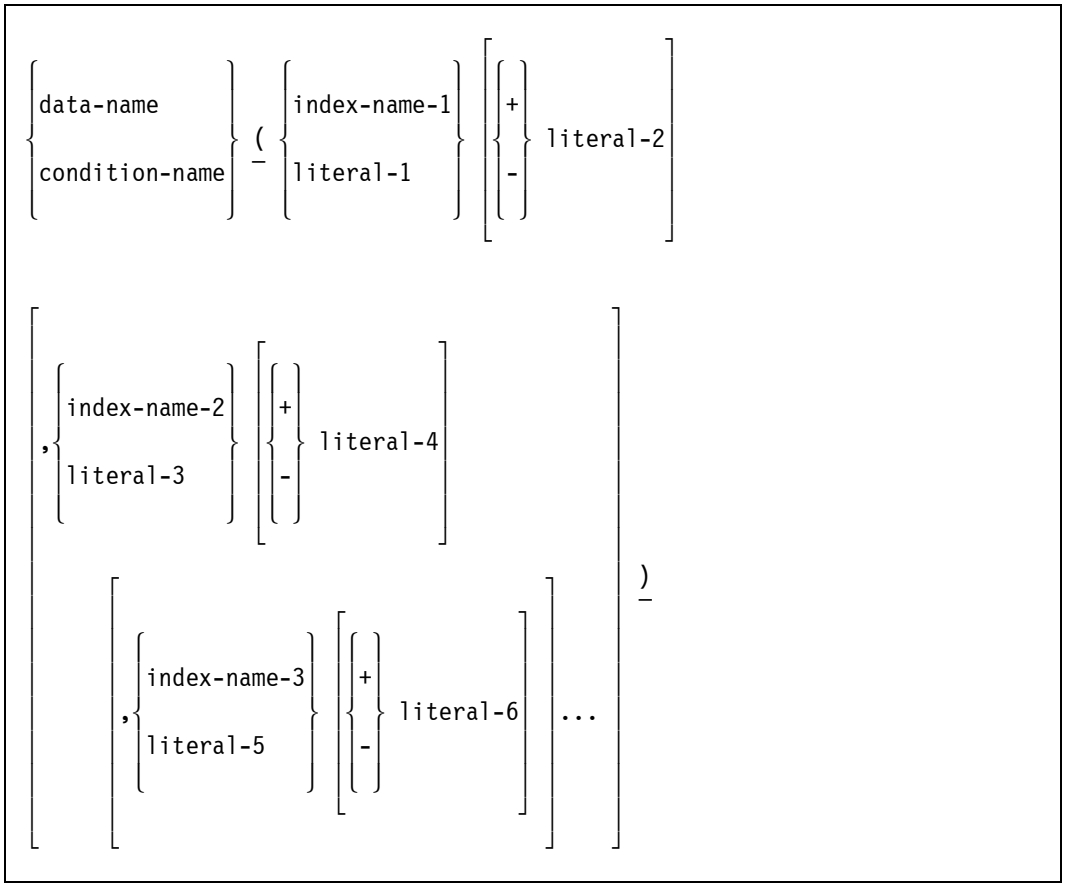
- For information on defining the number of items in a table, refer to "OCCURS Clause" in Section 7, "DATA DIVISION."
- For information on using the OPTIMIZE compiler option, see Section 17, "Control of the Compilation Process."

Indexing

An index, like a subscript, identifies the individual elements in a table of like elements. The compiler assigns an index to the level of the table when the program defines the table with an INDEXED BY phrase in the OCCURS clause. The name given in the INDEXED BY phrase is called an index-name and refers to the assigned index. The value of an index corresponds to the occurrence number of an element in the associated table.

To assign a value to an index-name, the program must execute a SET statement, a SEARCH ALL statement, or a Format 4 PERFORM statement.

The general format for indexing follows:



Explanation of Format

An index-name has the same internal representation as an index data item. If a value to be stored in an index-name or an index data-name exceeds the largest value that can be held in that index-name or index data-name, the value is truncated. The truncation is executed according to the rules for size error conditions in an arithmetic statement without a SIZE ERROR phrase. (This is an extension to ANSI X3.23-1974 COBOL.)

An index-name assigned to one table cannot be used to index another table. (This is an extension to ANSI X3.23-1974 COBOL.)

You can specify direct indexing by using an index-name in the form of a subscript. You can specify relative indexing by following the index-name with the operator plus sign (+) or minus sign (–), followed by an unsigned integer numeric literal, all of which are delimited by parentheses following the table element data-name. The compiler determines the occurrence number resulting from relative indexing by incrementing (when the operator + is used) or decrementing (when the operator – is used) the occurrence number by the value of the literal. The occurrence number is represented by the value of the index. When more than one index-name is required, the names are written in descending order of inclusive dimensions of the data organization.

When the program executes a statement that refers to an indexed table element, the value of each direct or relative index should *not* be as follows:

- Less than a value that corresponds to the beginning of the first occurrence of the table element
- Greater than a value that corresponds to the beginning of the last occurrence of the table element (as determined by the OCCURS clause)

The index value need not precisely address the beginning of a table element. For example, you can set an index-name to the value of an index data item that has been set to the value of another index-name. (Such assignments are made without conversion.)

Subscripts may be used even when the table is declared as INDEXED BY. (This is an extension to ANSI X3.23-1974 COBOL.) See the discussion on subscripting earlier in this section for the applicable conventions.

An invalid index occurs at run time if the program attempts to access an item beyond the end of the 01-level record. No other range checking is performed on index-names, either during the execution of the SET statement that modifies the value of the index-name, or during references to the table.

Thus, you must make sure that the contents of an index-name are valid for the table that it refers to at execution time.

Failing to observe the table limits can produce unexpected results in those cases where the values of the index-names are incorrect, but do not cause access outside the bounds of the 01-level record. Using inappropriate index values to access parts of an 01-level record outside the bounds of the table is strongly discouraged.

For More Information

- For information about specifying indexes, refer to “OCCURS Clause” in Section 7,“ DATA DIVISION.”
- For information about specifying the format of a data item in storage, refer to “USAGE Clause” in Section 7, “DATA DIVISION.”

Editing

Editing is the process of using symbols in the PICTURE clause to specify the format of data for output reports. The editing operation occurs as the data item is moved from the sending field to the receiving field.

Some of the applications for which you might want to use editing include the following:

- Separating fields with delimiters such as zeros or spaces
- Adding asterisks for protection of amounts on checks
- Adding credit or debit symbols for accounting purposes
- Formatting monetary values with commas (,) and dollar signs ($)
- Indicating positive or negative values with plus (+) and minus (–) signs
- Preceding the fractional part of an amount with a decimal point (.)

For More Information

A complete description of editing is provided with the PICTURE clause in Section 7, “DATA DIVISION.”

Section 7
DATA DIVISION

The third division of a source program, the DATA DIVISION, describes the data that the object program accepts as input to manipulate, create, or produce as output. Data to be processed falls into one of three categories:

- Data that is contained in files and that enters or leaves the internal memory of the computer from a specified area or areas
- Data developed internally and placed in intermediate or working storage or in a specific format for output-reporting purposes
- Constants that you define

Sections of the DATA DIVISION

The DATA DIVISION, one of the required divisions in a program, is subdivided into seven sections:

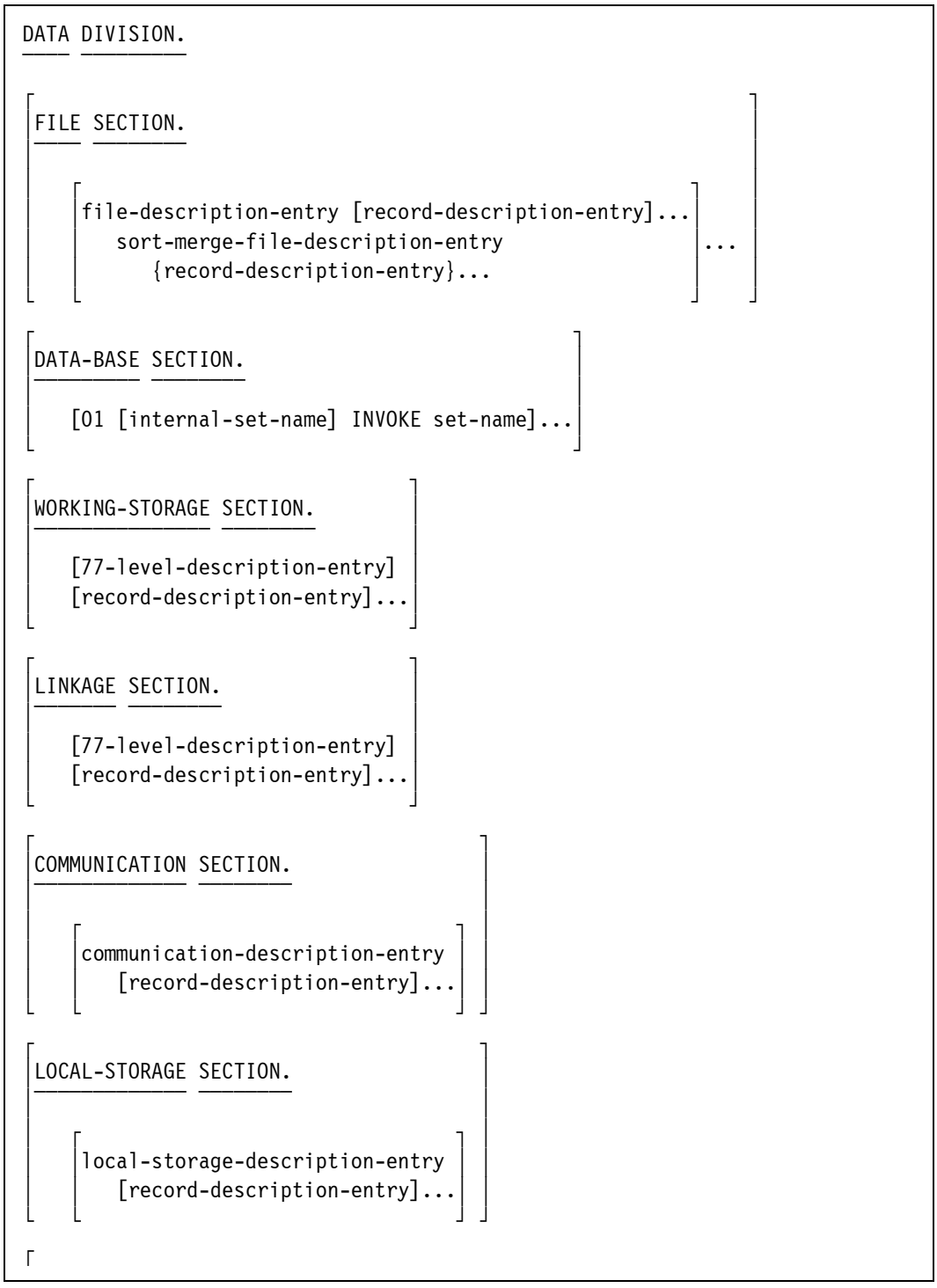
- The FILE SECTION defines the structure of data files.
- The DATA-BASE SECTION, an extension to ANSI X3.23-1974 COBOL, describes one or more databases that can be used by the COBOL program.
- The WORKING-STORAGE SECTION describes records and noncontiguous data items that are not part of external data files but are developed and processed internally.
- The LOCAL-STORAGE SECTION, an extension to ANSI X3.23-1974 COBOL, describes parameters to be received by separate tasks or by procedures to be bound from another program.
- The LINKAGE SECTION appears in the called program and describes data items to be referenced by the calling program and the called program.
- The COMMUNICATION SECTION describes the data items in the source program that serve as the interface between the data communications interface (DCI) library and the program.
- The REPORT SECTION describes the contents and format of generated formats.

For More Information

- For information about the REPORT SECTION, refer to Section 12, “Report Writer.”
- For information about the LINKAGE SECTION, refer to Section 13, “ANSI Inter-Program Communication (IPC).”

- For details about the COMMUNICATION SECTION, refer to Section 14, "COMMUNICATION SECTION."
- For information about the DATA-BASE SECTION and the INVOKE clause, refer to Volume 2 of this manual.
- For information about the SAME RECORD AREA clause, refer to Volume 2 of this manual.

The general structure of the DATA DIVISION is as follows. The sections FILE, WORKING-STORAGE, and LOCAL-STORAGE are explained on the following pages.


DATA DIVISION.
FILE SECTION.
file-description-entry [record-description-entry]...
sort-merge-file-description-entry
{record-description-entry}...
DATA-BASE SECTION.
[01 [internal-set-name] INVOKE set-name]...
WORKING-STORAGE SECTION.
[77-level-description-entry]
[record-description-entry]...
LINKAGE SECTION.
[77-level-description-entry]
[record-description-entry]...
COMMUNICATION SECTION.
communication-description-entry
[record-description-entry]...
LOCAL-STORAGE SECTION.
local-storage-description-entry
[record-description-entry]...

```
                                        ┐
REPORT SECTION.                         │
                                        │
   [                                ]   │
   report-description-entry             │
      {report-group-description-entry}... │
   [                                ]   ┘
```

FILE SECTION

The FILE SECTION defines the structure of data files used in the program. These files have been previously named and assigned to a device in the SELECT clause. Typically, each file is defined by a file-description (FD) entry and one or more record descriptions. Record descriptions are written immediately following the FD entry.

When the file description (FD) specifies a file to be used as a Report Writer output file, this file is defined by a FD entry, but no record-description entries are permitted. Report-description entries appear in the REPORT SECTION.

File-Description (FD) Entry

The file-description (FD) entry identifies a file previously declared in the SELECT clause and provides information about its physical structure. Record descriptions for the file immediately follow its file description.

There are two formats for describing files. These formats are used as follows:

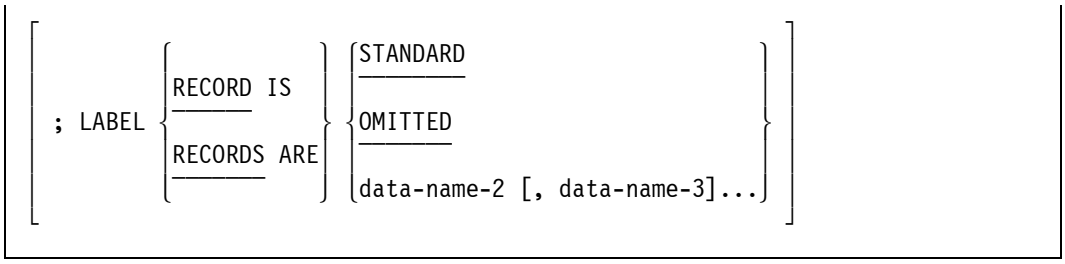
Format	Explanation
1	This format describes input and output files.
2	This format describes sort and merge files.

Format 1: File Description (FD) Entry

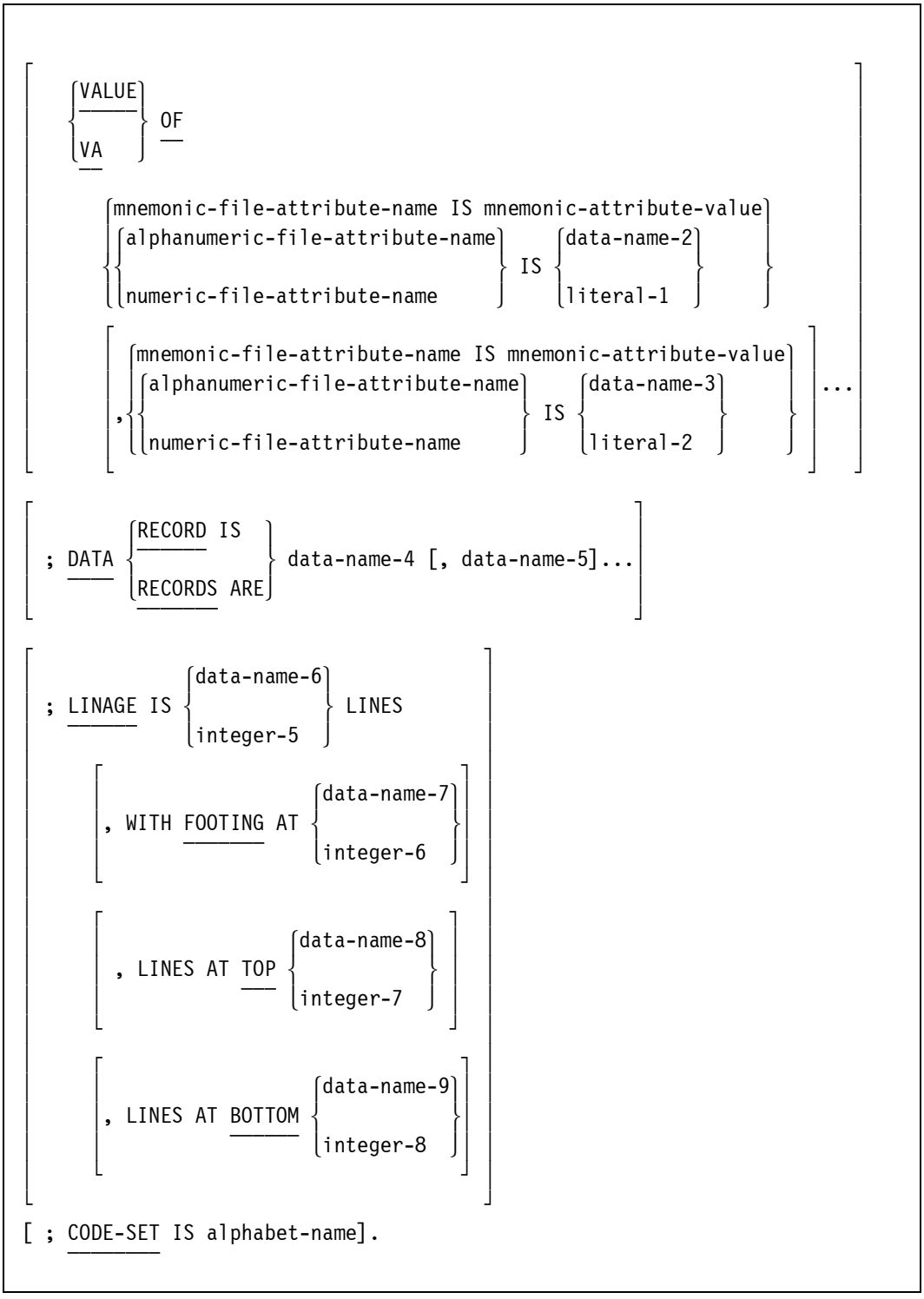
```
FD file-name
──

[                                                    ]
                                                {RECORDS   }
 ; BLOCK CONTAINS [integer-1 TO] integer-2 {──────────}
                                                {CHARACTERS}
[                                                    ]

[                                                    ]
 ; RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS
                                                     
     [DEPENDING ON data-name-1]                      
[                                                    ]
```


; LABEL
RECORD IS
RECORDS ARE
STANDARD
OMITTED
data-name-2 [, data-name-3]...

Format 1: File Description (FD) Entry



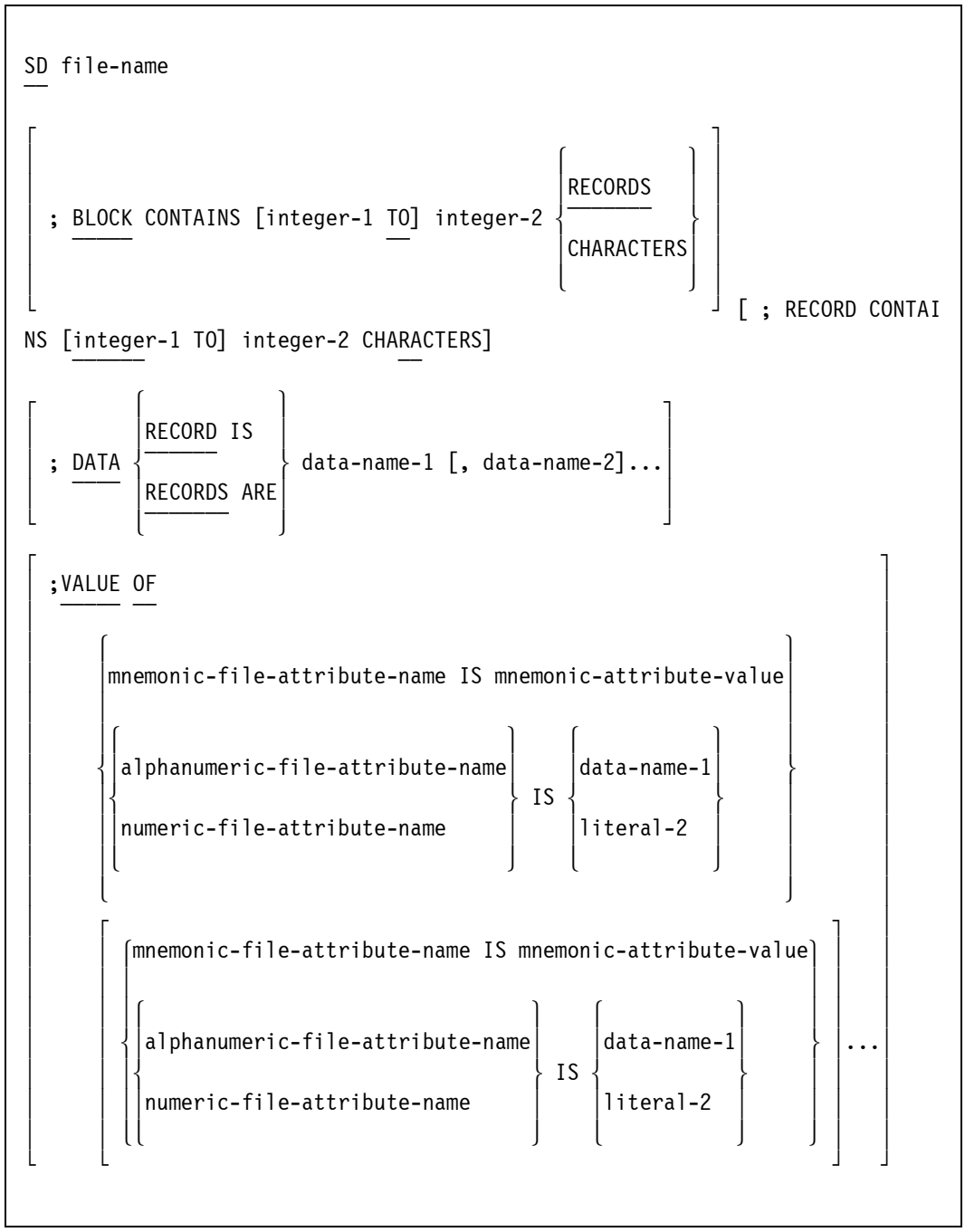
Explanation of Format 1

In a COBOL74 program, the file-description (FD) entry represents the highest level of organization in the FILE SECTION. The FILE SECTION header is followed by a file-description (FD) entry consisting of a level indicator (FD), a file-name, and a series of independent clauses. An FD identifies the beginning of a file description and must provide the file-name.

The FD clauses specify the size of the logical and physical records, the presence or absence of label records, the value of file attributes, the names of the data records that make up the file, the character code set, the number of lines to be written on a logical printer page, and the name or names of the reports pertaining to a given file. The clauses that follow the name of the file are optional in many cases, and their order of appearance is immaterial. The FD clauses are described on the following pages.

One or more record description entries must follow the file-description (FD) entry, except when the REPORT clause in Report Writer is specified.

Format 2: Sort Merge Description (SD) Entry



Explanation of Format 2

A sort merge description (SD) entry gives information about the size and names of the data records associated with the file to be sorted. No label procedures can be controlled by users, and the rules for blocking and internal storage are peculiar to the SORT statement.

The level indicator SD identifies the beginning of the sort-merge description entry and must precede the file-name. Other clauses that follow the name of the file are optional, and their order of appearance is immaterial. The SD clauses are described on the following pages.

One or more record-description entries must follow the SD entry; however, no I/O statements (except RELEASE and RETURN) can be executed for this file.

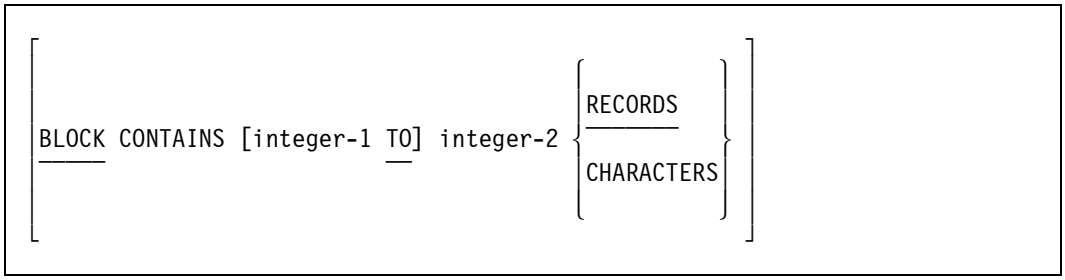
BLOCK CONTAINS Clause

The BLOCK CONTAINS clause specifies the size of a physical record, also known as the blocking factor.

This clause is required when the physical record contains more than one logical record. If this clause is not specified, the physical record is assumed to contain one logical record as large as the largest record specified for the file. (This is an extension to
ANSI X3.23-1974 COBOL).

This clause is not required in an SD file entry and has no effect on the SD file entry if specified.

The general format of this clause is as follows:



Explanation of Format

If only integer-2 is shown, it represents the exact number of records or characters in the physical record. If integer-1 and integer-2 are both shown, they refer to the minimum and maximum size of the physical record, respectively.

When the word RECORDS is not specified, the value of integer-2 must not be less than the largest record specified for the file.

When RECORDS is specified, the physical record size is considered to be integer-2 multiplied by the size of the largest record specified for this file.

When the word CHARACTERS is specified, the physical record size is considered to be integer-2. If integer-2 is not a multiple of the size of the largest record specified for the file, the physical record size is adjusted to be a multiple of the size of the largest record specified, not to exceed the value of integer-2. (This is an extension to ANSI X3.23-1974 COBOL).

If logical records of differing sizes are grouped into one physical record, the amount of data transferred from the record area to the physical record depends on the size of the record named in the WRITE or REWRITE statement. In this case, the logical records are aligned on maximum record-size boundaries. If the size of the record named does not equal the maximum record size specified for the file, the data is transferred to the physical record according to the rules specified for the MOVE statement without the CORRESPONDING phrase. The sending area is considered to be a group item.

If variable-length records are specified, then the physical record size is determined as follows:

- If the word RECORDS is shown and only integer-2 is shown, the physical record size equals integer-2 multiplied by the maximum record size.
- If the word RECORDS is shown and integer-1 and integer-2 are both shown, the physical record size equals integer-1 multiplied by the maximum record size or integer-2 multiplied by the minimum record size, whichever is larger.
- If the word CHARACTERS is shown, the physical record size equals integer-2 or the maximum record size, whichever is larger. If the maximum record size is larger, a warning is issued. Integer-1 is shown for documentation only.

In the case of relative file organization, the physical record size is adjusted by the I/O subsystem to be integer-2 multiplied by a value that is 6 bytes larger than that which would be determined by the previously stated methods.

For More Information

- For more information about specifying record size, refer to "RECORD CONTAINS Clause," in this section.
- For information about record-blocking techniques, refer to the *I/O Subsystem Programming Guide.*

RECORD CONTAINS Clause

The RECORD CONTAINS clause specifies the size of the data records. It can be used to specify variable-length records.

The general format of this clause is as follows:

```
RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS

   [DEPENDING ON data-name-1]
```

Explanation of Format

The DEPENDING ON clause is valid only if the *integer-3 TO* clause is present. The *integer-3 TO* clause is ignored for indexed or relative files; the DEPENDING ON clause is not allowed for indexed or relative files.

If the data-name-1 option is in the record of the file, then it must reference an elementary unsigned numeric item of USAGE IS DISPLAY (4 characters long), which is the first item in the record. If data-name-1 is not in the record of the file, then it must be an elementary unsigned numeric item.

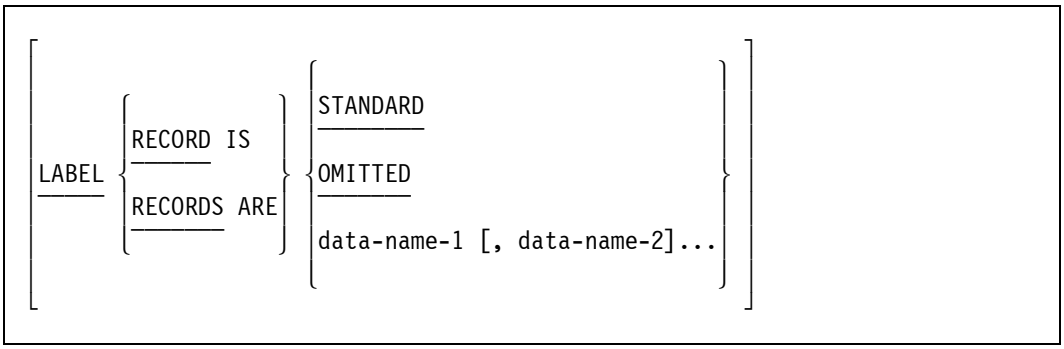
The size of each data record is completely defined in the record-description entry; therefore, the RECORD CONTAINS clause is never required. When this clause is present, however, the following rules apply:

- For fixed-length records, use integer-4 alone when all data records in the file have the same size. In this case, integer-4 represents the exact number of characters in the data record. For fixed-length records, do not use the TO clause.
- The record size 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 in the logical record. The size of a record is determined by the sum of the number of characters in any variable-length item subordinate to the record. This sum can be different from the actual size of the record.
- For variable-length records, integer-3 and integer-4 are both used. They refer, respectively, to the minimum number of characters in the smallest data record and the maximum number of characters in the largest data record. The following rules apply:
  - If the DEPENDING ON clause is not specified, the logical-record length is supplied by the system and is written to the record as the first four characters of the record. This length cannot be referenced by the program. The format of the length depends on the USAGE value of the first 01-level record of the FD. For DISPLAY, the length is four EBCDIC characters; for COMP, four packed decimal characters; for BINARY, one binary word. The length is part of the record when the type of the file is DISK or TAPE, but is not written if the file is REMOTE, PRINTER, or PORT.
  - If the DEPENDING ON clause is specified, the logical-record length is supplied by the program at run time in data-name-1. Data-name-1 must follow the previously stated syntax rules. If data-name-1 is in the record, the logical-record length of data-name-1 must include the four bytes of data-name-1.
  - When a READ statement is executed, the contents of the data item referred to by data-name-1 indicate the maximum record size of the record just read. At end-of-file (EOF), data-name-1 is set to 0 (zero).

LABEL RECORDS Clause

The LABEL RECORDS clause specifies the presence or absence of label information.

The general format of this clause is as follows:



Explanation of Format

STANDARD

The STANDARD phrase should be used if you wish to take advantage of the automatic file allocation and handling procedures in the operating system. (Disk devices maintain a directory instead of a system of labels.) The format of labels is dependent upon the device containing the file.

If the LABEL RECORDS clause is not used, the STANDARD phrase is assumed.

OMITTED

The OMITTED phrase must be used if an input file does not have standard labels or if labels are not desired on output files.

data-name-1

All references to data-name-1 also apply to data-name-2 and so on.

You should use the data-name-1 option to include header and trailer records in the standard tape label. This format can be used only with magnetic tape files. A maximum of nine label records can be specified.

Data-name-1 identifies the tape label user record descriptions to be used by the label USE procedures for a given file. The file handling routine of the operating system performs the USE statement when the file is opened, closed, or when a volume is switched. Label records are made available only when both the LABEL RECORDS clause and the USE statement are specified.

Data-name-1 must be defined in the file-description (FD) entry for which it is defined or must be in the WORKING-STORAGE SECTION. The subordinate items identify the fields to be accessed in the label records and must add up to a total record size of 80 characters. The first four characters of every label record are reserved for use by the operating system.

Example

Example 7–1 shows a file-description (FD) entry with a LABEL RECORDS clause using the data-name-1 option. TAPE-LABEL is a record description in the file TAPE-FILE. The record description identifies the label to be used in the USE procedures.

```
FD  TAPE-FILE
    LABEL RECORD IS TAPE-LABEL
    VALUE OF FILENAME "TAPELABEL1"
    DATA RECORD IS TAPE-DATA.
01  TAPE-LABEL.
    03  LABEL-RESERVE              PIC X(4).
    03  DATE-OF-CREATION           PIC 9(6).
    03  FORMAT-TYPE                PIC XX.
    03  REMAINING-DATA             PIC X(68).
01  TAPE-DATA.
     :
```

Example 7–1. Coding the LABEL RECORDS Clause

For More Information

- For information about the format of the USE procedures, refer to Format 2 of the USE statement in Section 9, "PROCEDURE DIVISION Statements."
- For information about label formats, refer to the *I/O Subsystem Programming Guide.*
- For an example of tape label access by way of USE routines, refer to Appendix D.

VALUE OF Clause

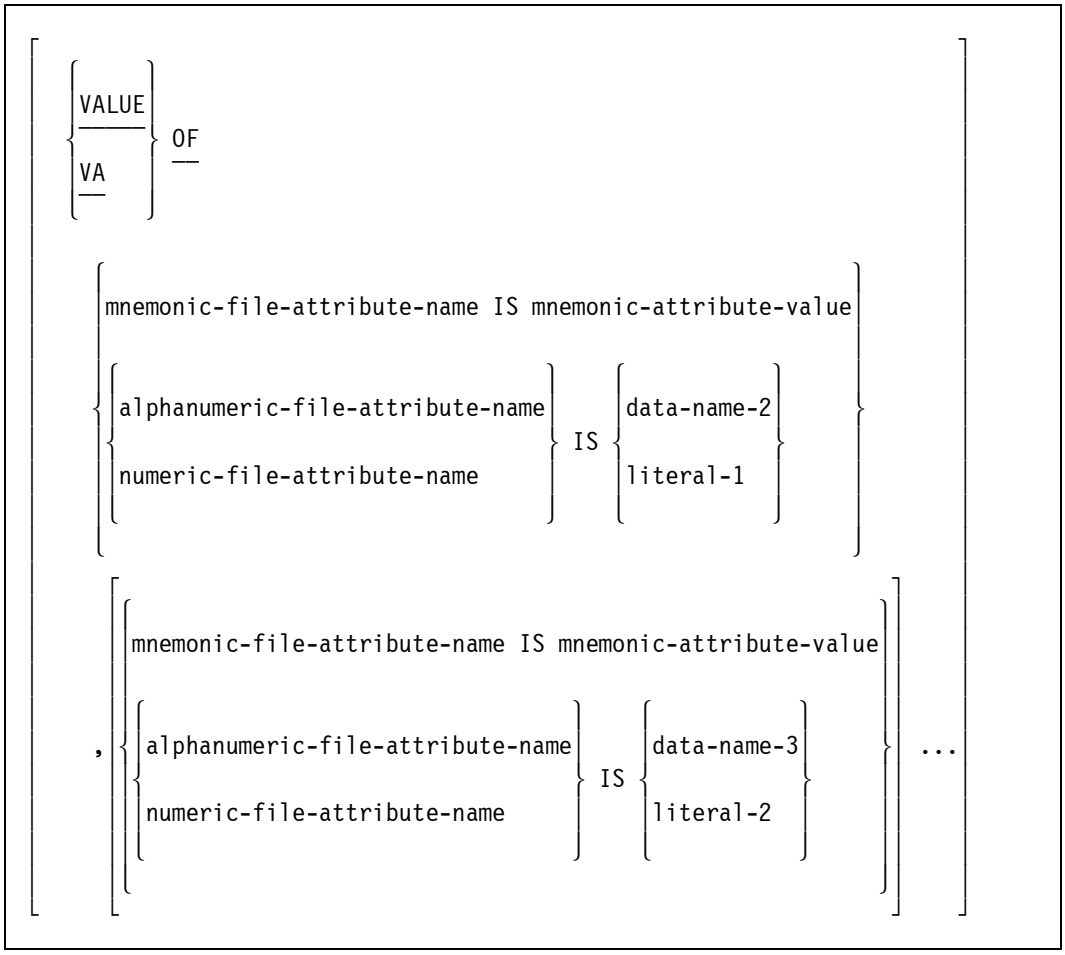
The VALUE OF clause defines the initial values for the attributes of a file.

The descriptive clauses and phrases of the INPUT-OUTPUT SECTION and the file record descriptions (other than the VALUE OF clause) implicitly determine the initial values for appropriate attributes of a file. These attribute values, however, can be overridden, or other attributes can be specified, by the VALUE OF clause.

Note: *File attributes provide you with access to functionality not otherwise available within the language. Also, file attributes can be used to declare and access files. When both a file attribute and the standard COBOL syntax are available to accomplish a desired function, it is always preferable to use the standard COBOL syntax because changing the attribute can lead to unexpected results in cases when the attribute is also used or altered by the compiler.*

This clause is not required in a sort merge description (SD), and has no effect on the sort merge description if specified.

The general format of this clause is as follows:



Explanation of Format

A mnemonic file attribute must be assigned a mnemonic-attribute-value. An alphanumeric or numeric file attribute can be assigned a value that is either a data-name or a literal.

The mnemonic-attribute-value must be associated with the attribute specified. (Mnemonic-attribute-value is an extension to ANSI X3.23-1974 COBOL.)

If literals (literal-1, literal-2, . . .) are used with alphanumeric file attributes, they must be nonnumeric literals.

If data-names (data-name-2, data-name-3, . . .) are used with alphanumeric file attributes, they must be USAGE DISPLAY data items, and the data contained in them must end with a period.

If literals (literal-1, literal-2, . . .) are used with numeric file attributes, they must be numeric literals.

If data-names (data-name-2, data-name-3, . . .) are used with numeric file attributes, they must be numeric data items representing integers.

When an attribute is equated to a literal value, the value becomes a part of the file description given by the file when the file is first referenced at run time. Any specification in this file description (FD) can be overridden by a file equation.

When an attribute is equated to a data-name value, the attribute is implicitly changed to this value just before execution of any explicit OPEN, SORT, or MERGE statement that references the file.

Data-name-2, data-name-3, and so on can be qualified but cannot be subscripted or indexed.

If an alphanumeric file attribute is specified, the contents of data-name must be followed by a period.

File titles must not contain special characters.

Port Files

Using data-name-2 in file descriptions for port files is not recommended for programs specifying subfiles that are to be opened independently and are to remain open simultaneously. The compiler explicitly sets all dynamic attributes for the entire file on each OPEN statement. An OPEN statement for a subfile of a port file is rejected by the operating system if any other subfile of the port file is open and if the file declaration contains a dynamic file attribute that can be modified only when the file is closed.

Using the CHANGE statement is recommended for dynamically changing attributes of port files that have multiple subfiles explicitly opened. Note that the CHANGE statement must still be executed while the port file is closed.

This restriction does not apply to programs that open the entire port file, to programs that have only one subfile of a port file open at any given time, or to file attributes that are not limited as to when they can be modified.

Example

Example 7–2 shows the coding of the VALUE OF clause.

```
026000 FILE SECTION.
028000 FD  INFIL
030000     LABEL RECORDS ARE STANDARD
032000     VALUE OF FILENAME IS "TAPEIN"
034000              SAVEFACTOR 30
036000              AREAS IS 10
038000              AREASIZE IS 1000.
040000 01  TAPE-REC  PIC X(80).
042000 FD  WORK-FILE
044000     VALUE OF FAMILYNAME IS "PACK01"
046000     VALUE OF AREAS IS 20
048000     VALUE OF FILENAME IS "TEMP01".
050000 01  WORK-REC.
```

```
052000     05 KEY-REC       PIC 9(8)  COMP.
054000     05 REM-WORK      PIC X(76).
```

Example 7–2. Coding the VALUE OF Clause

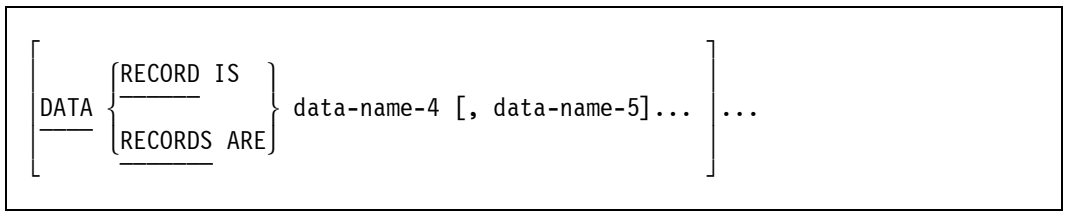
For More Information

- For general information about files, refer to Section 3, "File and Task Concepts."
- For more information about the CHANGE statement, an extension to ANSI X3.23-1974 COBOL that enables you to modify a file attribute or a task attribute, refer to the CHANGE statement in Section 9, "PROCEDURE DIVISION Statements."
- For a description of available attributes and their values, refer to the *File Attributes Reference Manual*.
- For information about using file attributes, refer to the *I/O Subsystem Programming Guide*.

DATA RECORDS Clause

The DATA RECORDS clause is an optional clause that documents the names of data records associated with a file.

The general format of this clause is as follows:



Explanation of Format

Data-name-4 and data-name-5 are the names of data records that should have 01-level record descriptions (with the same names) associated with them.

The presence of more than one data-name indicates that the file contains more than one type of data record. These records can be different (for example, in size or in format), and their listed order is not significant.

Conceptually, all data records in a file share the same area, even if more than one type of data record is present in the file.

Example

Example 7–3 shows the coding of an FD and the DATA RECORDS clause.

```
FD  INPUT-FILE.
    DATA RECORDS ARE PRODUCTION, SALES, INVENTORY.
01  PRODUCTION.
    03  REC-TYPE-1      PIC 99.
    03  REC-PROD        PIC X(78).
01  SALES.
    03  REC-TYPE-2      PIC 99.
    03  REC-SALES       PIC X(78).
01  INVENTORY.
    03  REC-TYPE-3      PIC 99.
    03  REC-INVEN       PIC X(78).
```

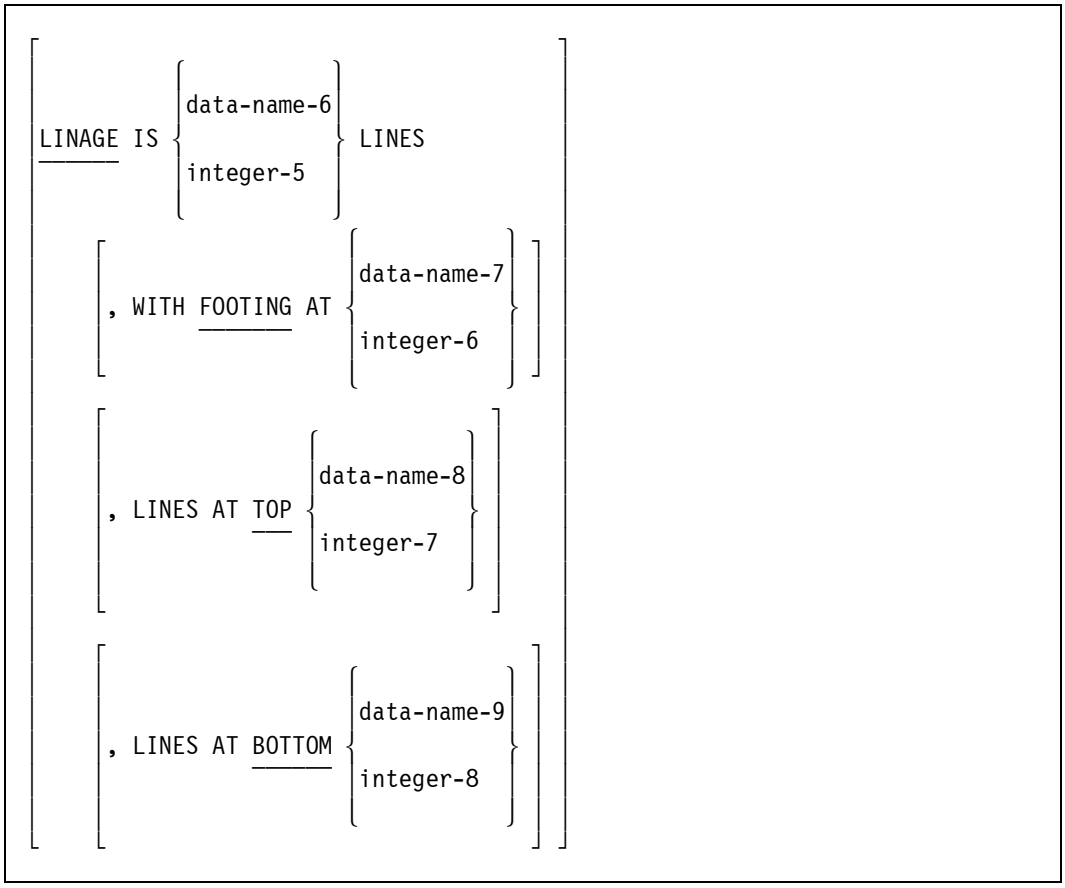
Example 7–3. Coding the DATA RECORDS Clause

LINAGE Clause

The LINAGE clause enables you to specify the number of lines per page, the size of the top and bottom margins on the logical page, and the line number within the page body at which the footing area begins.

Each logical page is contiguous to the next; no additional spacing is provided.

The general format of this clause is as follows:



Explanation of Format

The LINAGE clause enables you to specify the size (number of lines) of a logical page. The logical page size is the sum of the values referenced by each phrase except the FOOTING phrase. If the LINES AT TOP or LINES AT BOTTOM phrase is not specified, the value for this function is 0. If the FOOTING phrase is not specified, the assumed value is equal to integer-5 or the contents of the data item referenced by data-name-6, whichever is specified.

No particular relationship exists between the size of the logical page and the size of a physical page.

LINAGE IS

The value of integer-5 or of the data item referenced by data-name-6 specifies the number of lines that can be written, spaced, or both written and spaced on the logical page. The value must be greater than 0. The part of the logical page in which these lines can be written, spaced, or both written and spaced is called the page body. This value is used for all logical pages written for the file during a given execution of the program. If a WRITE ... ADVANCING PAGE statement is executed, or if a Page-Overflow condition occurs, the value specifies the number of lines for the next logical page.

LINES AT TOP

The value of integer-7 or of the data item referenced by data-name-8 specifies the number of lines desired for the top margin on the logical page. Integer-7 must be in the range from 0 through 65,535. If the value of data-name-8 is less than the number 0, the results are unpredictable. Data-name-8 can be in the range from 0 through 549,755,813,887. This value is used for all logical pages written for the file during a given execution of the program. If a WRITE ... ADVANCING PAGE statement is executed, or when a Page-Overflow condition occurs, the value specifies the top margin for the next logical page.

LINES AT BOTTOM

The value of integer-8 or of the data item referenced by data-name-9 specifies the number of lines desired for the bottom margin on the logical page. Integer-8 must be in the range from 0 through 65,535. If the value of data-name-9 is less than the number 0, the results are unpredictable. Data-name-9 can be in the range from 0 through 549,755,813,887. This value is used for all logical pages written for the file during a given execution of the program. If a WRITE ... ADVANCING PAGE statement is executed, or if a Page-Overflow condition occurs, the value specifies the bottom margin for the next logical page.

WITH FOOTING AT

The value of integer-6 or of the data item referenced by data-name-7 specifies the line number in the page body at which the footing area begins. The value must be greater than 0, and less than or equal to the value of integer-5 or the data item referenced by data-name-6.

The footing area consists of the area of the logical page between the line represented by the value of integer-6 or the data item referenced by data-name-7 and the line represented by the value of integer-5 or the data item referenced by data-name-6, inclusive. This value is used for all logical pages written for the file during a given execution of the program. If a WRITE ... ADVANCING PAGE statement is executed, or if a Page-Overflow condition occurs, the value specifies the footing area for the next logical page.

LINAGE-COUNTER Special Register

The LINAGE-COUNTER, a special register, is generated by the presence of a LINAGE clause. The value in the LINAGE-COUNTER register at any time represents the line number at which the device is positioned in the current page body. The rules governing the LINAGE-COUNTER register are as follows:

- A separate LINAGE-COUNTER register is supplied for each file described in the FILE SECTION that has a LINAGE clause.
- The data-name, LINAGE-COUNTER, can be referenced, but not modified, by PROCEDURE DIVISION statements. Because more than one LINAGE-COUNTER can exist in a program, you must qualify LINAGE-COUNTER by using a file-name when necessary.

- The LINAGE-COUNTER register is automatically modified according to the following rules during the execution of a WRITE statement to an associated file:
  - If the ADVANCING PAGE phrase of the WRITE statement is specified, the LINAGE-COUNTER register is automatically reset to 1.
  - If the ADVANCING identifier-6 or integer phrase of the WRITE statement is specified, the LINAGE-COUNTER register is incremented by integer or by the value of the data item referenced by identifier-6.
  - If the ADVANCING phrase of the WRITE statement is not specified, the LINAGE-COUNTER register is incremented by 1.
  - The value of the LINAGE-COUNTER register is automatically reset to 1 when the device is repositioned to the first line on which writing can occur for each of the succeeding logical pages.
- The value of the LINAGE-COUNTER register is automatically reset to 1 when the file is opened.

CODE-SET Clause

The CODE-SET clause specifies the character code set used to represent data on the external media.

The general format of this clause is as follows:

```
[CODE-SET IS alphabet-name]
```

Explanation of Format

If the CODE-SET clause is specified, alphabet-name specifies both the character code set used to represent data on the external media and the algorithm for converting the character codes on the external media to or from EBCDIC. This code conversion occurs during execution of an input or output operation.

If the CODE-SET clause is not specified, the native character code set (EBCDIC) is assumed for data on the external media.

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

The alphabet-name referenced by the CODE-SET clause must not specify the literal phrase. The CODE-SET clause can be specified only for files that are not on a mass-storage device.

For information about specifying an alphabet-name, refer to “SPECIAL-NAMES” in Section 5, “ENVIRONMENT DIVISION.”

Record Description

One or more record descriptions must follow each file description (FD). A record description consists of a set of data-description entries that describe the characteristics of a particular record. Each data-description entry consists of a level-number (followed by a data-name, if required) followed by a series of independent clauses, as required. A record description has a hierarchic structure; therefore, the clauses used with an entry can vary considerably, depending on whether or not the entry is followed by subordinate entries.

Data-Description Entry Formats

A data-description entry specifies the characteristics of a particular data item. The following four formats are used for the data-description entry:

Format	Use	Refer to
1	Defines a record	"Data-Description Entry for Record Structure"
2	Renames entries	"Data-Description Entry for Renaming Entries"
3	Specifies condition-names	"Data-Description Entry for Condition-Names"
4	Invokes a dictionary	Volume 2 of this manual

For More Information

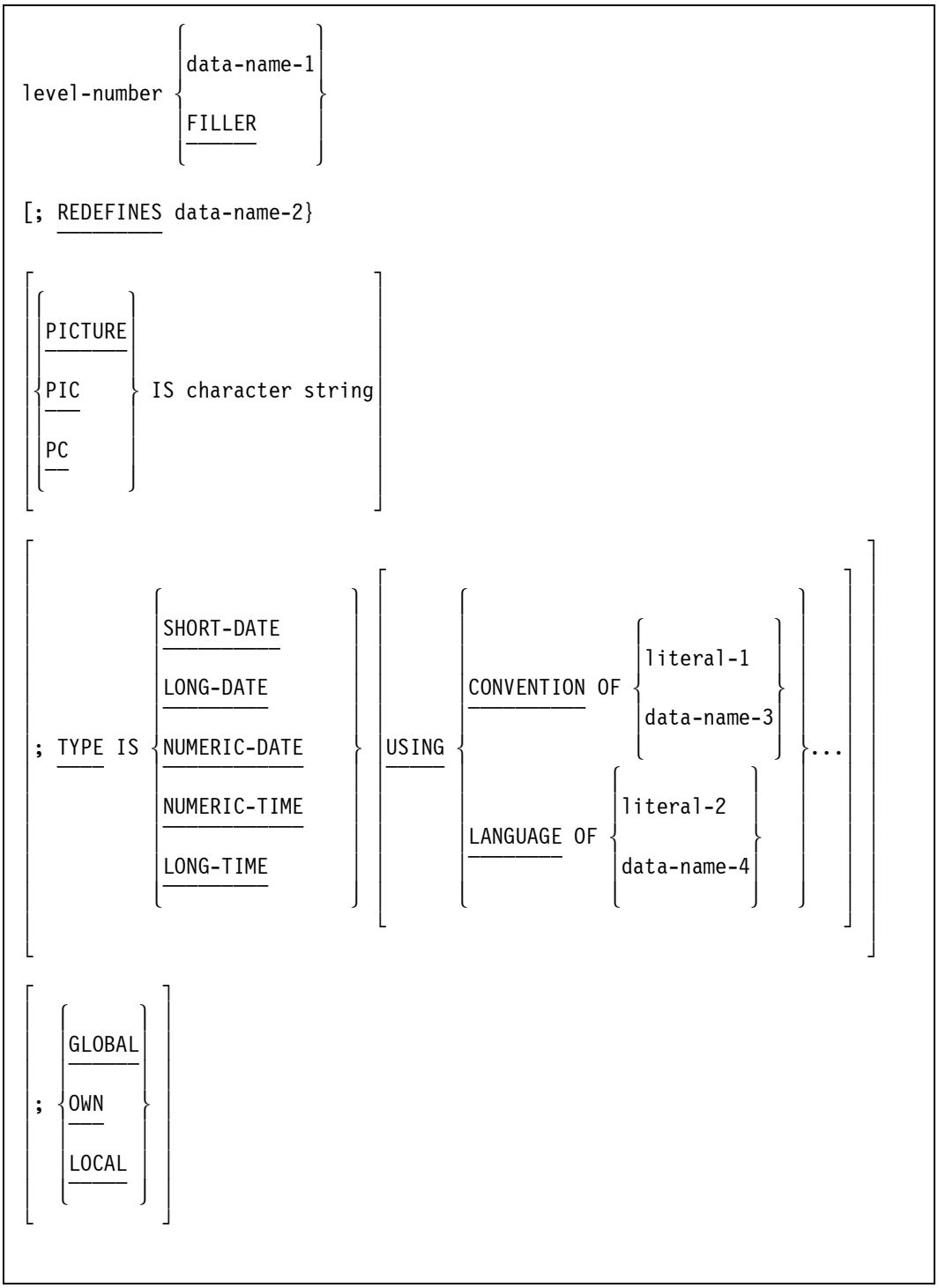
- For information about the structure of a record description, refer to "Levels" in Section 6, "Data Concepts."
- For information about the elements allowed in a record description, refer to "Data" in Section 6, "Data Concepts."

Data-Description Entry for Record Structure

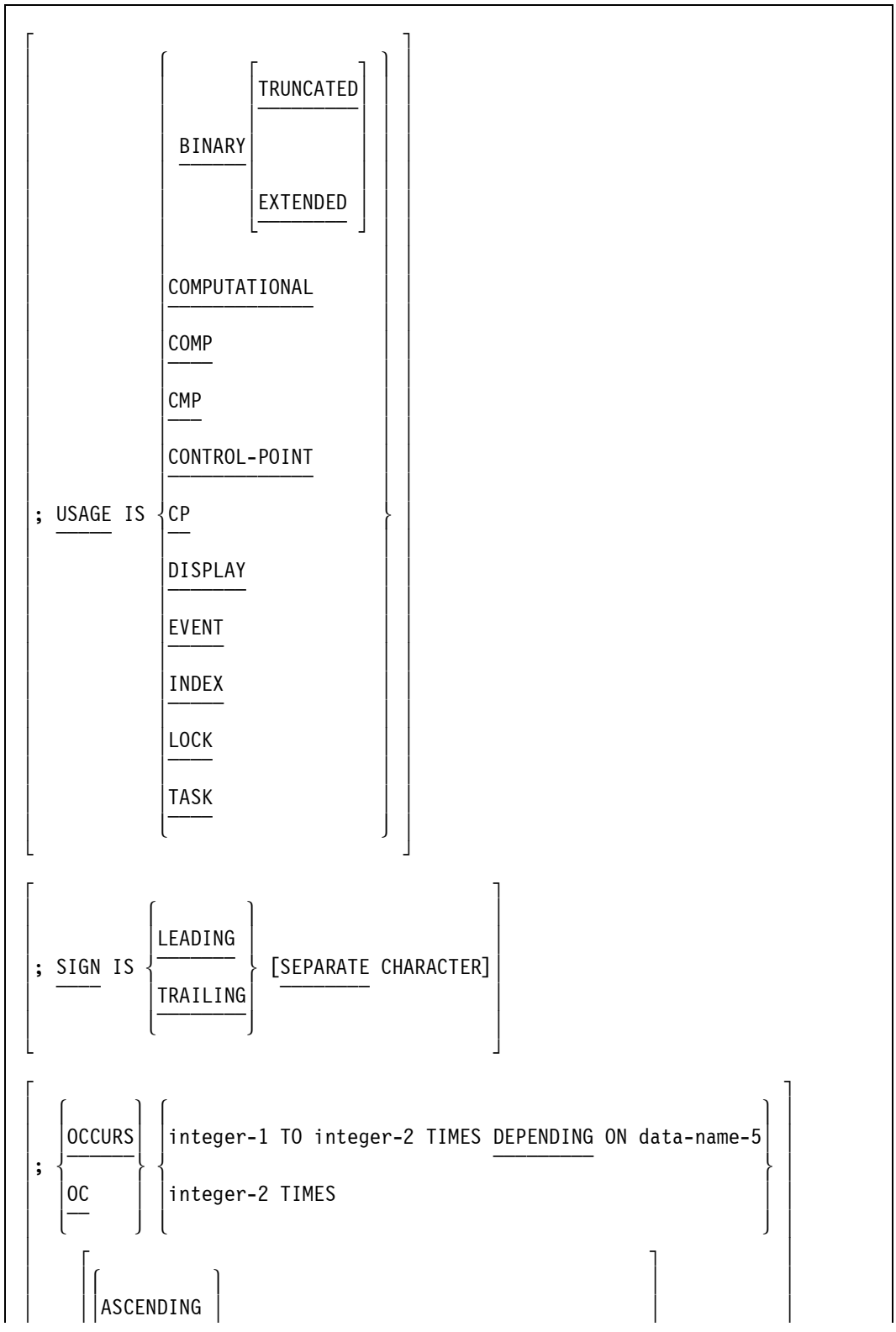
The following general format shows the complete syntax for defining a record. The optional clauses are described on the following pages.

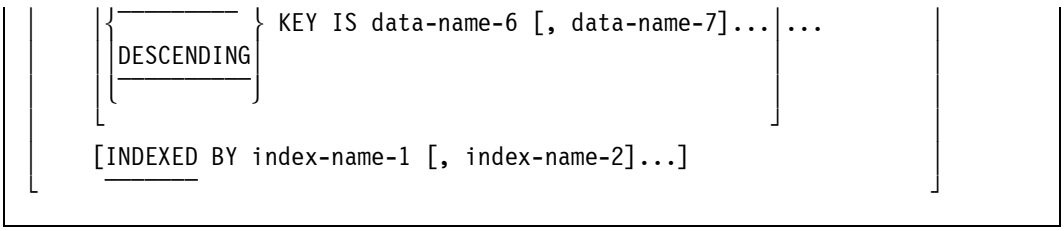
Note: *Refer to Volume 2 for information about the USER and VERSION clauses.*

Format 1: Record Structure

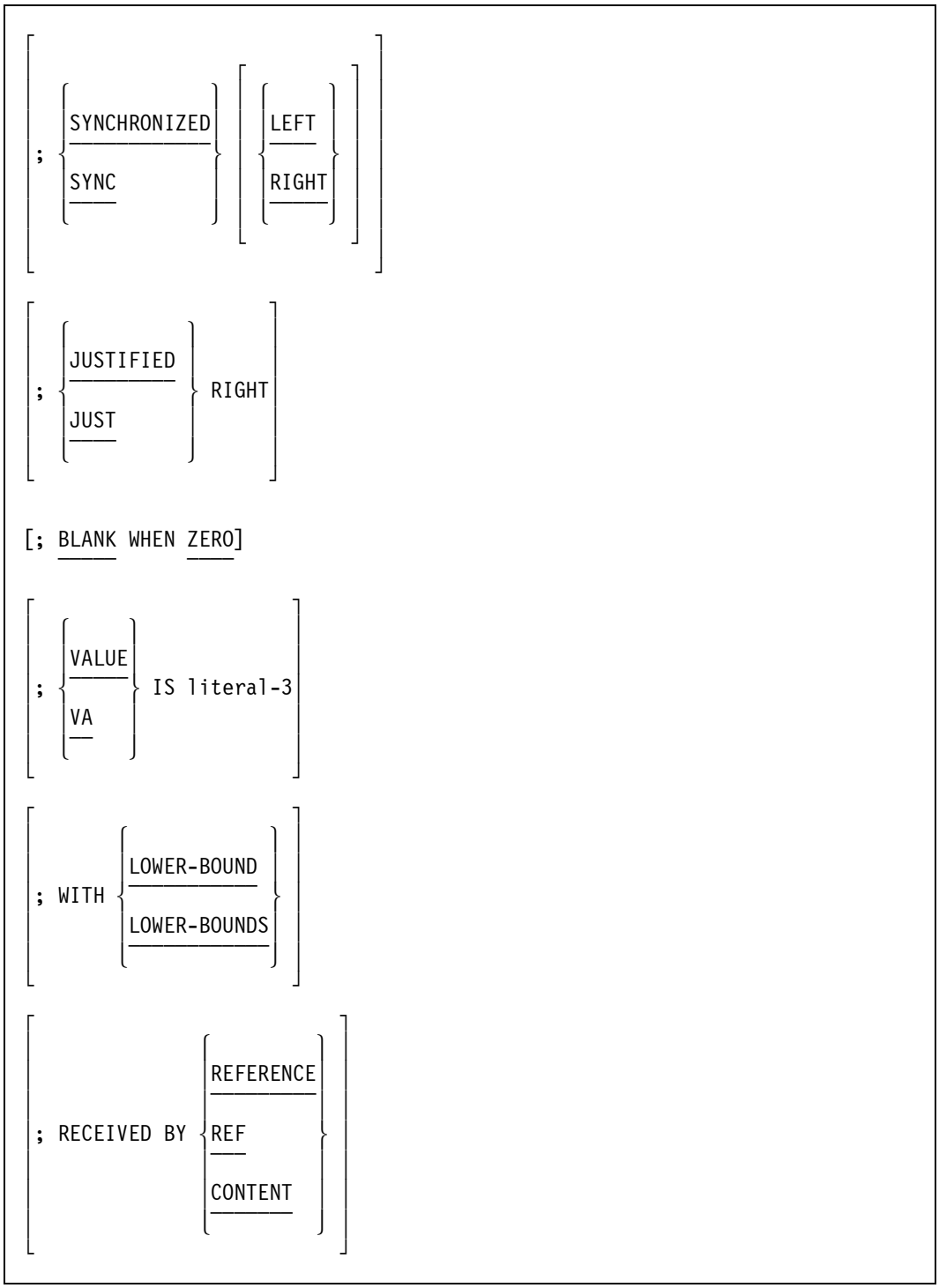


Format 1: Record Structure




{ASCENDING / DESCENDING} KEY IS data-name-6 [, data-name-7]... ...
[INDEXED BY index-name-1 [, index-name-2]...]

Format 1: Record Structure



Explanation of Format 1

The level-number can be any number from 01 through 49, or 77. Each record of a file begins with the level-number 01. This number is reserved for the record-name only,

because it is the most inclusive grouping for a record. Less inclusive groupings are given higher numbers; these numbers are not necessarily successive.

The clauses of the data-description entry can be written in any order, with the following exceptions:

- The data-name-1 or FILLER clause must immediately follow the level-number.
- The REDEFINES clause must immediately follow the data-name-1 clause.

The PICTURE clause must be specified for every elementary item except an index data item. The PICTURE clause cannot be used for an index data item.

The clauses SYNCHRONIZED, PICTURE, JUSTIFIED, and BLANK WHEN ZERO must not be specified except for an elementary data item.

Multiple 01-level entries in a given file description (FD) of the FILE SECTION represent redefinition of the same memory area.

If a file is selected using Format 5 of the SELECT statement, only record descriptions or form libraries invoked from the dictionary are allowed; record descriptions coded as usual are not permitted. If the file is not selected with Format 5 of the SELECT statement, then its record descriptions can be either coded as usual or invoked from the dictionary.

For More Information

- For more information about defining the hierarchic structure of a record, refer to "Levels" in Section 6, "Data Concepts."
- For Format 5 of the SELECT statement and for information about the rules and syntax for invoking the dictionary in order to use a previously defined data item, refer to Volume 2.

Data-Name or FILLER Clause

A data-name specifies the name of the data being described. The keyword FILLER specifies an elementary item of the logical record that cannot be referenced explicitly.

The general format is as follows:

```
┌           ┐
│data-name  │
└           ┘
┌           ┐
│FILLER     │
└           ┘
```

Explanation of Format

In the FILE, COMMUNICATION, LINKAGE, and WORKING-STORAGE sections, a data-name or the keyword FILLER must be the first word following the level-number in each data-description entry.

The keyword FILLER can be used to name an elementary item in a record. A FILLER item can never be referenced explicitly. However, the keyword FILLER can be used as a conditional variable because such use does not require explicit reference to the FILLER item, but instead requires reference to the value of the FILLER item.

BLANK WHEN ZERO Clause

This clause is used with the PICTURE clause to print spaces if the value of the data item is 0.

The general format of this clause is as follows:

```
BLANK WHEN ZERO
```

Explanation of Format

The BLANK WHEN ZERO clause can be used only for an elementary item with the PICTURE clause specified as numeric or numeric-edited.

When the BLANK WHEN ZERO clause is used, the item contains nothing but spaces if the value of the item is 0.

When the BLANK WHEN ZERO clause is used for an item that has a numeric picture, the category of the item is considered to be numeric-edited.

Example

Example 7–4 illustrates the effect of the BLANK WHEN ZERO clause.

```
  Input File                    Output File

PICTURE   Data            PICTURE          Data
Clause                    Clause

9V99       000            $.99             $.00
9V99       000            $.99 BLANK
                               WHEN ZERO
```

Example 7–4. Effect of the BLANK WHEN ZERO Clause

GLOBAL Clause (Extension to ANSI X3.23-1974 COBOL)

The GLOBAL clause allows COBOL programs compiled at lexicographic level 3 or higher to use untyped procedures, files, and certain variables in the outer block of the host program by declaring these items as global items.

The general format of this clause is as follows:

```
GLOBAL
------
```

Explanation of Format

Level-77 data items with BINARY, REAL, DOUBLE, EVENT, LOCK, or TASK usage, or 01-level items that are declared in the WORKING-STORAGE SECTION of a host program can be passed as parameters. These items can be declared global in a bound procedure by using the GLOBAL clause in the operand or item data-description entry. GLOBAL declarations are matched by name and type to the GLOBAL directory of the host. The GLOBAL clause must not be specified in the host program.

Index-names generated for a global array are not themselves global items, but are treated as if they had been described with an OWN clause. Index-names for a local array are treated as local variables.

If most or all of the variables declared in the WORKING-STORAGE SECTION need to be declared global, the compiler control option GLOBAL can be used. The compiler control option can be assigned the value TRUE throughout the compilation, although this designation affects only variables that are candidates for GLOBAL declaration and that are in the WORKING-STORAGE SECTION. The LOCAL or OWN clause can be used to override the compiler control option.

Examples

Example 7–5 shows how to declare data items to be global in the WORKING-STORAGE SECTION.

```
77 GLASTATUS GLOBAL BINARY PIC 9(11).
77 BL-EVENT GLOBAL EVENT.
01 GL-EBCRAY GLOBAL.
   03 CMP-ITM COMP PIC 9(11) OCCURS 100 INDEXED BY I.
```

Example 7–5. Coding the GLOBAL Clause

In Example 7–6, the GLOBAL compiler option declares G1, G2, and G3 to be global. L1 is declared as local because the LOCAL clause overrides the GLOBAL compiler option. J is declared with an OWN clause because it is an index-name.

```
$ SET GLOBAL

     77 G1 BINARY PIC 9(11).
     77 G2 BINARY PIC 9(11).
     77 L1 LOCAL COMP PIC 9(11).
     01 G3.
        03 FLD PIC 9(11) COMP OCCURS 10 INDEXED BY J.
```

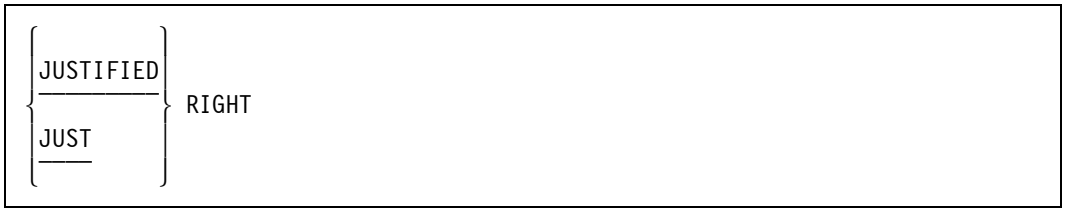
Example 7–6. Using the GLOBAL Compiler Option

JUSTIFIED Clause

The JUSTIFIED clause changes the rules for alphanumeric move operations. Normally, data that is moved is left-justified. The JUSTIFIED clause causes alphanumeric data to be right-justified in the receiving data item.

When the JUSTIFIED clause is omitted, the standard rules for data alignment in an elementary item apply.

The general format of this clause is as follows:



Explanation of Format

The JUSTIFIED clause can be specified only at the elementary item level.

JUST is an abbreviation for JUSTIFIED.

The JUSTIFIED clause cannot be specified for any data item described as numeric or for any data item for which editing is designated.

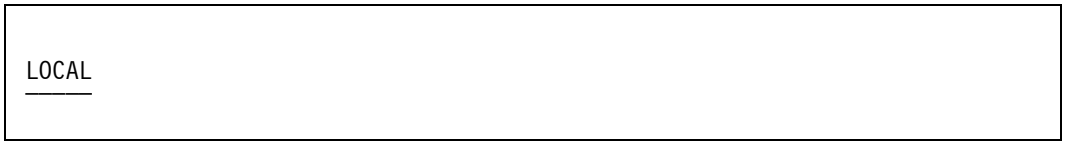
When a receiving data item is described with the JUSTIFIED clause and the sending data item is larger than the receiving data item, the leftmost characters are truncated. When the receiving data item is described with the JUSTIFIED clause and is larger than the sending data item, the data is aligned at the rightmost character position in the data item with space-fill for the leftmost character positions.

For information about data alignment, refer to Section 6, “Data Concepts.”

LOCAL Clause (Extension to ANSI X3.23-1974 COBOL)

A local data-name is referenced in the same procedure in which it is declared. Any value stored in it is lost upon exit from that procedure.

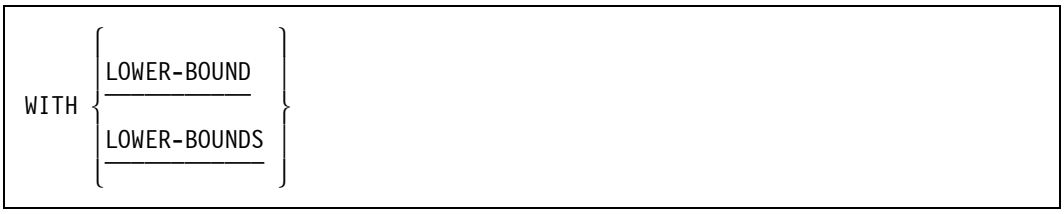
The general format of this clause is as follows:



In COBOL procedures compiled at level 3 or higher, data-names are implicitly declared as local unless the GLOBAL or the OWN clause is specified.

LOWER-BOUNDS Clause (Extension to ANSI X3.23-1974 COBOL)

The LOWER-BOUNDS clause permits bound or host COBOL74 programs to pass or receive array parameters compatible with the FORTRAN and ALGOL constructs that generate a lower-bound stack item.



Explanation of Format

This clause is used in the data description of a 01-level item in the LINKAGE SECTION (if array parameters are received) or the LOCAL-STORAGE SECTION (if array parameters are to be passed).

The LOWER-BOUNDS clause is not meaningful for 77-level items. A warning is issued if a 77-level data item is declared with the WITH LOWER-BOUNDS clause.

The LOWER-BOUNDS clause affects only bound procedures. The purpose of this clause is to declare formal parameters for binding that are compatible with FORTRAN and ALGOL. The clause must be used when communicating with FORTRAN programs or with ALGOL programs that contain formal array parameters declared with a variable lower-bound description (that is, ARRAYNAME [*]). The actual lower-bound parameter passed by a COBOL program to another program always has a value of 0.

The actual lower-bound parameter received by a COBOL program is not used in addressing the array.

Table 7–1 shows the matching of parameters between the COBOL74 and ALGOL programming languages. The <integer> variable, in ALGOL, is the specified lower-bound parameter.

Table 7–1. COBOL74 and ALGOL Parameter Matching

COBOL Parameter	Corresponding ALGOL Parameter
01 BINARY	REAL array [<integer>]
01 BINARY WITH LOWER-BOUNDS	REAL array [*]
01 COMP	EBCDIC character array [<integer>]
01 COMP WITH LOWER-BOUNDS	EBCDIC character array [*]

Table 7–1. COBOL74 and ALGOL Parameter Matching

COBOL Parameter	Corresponding ALGOL Parameter
01 DISPLAY	EBCDIC character array [<integer>]
01 DISPLAY WITH LOWER-BOUNDS	EBCDIC character array [*]
01 DOUBLE	REAL array [<integer>]
01 DOUBLE WITH LOWER-BOUNDS	REAL array [*]
01 REAL	REAL array [<integer>]
01 REAL WITH LOWER-BOUNDS	REAL array [*]

For library calls and tasking calls, the LOWER-BOUNDS clause is ignored.

For COBOL74 tasks, parameters with or without lower-bounds can be passed to COBOL74 programs and received from COBOL74 programs; the operating system handles the coercion. COBOL74 does not use the value of the lower-bound parameter in addressing the array.

When a user program passes an array parameter with a lower-bound to a COBOL74 library, the user program actually sends two parameters: a by-reference array followed by a by-value integer. The COBOL74 program must declare an extra parameter, a 77-level PIC 9(11) BINARY item, to receive the lower-bound parameter.

OCCURS Clause

The OCCURS clause defines tables and other homogeneous data items. If the OCCURS clause is used, the data-name that is the subject of this entry must be either subscripted or indexed whenever it is referenced in a statement other than the SEARCH or USE FOR DEBUGGING statement. In addition, if the subject of this entry 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 when the data-names are the objects of a REDEFINES clause.

The OCCURS clause eliminates the need for separate entries for repeated data items and supplies information required to apply subscripts or indexes.

Except for the OCCURS clause itself, all data-description clauses associated with an item that has an OCCURS clause in its description apply to each occurrence of the item described.

The OCCURS clause has the following two formats:

Format	Explanation
1	Specifies that an item occurs an exact number of times.
2	Specifies that an item occurs a variable number of times, depending on the data item referenced by data-name-1.

Format 1

The general format of this clause is as follows:

```
{OCCURS}
{OC    } integer-2 TIMES

[{ASCENDING }                                    ]
[{          } KEY IS data-name-2 [, data-name-3]...]
[{DESCENDING}                                    ]

[INDEXED BY index-name-1 [, index-name-2]...]
```

Explanation of Format 1

Format 1 of the OCCURS clause cannot be specified in a data-description entry that meets either of the two following conditions:

- The entry has a 01, 66, 77, or 88 level-number.
- The entry describes an item of variable size. The size of an item is considered variable if the data description of any subordinate item contains Format 2 of the OCCURS clause.

OCCURS or OC

OC is an abbreviation for OCCURS. (This is an extension to ANSI X3.23-1974 COBOL.)

integer-2

Integer-2 cannot exceed the maximum record size.

KEY IS

The KEY IS phrase indicates that the repeated data is arranged in ascending or descending order according to the values contained in data-name-2 and data-name-3. The data-names are listed in descending order of significance.

Data-name-2 and data-name-3 can be qualified.

INDEXED BY

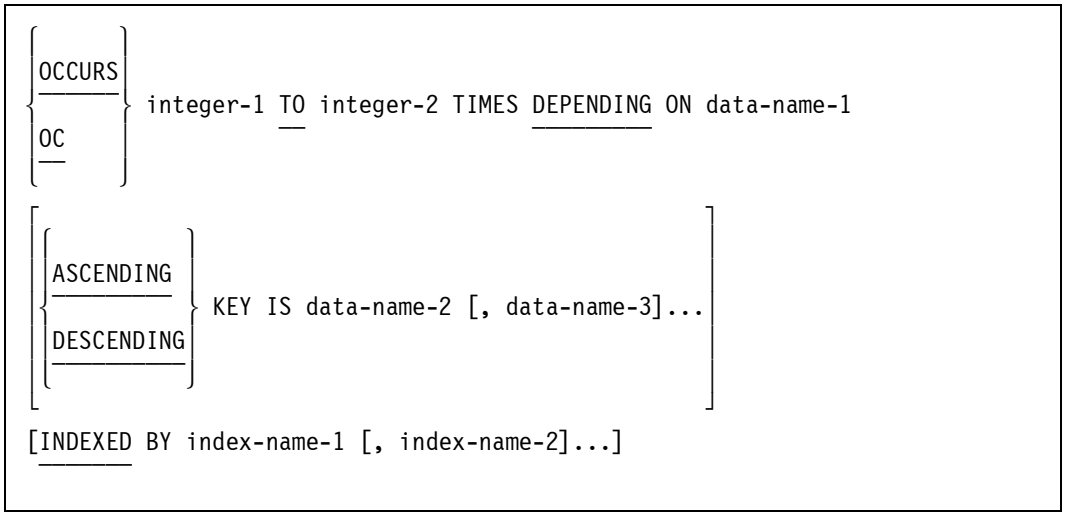
An INDEXED BY phrase is required if the subject of the entry or an entry subordinate to this entry is referenced by indexing. The index-name identified by this clause is not defined elsewhere because its allocation and format depend on the hardware, and the index-name cannot be associated with any data hierarchy because it is not a data item.

Index-name-1 and index-name-2 must be unique words in the program.

Note: When the OCCURS clause is used on a group item, the sum of the number of index-names in any associated INDEXED BY clause and the number of data-names declared subordinate to the group item cannot exceed 511.

Format 2

The general format of this clause is as follows:



Explanation of Format 2

Format 2 specifies that the subject of this entry has a variable number of occurrences. The value of integer-2 represents the maximum number of occurrences, and the value of integer-1 represents the minimum number of occurrences. The length of the subject of the entry is not variable, but the number of occurrences is variable.

A data-description entry that contains Format 2 of the OCCURS clause can be followed in that record description only by data-description entries that are subordinate to it.

integer-1 TO integer-2

The value of integer-1 must be less than the value of integer-2. Integer-1 and integer-2 cannot exceed the maximum record size. A syntax error results if the limit is exceeded.

DEPENDING ON

The value of data-name-1 is used to determine the last table element that can be referenced. When the value of data-name-1 is less than integer-2, the data items with occurrence numbers exceeding the value of data-name-1 are inaccessible. Reducing the value of the data item referenced by data-name-1 has no effect on the contents of data items with occurrence numbers that exceed the value of the data item referenced by data-name-1. (This is an extension to ANSI X3.23-1974 COBOL.)

When a table element is referenced, the value of data-name-1 must fall in the range integer-1 through integer-2, inclusive. If the value of data-name-1 is outside this range, the program ends abnormally.

Data-name-1 can be qualified.

The data description of data-name-1 must be that of an unsigned integer. Integer-1 can be 0, which is a relaxation of the ANSI-74 standard rule requiring a minimum of one occurrence. If data-name-1 takes on a value of 0 at run time, then no occurrences exist until the value of data-name-1 becomes nonzero. Any attempt to refer to an occurrence outside the current range produces an error.

The data item defined by data-name-1 must not occupy a character position in the range between the first character position defined by the data-description entry containing the OCCURS clause and the last character position defined by the record-description entry containing that OCCURS clause.

KEY IS

If data-name-2 is not the subject of this entry, then the following three conditions apply:

- All items identified by the data-names in the KEY IS phrase must be in the group item that is the subject of this entry.
- Items defined by the data-name in the KEY IS phrase must not contain an OCCURS clause.
- No entry can contain an OCCURS clause between the items identified by the data-names in the KEY IS phrase and the subject of this entry.

When a group item is referenced that has a subordinate entry that uses Format 2 of the OCCURS clause, only that part of the table area specified by the value of data-name-1 is used in the operation.

The KEY IS phrase indicates that the repeated data is arranged in ascending or descending order according to the values contained in data-name-2 and data-name-3. The data-names are listed in descending order of significance.

Data-name-2 and data-name-3 can be qualified.

INDEXED BY

An INDEXED BY phrase is required if the subject of the entry or an entry subordinate to this entry is referenced by indexing. The index-name identified by this clause is not defined elsewhere because its allocation and format depend on the hardware, and the index-name cannot be associated with any data hierarchy because it is not a data item.

Index-name-1 and index-name-2 must be unique words in the program.

OWN Clause (Extension to ANSI X3.23-1974 COBOL)

COBOL procedures compiled at level 3 or higher can declare certain variables to be OWN. These variables retain their values or states throughout repeated exits and reentries of the procedure in which they are declared.

The general format of this clause is as follows:

```
OWN
```

Any item declared in the WORKING-STORAGE SECTION can be made OWN by using the OWN clause or the compiler control option OWN.

All related index-names and copy descriptors for OWN items are also OWN; redefinitions of OWN items are implicitly OWN and need not use the OWN clause.

Use of the compiler control option OWN throughout the compilation causes all stack locations obtained in the WORKING-STORAGE SECTION to be OWN, unless overridden temporarily by a GLOBAL or LOCAL clause on an individual item.

Example

Example 7–7 shows the declaration of OWN data items in the WORKING-STORAGE SECTION.

```
77 X PIC X(10) OWN.
77 Y REDEFINES X PIC 9(10).
01 A OWN.
   03 CMP-ITEM COMP PIC 9(11) OCCURS 100 INDEXED BY J.
```

Example 7–7. Coding the OWN Clause

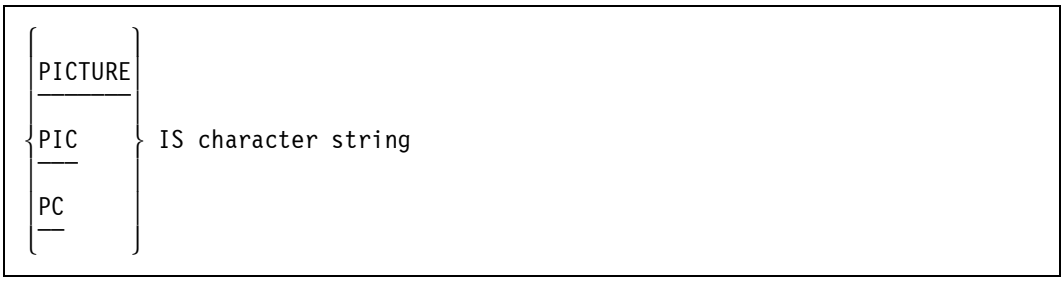
PICTURE Clause

The PICTURE clause describes the type of data item, the size of a data item, and the editing requirements of an elementary data item.

The following two methods exist for performing editing with the PICTURE clause:

- Insertion editing
- Zero-suppression and replacement editing

The general format of the PICTURE clause is as follows:



Explanation of Format

A PICTURE clause can be specified only at the elementary item level and must be specified for every elementary data item except an index data item. The PICTURE clause cannot be specified for an index data item.

A character string consists of certain allowable combinations of characters in the COBOL character set that are used as symbols. The allowable combinations determine the category of the elementary item. The maximum number of characters allowed in the character string is 30.

PIC and PC are synonyms for PICTURE. (PC is an extension to ANSI X3.23-1974 COBOL.)

The asterisk, when used as the zero-suppression symbol, and the clause BLANK WHEN ZERO cannot appear in the same entry.

Defining Data Categories

The following categories of data can be described with a PICTURE clause: alphabetic, numeric, alphanumeric, alphanumeric-edited, numeric edited, Kanji, and Kanji-edited.

Table 7–2 shows the rules for defining the seven categories of data.

Table 7–2. Defining Items with the PICTURE Clause

Data Type	Rules for Defining the Data Type
Numeric	The character string can contain only the symbols P, S, V, and 9. The number of digit positions that can be described by the character string must be from 1 to 23. If unsigned, the contents of the item (when represented in standard data format) must be a combination of the numerals 0 through 9; if signed, the item can also contain a plus sign (+), a minus sign (–), or any other representation of an operational sign.
Alphabetic	The character string of the item can contain only the symbols A and B. The contents of the item (when represented in standard data format) must be a combination of the 26 letters of the alphabet and the space from the COBOL character set.
Alphanumeric	The character string of the item is restricted to certain combinations of the symbols A, X, and 9; the item is treated as if the character string contained all Xs. A PICTURE character string that contains all As or 9s does not define an alphanumeric item. The contents of the item (when represented in standard data format) are allowable characters in the character set.
Alphanumeric-edited	The character string is restricted to certain combinations of the symbols A, B, I, X, 9, 0 (zero), and slash (/). In addition, one of two conditions must apply as follows: • The character string must contain at least one B and one X, at least one 0 (zero) and one X, or at least one slash (/) and one X. • The character string must contain at least one 0 (zero) and one A, or at least one slash and one A. The contents of the item (when represented in standard format) are allowable characters in the character set.

Table 7–2. Defining Items with the PICTURE Clause

Data Type	Rules for Defining the Data Type
Numeric-edited	The character string of the item is restricted to certain combinations of the symbols B, I, P, V, Z, 9, 0 (zero), slash (/), comma (,), period (.), plus sign (+), minus sign (–), CR, DB, asterisk (*), and dollar sign ($). The allowable combinations are determined from the order of precedence of symbols and the editing rules. In addition, both of the following conditions must apply: • The number of digit positions that can be represented in the character string must be from 1 through 23. • The character string must contain at least one B, Z, 0 (zero), slash (/), comma (,), period (.), plus sign (+), minus sign (–), CR, DB, asterisk (*), or dollar sign ($). The contents of the character positions of symbols that are allowed to represent a digit in standard data format must be one of the numerals.
Kanji	The character string can contain only the symbol X. The contents of the item (when represented in standard data format) are represented in a 2-byte (16-bit) format.
Kanji-edited	The character string of the item is restricted to certain combinations of the symbols X, B, 0 (zero), and slash (/). In addition, the character string must contain at least one B and one X, at least one 0 (zero) and one X, or at least one slash (/) and one X. The contents of the item (when represented in standard format) are represented in a 2-byte (16-bit) format.

For information about specifying an operational sign for numeric data, refer to "SIGN Clause" later in this section.

Determining the Size of the Elementary Item

The size of an elementary item is the number of character positions it occupies in standard data format. You indicate the size of an elementary item by using the number of allowable symbols that represent character positions. For example, 9999 indicates a field with four digits.

The symbols A, B, P, X, Z, 9, 0 (zero), slash (/), plus sign (+), minus sign (–), asterisk (*) or dollar sign ($) can appear more than once in a given PICTURE clause. You can specify a number of consecutive occurrences of a symbol by using an integer enclosed in parentheses after the symbol. For example, X(8) indicates eight alphanumeric characters.

Describing Elementary Items with Symbols

Elementary items are described in the PICTURE clause with the symbols shown in the following table. The following symbols can appear only once in a given PICTURE clause:

- S
- V
- . (period)
- CR
- DB

Table 7–3. Describing Elementary Items Using Symbols

Symbol	Explanation
A	Each A in the character string represents a character position that can contain only a letter of the alphabet or a space.
B	Each B in the character string represents a character position into which the space character is inserted.
I	Each I in the character string represents a character position into which the single nonblank character following the letter I in the PICTURE string is inserted. The inserted character is counted in the character size of the item. (The use of the letter I in a PICTURE string is an extension to ANSI X3.23-1974 COBOL.)
P	Each P indicates an assumed decimal scaling position and specifies the location of an assumed decimal point when the point is not within the number that appears in the data item. The scaling position character P is not counted in the size of the data item, but scaling position characters are counted in determining the maximum number of 23 digit positions in numeric-edited items or numeric items. The scaling position character P can appear only to the left or right as a continuous string of Ps in a PICTURE description. Because the scaling position character P implies an assumed decimal point to the left of the P characters if these characters are the leftmost PICTURE characters, and to the right if the P characters are the rightmost PICTURE characters, the assumed decimal point symbol V is redundant as either the leftmost or the rightmost character in such a PICTURE description. The character P and the insertion character period (.) cannot occur together in the same PICTURE character string. If, in any operation involving conversion of data from one form of internal representation to another, the data item being converted is described with the PICTURE character P, each digit position described by a P is considered to contain the value 0 and the size of the data item is considered to include the digit positions so described.

Table 7–3. Describing Elementary Items Using Symbols

Symbol	Explanation
S	The letter S in a character string indicates the presence of an operational sign in the internal representation of a numeric data item. The S must be the first (leftmost) character in the character string. The symbol S can be used in the PICTURE character string of any data item with the USAGE clause equal to DISPLAY, COMPUTATIONAL, or BINARY. The SIGN clause can be used to specify the exact representation and position of the operational sign. When an operational sign is specified for a DISPLAY data item and a SIGN clause is not specified, the sign is maintained and expected in the zone of the least significant (rightmost) character. When the data item is in the receiving field in an arithmetic statement and when the native character set is EBCDIC, the four zone bits are set to binary 1101 for negative values and to binary 1100 or 1111 for positive values. When the data item is used in an algebraic comparison or operation to supply an algebraic value, specification of the least significant zone as binary 1101 causes the value to be considered negative. Only the zone values 1100, 1101, and 1111 qualify the data item as NUMERIC if it is tested by the numeric class condition. For DISPLAY data items, the presence or absence of an operational sign has no effect on the amount of storage required to contain the data item, unless the SIGN SEPARATE clause is specified. When an operational sign is specified for a COMPUTATIONAL data item and a SIGN clause is not specified, the sign is maintained and expected as a leading, separate 4-bit character to the left of the most significant digit position. When the native character set is EBCDIC, the binary pattern of the sign character is 1101 for negative values and 1100 for positive values. Like DISPLAY data items, only these values allow the item to be considered NUMERIC in the class condition test. Unlike DISPLAY data items, the specification of an operational sign for COMPUTATIONAL data items increases by one the number of 4-bit character positions occupied by the data item in storage.
V	The letter V in a character string indicates the location of the assumed decimal point and can appear only once in a character string. The symbol V does not represent a character position and, therefore, is not counted in the size of the elementary item. When the assumed decimal point is to the right of the rightmost character in the string, the V is redundant.
X	Each letter X in the character string represents a character position that contains any allowable character from the character set.
Z	Each letter Z in a character string can represent only the leftmost leading numeric character positions that are replaced by space characters when the contents of the character positions are 0. Each symbol Z is counted in the size of the item.
9	Each numeral 9 in the character string represents a character position that contains a numeral and is counted in the size of the item.

Table 7–3. Describing Elementary Items Using Symbols

Symbol	Explanation
0	Each numeral zero (0) in the character string represents a character position in which the numeral 0 is inserted. The numeral 0 is counted in the size of the item.
/ (slash)	Each slash (/) in the character string represents a character position into which the slash character is inserted. The slash is counted in the size of the item.
, (comma)	Each comma (,) in the character string represents a character position into which the comma character is inserted. This character position is counted in the size of the item. The comma must not be the last character in the PICTURE character string.
. (period)	When the period (.) appears in the character string, it represents the decimal point for alignment purposes and also represents a character position into which the period character is inserted. The period is counted in the size of the item. For a given program, the functions of the period and comma are exchanged if the clause DECIMAL-POINT IS COMMA is stated in the SPECIAL-NAMES paragraph. In such an exchange, the rules for the period apply to the comma, and the rules for the comma apply to the period wherever these characters appear in a PICTURE clause. A period immediately followed by a nonblank character is considered to be an insertion character, even if the nonblank character is not part of the PICTURE character string. For a period in the last character position of the PICTURE character string to be treated as an insertion character, it must be immediately followed by another period. The second period ends the data-description entry. Thus, a PICTURE clause must be the last clause in the data-description entry if it has a period as an insertion character in the last character position of the PICTURE character string.
+, –, CR, DB	These symbols are editing sign-control symbols. When used, these symbols represent the character position into which the editing sign-control symbol is placed. The symbols are mutually exclusive in any character string, and each character used in the symbol is counted in determining the size of the data item.
* (asterisk)	Each asterisk (*) in the character string represents a leading numeric character position into which an asterisk is placed when the content of that position is 0 (zero). Each asterisk is counted in the size of the item. Try to avoid the use of an asterisk in a PICTURE character string for purposes other than as a leading-zero-replacement mechanism. Although this use of an asterisk may produce expected results under certain circumstances, it is not a supported feature.
$ (dollar sign)	The dollar sign ($) in the character string represents a character position into which a currency symbol is to be placed. The currency symbol in a character string is represented by either the dollar sign or the single character specified in the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. The currency symbol is counted in the size of the item.

Insertion Editing

You can use the following four types of insertion editing:

- Simple insertion editing
- Special insertion editing
- Fixed insertion editing
- Floating insertion editing

Zero-suppression and replacement editing is described later in this section.

Simple Insertion Editing

Simple insertion editing uses the space character (B), slash (/), zero (0), and comma (,) as insertion characters.

There are two mechanisms for using an arbitrary nonblank character as a simple insertion character.

When the STRICTPICTURE compiler control option is set, only the previously mentioned characters can be used as simple insertion characters in a PICTURE string.

When the STRICTPICTURE compiler control option is reset, any nonblank character following the letter I in the PICTURE string is treated as a simple insertion character. This capability is a Unisys ClearPath MCP extension to ANSI X3.23-1974 COBOL.

Manual insertion ("I-insertion") of a B character in the PICTURE character-string results in the insertion of a B character in the output. In contrast, the use of the B character as an editing character in the PICTURE character-string (not preceded by "I") results in the insertion of a blank.

The precedence rules associated with any Manual Insertion Editing character in a PICTURE character-string are the same as those for the simple insertion editing character 0 (zero).

You might want to use manual insertion editing to format telephone numbers, dates, employee numbers, and so on.

The insertion characters are counted in the size of the item and represent the position in the item into which the character is inserted. The following table shows examples of valid PICTURE clauses using simple insertion editing.

Table 7–4. Simple Insertion Editing Examples

Picture	Data	Actual Representation	
		STRICTPICTURE Set	STRICTPICTURE Reset
9//09	47	4//07	4//07
9RQ09	47	**	**
9IRIQ09	47	**	4RQ07
/900/	5	/500/	/500/
I(900I)	5	**	(500)
9/09/	88	8/08/	8/08/
9I)0I3I)	88	**	8)03)
9(03)	88	088	088
99/99/99	123456	12/34/56	12/34/56
99I-99I-99	123456	**	12-34-56
99BBB99	1234	12bbb34	12bbb34
99BIBIB99	1234	**	12bBB34
99BIBB99	1234	**	12bBb34

Note: *The double asterisks (**) indicate a syntax error. The "b" character in the actual representation columns of the table represents a single space character.*

Special Insertion Editing

Special insertion editing uses the period (.) as the insertion character. Besides being an insertion character, the period represents the decimal point for alignment purposes. The insertion character used for the actual decimal point is counted in the size of the item. You cannot use both the assumed decimal point symbol (V) and the actual decimal point symbol (.) in the same PICTURE character string. Special insertion editing results in the insertion character appearing in the item in the same position as is shown in the character string.

The following table illustrates the use of special insertion editing.

Table 7–5. Special Insertion Editing Example

Picture	Data	Actual Representation
9.9	470	4.70
99.9	1257	12.57
9.9	38	3.8

Fixed Insertion Editing

Fixed insertion editing uses the dollar sign ($) and the editing sign-control symbols, the plus sign (+), the minus sign (–), the credit symbol (CR), and the debit symbol (DB) as the insertion characters. Only one dollar sign and only one of the editing sign-control symbols can be used in any PICTURE character string.

When the symbol CR or DB is used, it represents two character positions in determining the size of the item, and the symbol must represent the rightmost character position that is counted in the size of the item. The plus and minus sign symbols, when used, must be in either the leftmost or the rightmost character position to be counted in the size of the item. The dollar sign must be the leftmost character position to be counted in the size of the item, except that it can be preceded by either a plus or a minus sign.

Fixed insertion editing results in the insertion character occupying the same character position in the edited item as it occupied in the PICTURE character string. The results produced by editing sign-control symbols depend on the value of the data item, as shown in the following table. For example, if the edit symbol is the minus sign and the data item is negative, the data item has a minus sign. However, if the edit symbol is the minus sign and the data item is positive or 0 (zero), the data item has a space.

Table 7–6. Data Item Values and Results of Editing Sign Control Symbols

Editing Symbol in PICTURE Character String	Results If Data Item Is Positive or Zero	Results If Data Item Is Negative
+	+	–
–	Space	–
CR	2 spaces	CR
DB	2 spaces	DB

Floating Insertion Editing

Floating insertion editing uses the dollar sign ($) and the two editing sign-control symbols, the plus sign (+) and the minus sign (–), as insertion characters. These insertion characters are mutually exclusive in the same PICTURE character string.

Floating insertion editing is indicated in a PICTURE character string by using a string of at least two of the same floating insertion characters. This string of floating insertion characters can contain any of the fixed insertion symbols or have fixed insertion characters immediately to the right of the string. These insertion characters are part of the floating string.

The leftmost character of the floating insertion string represents the leftmost symbol in the data item. The rightmost character of the floating string represents the rightmost limit of the floating symbols in the data item.

The second floating character from the left represents the leftmost limit of the numeric data that can be stored in the data item. Nonzero numeric data can replace all characters at, or to the right of, this limit.

In a PICTURE character string, floating insertion editing can be represented in two ways:

- Any or all of the leading numeric character positions to the left of the decimal point can be represented by insertion characters.
- All numeric character positions in the PICTURE character string can be represented by insertion characters.

If the insertion characters are present only to the left of the decimal point in the PICTURE character string, a single floating insertion character is placed in the character position immediately preceding either the decimal point or the first nonzero digit in the data represented by the insertion symbol string, whichever is farther to the left in the PICTURE character string. The character positions preceding the insertion character are replaced with spaces.

If all numeric character positions in the PICTURE character string are represented by the insertion character, the result depends on the value of the data. If the value is 0 (zero) the entire data item contains spaces. If the value is not 0, the result is the same as when the insertion character is present only to the left of the decimal point.

To avoid truncation, the minimum size of the PICTURE character string for the receiving data item must be the number of characters in the sending data item plus the number of nonfloating insertion characters being edited into the receiving data item plus one for the floating insertion character.

The following table illustrates floating insertion editing.

Table 7–7. Floating Insertion Editing Examples

Editing Symbol in PICTURE Character String	Data Item	Actual Representation
$$,$$$	1234	$1,234
–(5)	0012	–12
–(5)	0123	–123
+++99	1234	+1234
+++99	001	+01

Zero-Suppression and Replacement Editing

Zero-suppression editing can replace zeros either with spaces or with asterisks.

The suppression of leading zeros in numeric character positions is indicated by the use of the symbol Z or the asterisk (*) as the suppression symbol in a PICTURE character string. These symbols are mutually exclusive in any PICTURE character string. Each suppression symbol is counted in determining the size of the item. If the letter Z is used, the replacement character is the space; if the asterisk is used, the replacement character is the asterisk.

Zero-suppression and replacement is indicated in a PICTURE character string by using a string of one or more allowable symbols to represent leading numeric character positions that are replaced when each associated character position in the data contains a zero. Any simple insertion characters embedded in the string of symbols or to the immediate right of this string are part of the string.

In a PICTURE character string, zero-suppression can be represented in two ways:

- Any or all leading numeric character positions to the left of the decimal point can be represented by suppression symbols.
- All numeric character positions in the PICTURE character string can be represented by suppression symbols.

If the suppression symbols appear only to the left of the decimal point, any leading zero in the data is replaced by the replacement character. Suppression ends at the first nonzero digit in the data represented by the suppression symbol string or at the decimal point, whichever is encountered first.

If all numeric character positions in the PICTURE character string are represented by suppression symbols and the value of the data is not 0 (zero), the result is the same as if the suppression characters were located only to the left of the decimal point. If the value of the data is the number 0 (zero) and the suppression symbol is Z, the entire data item consists of spaces. If the value of the data is 0 (zero) and the suppression symbol is an asterisk, then the data item consists of all asterisks except for the actual decimal point.

The asterisk, when used as the zero-suppression symbol, cannot appear in the same entry as a BLANK WHEN ZERO statement.

The following table illustrates zero-suppression and replacement editing.

Table 7–8. Zero-Suppression and Replacement Editing Examples

Picture	Data	Actual Representation
ZZZ	123	123
ZZ9	001	1
***	001	**1
Z99	012	12

Editing Methods and Data Categories

The type of editing that can be performed on an item depends on the category to which the item belongs. The following table specifies the type of editing that can be performed on a given item category.

Table 7–9. Data Categories and Editing Methods Allowed

Category	Type of Editing Allowed
Alphabetic	Simple insertion using the space character (B)
Numeric	None
Alphanumeric	None
Alphanumeric-edited	Simple insertion using the space character (B), the number 0 (zero), and the slash (/)
Numeric-edited	All (subject to the following note)
Kanji	None
Kanji-edited	Simple insertion using the space character (B), the number 0 (zero), and the slash (/)

Note: *Floating insertion editing and zero-suppression and replacement editing are mutually exclusive in a PICTURE clause. Only one type of replacement can be used with zero-suppression in a PICTURE clause.*

Editing Application of the PICTURE Clause

The following table provides various examples of the editing function of the PICTURE clause.

Table 7–10. Editing Application of the PICTURE Clause

Sending Area		Receiving Area	
PICTURE Clause	**Data**	**Editing PICTURE Clause**	**Edited Data**
9(5)	12345	$ZZ,ZZ9.99	$12,345.00
V9(5)	12345	$$$,$$9.99	$0.12
V9(5)	12345	$ZZ,ZZ9.99	$ 0.12
9(5)	00000	$$$,$$9.99	$0.00
9(3)V99	12345	$ZZ,ZZ9.99	$ 123.45
9(5)	00000	$$$,$$$.$$	
9(5)	01234	$$$,$$$.$$	$1,234.00
9(5)	00000	$**,***.**	*******.**
9(5)	00123	$**,***.**	$***123.00
9(3)V99	00012	$ZZ,ZZ9.99	$ 0.12
9(3)V99	12345	$$$,$$9.99	$123.45
9(3)V99	00001	$ZZ,ZZZ.99	$ .01
9(5)	12345	$$$,$$9.99	$12,345.00
9(5)	00000	$ZZ,ZZZ.ZZ	
9(3)V99	00001	$$$,$$$.$$	$.01
S9(5)	(+) 12345	ZZZZ9.99+	12345.00+
S9(5)	(–) 00123	–99999.99	–00123.00
9(3)V99	12345	999.00	123.00
S9(5)	(–) 12345	ZZZZ9.99–	12345.00–
S9(5)	(+) 12345	ZZZZ9.99–	12345.00
9(5)	12345	BBB99.99	45.00
S9(5)V	(–) 12345	–ZZZZ9.99	–12345.00
S9(5)	(–) 12345	$$$$$$.99CR	$12345.00CR
S99V9(3)	(–) 12345	–.99	–12.34
S9(5)	(+) 12345	$$$$$$.99CR	$12345.00
9(3)V99	12345	999.BB	123.
9(5)	12345	00999.00	00345.00

Understanding Precedence Rules

The chart in Figure 7–1 shows the order of precedence when characters are used as symbols in a character string. An X at an intersection indicates that the symbol or symbols at the top of the column can precede the symbol or symbols left of the row in a character string. Arguments appearing in braces indicate that the symbols are mutually exclusive.

At least one of the symbols A, X, Z, 9, or asterisk (*), or at least two of the symbols, plus sign (+), minus sign (–), or dollar sign ($), must be present in a PICTURE string.

Nonfloating insertion symbols plus sign (+), minus sign (–), and floating insertion symbols Z, asterisk (*), plus sign (+), minus sign (–) and dollar sign ($), and another symbol P appear twice in Figure 7–1. The leftmost column and uppermost row for each symbol represent the use of the symbol to the left of the decimal point position; the second appearance of the symbol in the chart represents its use to the right of the decimal point position.

First Symbol / Second Symbol		Nonfloating Insertion Symbols									Floating Insertion Symbols						Other Symbols					
		B	0	/	,	.	+ −	+ −	CR DB	$	Z *	Z *	+ −	+ −	$	$	9	A X	S	V	P	P
Nonfloating Insertion Symbols	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
	$						X															
Floating Insertion Symbols	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
	$	X	X	X	X		X								X							
	$	X	X	X	X	X	X								X	X				X		X
Other Symbols	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

Figure 7–1. PICTURE Character Precedence Chart

RECEIVED Clause (Extension to ANSI X3.23-1974 COBOL)

This clause identifies those items that are received as parameters by name or by value from another procedure, or items that are to be passed to another procedure by name.

```
RECEIVED BY { REFERENCE }
            { REF       }
            { CONTENT   }
```

Explanation of Format

The RECEIVED clause can appear only on a 77-level data item in the LOCAL-STORAGE, LINKAGE, or WORKING-STORAGE SECTION.

When you do not specify the RECEIVED BY clause, all items and files are received by reference to bound procedures except for 77-level parameters with the following usage: BINARY, DOUBLE, or REAL. For bound procedures, these parameters are received by content. However, you can declare these parameters to be received by reference to allow passing by reference. For tasking calls, 77-level parameters with BINARY, DOUBLE, or REAL usage can be declared as received by content to allow passing by value.

A data-description entry containing a RECEIVED clause must not contain a VALUE clause.

RECEIVED BY REFERENCE

This clause allows two or more procedures to share an item. Any reference to the identifier in one of the procedures that shares the identifier specifies the same common data area as the other procedures. REF is synonymous with REFERENCE.

RECEIVED BY REFERENCE 77-level items with BINARY, DOUBLE, or REAL usage are a special case of parameter passing. Parameters declared in this way refer to data declared in the hardware stack. All other RECEIVED BY REFERENCE parameters address items within an array. This special case of the RECEIVED BY REFERENCE clause allows COBOL74 programs to communicate with ALGOL programs and other COBOL programs that pass stack references instead of array references.

RECEIVED BY CONTENT

This clause identifies parameters passed by value. The current value of the identifier is received by procedure. Another procedure can change the value of that data-name, but the change merely affects the copy of the item for that procedure. Likewise, the receiving procedure can make changes to data-names that do not affect the original item.

The RECEIVED BY CONTENT clause cannot appear with any item that has usage described as TASK (CONTROL-POINT), EVENT, or LOCK or with any item described at the 01-level.

If an item declared as RECEIVED BY CONTENT is referred to in a PROCEDURE DIVISION USING declaration, the program cannot be used as a library and cannot be the subject of an Inter-Program Communication (IPC) *CALL* statement. The program can be used as a bound procedure, or it can be called from the Work Flow Language (WFL).

REDEFINES Clause

The REDEFINES clause allows the same computer storage area to be described by different data-description entries. This clause redefines the storage area, not the data items occupying the area.

```
level-number data-name-1; REDEFINES data-name-2
```

Note: *Level-number, data-name-1, and the semicolon (;) are shown for clarity. Level-number and data-name-1 are not part of the REDEFINES clause.*

Explanation of Format

The REDEFINES clause, when specified, must immediately follow data-name-1. The level-numbers of data-name-1 and data-name-2 must be identical but must not be 66 or 88.

The clause must not be used in 01-level entries in the FILE SECTION or COMMUNICATION SECTION. The REDEFINES clause can be used at the 01-level in the WORKING-STORAGE SECTION.

The data-description entry for data-name-2 cannot contain a REDEFINES clause. However, data-name-2 can be subordinate to an entry that contains a REDEFINES clause. In addition, the data-description entry for data-name-2 cannot contain an OCCURS clause; however, data-name-2 can be subordinate to an item that has an OCCURS clause in its data-description entry. In this case, the reference to data-name-2 in the REDEFINES clause cannot be subscripted or indexed. Neither the original definition nor the redefinition can include an item of variable size as defined in the OCCURS clause.

No entry with a level-number numerically lower than the level-number of data-name-2 and data-name-1 can occur between the data-description entries of data-name-2 and data-name-1.

Redefinition starts at data-name-2 and ends when a level-number less than or equal to that of data-name-2 is encountered.

Multiple redefinitions of the same character positions are permitted. The entries giving the new descriptions of the character positions must follow the entries for the area being redefined, without intervening entries that define new character positions. Multiple redefinitions of the same character positions must all use the data-name of the entry that originally defined the area.

The entries that give the new description of the character positions must not contain any VALUE clauses except in condition-name entries.

Multiple 01-level entries subordinate to any given level indicator represent implicit redefinitions of the same area.

The following paragraphs refer to extensions to ANSI X3.23-1974 COBOL.

The REDEFINES clause specifies the redefinition of a storage area, not the data items occupying the area. Therefore, the usage of data-name-1 need not be the same as the usage of data-name-2, except that DISPLAY or group data items cannot redefine elementary COMPUTATIONAL or INDEX data items that do not begin on a byte boundary. When redefinition occurs at a level other than the 01-level, the amount of storage allocated for data-name-2 must be the same as the amount of storage implied by the declared size and usage of data-name-1, with the following exceptions:

- A DISPLAY or group data item can redefine an elementary COMPUTATIONAL data item that begins, but does not end, on a byte boundary if the difference in size results from the generation of a 4-bit filler so that the redefining item ends on a byte boundary.
- A DISPLAY or group item can be redefined by an elementary COMPUTATIONAL data item, although the actual size (including sign position, if described) is one 4-bit character less than the number of 4-bit characters in the storage area. The redefining item is aligned to begin on a byte boundary and end at the middle of the last byte of storage.

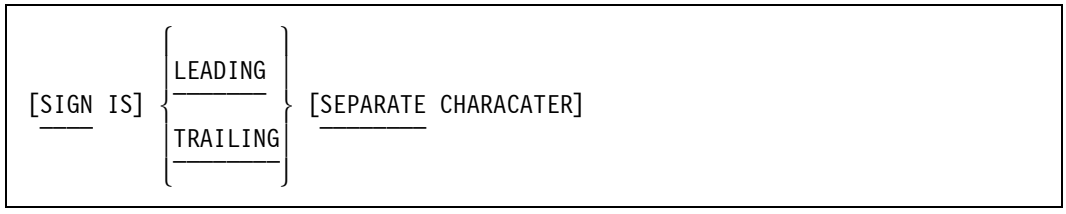
For More Information

Refer to Volume 2 of this manual for information about using the REDEFINES clause with form libraries.

SIGN Clause

The SIGN clause specifies the position and mode of representation of the operational sign when these properties must be described explicitly.

The general format of this clause is as follows:



Explanation of Format

SIGN

The optional SIGN clause specifies the position and mode of representation of the operational sign for the numeric data-description entry to which the clause applies or for each numeric data-description entry subordinate to the group to which the group applies. The SIGN clause applies only to numeric data-description entries with the character S in the PICTURE clause. The letter S indicates the presence of, but not the representation or position of, the operational sign.

A numeric data-description entry with an S in the PICTURE clause, but to which no optional SIGN clause applies, has an operational sign that is positioned and represented according to the standard default position and representation of operational signs.

Every numeric data-description entry with the character S in the PICTURE clause is a signed, numeric data-description entry. If a SIGN clause applies to such an entry and conversion is necessary for computation or comparisons, conversion takes place automatically.

The SIGN clause can be specified only for a numeric data-description entry with the character S in the PICTURE clause or for a group item containing at least one such numeric data-description entry.

The numeric data-description entries to which the SIGN clause applies must be described as USAGE IS DISPLAY or USAGE IS COMPUTATIONAL. (Use of the SIGN clause with USAGE IS COMPUTATIONAL is an extension to ANSI X3.23-1974 COBOL.)

At most, one SIGN clause can apply to any given numeric data-description entry.

If a SIGN clause without a SEPARATE CHARACTER phrase applies to a numeric data-description entry, then the following rules apply:

- When the data-item usage is DISPLAY, the operational sign is maintained and expected as binary number 1100 or 1101 in the zone of the leading or trailing character and does not cause additional storage to be allocated for the data item.
- If the data-item usage is COMPUTATIONAL, the operational sign is maintained and expected as binary number 1100 or 1101 leading or trailing 4-bit character. This sign increases by one 4-bit character the amount of storage allocated for the data item, in addition to that storage allocated for an unsigned COMPUTATIONAL data item. The presence or absence of the SEPARATE CHARACTER phrase has no effect on the position or representation of the operational sign for COMPUTATIONAL data items.

SEPARATE CHARACTER

If the CODE-SET clause is specified, any signed numeric data-description entries associated with that file must be described with the SIGN IS SEPARATE clause.

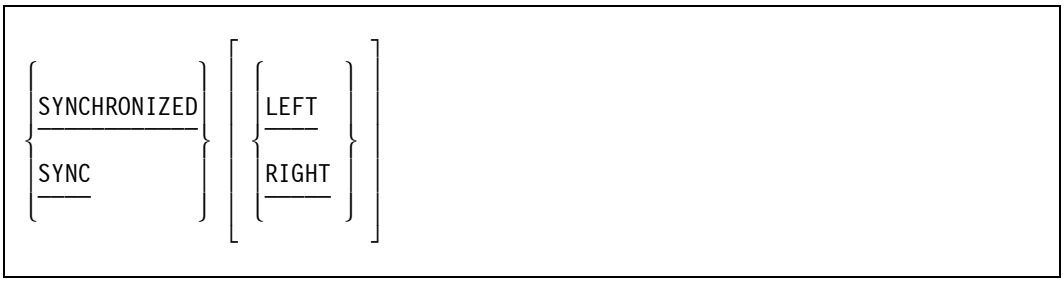
If a SIGN clause with a SEPARATE CHARACTER phrase applies to a numeric data-description entry, then the following rules apply:

- If data-item usage is DISPLAY, the operational sign is maintained and expected as a leading or trailing character separate from, and in addition to, the numeric character positions. The operational sign for negative values is the minus sign (–) and for nonnegative values is the plus sign (+).
- When the data item usage is COMPUTATIONAL, the operational sign is maintained and expected as a binary 1100 for positive values or a binary 1101 for negative values in the zone of the leading or trailing character. The sign increases by one 4-bit character the amount of storage allocated for an unsigned COMPUTATIONAL data item. The binary number 1111 is also allowed as a positive value because any combination that is neither 1101 nor a digit is interpreted by the hardware as positive. The operators produce 1100 for a positive sign and 1101 for a negative sign. The SEPARATE CHARACTER phrase does not affect the position or representation of the operational sign for COMPUTATIONAL data items.

SYNCHRONIZED Clause

The SYNCHRONIZED clause specifies the alignment of an elementary item on the natural boundaries of the computer memory.

The general format of this clause is as follows:



Explanation of Format

This clause can appear only with an elementary item.

SYNC is an abbreviation for SYNCHRONIZED.

This clause cannot appear with items of type INDEX, TASK, EVENT, or LOCK.

If the subject data item is of type COMPUTATIONAL, it is aligned on a byte boundary. If the data item is a BINARY type, it is aligned on a word boundary. If the previous data item did not end on a byte (or word) boundary, an implicit FILLER keyword is generated. This unused FILLER keyword is included in the size of any group item or items to which the elementary item belongs.

The RIGHT or LEFT option following SYNCHRONIZED is treated as a comment entry.

Whenever a SYNCHRONIZED item is referenced in the source program, the original size of the item, as shown in the PICTURE clause, is used in determining any action that depends on size, such as justification, truncation, or overflow.

If the data description of an item contains the SYNCHRONIZED clause and an operational sign, the sign of the item appears in the normal operational sign position, regardless of whether the SYNCHRONIZED LEFT or SYNCHRONIZED RIGHT clause is used with the item.

BINARY, DOUBLE, and REAL data items subordinate to a data-description entry containing an OCCURS clause are not synchronized.

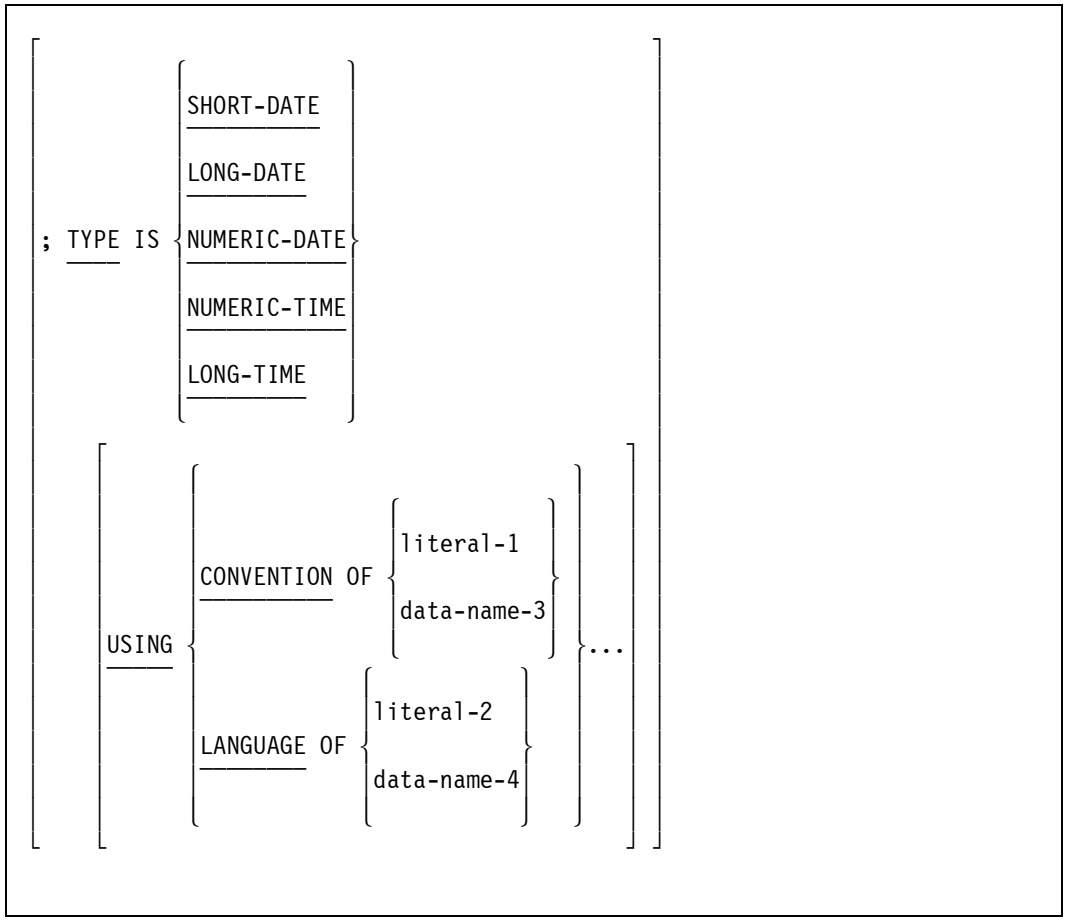
In all other cases, when the SYNCHRONIZED clause is specified in a data-description entry of a data item that also contains an OCCURS clause, or in a data-description entry of a data item subordinate to a data-description entry containing an OCCURS clause, the following rules apply:

- Each occurrence of the data item is synchronized.
- Any implicit FILLER keyword generated for other data items in that same table is generated for each occurrence of those data items.

TYPE Clause (Extension to ANSI X3.23-1974 COBOL)

The TYPE clause provides automatic date and time editing based on the CONVENTION option, LANGUAGE option, or CONVENTION and LANGUAGE options you specify. This clause can be used only for internationalization purposes.

The general format of the TYPE clause is as follows:



Explanation of Format

Data items can be declared to be one of the following date or time types:

Type	Example
SHORT-DATE	Fri, Aug 31, 1990
LONG-DATE	Friday, August 31, 1990
NUMERIC-DATE	08/31/1990
NUMERIC-TIME	13:37:20
LONG-TIME	14:37:20.2200

Data items can also be declared with an associated LANGUAGE or CONVENTION option.

Each convention has a specified format for the five date or time data items. The program formats an item declared to be one of the five date or time types according to the predefined format of the specified convention. For the SHORT-DATE, LONG-DATE, and LONG-TIME options, the specified language is also used in formatting the output. If the convention or language is not specified, the system determines the language, the convention, or the language and convention to be used based on system-defined hierarchy.

The only clauses that can be used with the TYPE clause are the PICTURE clause and the USAGE clause. If the USAGE clause is specified, it can only designate USAGE IS DISPLAY. The data type of an item with a TYPE clause is implicitly alphanumeric (without editing).

Whenever a data item with a TYPE clause is used as a destination, the information in that item is alphanumeric regardless of the contents of the PICTURE clause. For that reason, the results of retrieving the information contained in such a data item are unpredictable unless the PICTURE clause for that item specifies that the item is alphanumeric. In such cases, the compiler issues a syntax warning when it encounters this conflict. The syntax warning is best avoided by ensuring that any item that has a TYPE clause also has a PICTURE clause containing only instantiations of the character X sufficient to describe an item of the appropriate length.

The total length of the data item must be greater than or equal to the length required by the format of the specified convention. If the length of a data item is shorter than the required length, the compiler issues a truncation warning message.

Example

Example 7–8 shows coding of the TYPE clause. NUM-DATE-ITEM is declared to be of type NUMERIC-DATE, and is formatted using the ASERIESNATIVE convention. LONG-DATE-ITEM has data formatted according to the convention and language determined by the system hierarchy. LONG-TIME-ITEM is declared to be of type LONG-TIME, and is formatted using the UNITEDKINGDOM1 convention and the ENGLISH language.

```
01  NUM-DATE-ITEM       PIC X(10)  TYPE IS NUMERIC-DATE
                                    USING CONVENTION OF "ASERIESNATIVE".
01  LONG-DATE-ITEM      PIC X(29) TYPE IS LONG-DATE.
01  LONG-TIME-ITEM      PIC X(20) TYPE IS LONG-TIME
                                    USING CONVENTION OF "UNITEDKINGDOM1"
                                          LANGUAGE OF "ENGLISH".
```

Example 7–8. Coding the TYPE Clause

USAGE Clause

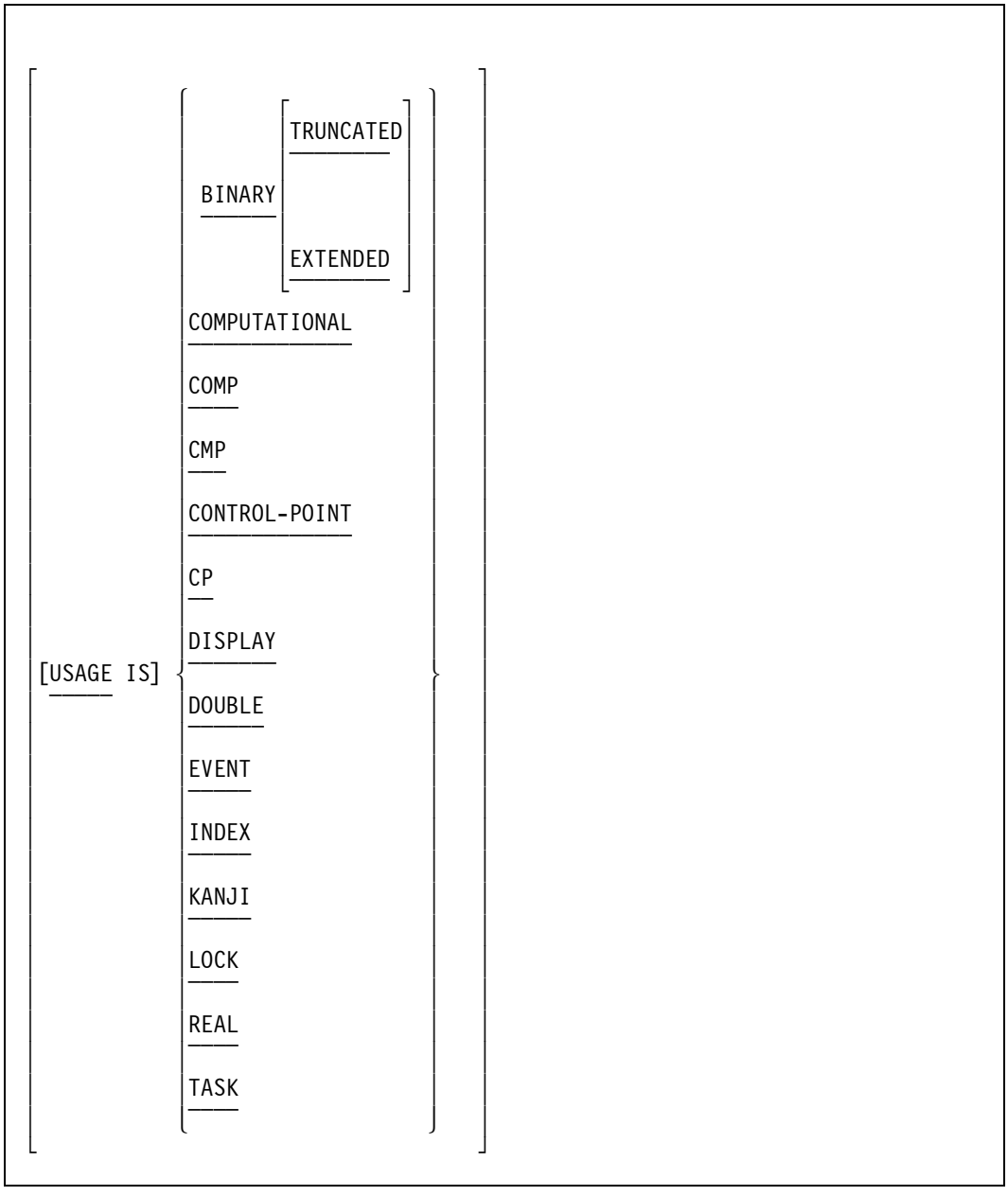
The USAGE clause specifies the format of a data item.

If the USAGE clause is specified at a group level, it applies to each elementary item in the group.

The USAGE clause can appear in any data-description entry with a level-number other than 66 or 88. (This is an extension to ANSI X3.23-1974 COBOL.)

If the USAGE clause is written in the data-description entry for a group item, it can also be written in the data-description entry for any subordinate elementary item of the group item, but the same USAGE clause must be specified by both entries. Items with different USAGE clauses can appear in the same record. (This is an extension to ANSI X3.23-1974 COBOL.)

The general format of this clause is as follows:



Explanation of Format

BINARY (Extension to ANSI X3.23-1974 COBOL)

The USAGE IS BINARY clause indicates that data is in a binary-coded format. A binary item is capable of representing a value to be used in computations and therefore is always numeric.

Binary items occupy memory as follows:

- When the declared size is less than or equal to 11 decimal digits, the actual size is equal to 1 computer word; however, the item is not necessarily aligned on a word boundary. (This size is equivalent to 6 DISPLAY digits or 12 COMPUTATIONAL digits.)
- When the declared size is greater than 11 digits, the actual size is equal to 2 computer words (the equivalent of 12 DISPLAY digits); however, the item is not necessarily aligned on a word boundary.
- The size of the record is determined by the actual size of the item (1 or 2 computer words).

Although BINARY items are not required to start at a word boundary, faster execution results when these items start at a word boundary.

The value stored in a USAGE BINARY data item is maintained internally as an integer. If the associated PICTURE clause contains an explicit decimal point, the compiler takes this into account in any operations. High-order digit truncation and the applicability of ON SIZE ERROR clauses to a USAGE BINARY data item is dependent on whether such data items are implicitly USAGE BINARY TRUNCATED or USAGE BINARY EXTENDED.

If the item is declared as an elementary item at Level 01, it is treated as if it had been declared USAGE IS BINARY TRUNCATED.

If the item is declared as an elementary item at Level 77, or as a subordinate item in a group, it is treated as if it had been declared USAGE IS BINARY EXTENDED.

Note: *In a future release, this inconsistency may be eliminated.*

See BINARY EXTENDED and BINARY TRUNCATED below for details on these clauses.

Unisys recommends that the user explicitly specify either USAGE BINARY EXTENDED or USAGE BINARY TRUNCATED for USAGE BINARY items.

BINARY EXTENDED (Extension to ANSI X3.23-1974 COBOL)

If the EXTENDED phrase is specified or implied for the item, then high-order digits are not explicitly truncated. When the item is used as a destination, the size-error determination is limited to arithmetic faults, such as integer-overflow conditions.

The maximum internal magnitude that can be stored in such a data item without the risk of precision loss is 549,755,813,887 if the PICTURE clause specifies 11 or fewer digits; otherwise, the maximum safe value is 302,231,454,903,657,293,676,543.

When arithmetic statements with ON SIZE ERROR clauses produce internal results that exceed these values, the ON SIZE ERROR condition is set. For other statements, INTEGER OVERFLOW program terminations might occur.

BINARY TRUNCATED (Extension to ANSI X3.23-1974 COBOL)

If the TRUNCATED phrase is specified or implied for an item, then the contents of the PICTURE clause are used for high-order digit truncation, and the data item is used as a destination for size-error determination.

The value stored in a USAGE BINARY TRUNCATED data item is maintained internally as an integer. If the associated PICTURE clause contains an explicit decimal point, then the compiler takes this into account in any operation. High-order digits outside the range specified by the PICTURE clause are truncated. For ON statements with an ON SIZE ERROR clause, the SIZE ERROR condition is set.

COMPUTATIONAL, CMP, or COMP

CMP and COMP are abbreviations for COMPUTATIONAL.

Elementary COMPUTATIONAL data items are represented internally as contiguous 4-bit digits.

A COMPUTATIONAL item can represent a value to be used in computations and must be numeric. A numeric literal is a character string with characters selected from the digits "0" through "9", the plus sign (+), the minus sign (-), and the decimal point. Digits "A" through "F" are NOT numeric. COMPUTATIONAL fields are packed-decimal numeric items, not hexadecimal strings.

If a group item is described as COMPUTATIONAL, the elementary items in the group are COMPUTATIONAL. The group itself is not COMPUTATIONAL and cannot be used in computations.

If the information contained in a data item does not match its description in the PICTURE clause, the results are unpredictable and vary depending on the machine for which the program was compiled, the machine on which the program is run, and the nature of the difference between the PICTURE clause and the data in the field.

The sign position of a data item described as signed must contain one of the following values:

- A hexadecimal digit C to indicate the item is positive
- A hexadecimal digit D to indicate the item is negative

The digit positions of a COMP item must contain only the hexadecimal digits 0 (zero) through 9. The hexadecimal digits A through F are invalid in the digit positions of such an item, and the results of attempting to access an item that violates these rules are unpredictable.

CONTROL-POINT, CP, or TASK

The USAGE IS CONTROL-POINT clause is a synonym for USAGE IS TASK. CP is an abbreviation for CONTROL-POINT.

If a group item is described with the USAGE IS TASK clause, the elementary items in the group are all task items. The group itself is not a task item and cannot be used in any statement except the USING phrase or within a parameter list. Elementary TASK items are data descriptors and, as such, occupy a single word of memory.

An elementary TASK item can be referred to directly only in an ATTACH, CALL, DETACH, RUN, EXECUTE, PROCESS, CHANGE, or SET statement, or in the USING phrase in a task-attribute expression, or in a parameter list. Further explanation of TASK items can be found in the descriptions of statements that reference them and in the task-attribute descriptions.

When a data usage is declared as TASK, the item can be a 77-level or a 01-level item or can be subordinate to a 01-level item declared with USAGE IS TASK.

Task items cannot be doubly subscripted. That is, a task item with an OCCURS clause cannot have a subordinate task item with an OCCURS clause. Task items cannot be redefined by items of any other usage. No other clauses are allowed on an item with USAGE IS TASK.

DISPLAY

DISPLAY data items are depicted internally as contiguous 8-bit characters represented in the EBCDIC character set.

The group item is considered to be a group data item that has an alphanumeric class, USAGE IS DISPLAY, and can be referenced at any place in the syntax acceptable for such an item. The size of the group item is in terms of DISPLAY characters aligned according to the rules for the DISPLAY phrase. The rule is that one character exists for every two 4-bit digits that form a part of the group item. (This is an extension to ANSI X3.23-1974 COBOL.)

The USAGE IS DISPLAY clause indicates that data is in a standard data format.

If the USAGE clause is not specified for an elementary item or for any group to which the item belongs, the usage is implicitly DISPLAY.

Every occurrence of a DISPLAY data item begins and ends on a byte boundary. In a record description, the declaration of a DISPLAY data item immediately following a COMPUTATIONAL or INDEX data item that does not end on a byte boundary causes automatic generation of a 4-bit filler between the two items. This filler area between the two data items is not included in the size of either item, but is included in the size of all group items to which the two items are subordinate. Similarly, if the last item declared in a group item at the next-lowest hierarchic level is a COMPUTATIONAL or an INDEX data item that does not end on a byte boundary, automatic generation of a 4-bit filler occurs. This filler is included in the size of the group item.

The PICTURE and USAGE clauses are the only clauses valid when the TYPE clause is specified. When the USAGE clause is used with the TYPE clause, the usage must be DISPLAY. (This paragraph is an extension to ANSI X3.23-1974 COBOL.)

If the information contained in a data item does not match its description in the PICTURE clause, the results are unpredictable and vary depending on the machine for which the program was compiled, the machine on which the program is run, and the nature of the difference between the PICTURE clause and the data in the field.

The sign position of a numeric data item described as signed must contain one of the following values:

- A hexadecimal digit C to indicate the item is positive
- A hexadecimal digit D to indicate the item is negative

The digit positions of a numeric DISPLAY item must contain only the hexadecimal digits 0 (zero) through 9. The hexadecimal digits A through F are invalid in the digit positions of such an item, and the results of attempting to access an item that violates these rules are unpredictable.

DOUBLE or REAL

The USAGE IS REAL and USAGE IS DOUBLE clauses indicate that data is in an internal floating-point format. A REAL or DOUBLE item is capable of representing a value to be used in computations and is always numeric. Neither the PICTURE clause nor the SIGN clause are permitted for REAL or DOUBLE items.

REAL and DOUBLE items occupy memory as follows:

- A REAL item is single precision; the actual size is equal to 1 computer word.
- A DOUBLE item is double precision; the actual size is equal to 2 computer words.
- Both REAL and DOUBLE items are not necessarily word-aligned.

Although REAL or DOUBLE items are not required to start at a word boundary, faster execution results when these items do start at a word boundary.

EVENT

Items described with the USAGE IS EVENT clause are used as a common interlock between two or more processes, thus providing an efficient means of correlating the activities of one process with its related processes. Elementary EVENT items occupy 2 words of memory. For information and syntax for controlling and testing event-names, refer to the CAUSE, RESET, IF, and WAIT statements.

EVENT usage is allowed only on a 77-level or a 01-level item; if used on a 01-level item, a subordinate OCCURS clause is allowed. No other entries are permitted with an event-name. EVENT items cannot be redefined by items of any other type.

INDEX

An INDEX data item (an elementary item described with the USAGE IS INDEX clause) contains a value that must correspond to the occurrence number of a table element. The elementary item cannot be a conditional variable. If a group item is described with the USAGE IS INDEX clause, the elementary items in the group are all index data items. The group itself is not an index data item and cannot be used in the SEARCH or the SET statement, or in a relation condition.

An index data item can be referenced explicitly only in a SEARCH or a SET statement, a relation condition, the USING phrase of a PROCEDURE DIVISION header, or the USING phrase of a CALL statement.

A group item is also considered to be a group data item if its class is numeric, if it has index usage, and if it can be referenced at any place in the syntax that is acceptable for such an item. The size of the group item is considered in terms of DISPLAY characters (4 characters for each subordinate index data item).

An index data item can be part of a group that is referenced in a MOVE or I/O statement; in this case, no conversion takes place.

An index data item can contain a signed value. An index data item occupies the same space and has the same alignment as an item declared PICTURE S9(7) USAGE IS COMPUTATIONAL.

The SYNCHRONIZED, JUSTIFIED, PICTURE, VALUE, and BLANK WHEN ZERO clauses cannot be used to describe group or elementary items declared with the USAGE IS INDEX clause.

KANJI (Extension to ANSI X3.23-1974 COBOL)

Kanji data items are depicted internally as contiguous 16-bit characters represented in the Japanese Kanji character set. Each Kanji character consists of 16 bits and occupies 2 bytes of memory.

When the Kanji phrase is specified for a group item, it implies that all subordinate elementary items are declared as USAGE IS KANJI. Any Kanji group item is regarded as an alphanumeric item whose frame size is 8 bits (DISPLAY), except when a Kanji group item is compared with a figurative constant and when a figurative constant is moved to a Kanji group item. If a usage is specified for a data item that is subordinate to the group item, the usage for both items must be identical. An item with a KANJI clause contains Kanji items.

All Kanji data items belong to the class alphanumeric, and the Kanji category is subdivided into Kanji and Kanji-edited.

Kanji usage is a type of double-octet usage.

LOCK

The same rules apply to declaring data usage as LOCK as those declaring data usage as EVENT.

For information and syntax for controlling and testing LOCK items, refer to the LOCK and UNLOCK statements in “PROCEDURE DIVISION Statements.”

Elementary LOCK items occupy 2 words of memory.

This item does not affect the use of the data item, although the specifications for some statements in the PROCEDURE DIVISION can restrict the USAGE clause of the reference operands. The USAGE clause can affect the radix or type of character representation of the item.

For More Information

- For information on using the CAUSE statement to initiate communication between processes in an asynchronous processing environment, refer to "CAUSE (Extension to ANSI X3.23-1974 COBOL)" in Section 9, "PROCEDURE DIVISION Statements."
- For information about evaluating conditions, refer to "IF" in Section 9, "PROCEDURE DIVISION Statements."
- For information on using the LOCK statement to deny related processes access to a common storage area, refer to "LOCK (Extension to ANSI X3.23-1974 COBOL)" in Section 9, "PROCEDURE DIVISION Statements."
- For information on using the RESET statement to control communication between processes in an asynchronous processing environment, refer to "RESET (Extension to ANSI X3.23-1974 COBOL)" in Section 9, "PROCEDURE DIVISION Statements."
- For information on using the UNLOCK statement to unlock a previously locked common storage area, refer to "UNLOCK (Extension to ANSI X3.23-1974 COBOL)" in Section 9, "PROCEDURE DIVISION Statements."
- For information on using the WAIT statement to suspend program execution for a specified time or until one or more conditions are TRUE, refer to "WAIT (Extension to ANSI X3.23-1974 COBOL)" in Section 9, "PROCEDURE DIVISION Statements."

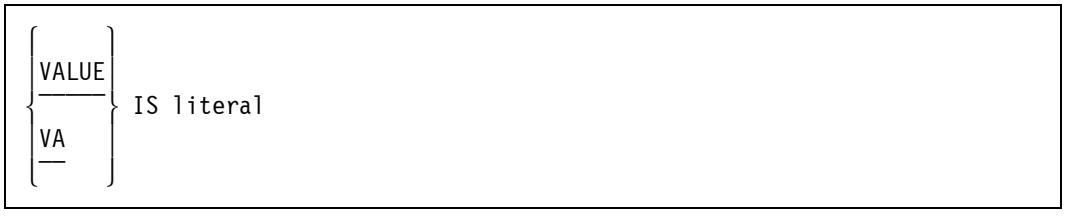
VALUE Clause

The VALUE clause defines the initial value of COMMUNICATION section and WORKING-STORAGE section data items, and the values associated with condition-names.

The VALUE clause cannot be

- Stated for any items of variable size
- Used with the GLOBAL or OWN clause
- Used for any item declared explicitly in the data declaration, or declared implicitly by using the GLOBAL or OWN compiler control options in bound procedures

Format 1



Format 2

Refer to "Data-Description Entry for Condition-Names" later in this section.

Explanation of Format 1

VALUE or VA

VA is an abbreviation for VALUE. (This is an extension to ANSI X3.23-1974 COBOL).

literal

A signed numeric literal must have a signed numeric PICTURE character string associated with it.

A numeric literal must have a value in the range of values indicated by the PICTURE clause and must not have a value that requires truncation of nonzero digits. A nonnumeric literal must not exceed the size indicated by the PICTURE clause.

A figurative constant can be substituted for the literal.

Rules Related to Data Category

The contents of the VALUE clause must not conflict with the description of the data item contained in the PICTURE or USAGE clauses associated with that data item. The following table shows the rules that apply:

Table 7–11. VALUE Clause Rules by Data Category

Data Category	Rules
Numeric	The literal 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 rules for data alignment.
Alphabetic, alphanumeric, alphanumeric-edited, or numeric-edited	The literal must be a nonnumeric literal. The literal is aligned in the data item as if the data item were 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 of the data item. Therefore, the VALUE clause for an edited item is presented in an edited form.
Kanji or Kanji-edited	The literal clause must be a Kanji literal. If the VALUE clause is specified for an edited item, no edit operation occurs.

Rules Related to Other Data-Description Entries

The VALUE clause must not conflict with other clauses in the data description of the item or in the data description within the hierarchy of the item. The following rules apply:

- Initialization occurs independently of the BLANK WHEN ZERO and the JUSTIFIED clause.

- The VALUE clause must not be stated in a data-description entry that contains an OCCURS clause or in an entry that is subordinate to an entry containing an OCCURS clause. This rule does not apply to condition-name entries.
- The VALUE clause must not be stated in a data-description entry that contains a REDEFINES clause or in an entry that is subordinate to an entry containing a REDEFINES clause. This rule does not apply to condition-name entries.
- 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 this group. The VALUE clause cannot be stated at the subordinate levels within this group.
- The VALUE clause must not be written for a group containing items for which USAGE (other than USAGE IS DISPLAY) is explicitly or implicitly specified.

Rules Related to DATA DIVISION Sections

The following table shows the rules that govern the use of the VALUE clause in each section of the DATA DIVISION.

Table 7–12. VALUE Clause Rules by Section

Section	Rules
FILE	The VALUE clause can be used only in condition-name entries.
WORKING-STORAGE and COMMUNICATION	The VALUE clause must be used in condition-name entries. The VALUE clause can also be used to specify the initial value of any other data item; in this case, the clause causes the item to assume the specified value at the start of the object program. If the VALUE clause is not used in the description of an item, the initial value is undefined.
LINKAGE	The VALUE clause can be used only in condition-name entries.
REPORT	If the elementary report entry containing the VALUE clause does not contain a GROUP INDICATE clause, then the printable item assumes the specified value each time the report group is printed. However, when the GROUP INDICATE clause is included, the specified value is presented only when certain execution-time conditions are met.

Rules Related to Data

Be careful when specifying a Format 1 VALUE clause on a data item in either the WORKING-STORAGE or the COMMUNICATION section. Using the VALUE clause in this way is analogous to placing a "MOVE literal TO identifier-2 [, identifier-3]" statement at the beginning of the PROCEDURE DIVISION. In this situation, the compiler overwrites information provided to the program as a parameter with the contents of the VALUE clause before it executes any other user statements. Information provided to the program as a parameter is data the program lists in the USING clause of the PROCEDURE DIVISION header.

For More Information

- For information on determining the category of a data item, refer to "PICTURE Clause" in this section.
- For information on the use of the VALUE clause with condition-names, refer to "Data-Description Entry for Condition-Names" in this section.

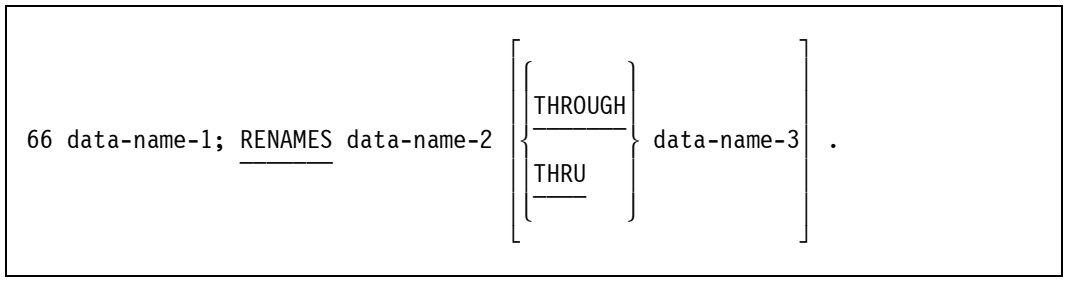
Data-Description Entry for Renaming Entries

Format 2 of the data-description entry uses the RENAMES clause. The RENAMES clause permits alternative, possibly overlapping, groupings of elementary items.

One or more RENAMES clauses can be written for a logical record.

All RENAMES clauses that refer to data items in a given logical record must immediately follow the last data-description entry of the associated record-description entry.

Format 2: Renaming Entries



Note: *Level-number 66, data-name-1, and the semicolon (;) are shown in the preceding format for clarity. Level-number 66 and data-name-1 are not part of the RENAMES clause.*

Explanation of Format 2

Data-name-1 cannot be used as a qualifier and can be qualified only by the names of the associated 01-level, field description (FD), communication description (CD), and sort merge description (SD). Neither data-name-2 nor data-name-3 can have an OCCURS clause in the data-description entry or be subordinate to an item that has an OCCURS clause in the data-description entry.

Data-name-2 and data-name-3 must be names of elementary items or groups of elementary items in the logical record and cannot be the same data-name. A 66-level entry cannot rename another 66-level entry, nor can it rename a 77-, 88-, or 01-level entry.

The beginning of the area described by data-name-3 must not be to the left of the beginning of the area described by data-name-2. The end of the area described by data-name-3 must be to the right of the end of the area described by data-name-2. Data-name-3, therefore, cannot be subordinate to data-name-2.

Data-name-2 and data-name-3 can be qualified.

The words THRU and THROUGH are equivalent.

None of the items in the range, including data-name-2 and data-name-3 (if specified), can be an item of variable size as defined in the OCCURS clause.

When data-name-3 is specified, data-name-1 is a group item that

- Includes all elementary items starting with data-name-2 if data-name-2 is an elementary item.
- Includes the first elementary item in data-name-2 if data-name-2 is a group item.
- Concludes with data-name-3 if data-name-3 is an elementary item.
- Concludes with the last elementary item in data-name-3 if data-name-3 is a group item.

When data-name-3 is specified and data-name-2 is an elementary COMPUTATIONAL or INDEX data item, data-name-3 must be positioned to begin at an 8-bit character boundary. (This is an extension to ANSI X3.23-1974 COBOL.)

When data-name-3 is specified and is an elementary COMPUTATIONAL or INDEX data item, data-name-3 must be positioned to end at the end of an 8-bit character boundary. (This is an extension to ANSI X3.23-1974 COBOL.)

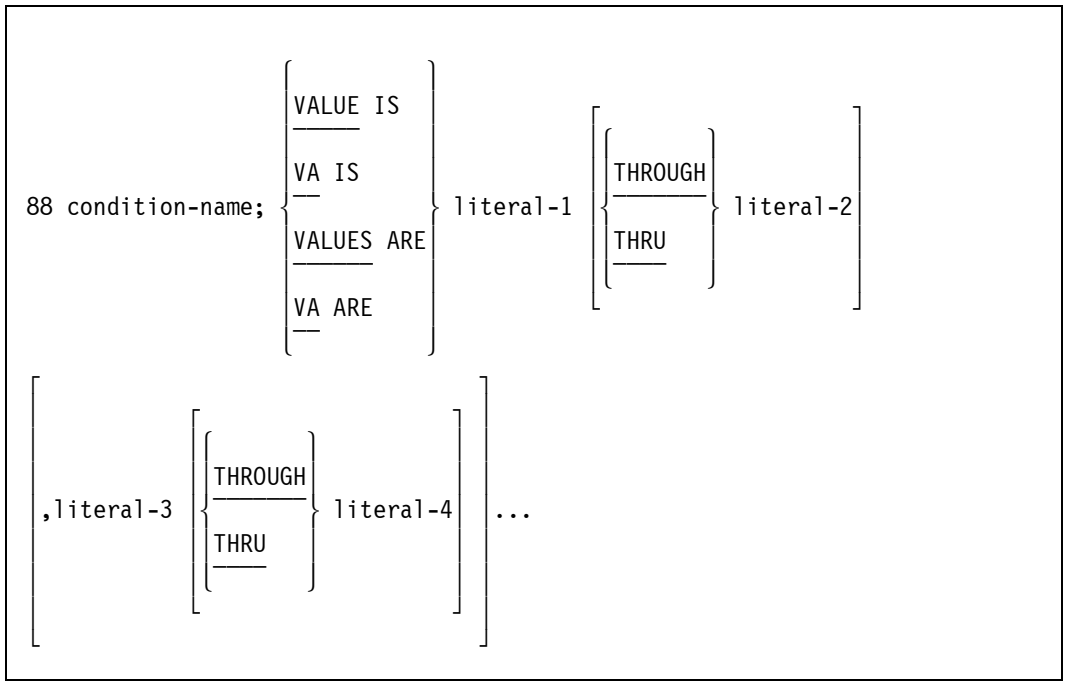
When data-name-3 is not specified, data-name-2 can be either a group or an elementary item. When data-name-2 is a group item, data-name-1 is treated as a group item; when data-name-2 is an elementary item, data-name-1 is treated as an elementary item.
Data-name-1 assumes all characteristics of data-name-2 as determined from the data description of data-name-2, including usage, justification, synchronization, editing requirements, and so forth.

Data-Description Entry for Condition-Names

Format 3 of the data-description entry assigns values to condition-names.

A condition-name is a name assigned to a specific value, a set of values or a range of values within a complete set of values that a data item can assume. The data item itself is called a conditional variable.

Format 3 of the Data-Description Entry



Explanation of Format 3

Each condition-name requires a separate entry with level-number 88. Format 3 contains the name of the condition and the value, values, or range of values associated with the condition-name. The condition-name entries for a particular conditional variable must follow the entry describing the item with which the condition-name is associated.

A condition-name can be associated with any data-description entry that contains a level-number except the following:

- Another condition-name.
- A level-66 item.
- A group item containing items for which USAGE (other than USAGE IS DISPLAY) is explicitly or implicitly specified. (This is an extension to ANSI X3.23-1974 COBOL.)
- An index data item.

In a condition-name entry, the VALUE clause is required. The VALUE clause and the condition-name itself are the only two clauses permitted in the entry. The characteristics of a condition-name are implicitly those of the conditional variable.

Wherever the THROUGH phrase is used, literal-1 must be less than literal-2, literal-3 less than literal-4, and so forth.

***Note**: The compiler does not verify the ascending relation of the literal-1 and literal-2. If literal-2 is less than literal-1, the evaluation of the condition returns FALSE. The words THROUGH and THRU are equivalent.*

Example

Example 7–9 shows the coding of condition-names. THIS-MONTH is the conditional variable, and the months JANUARY through DECEMBER are the condition-names.

```
01  THIS-MONTH      PIC 99.
    88  JANUARY             VALUE 1.
    88  FEBRUARY            VALUE 2.
    88  MARCH               VALUE 3.
    88  APRIL               VALUE 4.
    88  MAY                 VALUE 5.
    88  JUNE                VALUE 6.
    88  JULY                VALUE 7.
    88  AUGUST              VALUE 8.
    88  SEPTEMBER           VALUE 9.
    88  OCTOBER             VALUE 10.
    88  NOVEMBER            VALUE 11.
    88  DECEMBER            VALUE 12.
    88  SPRING              VALUE 3 THRU 5.
    88  THIRTY DAYS         VALUE 4, 6, 9, 11.
```

Example 7–9. Coding Condition-Names

In Example 7–9, if JUNE has the assigned value 6, the following entry could be written using the condition-name JUNE:

```
IF JUNE PERFORM ...
```

This coding is logically equivalent to using the conditional variable THIS-MONTH and writing:

```
IF THIS-MONTH IS EQUAL TO 6 PERFORM ...
```

Example

Example 7-10 shows the coding of condition-names that violates ascending relation of literal-1 less than literal-2.

```
01 COND-GROUP    PIC S9(11) BINARY.
   88   GROUP1 VALUE   1  THRU  10.
   88   GROUP2 VALUE  30  THRU  20.
   88   GROUP3 VALUE -62  THRU -68.
   88   GROUP4 VALUE -20  THRU -10.
MOVE 8 TO COND-GROUP.
IF GROUP1 DISPLAY "GROUP1".
MOVE 25 TO COND-GROUP.
IF GROUP2 DISPLAY "GROUP2".
MOVE 30 TO COND-GROUP.
IF GROUP2 DISPLAY "IT'S GROUP2!".
MOVE -66 TO COND-GROUP.
IF GROUP3 DISPLAY "GROUP3".
MOVE -68 TO COND-GROUP.
IF GROUP3 DISPLAY "IT'S GROUP3!".
MOVE -15 TO COND-GROUP.
IF GROUP4 DISPLAY "GROUP4".
```

At runtime, this program displays the following output:

- DISPLAY:GROUP1.
- DISPLAY:GROUP4.

For More Information

- Refer to “VALUE Clause” in this section for the details on this clause.
- Refer to “Condition-Name Condition” in Section 8, “PROCEDURE DIVISION Concepts” for information on this simple relation condition.

WORKING-STORAGE SECTION

The WORKING-STORAGE SECTION is composed of the section header followed by data-description entries for noncontiguous data items, record-description entries, or both entries. Each WORKING-STORAGE SECTION record-name and noncontiguous item name must be unique because these names cannot be qualified. Subordinate data-names need not be unique if they can be made unique by qualification.

Noncontiguous WORKING-STORAGE Items

Items and constants in the WORKING-STORAGE SECTION that have no hierarchic relationship to one another need not be grouped into records, provided that they do not need to be further subdivided. Instead, these items and constants are classified and defined as noncontiguous elementary items. Each of these items is defined in a separate data-description entry, beginning with the special level-number 77.

The general format of this section is as follows:

```
                ⎧           ⎫
                ⎪ PICTURE   ⎪
77 data-name    ⎨           ⎬
                ⎪ USAGE IS INDEX ⎪
                ⎩           ⎭

[Any other data-description clauses].
```

WORKING-STORAGE Records

Data elements and constants in the WORKING-STORAGE SECTION that bear a definite hierarchic relationship to one another must be grouped into records according to the rules for formation of record descriptions. All clauses used in record descriptions in the FILE SECTION can be used in record descriptions in the WORKING-STORAGE SECTION.

Example

Example 7–10 shows the coding of the WORKING-STORAGE SECTION.

```
WORKING-STORAGE SECTION
  77  MASTER-KEY              PIC 9(8)    COMP.
  77  TOTAL-SALES             PIC 9(10)   COMP  VALUE ZEROS.
  01  STATE-TABLE.
      03  STATES.
          05  CALIF           PIC 9(6).
          05  MICH            PIC 9(6).
          05  FLORIDA         PIC 9(6).
          05  ARIZONA         PIC 9(6).
```

Example 7–10. Coding the WORKING-STORAGE SECTION

```
03  STATE-KEY  REDEFINES STATES     OCCURS 4 TIMES.
          05  STATE-CODE      PIC 99.
05  COUNTY          PIC 99.
          05  CAPITAL-CODE    PIC 99.
  01  HEADING-LINE.
      03  FILLER              PIC X(52)   VALUE SPACES.
      03  FILLER              PIC X(17)   VALUE
                "SALES PERFORMANCES".
      03  FILLER              PIC X(10)   VALUE SPACES.
      03  PAGE-NO             PIC 9999.
      03  FILLER              PIC X(49)   VALUE SPACES.
```

Example 7–10. Coding the WORKING-STORAGE SECTION

LOCAL-STORAGE SECTION (Extension to ANSI X3.23-1974 COBOL)

The optional LOCAL-STORAGE SECTION describes parameters to be received, allowing separate tasks or procedures to be bound from another program. These parameters are described with USE EXTERNAL statements.

The general format of items in this section is as follows:

```
LD local-storage-name. [Any data description clauses].
```

Explanation of Format

The local-storage-description (LD) entry is followed by item descriptions used in the WORKING-STORAGE SECTION.

Local-storage items are associated with a specific procedure by being listed in the WITH clause of the USE statement for the procedure. All local-storage-names must be unique. An LD entry is required for each procedure that receives data as parameters. That is, a USING clause is present in both the invocation of the procedure and the USE statement in the section header.

Section 8
PROCEDURE DIVISION Concepts

The last division of a program, the PROCEDURE DIVISION, must be included in every COBOL program. This division can contain declarative and nondeclarative procedures.

PROCEDURE DIVISION Header

The PROCEDURE DIVISION is identified by, and must begin with, the following format:

```
                            ┌                         ┐
                            │       ┌            ┐    │
                            │       │ data-name  │    │
                            │       │ file-name  │    │
PROCEDURE  DIVISION         │ USING ┤ task-name  ├... │ .
---------  --------         │ ----- │ event-name │    │
                            │       │ lock-name  │    │
                            │       └            ┘    │
                            └                         ┘
```

Explanation of Format

The optional USING clause names the identifiers received as parameters in a tasking, bound procedure, library, or Inter-Program Communication (IPC) environment. When the USING clause is present, the object program operates as if each identifier in the list were replaced by the corresponding identifier from the USING clause of the CALL, PROCESS, or RUN statement of the calling program or Work Flow Language (WFL) job.

A data-name, file-name, task-name, event-name, or lock-name in the USING clause of the PROCEDURE DIVISION header must be defined in the LINKAGE SECTION of the program in which this header occurs. The data-name, task-name, event-name, and lock-name must be declared at level 77 or 01, and must not be redefined items.

When the RECEIVED BY REF clause appears in a data description for an identifier, the identifier refers to a single set of data available to the calling and the called program.

When the data-name is RECEIVED BY CONTENT, the invocation of the procedure initializes the corresponding data-name in the USING clause of the called program to the value in the initiating program. The correspondence is by position and not by symbolic name. Only tasking and bound-procedure environments support by-value parameters.

You must to ensure that the information passed into the parameters described in the USING clause of the PROCEDURE DIVISION header is compatible with the parameters in both format and size, or that the COBOL74 program does not attempt to access beyond the limits of the actual data as passed to the program (as distinct from the data limits described in the program). Failure to observe this convention can result in program failures such as INVALID INDEX and SEG ARRAY ERROR.

The rules for parameter mapping between COBOL74 programs called as tasks and programs written in other languages (including ALGOL and WFL) are discussed in more detail under "CALL" in Section 9, "PROCEDURE DIVISION Statements." The rules for parameter matching for the USING clause as described under "Explanation of Format 1" of the CALL verb apply as well to the USING clause of the PROCEDURE DIVISION header. In particular, Table 9–3, "Parameter Mapping for Tasking Calls," and Table 9–4, "WFL and COBOL74 Parameters," are of importance in determining the matching rules for COBOL74 programs called as tasks, whether they are called from WFL jobs or from ALGOL programs.

For string parameters passed from WFL jobs to COBOL74 programs as group or USAGE DISPLAY items, program failures can often be avoided by relying on the format of such parameters as documented in the *WFL Reference Manual*. Such parameters are terminated internally by the null (@00@) character, regardless of length. In cases in which the actual data is shorter than the description in the program, "STRING ... DELIMITED BY @00@..." is an effective tool in extracting all valid information while remaining within the confines of the actual data passed to the program.

For More Information

- For information about the USING clause with IPC, refer to "PROCEDURE DIVISION in the IPC Module" in Section 13, "ANSI Inter-Program Communication (IPC)."
- For a description of the USING clause with libraries, refer to Section 15, "Libraries."

PROCEDURE DIVISION Body

The body of the PROCEDURE DIVISION must conform to one of two formats:

Format 1

```
[                                                                    ]
[DECLARATIVES.                                                       ]
[                                                                    ]
[  {section-name SECTION [segment-number]. declarative sentence.}... ]
[  {     [paragraph-name. [sentence]...]...                      }   ]
[                                                                    ]
[  END DECLARATIVES.                                                 ]
[                                                                    ]

{section-name SECTION [segment-number].  }
{     [paragraph-name. [sentence]...]...}...
```

Format 2

```
{paragraph-name. [sentence]...}...
```

Explanation of Format 1 and Format 2

DECLARATIVES SECTIONs must be grouped at the beginning of the PROCEDURE DIVISION and must be preceded by the keyword DECLARATIVES and followed by the keywords END DECLARATIVES.

A section consists of a section header followed by one or more successive paragraphs. A section ends as follows:

- Immediately before the next section
- At the end of the PROCEDURE DIVISION
- At the keywords END DECLARATIVES in the DECLARATIVES SECTION

A procedure is composed of a paragraph, a group of successive paragraphs, a section, or a group of successive sections in the PROCEDURE DIVISION. If one paragraph is in a section, then all paragraphs must be in sections. A procedure-name is a word used to refer to a paragraph or a section in the source program in which the name occurs. The procedure-name consists of a paragraph-name (which can be qualified) or a section-name.

A paragraph consists of a paragraph-name followed by a period and by one or more successive sentences. A paragraph ends as follows:

- Immediately before the next paragraph-name or section-name
- At the end of the PROCEDURE DIVISION

- At the keywords END DECLARATIVES in the DECLARATIVES SECTION

A sentence consists of one or more statements and ends in a period.

A statement is a valid combination of words and symbols beginning with a verb.

The end of the PROCEDURE DIVISION and the physical end of the program is the physical position in a COBOL source program after which no further procedures appear.

Example

Example 8–1 is a sample program that illustrates the use of declaratives.

```
IDENTIFICATION DIVISION.
  ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
      SELECT INPUT-FILE ASSIGN TO DISK
      FILE STATUS IS INPUT-STATUS.

  DATA DIVISION.
  FILE SECTION.
  FD  INPUT-FILE
      VALUE OF TITLE IS "DISK-FILE".
  01  INPUT-REC PIC X(80).
  WORKING-STORAGE SECTION.
  77  INPUT-STATUS PIC XX.

  PROCEDURE DIVISION.
  DECLARATIVES.
  DECL-1 SECTION.
      USE AFTER STANDARD ERROR PROCEDURE ON INPUT.
  D-0010.
      DISPLAY "I/O ERROR READING FILE".
      DISPLAY "FILE STATUS IS " INPUT-STATUS.
      STOP RUN.
  END DECLARATIVES.

  MAIL-1000 SECTION.
  P-1010.
      MOVE "00" TO INPUT-STATUS.
      OPEN INPUT INPUT-FILE.
      READ INPUT-FILE.
      STOP RUN.
```

Example 8–1. Use of Declaratives

Categories of Statements and Sentences

Statements and sentences can be of the following three types:

- Conditional
- Compiler-directing imperative
- Program-directing imperative

Conditional Statements and Sentences

A conditional statement contains a condition that can be TRUE or FALSE, and specifies actions to be taken that depend on the truth value of the condition. The program performs a test to determine the value of the condition and performs a specified action when the condition is TRUE and another specified action when the condition is FALSE. For example, the *IF condition THEN imperative-statement* statement directs the program to perform the THEN statement when the specified condition is TRUE.

A conditional sentence can contain several imperative statements associated with a condition. A conditional sentence must end with a period.

Compiler-Directing Imperative Statements and Sentences

A compiler-directing imperative statement causes the compiler to take a specific action during the compilation process. The compiler-directing imperative statements include the following statements:

- The COPY statement, which directs the compiler to incorporate text from a library program into the program that contains the COPY statement
- The USE statement, which
  - Specifies procedures to be used for I/O exception handling
  - Identifies a separately compiled program to be used as a task or to be bound into the COBOL74 host program
  - Specifies a declarative as an interrupt procedure

A compiler-directing sentence is a single compiler-directing statement ending in a period.

Program-Directing Imperative Statements and Sentences

A program-directing imperative statement causes the program to take a specific unconditional action. An imperative statement can consist of a sequence of imperative statements.

The program-directing imperative statements are listed in Table 8–8 at the end of this section.

When the user-defined word *imperative-statement* appears in a general format notation, it refers to one of the following:

- A sequence of consecutive imperative sentences ended by a period
- An ELSE phrase associated with a previous IF statement
- A WHEN phrase associated with a previous SEARCH statement

An imperative sentence is an imperative statement ending in a period.

Arithmetic Expressions

An arithmetic expression can be one of the following:

- An identifier of a numeric elementary item
- A numeric literal
- A numeric function or an extended function with a numeric result
- Identifiers and literals separated by arithmetic operators
- Two arithmetic expressions separated by an arithmetic operator
- An arithmetic expression enclosed in parentheses

Any arithmetic expression can be preceded by a unary operator.

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

In a conditional expression, the use of a simple variable in parentheses, particularly when that simple variable is subscripted, can have undesirable results. It is recommended that you avoid this type of coding.

Arithmetic Operators

Five binary and two unary arithmetic operators can be used in arithmetic expressions and are listed in Tables 8–1 and 8–2.

Table 8–1. Binary Arithmetic Operators

Symbol	Meaning
+ (plus sign)	Addition
– (minus sign)	Subtraction
* (asterisk)	Multiplication
/ (slash)	Division
**	Exponentiation

Table 8–2. Unary Arithmetic Operators

Symbol	Meaning
+ (plus sign)	The effect of multiplication by +1
– (minus sign)	The effect of multiplication by –1

Formation and Evaluation Rules

Parentheses in arithmetic expressions define the order in which elements are evaluated. Expressions in parentheses are evaluated first; in nested parentheses, evaluation proceeds from the least inclusive set. When parentheses are not used, or when parenthesized expressions are at the same level of inclusiveness, the following ordering is used:

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

Parentheses either eliminate ambiguities in logic where consecutive operations of the same procedure appear or modify the normal hierarchic sequence of execution in expressions where deviation from the normal precedence is necessary. When the sequence of execution is not specified by parentheses, the execution order for consecutive operations of the same procedure is from left to right.

An arithmetic expression can begin only with a left parenthesis, a plus sign (+), a minus sign (–), or a variable. A one-to-one correspondence must be maintained between left and right parentheses of an arithmetic expression so that each left parenthesis is to the left of its corresponding right parenthesis.

The COBOL operator for exponentiation (**) causes any operand less than 12 digits long to be converted to double-precision representation before the operation. The exponentiation is performed on the double-precision operands. The result is truncated to single precision for any receiving field that is less than 12 digits long.

Furthermore, for all noninteger operands, the operand value is scaled into a double-precision floating-point value as part of the preparation for the operation, and various arithmetic operations are performed during the operation itself. The result of exponentiation should always be regarded as an approximation. Performing the appropriate calculations directly within the program might produce more precise results than exponentiation, particularly when the exponent is known to be an integer.

The valid combinations of operators, variables, and parentheses that can be arranged in an arithmetic expression are summarized in Table 8–3. If a combination is shown as not valid, it means that the symbols cannot occur next to each other in an arithmetic expression.

Table 8–3. Combination of Symbols in Arithmetic Expressions

First Symbol	Second Symbol				
	Identifier or Literal	*** / ** – +**	**Unary + or –**	**(**	**)**
* / ** + –	P	N	P	P	N
Unary + or –	P	N	N	P	N
(	P	N	P	P	N
)	N	P	N	N	P

Legend

P Combination of symbols is permitted.

N Combination of symbols is not valid.

Numeric Functions

A numeric function can be specified as a sending operand in an arithmetic statement, a MOVE statement, or an arithmetic expression.

The functions described in the following text are extensions to ANSI X3.23-1974 COBOL. An extended function can be specified as an arithmetic expression in the COMPUTE statement.

Refer to Volume 2 for information about the FORM-KEY function.

DIV

The DIV function is an integer division function that returns an integer equal to the integer part of the quotient after division.

The format for the DIV function is as follows:

```
FUNCTION DIV (argument-1, argument-2)
```

Explanation of Format

Argument-1 represents the dividend, and argument-2 the divisor. Argument-1 and argument-2 can be any language-defined arithmetic expression, including a valid function call that returns a numeric value. The result returned can be a real or an integer value, depending on the argument type. The result of argument-2 must not be 0 (zero).

Example

In the three COMPUTE statements in Example 8–2, the DIV function returns the integers 5, 3, and 30, respectively.

```
COMPUTE A = FUNCTION DIV(10,2).

COMPUTE A = FUNCTION DIV(10,3).

COMPUTE A = FUNCTION DIV(10,.33).
```

Example 8–2. Coding the DIV Function

FORMATTED-SIZE

The FORMATTED-SIZE function returns as a value the formatted size of the data-name. The returned value is equal to the following:

```
Length of the data-name in bytes +
    (number of Kanji data items subordinate to the data-name * 2)
```

The format for the FORMATTED-SIZE function is as follows:

```
FUNCTION FORMATTED-SIZE (data-name)
```

Explanation of Format

The number of Kanji data items subordinate to the data-name is determined according to the same rules that are used for record formatting in the WRITE DELIMITED statement.

The data-name can be qualified. It cannot be subscripted or indexed, nor can the data-name be a RENAMES entry.

The data-name must be either a group item or any category of elementary item described implicitly or explicitly as USAGE IS DISPLAY or USAGE IS KANJI.

Example

Example 8–3 shows the results returned by the FORMATTED-SIZE function.

```
000200 IDENTIFICATION DIVISION.
000300 ENVIRONMENT DIVISION.
000400 DATA DIVISION.
000500 WORKING-STORAGE SECTION.
000600 77  THE-SIZE PIC 9(8).
000700 01  ACCOUNT-FILE-RECORD.
000800     03  ACCT-NO PIC 9(8).
000900     03  NUMBER-OF-DETAILS PIC 99.
001000     03  TRANSACTION-DETAILS OCCURS 1 TO 100
001100          DEPENDING ON NUMBER-OF-DETAILS.
001200*  The physical size of TRANSACTION-DETAILS is 138 bytes.
001300*  The formatted size of the same group item is 142
001400*  bytes because of the leading and trailing delimiters
001500*  accounted for around each Kanji elementary item.
001600         05  DETAIL-IDENT    PIC 9(8).
001700         05  DETAIL-TEXT     PIC X(20)           KANJI.
001800         05  DETAIL-AMOUNT   PIC S9(8)V99.
001900         05  DETAIL-DESCRIPTION PIC X(40)        KANJI.
002000 PROCEDURE DIVISION.
002100 ONLY-HEADER.
002200     MOVE 1 TO NUMBER-OF-DETAILS.
002300*  Function FORMATTED-SIZE (account-file-record) should now
002400*  return a value of 152:  10 + (1 * (138 + (2 * 2)))
002500     COMPUTE THE-SIZE =
002600         FUNCTION FORMATTED-SIZE (ACCOUNT-FILE-RECORD).
002700     DISPLAY THE-SIZE.
002800*  Function FORMATTED-SIZE (account-file-record) should now
002900*  return a value of 2850:  10 + (20 * (138 + (2 * 2)))
003000     MOVE 20 TO NUMBER-OF-DETAILS.
003100     COMPUTE THE-SIZE =
003200         FUNCTION FORMATTED-SIZE (ACCOUNT-FILE-RECORD).
003300     DISPLAY THE-SIZE.
003400     STOP RUN.
```

Example 8–3. Results of FORMATTED-SIZE Function

For information about the WRITE DELIMITED statement, refer to "WRITE" in Section 9, "PROCEDURE DIVISION Statements."

MOD

The MOD function returns the modulus value of argument-1 divided by argument-2. The modulus value is equal to the following:

```
argument-1 - [argument-2 * INTEGER(argument-1/argument-2)]
```

The INTEGER function, implicit in the MOD function, is defined as the largest integer less than or equal to the argument. For example, INTEGER(+1.5) returns 1, and INTEGER(–1.5) returns –2.

The format for the MOD function is as follows:

```
FUNCTION MOD (argument-1,argument-2)
```

Explanation of Format

The arguments must be integer literals or data-names of type numeric. The scale should be equal to 0 (zero). No numbers can follow the character V in the character string declaration. The specifications follow the CODASYL proposal for the MOD function. Argument-2 must not be 0 (zero).

Example

Example 8–4 shows four COMPUTE statements that use the MOD function. The results of the example statements would be 1, 4, –4, and –1, respectively. The calculations that follow the example show how each result is obtained.

```
COMPUTE A = FUNCTION MOD(11,5).

COMPUTE A = FUNCTION MOD(-11,5).

COMPUTE A = FUNCTION MOD(11,-5).

COMPUTE A = FUNCTION MOD(-11,-5).
```

Example 8–4. Coding the MOD Function

A statement of the form *COMPUTE A = FUNCTION MOD(11,5).* produces a result of 1, as follows:

```
A = 11 - ( 5 * INTEGER (11 / 5) )
  = 11 - ( 5 * INTEGER (2.2) )
  = 11 - ( 5 * 2 )
  = 11 - 10
  = 1
```

A statement of the form *COMPUTE A = FUNCTION MOD(–11,5).* produces a result of 4, as follows:

```
A = -11 - ( 5 * INTEGER (-11 / 5) )
  = -11 - ( 5 * INTEGER (-2.2) )
  = -11 - ( 5 * -3 )
  = -11 + 15
  = 4
```

A statement of the form *COMPUTE A = FUNCTION MOD(11,–5).* produces a result of –4, as follows:

```
A = 11 - ( -5 * INTEGER (11 / -5) )
  = 11 - ( -5 * INTEGER (-2.2) )
  = 11 - ( -5 * -3 )
  = 11 - 15
  = -4
```

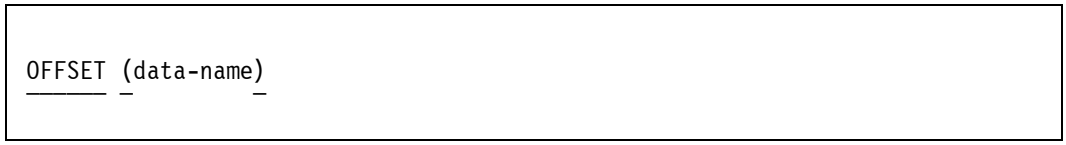
A statement of the form *COMPUTE A = FUNCTION MOD(–11,–5).* produces a result of –1, as follows:

```
A = -11 - ( -5 * INTEGER (-11 / -5) )
  = -11 - ( -5 * INTEGER (2.2) )
  = -11 - ( -5 * 2 )
  = -11 - ( -10 )
  = -11 + 10
  = -1
```

OFFSET

The OFFSET function returns a value equal to the number of characters that precede data-name in the logical record in which data-name is defined.

The format for the OFFSET function is as follows:



Explanation of Format

If data-name refers to a packed numeric data item that is not aligned on a character boundary, then the returned value is equal to the number of characters preceding the character with which data-name begins. If data-name is a record-name or a 77-level item, the value returned is 0. Data-name can be qualified.

Example

In Example 8–5, if Y is a 05-level data item within a 01-level group, the COMPUTE statement returns a value to Z that is the sum of the number of characters of the fields preceding Y in the same 01-level group data structure.

```
COMPUTE Z = OFFSET (Y).
```

Example 8–5. Coding the OFFSET Function

REM

The REM function returns as a value the remainder of a division.

The format for the REM function is as follows:

```
FUNCTION REM (argument-1,argument-2)
```

Explanation of Format

Argument-1 represents the dividend, and argument-2 the divisor. Argument-1 and argument-2 can be any language-defined arithmetic expression, including a valid function call that returns a numeric value. The result returned can be either a real or an integer value depending on the argument type. The result of argument-2 must not be 0 (zero).

Example

In Example 8–6, the four uses of the REM function return the values 1, 1, –1, and 0.1, respectively.

```
COMPUTE A = FUNCTION REM(10,3).

COMPUTE A = FUNCTION REM(11,-5).

COMPUTE A = FUNCTION REM(-11,5).

COMPUTE A = FUNCTION REM(10,0.33).
```

Example 8–6. Coding the REM Function

Multiple Function Calls in an Expression

Example 8–7 shows the way in which functions can be combined in a COMPUTE statement to form an arithmetic expression.

```
000200*  The first example shows that the values returned by
000300*  the DIV and REM functions are used in an addition operation.
000410*
000500 COMPUTE A = FUNCTION DIV(10,2) + FUNCTION REM(10,3).
000600*           = 5 + 1 = 6
000610*
000700*  In the next example, X and Y are numeric variables:
000800 COMPUTE A = 10 + FUNCTION DIV(10,X + Y).
000810*
000900* In the next example,  values returned by  DIV and REM functions
001000* are used as the arguments for a DIV function:
001050*
001100 COMPUTE A = FUNCTION MOD(-11,5) +
001200         FUNCTION DIV( (FUNCTION REM(10,3) *10),
001300         FUNCTION DIV(10,FUNCTION DIV(10,3) ) ).
001400*    = 4 +
001500*      FUNCTION DIV(FUNCTION REM(10,3) * 10,
001600*                   FUNCTION DIV(10,3) )
001700*    = 4 + FUNCTION DIV( (1 * 10) ,3)
001800*    = 4 + FUNCTION DIV( (1 * 10) ,3)
001900*    = 4 + 3
002000*    = 7
```

Example 8–7. Coding Multiple Function Calls in an Expression

Conditional Expressions

Conditional expressions identify conditions that are tested so that the object program can choose between paths of control that depend on the truth value of the condition. Conditional expressions are specified in IF, PERFORM, and SEARCH statements. The two categories of conditions associated with conditional expressions are simple conditions and complex conditions. Each can be enclosed in any number of paired parentheses; the paired parentheses do not change the category of the condition.

Simple Conditions

Simple conditions are relation, class, condition-name, sign, and event-identifier conditions. A simple condition has a truth value of TRUE or FALSE. The inclusion of parentheses in simple conditions does not change the simple truth value.

There is a permanent restriction on the use of parentheses in relational conditions. This restriction is not always enforced by a syntax error. Violating the following rule can result in an unexpected program failure or an incorrect program execution.

Parentheses in conditional expressions in COBOL74 should not be used to surround simple variables, particularly when the simple variables are subscripted.

Example

Example 8–8 shows correct and incorrect uses of parentheses. In the following code the first two statements are legitimate, but the third statement should be avoided.

```
IF A (I) IS EQUAL TO B (J)...
IF (A (I) IS EQUAL TO B (J))...
IF (A (I)) IS EQUAL TO (B (J))...
```

In the following code, the first line is technically correct, but it is better represented by the second or third line of code.

```
IF (C) IS EQUAL TO (D)
IF C IS EQUAL TO D
IF (C IS EQUAL TO D)
```

In the following code, the first line is technically correct, but it is better represented by the second or third line of code.

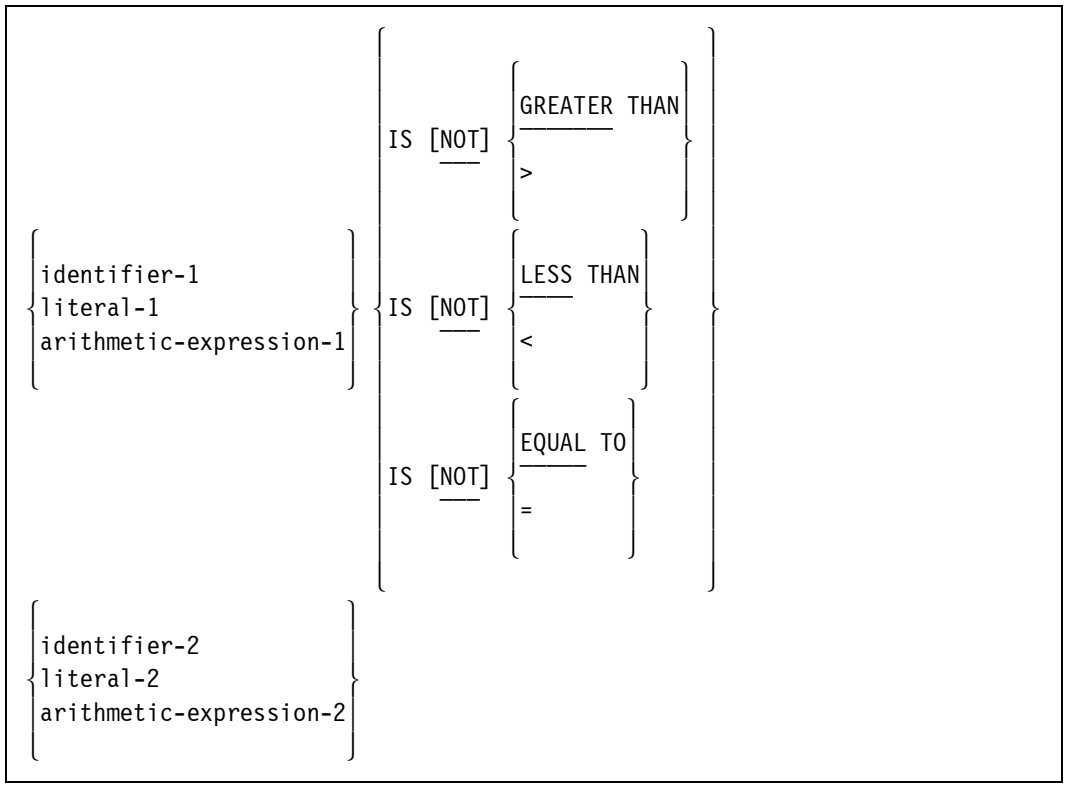
```
IF (LITERAL-1) IS EQUAL TO (LITERAL-2)
IF LITERAL-1 IS EQUAL TO LITERAL-2
IF (LITERAL-1 IS EQUAL TO LITERAL-2)
```

Example 8–8. Parentheses Restrictions in Simple Conditions

Relation Condition

A relation condition causes a comparison of two operands that can be data items referenced by an identifier, literals, or values resulting from an arithmetic expression. A relation condition has a truth value of TRUE if the relation exists between the operands. Comparison of two numeric operands is permitted regardless of their respective USAGE clauses. However, for all other comparisons, the operands must have the same usage. If either of the operands is a group item, the nonnumeric comparison rules apply.

The general format of a relation condition is as follows:



Note: *The required relational characters >, <, and = are not underlined to avoid confusion with other symbols, such as the symbol for "greater than or equal to."*

Explanation of Format

The first operand (identifier-1, literal-1, or arithmetic-expression-1) is called the subject of the condition; the second operand (identifier-2, literal-2, or arithmetic-expression-2) is called the object of the condition.

The relational operator specifies the type of comparison to be made in a relation condition. When used, NOT and the next keyword or relational character form one relational operator that defines the comparison to be executed for a truth value. For example, NOT EQUAL is a truth test for an unequal comparison, and NOT GREATER is a truth test for an equal or less than comparison. Table 8–4 shows the meanings of the relational operators.

Table 8–4. Meanings of Relational Operators

Operator	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

Comparing Numeric Operands

For numeric operands, a comparison is made with respect to the algebraic value of the operands. The length of the literal or the arithmetic expression operands, in terms of number of digits represented, is not significant. Zero is considered a unique value regardless of the sign.

A comparison of these operands is permitted regardless of the manner in which the usage is described. Unsigned numeric operands are considered positive for purposes of comparison.

Comparing Nonnumeric Operands

For nonnumeric operands, or for one numeric and one nonnumeric operand, a comparison is made with respect to a specified collating sequence of characters. If one of the operands is specified as numeric, it must be an integer data item or an integer literal and the following conditions apply:

- If the nonnumeric operand is an elementary data item or a nonnumeric literal, the numeric operand is treated as if it were moved to an elementary alphanumeric data item of the same size as the numeric data item (in terms of standard data format characters). The numeric operand is then treated as if the contents of this alphanumeric data item were compared with the nonnumeric operand.
- If the nonnumeric operand is a group item, the numeric operand is treated as if it were moved to a group item of the same size as the numeric data item (in terms of standard data format characters). The numeric operand is then treated as if the contents of this group item were compared with the nonnumeric operand.
- A noninteger numeric operand cannot be compared to a nonnumeric operand.

- If one of the operands is an undigit literal, then it cannot be compared to a numeric operand. Otherwise, it is treated as if it were moved to an elementary alphanumeric data item of the appropriate length. Each pair of hexadecimal digits represents one 8-bit character. (This is an extension to ANSI X3.23-1974 COBOL.)

The size of an operand is the number of standard data-format characters in the operand. Numeric and nonnumeric operands can be compared only when the usage is the same.

Operands of equal size and operands of unequal size are described as follows:

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

  The first pair of unequal characters encountered is compared to determine their relative position in the collating sequence. The operand that contains the character positioned higher in the collating sequence is considered to be the greater operand.
- If the operands are of unequal size, comparison proceeds as if the shorter operand were extended on the right by sufficient spaces to make the operands of equal size.

For More Information

- For a detailed description of move operations, refer to "MOVE" in Section 9, "PROCEDURE DIVISION Statements."
- For information about the collating sequence, refer to "OBJECT-COMPUTER" in Section 5, "ENVIRONMENT DIVISION."
- For details on the character P, refer to "PICTURE Clause" in Section 7, "DATA DIVISION."

Comparing Kanji Operands

For Kanji operands, the relational operators are restricted to the following:

```
IS [NOT] EQUAL TO
IS [NOT] =
```

If the operands are of unequal size, the right side of the shorter operand is filled with Kanji space characters to make both operands of equal size before the comparison is made.

Comparing Index-Names and Index Data Items

Relation tests can be made between the following:

- 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 that corresponds 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.

Any result of the comparison between an index data item and a data item or a literal that is not specified in the preceding list is not allowed.

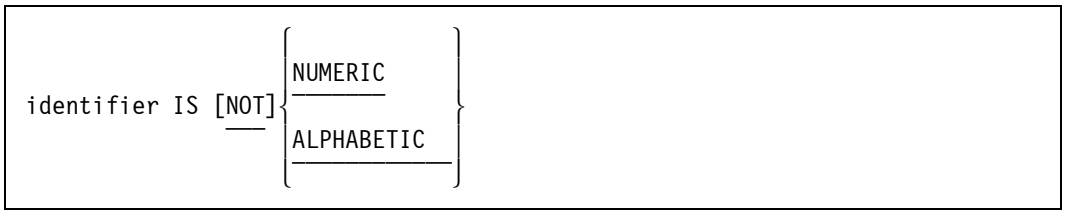
Class Condition

The class condition determines whether the operand is NUMERIC or ALPHABETIC. For example, you might want to use the class condition test to ensure that an item that is supposed to contain numeric data actually does contain numeric digits. You can code an IF statement to test for a class condition. The following is an example:

```
IF NEW-NUM IS NUMERIC
  PERFORM CALC-ROUTINE
  ELSE PERFORM ERROR-ROUTINE.
```

When the NUMERIC option is specified, the class condition tests whether the operand consists entirely of only the characters 0 through 9, with or without the operational sign.

When the ALPHABETIC option is specified, the class condition tests whether the characters are contained entirely in the alphabetic truthset. For most application programs, the alphabetic truthset includes the characters A through Z and the space. For programs using the internationalization features, the alphabetic truthset can be specified with respect to the system collating sequence. (The internationalization features are an extension to ANSI X3.23-1974 COBOL.)



Explanation of Format

The usage of the operand in a NUMERIC test must be DISPLAY or COMPUTATIONAL. (Allowing COMPUTATIONAL items in the class condition is an extension to ANSI X3.23-1974 COBOL.) The usage of the operand in an ALPHABETIC test must be DISPLAY.

When used, NOT and the next keyword specify a class condition that defines the class condition test to be executed for a truth value; for example, NOT NUMERIC is a truth test for determining that an operand is nonnumeric.

The NUMERIC test cannot be used with an item described as alphabetic in its data description or with an item described as a group item composed of elementary items with one or more operational signs.

If the data description of the item being tested does not indicate the presence of an operational sign, the item being tested is determined to be numeric only if the contents are numeric and an operational sign is not present. If the data description of the item indicates the presence of an operational sign, the item being tested is determined to be numeric only if the contents are numeric and a valid operational sign is present.

The ALPHABETIC test cannot be used with an item described as numeric in its data description. For most applications, the item being tested is determined to be alphabetic only if the contents consist of any combination of the alphabetic characters A through Z and the space. For applications using the internationalization features, the data item being tested is determined to be alphabetic only if the contents consist of any combination of the alphabetic characters in the truthset. To use a system collating sequence other than the characters A through Z and the space, the program must use the *alphabet-name IS CCSVERSION* phrase of the SPECIAL-NAMES paragraph. (The internationalization features are an extension to ANSI X3.23-1974 COBOL.)

For More Information

- The position and representation of valid operational signs are discussed under "PICTURE Clause" and "SIGN Clause" in Section 7, "DATA DIVISION."
- Refer to Section 16, "Internationalization," for information about truthsets.

Condition-Name Condition

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 a condition-name. The general format of the condition-name condition is as follows:

```
condition-name
```

Explanation of Format

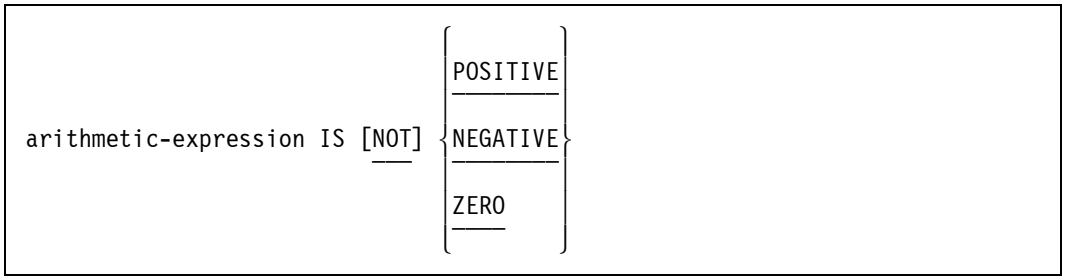
If the condition-name is associated with a range or ranges of values, then the conditional variable is tested to determine whether its value falls in this range, which includes the end values.

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

The result of the test is TRUE if one of the values corresponding to the condition-name equals the value of its associated conditional variable.

Sign Condition

The sign condition determines whether or not the algebraic value of an arithmetic expression is less than, greater than, or equal to 0.



Explanation of Format

When used, NOT and the next keyword specify a sign condition that defines the algebraic test to be executed for a truth value; for example, NOT ZERO is a truth test for a nonzero (positive or negative) value. An operand is positive if its value is greater than 0, negative if its value is less than 0, and 0 if its value is equal to 0. The arithmetic expression must contain at least one reference to a variable.

Event-Identifier Condition

In the event-identifier condition, an event variable, an event-valued file attribute, or an event-valued task attribute is tested to determine whether the condition is TRUE or FALSE.

Explanation of Format

The use of an event-identifier as a condition returns the value TRUE when the event has been caused and not reset. The event-identifier condition returns the value FALSE if the event is reset.

Complex Conditions

A complex condition is formed by combining simple conditions, combined conditions, and/or complex conditions with logical connectors (logical operators AND and OR) or by negating these conditions with a logical negator (the logical operator NOT). The truth value of a complex condition, whether parenthesized or not, is the truth value that results from the interaction of all the stated logical operators on the individual truth values of simple conditions or on the intermediate truth values of conditions logically connected or logically negated.

Table 8–5 shows the logical operators and explains their meanings.

Table 8–5. Logical Operators and Their Meaning

Logical Operator	Explanation
AND	Logical conjunction. The truth value is TRUE if both of the conjoined conditions are true, and the truth value is FALSE if at least one of the conjoined conditions is FALSE.
OR	Logical inclusive OR. The truth value is TRUE if at least one of the included conditions is TRUE, and the truth value is FALSE if both included conditions are FALSE.
NOT	Logical negation or reversal of the truth value. The truth value is TRUE if the condition is FALSE, and the truth value is FALSE if the condition is TRUE.

Negated Simple Conditions

A simple condition is negated by using the logical operator NOT.

The general format of a negated simple condition is as follows:

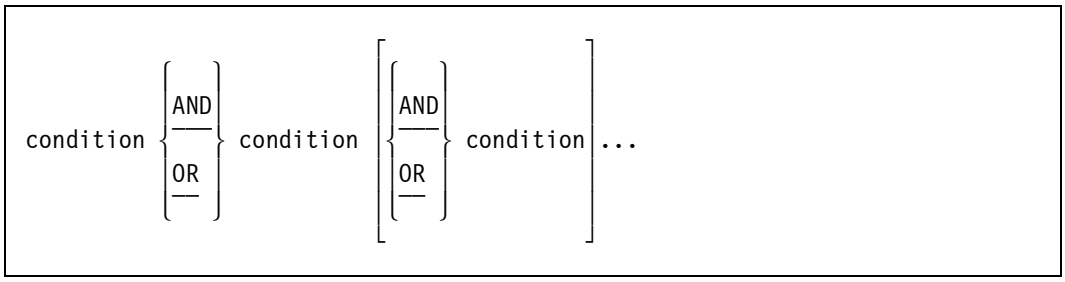
```
NOT simple-condition
```

Explanation of Format

The negated simple condition has a truth value opposite of the truth value for a simple condition. Thus, the truth value of a negated simple condition is TRUE if the truth value of the simple condition is FALSE; the truth value of a negated simple condition is FALSE only if the truth value of the simple condition is TRUE. Enclosing a negated simple condition in parentheses does not change the truth value.

Combined and Negated Combined Conditions

A combined condition results from connecting conditions with one of the logical operators AND or OR. The general format of a combined condition is as follows:



A condition can be any of the following:

- A simple condition
- A negated simple condition
- A combined condition
- A negated-combined condition
- A combination of the preceding conditions

The negated-combined condition is created if the NOT logical operator is followed by a combined condition enclosed in parentheses.

A combination of the preceding conditions must be specified according to the rules summarized in Table 8–6.

Although parentheses need never be used when either AND or OR is used exclusively in a combined condition, parentheses can be used to establish the precedence of operators when a mixture of the logical operators AND, OR, and NOT is used.

Table 8–6 shows the ways in which conditions and logical operators can be combined and parentheses used. A one-to-one correspondence must exist between left and right parentheses so that each left parenthesis is to the left of its corresponding right parenthesis.

Table 8–6. Combinations of Conditions, Logical Operators, and Parentheses

Base Element	First in Conditional Expression	Last in Conditional Expression	Left-to-Right Element Sequence Preceder	Left-to-Right Element Sequence Follower
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,)

Notes:

- *If a "left-to-right element sequence preceder" is not the first element, it must be preceded by the given example.*
- *If a "left-to-right element sequence follower" is not the first element, it must be followed by the given example.*

The element pair OR NOT is permissible, but the pair NOT OR is not permissible. NOT is permissible, but NOT NOT is not permissible.

Abbreviated Combined Relation Conditions

Combined relation conditions can be abbreviated when one of the following occurs:

- The sequence has no parentheses, and a succeeding relation condition with a subject is the same as the preceding relation condition.
- The sequence has no parentheses, and a succeeding relation condition with a subject and a relational operator is the same as the preceding relation condition.

One of the following abbreviations can be used for any relation condition except the first:

- Omit the subject of the relation condition.
- Omit the subject and relational operator of the relation condition.

The use of abbreviated relations in conditional statement sequences that contain parentheses is an extension to ANSI X3.23-1974 COBOL. When a condition is abbreviated, the subject and the relational operator are assumed, and they are declared immediately before the object for which the assumption was made. This situation occurs even if the expression in which the relational operator or the subject (or both) is embedded within a parenthetical expression.

Recommendations

- Avoid abbreviated combined relation conditions that require the extraction of a relational operator, a subject, or both from a parenthetical expression that does not include the object. For example,

```
IF (A = B AND C = D) OR E ...
```

- Avoid abbreviated combined relation conditions in which the subject to be implied immediately follows the logical operator AND (or the combination AND NOT). For example,

```
IF A = B AND C = D OR E ...
```

- Combined relation conditions without parentheses are always evaluated from left to right; the presence of an abbreviation in the condition does not override that rule. For example,

  The condition

```
IF A = B OR C = D OR E ...
```

  is evaluated the same as

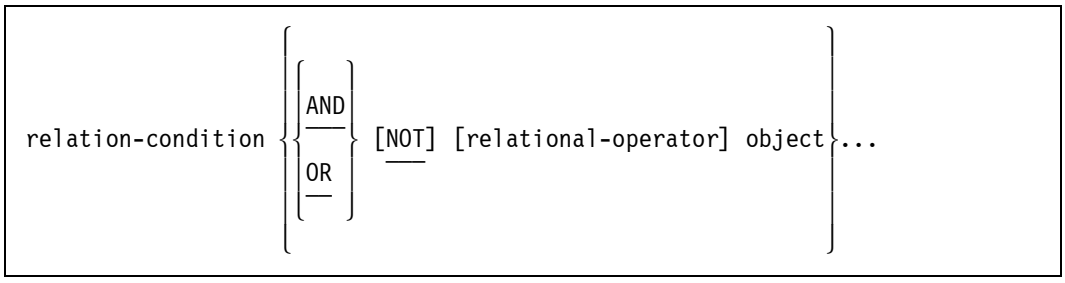
```
IF (A = B OR C = D) OR C = E ...
```

  and not as

```
IF A = B OR (C = D OR C = E) ...
```

  This last relation condition might be erroneously assumed on the basis of the presence of an abbreviation in the combined relation condition.

The format of an abbreviated combined relation condition is as follows:



Explanation of Format

In a sequence of relation conditions, both of the previously mentioned forms of abbreviation can be used. Using such abbreviations has the same effect as inserting the last stated subject in place of the omitted subject, and the last stated relational operator in place of the omitted relational operator. The result of such an implied insertion must comply with the rules listed in Table 8–6. The insertion of an omitted subject, an omitted relational operator, or both ends when a complete simple condition is encountered within a complex condition.

The interpretation applied to the use of the word NOT in an abbreviated combined relation condition is as follows:

- If the word or symbol immediately following NOT is GREATER, >, LESS, <, EQUAL, or =, then NOT is part of the relational operator.
- In any other situation, NOT is interpreted as a logical operator, causing the implied insertion of a subject or a relational operator to result in a negated relation condition.

Example

Table 8–7 contains some examples of abbreviated combined relation conditions and their expanded equivalents.

Table 8–7. Abbreviated Combined Relation Conditions

Abbreviated Combined Relation Condition	Expanded Equivalent
a > b AND NOT < c OR d	((a > b) AND (a NOT < c))
	OR (a NOT < d)
a NOT EQUAL b OR c	(a NOT EQUAL b) OR (a NOT EQUAL c)
NOT a = b OR c	(NOT (a = b)) OR (a = c)
NOT (a GREATER b OR < c)	NOT ((a GREATER b) OR (a < c))
NOT (a NOT > b AND c AND NOT d)	NOT (((a NOT > b) AND (a NOT > c))
	AND (NOT (a NOT > d)))

Condition Evaluation Rules

Parentheses can be used to define the order of evaluation for individual conditions of complex conditions when the implied evaluation precedence does not apply. Conditions in parentheses are evaluated first, and within nested parentheses, evaluation proceeds from the least inclusive condition to the most inclusive condition. When parentheses are not used, or parenthesized conditions are at the same level of inclusiveness, the following hierarchical order of logical evaluation is used until the final truth value is determined:

1. Values are established for arithmetic expressions.

2. Truth values for simple conditions are established in the following order:
   a. Relation (following the expansion of any abbreviated relation condition)
   b. Class
   c. Condition-name
   d. Switch-status
   e. Sign
3. Truth values for negated simple conditions are established.
4. Truth values for combined conditions are established: AND logical operators followed by OR logical operators.
5. Truth values for negated combined conditions are established.

   If the sequence of evaluation is not completely specified by parentheses, then consecutive operations of the same hierarchical level are evaluated from left to right.

For More Information

For information on the order in which elements of an expression are evaluated, refer to "Formation and Evaluation Rules" in this section.

Common Phrases in Statements

The ROUNDED phrase, the SIZE ERROR phrase, and the CORRESPONDING phrase can be used in a number of statements. Because the rules for these phrases are similar for each of the statements in which they are used, these phrases are explained only once. If there are unique aspects in the use of a phrase in a statement, the aspect is described in this section.

In the following paragraphs, a resultant identifier is the identifier associated with a result of an arithmetic operation.

ROUNDED Phrase

Truncation occurs if, after decimal point alignment, the number of places in the fractional portion of the result of an arithmetic operation is greater than the number of places provided for the fraction in the resultant identifier. The truncation occurs relative to the size provided for the resultant identifier in its PICTURE clause.

When rounding is requested, the absolute value of the resultant identifier is increased by adding the number 1 to its low-order digit whenever the absolute value of the next least-significant digit of the intermediate data item is greater than or equal to 5.

When the low-order integer positions in a resultant identifier are represented by the character P in the PICTURE clause for that resultant identifier, then rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

SIZE ERROR Phrase

The SIZE ERROR phrase enables you to specify procedures to be executed when a Size Error condition occurs. Size Error conditions exist under the following circumstances:

- If, after decimal point alignment, the absolute value of a result exceeds the largest value that can be contained in the associated resultant identifier.
- Division by 0 (zero) always causes a Size Error condition. If division by 0 is the cause of the Size Error condition, the program ends abnormally.

The Size Error condition applies only to the final results of an arithmetic operation and does not apply to intermediate results, except in the DIVIDE statement. If the ROUNDED phrase is specified, rounding takes place before checking for Size Error conditions. When such a Size Error condition occurs, subsequent action depends on whether or not the SIZE ERROR phrase is specified.

If the SIZE ERROR phrase is not specified and a Size Error condition occurs, the resultant value is stored in each of the receiving fields, and is left-truncated as required. If more than 23 significant digits are required, the program abnormally terminates with an error. Values of resultant identifiers for which no Size Error condition occurs are unaffected by size errors that occur for other resultant identifiers.

If the SIZE ERROR phrase is specified and a Size Error condition occurs, then the values of resultant identifiers affected by the size errors are not altered. Values of resultant identifiers for which no Size Error conditions occur are unaffected by size errors that occur for other resultant identifiers. After the arithmetic operation is completely executed, the imperative statement in the SIZE ERROR phrase is executed.

If an ADD CORRESPONDING or SUBTRACT CORRESPONDING statement produces a Size Error condition, the imperative statement in the SIZE ERROR phrase is not executed until all the individual additions and subtractions are completed.

CORRESPONDING Phrase

In the following description, AAA and BBB are identifiers that refer to group items. A pair of data items, one from AAA and one from BBB, correspond if all the following conditions are met:

- A data item in AAA and a data item in BBB are not designated by the keyword FILLER, and these data items have the same data-name and the same qualifiers up to, but not including, AAA and BBB.
- At least one of the data items is an elementary data item in a MOVE statement with the CORRESPONDING phrase. In an ADD or SUBTRACT statement with the CORRESPONDING phrase, both of the data items are elementary numeric data items.

- The description of AAA and BBB must not contain level-numbers 66, 77, or 88, or the USAGE IS INDEX clause.
- A data item subordinate to AAA or BBB that contains a REDEFINES, RENAMES, OCCURS, or USAGE IS INDEX clause is ignored, as are data items subordinate to the data item that contains the REDEFINES, OCCURS, or USAGE IS INDEX clause. However, AAA and BBB can have REDEFINES or OCCURS clauses, or can be subordinate to data items with REDEFINES or OCCURS clauses.

For information about repeated data items, refer to "OCCURS Clause" in Section 7, "DATA DIVISION."

Common Rules for Arithmetic Statements

ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT have the following in common:

- The data descriptions of the operands need not be the same; any necessary conversion and decimal point alignment are supplied throughout the calculation.
- The maximum size of each operand is 23 decimal digits. (ANSI X3.23-1974 extension.)
- Each arithmetic operation is evaluated using an intermediate data item for the result of the operation. If the number of significant digits in the result being developed is greater than the number of significant digits that can be held in an intermediate data item, the result is truncated to the size of an intermediate data item. The contents of the intermediate data item are moved to the resultant identifier according to the rules for the MOVE statement. Rounding is performed; the Size Error condition is determined only during this move. (This is an extension to ANSI X3.23-1974 COBOL.)

For More Information

- Refer to "MOVE" in Section 9, "PROCEDURE DIVISION Statements."
- Rounding and size conditions are described.
- For information on size conditions, see "SIZE ERROR Phrase" earlier in this section.

Calculating Multiple Results with One Arithmetic Statement

The ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements can produce multiple results. An arithmetic statement with multiple receiving fields produces results as if it had been written in the following way:

1. A statement performs all the operations necessary to arrive at the result that is ultimately to be stored in or combined with the receiving items, and stores that result in a temporary storage location.
2. A sequence of statements is performed such that for each of the multiple receiving fields, the value in that temporary location is stored in or combined with the value in the multiple receiving fields. This sequence of statements is considered to be written in the same left-to-right sequence in which the multiple receiving fields are listed in the original statement.

This sequence is required by ANSI for multiple-destination arithmetic statements (ADD, COMPUTE, DIVIDE, MULTIPLY and SUBTRACT), and the required sequence means that there are cases in which the results obtained with a multiple-destination arithmetic statement might differ from those produced with single-destination statements using the same expression and the same destinations.

For example, the result of the statement *ADD a, b, c TO c, d (c), e* is equivalent to the following statement. Temp is an intermediate result item provided by the compiler.

```
ADD a, b, c GIVING temp
ADD temp TO c
ADD temp TO d (c)
ADD temp TO e
```

Destination fields in multiple-destination arithmetic statements must all be fixed-point (including integer) or floating-point (REAL or DOUBLE). The compiler issues a syntax error if you mix floating-point destinations with fixed-point destinations.

If the destination fields are all floating-point (that is, USAGE REAL or USAGE DOUBLE), the intermediate data item is equivalent to a 77-level USAGE DOUBLE item.

If the destination fields are all fixed-point, the intermediate data item is equivalent to a 77-level USAGE BINARY item (not BINARY TRUNCATED) that is 23 digits and can contain 1 more digit to the right of the decimal point than the destination with the largest number of digits in its fractional part. This is done to ensure, wherever possible, that the intermediate result value is maintained as an integer regardless of the presence or absence of a ROUNDED clause on any destination.

This precision difference between the single- and multiple-destination statements might result in cases in which syntax errors or warnings occur in the multiple-destination cases that do not occur with single-destination equivalents. Also, arithmetic faults, rounding and truncation differences, and ON SIZE ERROR conditions might occur with multiple-destination statements that would not occur with single-destination statements that otherwise might be expected to produce equivalent results.

Since single-destination arithmetic statements often produce more precise results than multiple-destination arithmetic statements, it is suggested that you use the single-destination arithmetic forms rather than their multiple-destination equivalents if you are concerned about the potential for a precision difference, the increased risk of arithmetic faults such as INTEGER OVERFLOW, or the possibility of unexpected ON SIZE ERROR conditions or if you wish to avoid syntax errors or warning.

Handling Incompatible Data

When the contents of a data item referenced in the PROCEDURE DIVISION are not compatible with the class specified for that data item in the related PICTURE clause (except for the class condition), the result of such a reference is undefined.

For More Information

Details on numeric and alphabetic operands appear in “Class Condition” in this section.

I/O Exception Conditions

During the execution of an I/O statement, an exception condition that interrupts processing can happen. Often the exception is caused by an error in the program, but an exception can also occur when the compiler sends an informational message. The following mechanisms exist for handling I/O exception conditions in a program.

Mechanism	Explanation
Exception-handling directive	A direction to the compiler to perform a specified imperative statement when a particular condition occurs. The At End and Invalid Key conditions are examples of conditions to which imperative statements can be added to form an exception handling directive. For example, the AT END NEXT SENTENCE directive instructs the compiler to go to the next sentence when an At End condition occurs.
FILE STATUS clause	See I/O Status in Section 5, "ENVIRONMENT DIVISION."
USE procedures	See Format 1 of the USE verb in Section 9, "PROCEDURE DIVISION Statements."

Handling I/O Exception Conditions

You can use any of the three mechanisms to handle exceptions. The presence or absence of these mechanisms in the program impact the way the system responds to an I/O exception condition.

The following description of the process the system uses to handle exceptions explains the conditions in which each of the mechanisms becomes activated. You can then decide how best to handle exception conditions.

Programs with Exception-Handling Directives

The following table explains the process used by the compiler when the program contains an exception-handling directive.

IF the exception . . .	THEN the system takes the following actions:
Matches the directive	If the program contains a FILE STATUS clause for the file, the I/O status value is updated to reflect the exception. The imperative statement associated with the exception-handling directive is performed.
Does not match the directive	If the program contains a FILE STATUS clause for the file, the I/O status value is updated to reflect the exception. If a USE procedure is applicable, it is executed.

For a given I/O verb, the execution of the imperative statement associated with an exception-handling directive is mutually exclusive with the execution of a USE procedure applicable to the file and to the verb. In some contexts, the imperative statement is executed; in others, the USE procedure. No instance exists in which both are executed during the execution of the same COBOL statement.

Programs without Exception-Handling Directives

If the program omits an exception handling directive for the I/O verb, then the system takes the following action when an I/O exception condition occurs:

IF the program includes . . .	THEN the . . .
A FILE STATUS clause for the file	I/O status value is updated to reflect the exception.
An applicable USE procedure	USE procedure is executed.

Not all I/O verbs provide exception handling directives. For example, OPEN and CLOSE verbs lack them. When the program encounters I/O exception conditions during the execution of verbs for which exception handling directives are not available, the program must contain either USE procedures, the FILE STATUS clause, or both, if the program is to handle the condition.

Example

A *file not open* condition can occur on an OPEN AVAILABLE [EXTEND] verb. If the SELECT statement for the file does not contain a FILE STATUS clause, and DECLARATIVES contains no applicable USE procedure, then the program terminates if it encounters that condition. To handle the *file not open* condition, at least one of these mechanisms must be active at the time the OPEN verb is executed.

Not Handling I/O Exception Conditions

The system terminates the program if an I/O exception condition occurs on an I/O verb for which you have omitted all of the following mechanisms:

- An exception handling directive
- A FILE STATUS clause for the file
- An applicable USE procedure for the file in DECLARATIVES.

Functional Groupings of Verbs

The following lists categorize the COBOL74 verbs by function and summarize their purposes. Some verbs appear in more than one functional category.

PROCEDURE DIVISION Concepts

CALL	Passes control from a calling program to a called program.
CANCEL	Releases the memory areas occupied by the called program.
EXIT PROGRAM	Marks the logical end of a called program.
STOP RUN	Permanently ends the called program and all other programs in the run unit.

Arithmetic Verbs

ADD	Sums two or more numeric operands and stores the result.
COMPUTE	Assigns the value of an arithmetic expression to one or more data items.
DIVIDE	Divides a numeric operand into one or more other operands, and stores the quotient and remainder.
INSPECT (TALLYING)	Searches for and tallies the occurrences of specified characters in a data item.
MULTIPLY	Multiplies numeric operands and stores the result.
SUBTRACT	Subtracts one or the sum of two or more numeric operands from one or more items and stores the result.

Compiler-Directing Verbs

COPY	Incorporates text from a library program into a COBOL source program.
USE	Specifies procedures for handling input-output errors in addition to the standard procedures provided by the input-output control system.

Data Movement Verbs

ACCEPT (DATE, DAY, TIME, TIMER, TODAYS-DATE, TODAYS-NAME)	Makes low-volume data available to a specified data item. Data from the DATE, DAY, TIME, TIMER, TODAYS-DATE, or TODAYS-NAME register is moved to the specified item.
INSPECT (REPLACING)	Searches for and replaces occurrences of specified characters in a data item.
MOVE	Transfers data, according to the rules of editing, to one or more data areas.
STRING	Concatenates the partial or complete contents of one or more data items into a single data item.
UNSTRING	Causes contiguous data items in a sending field to be separated and placed into multiple receiving fields.

Imperative Verbs

ACCEPT	Makes low-volume data available to a specified data item from an operator display terminal (ODT).
ADD	Sums two or more numeric operands and stores the result.
ALLOW	Executes an interrupt procedure that has been attached to an EVENT item.
ALTER	Modifies the destination of a labeled GO TO statement.
ATTACH	Associates an interrupt procedure with an EVENT item.
CALL	Transfers control to a separate task or procedure.
CAUSE	Is used in communication between processes in an asynchronous processing environment to initiate specified events.
CHANGE	Modifies a file or a task attribute.
CLOSE	Ends the processing of a file, a reel, or a unit of a file. Also specifies the disposition of the file and of the device to which the file is assigned.

COMPUTE	Assigns the value of an arithmetic expression to one or more data items.
CONTINUE	Passes control to a previously called and exited synchronous process.
DELETE	Removes a record logically from a relative or indexed file.
DETACH	Dissociates a procedure from a task item or an EVENT item.
DISALLOW	Prevents execution of an interrupt procedure that has been attached to an event.
DISPLAY	Causes low-volume data to be transferred to an ODT.
DIVIDE	Divides a numeric operand into one or more other operands, and stores the quotient and remainder.
EXECUTE	Is synonymous with the RUN verb.
EXIT	Indicates a logical end for a series of sections or paragraphs referenced by a PERFORM statement.
EXIT PROCEDURE	Returns control from a bound procedure.
EXIT PROGRAM	Returns control from a dependent task to the program that initiated it.
GO TO	Transfers control unconditionally from one procedure to another. Control is not implicitly returned to the statement following the GO statement.
IF	Evaluates a condition. Subsequent action of the object program depends on whether the value of the condition is TRUE or FALSE.
INSPECT	Searches for and tallies or replaces specified characters in a data item.
LOCK	Enables a process to deny related processes access to a common storage area or to test a common storage area for a locked condition.
MERGE	Merges two or more identically sequenced files on a set of specified keys. The merged records then become available to an output procedure or an output file.
MOVE	Transfers data, according to the rules of editing, to one or more receiving data items.
MULTIPLY	Multiplies numeric operands and stores the result.
OPEN	Makes a file available for processing.
PERFORM	Transfers control unconditionally to one procedure or a group of consecutive procedures, and returns control to the statement following the PERFORM statement.
PROCESS	Initiates the parallel execution of another task.

READ	For sequential access, makes available the next logical record from a sequential file. For random access, makes available a specific record from a mass-storage file.
RELEASE	Transfers records to the initial phase of a sort operation, and writes records to a sort file.
RESET	Causes a specified event to be turned off in an asynchronous processing environment.
RETURN	Obtains sorted records from a sort operation or merged records from a merge operation.
REWRITE	Replaces a record logically in a mass-storage file.
RUN	Starts another program as an independent, asynchronous task.
SEARCH	Searches a table for a table element that satisfies a specified condition, and adjusts the associated index-name to indicate that table element.
SEEK	Repositions a mass-storage file to a specified record.
SET	Establishes reference points for table-handling operations by setting indexes associated with table elements. Also can alter the value of external switches and conditional variables.
SORT	Sequences the records in a file on a set of specified keys, and makes the sorted records available to output procedures or output files.
START	Positions records logically in a relative or an indexed file when the file is to be read sequentially.
STOP	Suspends a program either permanently or temporarily.
STRING	Concatenates the partial or complete contents of one or more data items into a single data item.
SUBTRACT	Subtracts one or the sum of two or more numeric operands from one or more items, and stores the results.
UNLOCK	Unlocks a common storage area so that related processes can access it.
UNSTRING	Causes contiguous data items in a sending field to be separated and placed into multiple receiving fields.
USE	Specifies procedures for I/O exception handling.
WAIT	Suspends program execution for a specified period of time.
WRITE	Releases a logical record for an output or an input-output file.

Input-Output Verbs

ACCEPT (identifier)	Transfers low-volume data from an ODT to a specified data item.
AWAIT-OPEN	Causes a port file subfile to wait for a start dialogue request from its correspondent endpoint.
CLOSE	Ends the processing of a file, and specifies the disposition of the file and of the device to which the file is assigned.
DELETE	Removes a logical record from a relative or an indexed file.
DISPLAY	Causes low-volume data to be transferred to an ODT.
OPEN	Makes a file available for processing.
READ (AT END, INVALID KEY)	For sequential access, makes available the next logical record from a sequential file. For random access, makes available a specific record from a mass-storage file.
RESPOND	Enables a program to accept or reject a request for a dialogue to be started or ended.
REWRITE	Replaces a record logically in a mass-storage file.
SEEK	Repositions a file to a specified record.
START	Provides a logical position for a relative or an indexed file when the file is to be read sequentially.
STOP (literal)	Suspends the execution of a program. The literal is communicated to the operator, and execution continues with the next executable statement in the program.
WRITE	Releases a logical record for an output or an input-output file.

Inter-Program Communication Verbs

CALL	Transfers control from one program to another.
CANCEL	Ensures that the next time a program referenced in a CALL statement is called, the program will be in its initial state.
EXIT PROGRAM	Indicates the logical end of a called program.

No Operation Verbs

CONTINUE	Indicates that no executable statement is present.
EXIT	Indicates a logical end to a series of sections or paragraphs referenced by a PERFORM statement.

Ordering Verbs

MERGE	Merges two or more identically sequenced files on a set of specified keys. The merged records then become available to an output procedure or an output file.
RELEASE	Transfers records to the initial phase of a sort operation, and writes records to a sort file.
RETURN	Causes the next record in a sort-merge file to be read.
SORT	Sequences a file on a set of specified keys, and makes the sort file available to output procedures or output files.

Report Writer Verbs

GENERATE	Links the PROCEDURE DIVISION to the Report Writer at process time.
INITIATE	Starts processing of a report.
TERMINATE	Terminates processing of a report.
USE BEFORE REPORTING	Specifies PROCEDURE DIVISION statements to be executed before a report group is produced.

Procedure-Branching Verbs

ALTER	Modifies the destination of a labeled GO TO statement.
CALL	Transfers control from one program to another during program execution.
EXIT	Indicates a logical end to a series of sections or paragraphs referenced by a PERFORM statement.
EXIT PROGRAM	Indicates the logical end of a called program.
GO	Transfers control unconditionally to a procedure-name. Control is not implicitly returned to the statement following the GO statement.

String-Handling Verbs

INSPECT (REPLACING, TALLYING)	Searches for and tallies or replaces the occurrences of specified characters in a data item.
STRING	Concatenates the partial or complete contents of one or more data items into a single data item.
UNSTRING	Causes contiguous data items in a sending field to be separated and placed into multiple receiving fields.

Table-Handling Verbs

SEARCH	Searches a table for a table element that satisfies a specified condition, and adjusts the associated index-name to indicate that table element.
SET	Establishes reference points for table-handling operations by setting indexes associated with table elements. Also alters the value of external switches and conditional variables.

Section 9
PROCEDURE DIVISION Statements

This section describes the statements in alphabetical order. The statements that provide an interface to other products (like Enterprise Database Server) are described in Volume 2.

ACCEPT

The ACCEPT statement transfers low-volume data to a data item. The incoming data is left-justified in the identifier, without consideration for editing, decimal point alignment, or operational sign position. Fill characters are inserted on the right if the size of the identifier is greater than the size of the incoming data. For identifiers of USAGE IS DISPLAY, the fill character is a space. For identifiers of USAGE IS COMPUTATIONAL, the fill character is 0 (zero).

Any necessary conversion of data from one form of internal representation to another takes place during the transfer.

Format	Explanation
1	Transfers data from a hardware device to a data item
2	Transfers data from date and time registers to a data item
3	Transfers a formatted system date or time to a data item based on the type, convention, and language in effect

Format 1

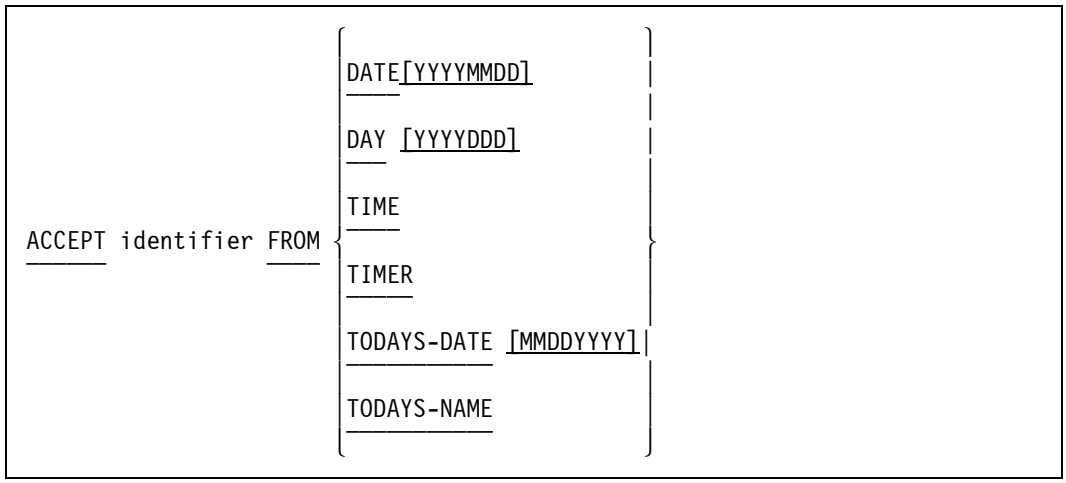
```
ACCEPT identifier [FROM mnemonic-name]
```

Explanation of Format 1

The ACCEPT statement transfers data from the hardware device. If the FROM phrase is not given, the device used is the ODT.

The mnemonic-name must also be specified in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION and must be associated with the hardware-name ODT.

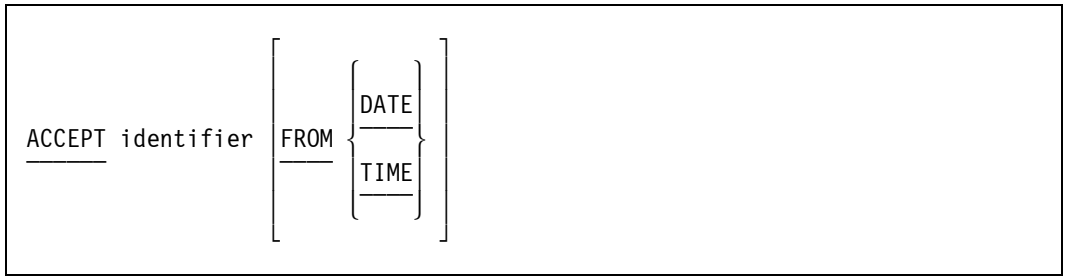
Format 2



Explanation of Format 2

The ACCEPT statement transfers the value of a special register to the data item that is specified by the identifier, according to the rules of the MOVE statement. The special registers and their uses and formats as required for the ACCEPT verb are explained in Table 2–4 in this document.

Format 3



Explanation of Format 3

Format 3 transfers the formatted system date or time to the data item specified by the identifier using the type, convention, and language in effect for the item. Format 3 is used when the identifier has an associated TYPE clause. If the convention or language are not declared for the item, the system use the default convention and language.

The FROM clause is optional and is used only for documentation. The specification of either DATE or TIME should match the type of the identifier. The DATE specification should be used when the receiving item is of type SHORT-DATE, LONG-DATE, or NUMERIC-DATE. The TIME specification should be used when the item is of type LONG-TIME or NUMERIC-TIME. If the type of the item and the special register do not match, the compiler issues a warning message, continues the compilation, and assumes the special register is valid for the type declared for the receiving item.

For More Information

- For the contents of each special register, see Section 2, "Language Elements."
- For information about the TYPE clause, refer to Section 7, "DATA DIVISION."
- For details on localizing a program, refer to Section 16, "Internationalization."

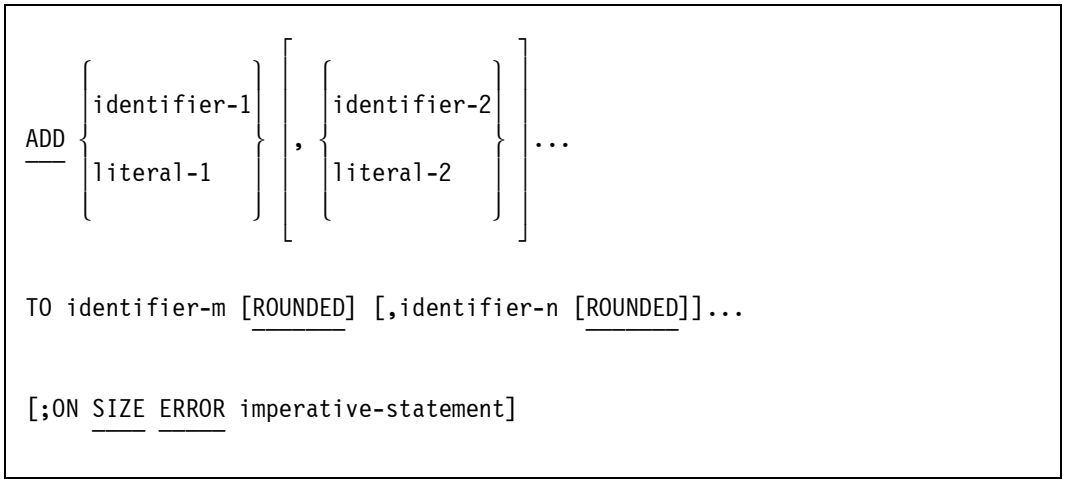
ADD

The ADD statement adds two or more numeric operands and stores the result.

When a sending item and a receiving item in the same ADD statement share a part—but not all—of their storage areas, the result of the execution of the statement is undefined.

Format	Explanation
1	The ADD...TO format stores the result of the add operation in the identifier following the word TO. The operands preceding the word TO are unchanged.
2	The ADD...GIVING format stores the result of the add operation in the identifier following the word GIVING. The values of the operands preceding the word GIVING are unchanged.
3	The ADD CORRESPONDING format adds the corresponding data items of two group items.

Format 1



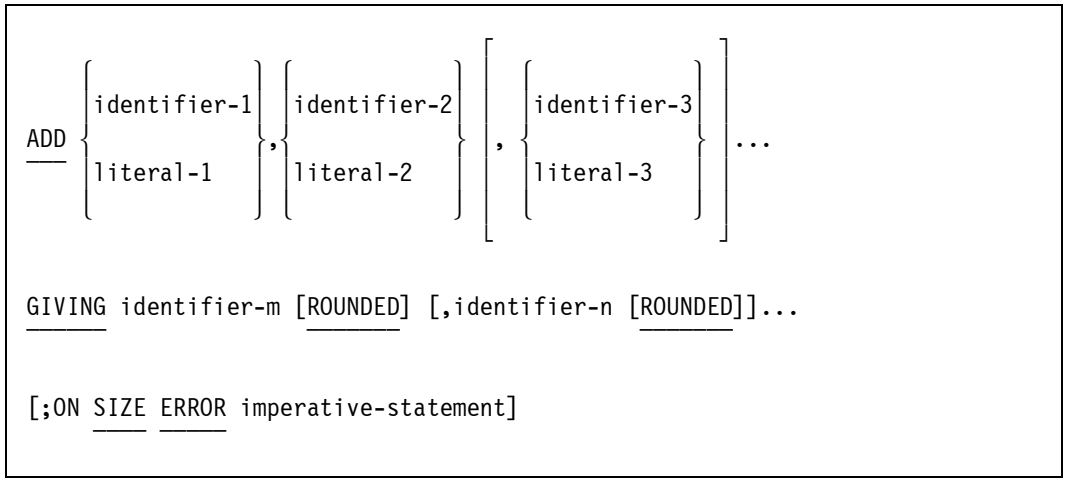
Explanation of Format 1

The values of the operands preceding the word TO are added together. The sum is then added to the current value of identifier-m, and the result is immediately stored in identifier-m. This process is repeated for each operand following the word TO.

Each identifier must refer to an elementary numeric item. Each literal must be a numeric literal.

The imperative-statement can be the NEXT SENTENCE phrase.

Format 2



Explanation of Format 2

The values of the operands preceding the word GIVING are added together. The sum is then stored as the value of each resultant identifier: identifier-m, identifier-n, and so on.

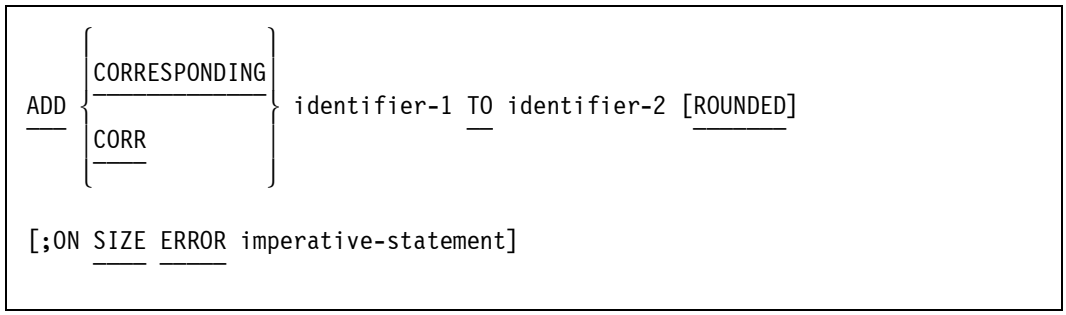
Each identifier preceding the word GIVING must refer to an elementary numeric item. Each identifier following the word GIVING must refer to either an elementary numeric item or an elementary numeric-edited item.

Each literal must be a numeric literal.

The imperative-statement can be the NEXT SENTENCE phrase.

You might want to use this format when you need to retain the contents of the operands.

Format 3



Explanation of Format 3

Data items in identifier-1 are added to and stored in corresponding data items in identifier-2. Each identifier must refer to a group item.

CORR is an abbreviation for CORRESPONDING.

The imperative-statement can be the NEXT SENTENCE phrase.

For More Information

- For a description of the features common to the arithmetic statements, refer to "Common Rules for Arithmetic Statements," in Section 8, "PROCEDURE DIVISION Concepts."
- For information on producing multiple results with one arithmetic statement, refer to "Calculating Multiple Results with One Arithmetic Statement" in Section 8, "PROCEDURE DIVISION Concepts."
- For information about the rounding of arithmetic result fields, refer to "ROUNDED Phrase" in Section 8, "PROCEDURE DIVISION Concepts."
- For information on Size Error conditions, refer to "SIZE ERROR Phrase" in Section 8, "PROCEDURE DIVISION Concepts."
- For information on adding group items, refer to "CORRESPONDING Phrase" in Section 8, "PROCEDURE DIVISION Concepts."

ALLOW (Extension to ANSI X3.23-1974 COBOL)

The ALLOW statement executes an interrupt procedure that has been attached to an EVENT item.

Format	Explanation
1	Causes the specified interrupt procedures to be specifically allowed
2	Enables you to remove the General Disallow Interrupt condition

Format 1

```
ALLOW section-name-1 [,section-name-2]...
```

Explanation of Format 1

Execution of an *ALLOW section-name* statement causes the specified interrupt procedures (section-names) to be specifically allowed. Whenever their attached events are caused, they are executed unless a General Disallow Interrupt condition is in effect.

Section-names used in this statement must be defined in the DECLARATIVES SECTION with the USE AS INTERRUPT clause.

Format 2

```
ALLOW INTERRUPT
```

Explanation of Format 2

The ALLOW INTERRUPT statement is the logical opposite of the DISALLOW INTERRUPT statement. It removes the General Disallow Interrupt condition. Interrupt procedures that are queued because of the General Disallow Interrupt condition are executed immediately after the ALLOW INTERRUPT statement is executed, unless their current status is specifically disallowed.

General Rules

The following information applies to both formats.

Interrupts previously disallowed by a *DISALLOW section-name* statement can be allowed by an *ALLOW section-name* statement during the time a General Disallow Interrupt condition is in effect. If the associated event is caused after an interrupt procedure has been specifically allowed, the interrupt procedure is queued until an ALLOW INTERRUPT statement removes the General Disallow Interrupt condition.

Interrupt procedures can be successfully allowed or disallowed regardless of whether they are attached to an event. However, performing an ATTACH statement for any interrupt procedure that has not been specifically disallowed causes an Automatic Implicit Allow condition for that procedure.

For More Information

- For information about associating an interrupt procedure with an EVENT item, refer to "ATTACH (Extension to ANSI X3.23-1974 COBOL)" in this section.
- For information about dissociating a procedure from a task item or an EVENT item, refer to "DETACH (Extension to ANSI X3.23-1974 COBOL)" in this section.
- For information about preventing execution of an interrupt procedure that has been attached to an event, refer to "DISALLOW" in this section.
- For information about specifying interrupt procedures, refer to Format 4 of "USE" in this section.

ALTER

The ALTER statement modifies a labeled GO TO statement.

```
ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2

[,procedure-name-3 TO [PROCEED TO] procedure-name-4]...
```

Explanation of Format

Execution of the ALTER statement modifies the GO TO statement in the paragraph named procedure-name-1, procedure-name-3, and so forth. The subsequent executions of the modified GO TO statements cause control to be transferred to procedure-name-2, procedure-name-4, and so forth. Modified GO TO statements in independent segments can, under some circumstances, be returned to their initial states.

Each procedure-name-1, procedure-name-3, and so forth is the name of a paragraph that contains a single sentence consisting of a GO TO statement without the DEPENDING phrase.

Each procedure-name-2, procedure-name-4, and so forth is the name of a paragraph or section in the PROCEDURE DIVISION.

ATTACH (Extension to ANSI X3.23-1974 COBOL)

The ATTACH statement associates an interrupt procedure with an EVENT item.

```
ATTACH section-name TO event-identifier
```

Explanation of Format

The section-name must be the name of a section in the DECLARATIVES SECTION that specifies the USE AS INTERRUPT clause.

The event-identifier cannot be an event-valued file attribute.

When an EVENT item is caused, all interrupt procedures that are then attached to that event-identifier are either executed immediately (interrupting the main program) or queued. If an interrupt procedure is allowed specifically, and if the General Disallow Interrupt condition is not in effect, the execution occurs. If an interrupt procedure is disallowed specifically, or if the General Disallow Interrupt condition is in effect, the execution is queued.

An ATTACH statement for an interrupt procedure that has not been specifically disallowed causes that procedure to be implicitly and automatically allowed. Execution of a subsequent *DISALLOW section-name* statement inhibits the immediate execution of that interrupt procedure so that when the associated event is caused, execution is queued. This specific *DISALLOW section-name* statement remains in effect until a specific *ALLOW section-name* statement is executed, even if that section-name has intervening DETACH and ATTACH statements.

Two or more interrupt procedures can be attached to a single event-identifier. The order of execution of these interrupt procedures when the event is caused is the reverse of their order of attachment.

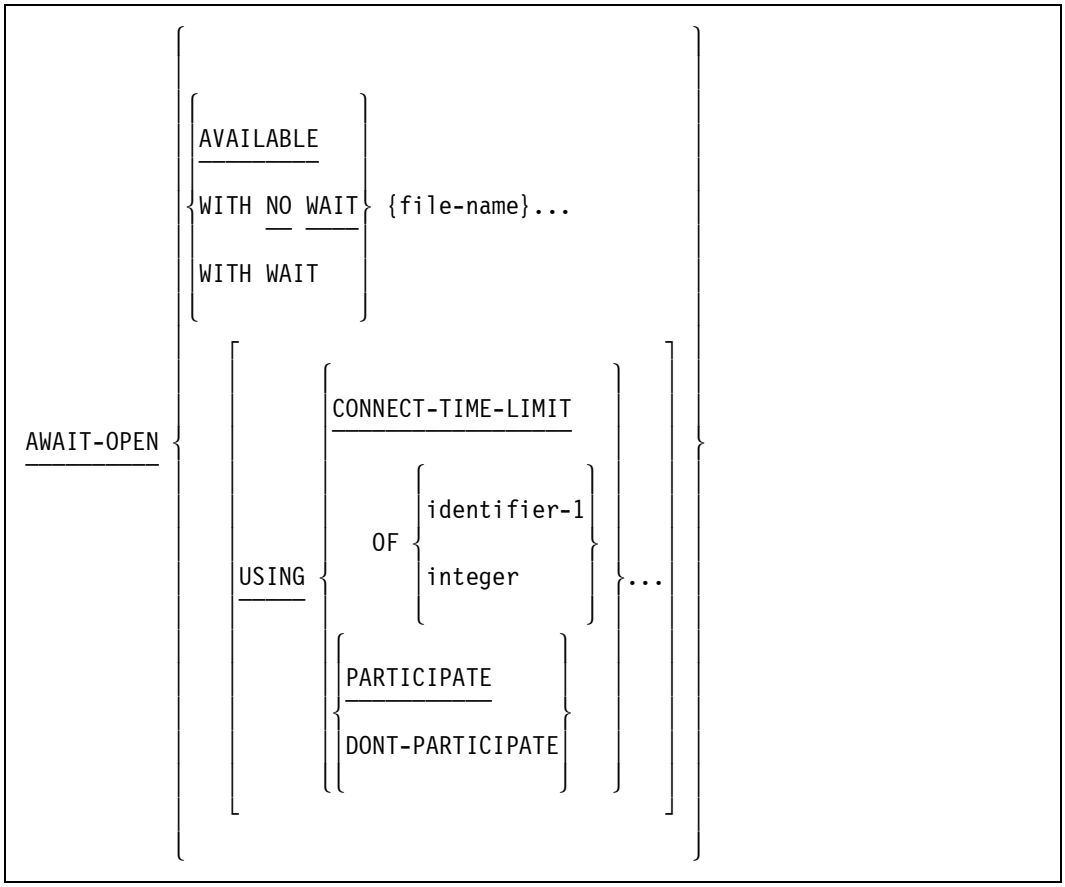
A particular interrupt procedure can be attached to only one event at any one time. If the interrupt procedure is already attached to an event when the ATTACH statement is executed, the interrupt procedure is automatically detached from the old event and then attached to the new event. Any queued invocations of the interrupt procedure are lost.

For More Information

- For information about executing an interrupt procedure that has been attached to an EVENT item, refer to "ALLOW (Extension to ANSI X3.23-1974 COBOL)" in this section.
- For information about dissociating a procedure from a task item or an EVENT item, refer to "DETACH (Extension to ANSI X3.23-1974 COBOL)" in this section.
- For information about preventing execution of an interrupt procedure that has been attached to an event, refer to "DISALLOW" in this section.
- For information about specifying interrupt procedures, refer to Format 4 of "USE" in this section.

AWAIT-OPEN (Extension to ANSI X3.23-1974 COBOL)

The AWAIT-OPEN statement is used only for port files. This statement causes a subfile to wait for a start dialogue request from its correspondent endpoint. The AWAIT-OPEN statement does not initiate a request for a dialogue to be established.



Explanation of Format

The file-name identifies one or more port files.

The AVAILABLE phrase opens a subfile if a matching subfile has already initiated a request for a dialogue to be started. If no request has been sent, the AWAIT-OPEN statement fails and the subfile is not considered for subsequent matching. The AVAILABLE phrase is a way of indicating an OPEN WAIT statement with the AVAILABLEONLY file attribute set to TRUE.

The NO WAIT phrase returns control to the program as soon as possible. The subfile waits for a request for a dialogue to be initiated while the program continues processing.

The WAIT phrase suspends the program until an attempt to match the subfile with an incoming dialogue request either succeeds or fails. If the program specifies subfiles, the AWAIT-OPEN WAIT statement suspends the program until the open operation on each affected subfile succeeds or fails. The AWAIT-OPEN WAIT statement is the default statement and is used if no OPEN option is specified in your program.

The USING option enables you to use each of the USING clauses once only.

The CONNECT-TIME-LIMIT phrase indicates the time in minutes that your program allows for the system to match a subfile with a dialogue request. The time in minutes is indicated by the value contained in identifier-1 or integer. The AWAIT-OPEN operation fails if a companion subfile does not initiate a dialogue request in that amount of time. If you do not use the CONNECT-TIME-LIMIT phrase or if you specify a value of 0 (zero), your program allows an indefinite amount of time. An error results if you specify a negative or noninteger value.

Identifier-1 designates an elementary integer data item. Integer specifies an integer numeric literal.

The PARTICIPATE phrase specifies that your program uses the RESPOND statement to accept or reject an offer for a dialogue to be started or ended.

The DONT-PARTICIPATE phrase specifies that your program unconditionally accepts all offers to open or close a dialogue with a correspondent endpoint.

General Rules

You must use the RESERVE NETWORK clause in the SPECIAL-NAMES paragraph for the compiler to recognize AWAIT-OPEN as a reserved word.

The ACTUAL KEY clause specifies the subfile that is awaiting an open request. If you do not code an ACTUAL KEY clause, the compiler assumes the number of subfiles is equal to the value of the MAXSUBFILES attribute. If you are using multiple subfiles, do the following:

1. Specify the total number of subfiles in your program by using the *CHANGE ATTRIBUTE MAXSUBFILES TO VALUE attribute-value* statement.
2. Specify the subfiles that are to await an open operation with the *SELECT port-file ASSIGN TO PORT; ACTUAL KEY IS subfile-num* clause.
3. Declare subfile-num and attribute-value in the WORKING-STORAGE SECTION.

Table 9–1 shows the way the value specified in the ACTUAL KEY clause determines which subfile awaits an open request.

Table 9–1. Designating Subfiles for the AWAIT-OPEN Statement

Value	Explanation
0 or none	Every closed subfile awaits an open request.
Nonzero	The specified subfile awaits an open request.
Greater than the MAXSUBFILES value, or a negative number	A run-time error occurs.

The system returns a value that indicates the result of an AWAIT-OPEN statement. You can access this value by including a *SELECT file-name FILE STATUS IS data-name* clause in your program. The operating system moves a value into the data-name storage area after the program performs the AWAIT-OPEN statement. You can then use an IF statement to test the value of the data-name and take the desired action, depending on the result. If you choose not to code actions for the AWAIT-OPEN results, the system provides a default action for each result.

Table 9–2 shows the I/O status values and their meanings.

Table 9–2. I/O Status Values for the AWAIT-OPEN Statement

Value	Explanation
00	Control was returned to the program after the AWAIT-OPEN statement completed correctly.
83	An error was detected during the execution of the AWAIT-OPEN statement.

Note: *The I/O status value "83" is an extension to ANSI X3.23-1974 COBOL.*

Examples

In the following examples, it is assumed that earlier in the program the port files were declared by using an ACTUAL KEY clause to specify a subfile index and that the subfile index was set to a particular subfile.

Example 9–1 indicates the program is suspended until the operation fails or until the subport on the port file PORTFILE1 receives an incoming dialogue request that matches and establishes a dialogue with the corresponding endpoint.

```
AWAIT-OPEN WITH WAIT PORTFILE1.
```

Example 9–1. Coding an AWAIT-OPEN WITH WAIT Statement

Example 9–2 indicates that control is returned to the program as soon as the AWAIT-OPEN statement is checked for semantic correctness. The subfile for port file PORTFILE1 awaits dialogue establishment while the program continues running.

```
AWAIT-OPEN WITH NO WAIT PORTFILE1.
```

Example 9–2. Coding an AWAIT-OPEN WITH NO WAIT Statement

Example 9–3 is the same as an AWAIT-OPEN WAIT statement with the AVAILABLEONLY file attribute set to TRUE. When AVAILABLEONLY is TRUE, the AWAIT-OPEN request is matched to dialogue requests that have already been received. If no matching requests have been received, the AWAIT-OPEN statement fails and the subfile is not considered for subsequent matching.

```
AWAIT-OPEN AVAILABLE PORTFILE1.
```

Example 9–3. Coding an AWAIT-OPEN AVAILABLE Statement

Example 9–4 indicates that the program waits for a request for a dialogue to be established on the subfile for the port file PORTFILE2. Control is not returned to the program until the subfile is matched to an incoming dialogue request. The PARTICIPATE option indicates that the dialogue request is not automatically accepted when it is matched to the subfile. When the system returns control to the program, the program can review and negotiate dialogue attributes, and accept or reject any offers by responding with the RESPOND statement.

```
AWAIT-OPEN WITH WAIT PORTFILE2 USING PARTICIPATE.
```

Example 9–4. Coding an AWAIT-OPEN...PARTICIPATE Statement

Example 9–5 indicates that the program does not wait for a dialogue request to be established on the subfile for the port file PORTFILE3. Control is returned to the program as soon as possible. In addition, the maximum amount of time that the system can wait for a successful match is equal to the value contained in the NUMER-ITEM item.

```
AWAIT-OPEN NO WAIT PORTFILE3 USING CONNECT-TIME-LIMIT OF NUMER-ITEM.
```

Example 9–5. Coding an AWAIT-OPEN...CONNECT-TIME-LIMIT Statement

For More Information

- For information about another type of open operation used to establish a dialogue, refer to “OPEN” in this section.
- For step-by-step information on coding port file applications, refer to the *I/O Subsystem Programming Guide.*
- For references to file attributes, refer to the *File Attributes Reference Manual.*

CALL

This statement transfers control from an object program to a separate task or procedure.

Format	Explanation
1	Starts an independently compiled program that is executed under the control of the calling program
2	Calls a bound-procedure
3	Starts a system dump operation
4	Initiates an independent task by submitting a Work Flow Language (WFL) job

For More Information

- For information on the use of the CALL statement in Inter-Program Communication (IPC), refer to "CALL Statement" in Section 13, "ANSI Inter-Program Communication (IPC)."
- For information on the use of the CALL statement with libraries, refer to "Referring to a Library" in Section 15, "Libraries."

Format 1 (Extension to ANSI X3.23-1974 COBOL)

```
CALL task-identifier WITH section-name [USING actual-parameter-list]
```

Explanation of Format 1

Format 1 is used for calls on tasks. A task is an independently compiled program that is executed under the control of the calling program.

The execution of the tasking CALL statement causes the program containing the CALL statement to be suspended and causes the program being called to begin execution as a separate task.

Upon execution of an EXIT PROGRAM statement in the called program, the called program is suspended and control is returned to the next statement of the calling program. The calling program can call the called program again with a CONTINUE statement, and shared data is not reinitialized. The called program begins execution at its first executable statement with each CONTINUE statement.

Execution of an EXIT PROGRAM RETURN HERE statement in the called program suspends the called program and returns control to the next statement of the calling program. Shared data is not reinitialized. A subsequent CONTINUE statement in the calling program returns control to the next statement after the EXIT statement in the called program.

In the called program, execution of a STOP RUN statement or abnormal termination returns control to the next statement of the calling program. Subsequent calls reinitialize the called program. The inclusion of a task-identifier in the USING clause allows a called program to make any reference to that task that is allowable in the calling program.

Files to be passed as parameters must have a record description. The record described for the file can be passed as a parameter. In the PROCEDURE DIVISION header of the called program, the USING phrase must not reference any data item in the FILE SECTION of the called program. Either or both programs can initiate I/O to the file passed as a parameter in the CALL statement.

The name of the program to be called can either be specified at the source level in the SPECIAL-NAMES paragraph or be assigned from a data item at run time by moving or reading the title into the data item named in the USE EXTERNAL statement of section-name. Standard file-naming conventions apply.

Each identifier in the *USING actual-parameter-list* clause must be defined as either a 77-level item that resides in the stack or a 01-level item. Items must correspond in level-number, usage, and size to those described in the corresponding positions of the USING clause of the PROCEDURE DIVISION in the called program. The exceptions are DISPLAY, COMP, BINARY group items, and 77-level BINARY items with RECEIVED BY REFERENCE clauses. These items are interchangeable as parameters of tasking CALL statements only; that is, each item can be passed to and received by the other items when the system is doing tasking calls. However, the lengths must be the same, or run-time errors, such as INVALID INDEX, occur.

The identifiers in the USING clause can be any combination of data items, task (control point) items, INDEX items, EVENT items, or lock items at either the group or elementary level.

The USING clause is included in the CALL statement if there is a USING clause in the USE statement of section-name, in the PROCEDURE DIVISION header of the called program, or both. The number, type, and order of items in each USING clause must be identical, except that DISPLAY, COMP, BINARY, REAL, and DOUBLE group items, and 77-level BINARY REAL, and DOUBLE items with RECEIVED BY REFERENCE clauses are interchangeable as parameters of tasking CALL statements. That is, each item can be passed to and received by the other. You must take care to ensure that the lengths of different types of items are the same, or run-time errors might occur.

Table 9–3 describes the matching of formal parameters between the COBOL74, ALGOL, and COBOL68 languages.

Table 9–3. Parameter Mapping for Tasking Calls

COBOL74 Parameter	ALGOL Parameter	COBOL68 Parameter
77-level BINARY or REAL item (single precision)	REAL INTEGER	77-level COMP or COMP-4 item (single precision)
77-level BINARY or DOUBLE item (double precision)	DOUBLE	77-level COMP or COMP-5 item (double precision)
01-level DISPLAY, COMP, BINARY, REAL, or DOUBLE item	REAL ARRAY [*]	01-level DISPLAY, COMP, or COMP-2 group item with or without LOWER-BOUNDS
	INTEGER ARRAY [*]	
	EBCDIC ARRAY [*]	
	HEX ARRAY [*]	
	REAL ARRAY [0]	
	INTEGER ARRAY [0]	
	EBCDIC ARRAY [0]	
	HEX ARRAY [0]	
77-level EVENT or LOCK item	EVENT	77-level EVENT or LOCK item
77-level or 01-level TASK elementary item	TASK	77-level or 01-level TASK elementary item
01-level EVENT or LOCK group item	EVENT ARRAY	77-level EVENT or LOCK group item
01-level TASK group item	TASK ARRAY	01-level TASK group item
FILE	FILE	FILE

The following notes describe the use of actual parameters and their correspondence to ALGOL variable types:

- BINARY items are single precision when they are declared with a length of 11 digits or less. These items are double precision if they are declared with a length of 12 to 23 digits. While BINARY items can match ALGOL REAL or INTEGER parameters, they are maintained in binary-coded format; their values are maintained with exponents of zero and no fractional parts.
- A BINARY 01-level group item with elements that are double precision (length of 12 to 23 digits) is not represented as a double array. Each BINARY01-level item is a real array. Two array elements are used for each item with a length of 12 to 23 digits (double) and one array element is used for each item with a length of 1 to 11 digits (single precision). ALGOL programs accessing the double-precision elements must use a double array equated to the real array that was used to receive or send the parameter. The ALGOL real array must be long enough to allow two elements for each item with a length of 12 to 23 digits.

- 77-level DISPLAY and COMP items are not allowed as parameters in COBOL74 tasking.
- An ALGOL subprogram that declares global libraries can be bound to a COBOL74 host program. However, all entry points of the libraries must be used in CALL statements in the COBOL74 program for the program to bind successfully.

  Any array row passed to or from an ALGOL-like language is considered a 01-level record in COBOL74. For example, consider an ALGOL program with an external procedure declaration specifying a formal parameter EBCDIC ARRAY A[*]. If this declaration is used to call a COBOL74 program that has a formal DISPLAY item parameter, the ALGOL-supplied array row is treated as a DISPLAY item of whatever length is declared in the COBOL74 program; the array row must be the declared length. The lower-bound could have been fixed when specified in the ALGOL procedure declaration (for example, A[0] or A[1]), instead of variable bound (A[*]); COBOL74 is not affected by the ALGOL lower-bound.
- In the opposite direction,(a COBOL74 item passed to an ALGOL procedure), the COBOL74 item is treated as an ALGOL array row; its length is determined by the COBOL74 declaration. If the ALGOL procedure specifies a variable bound (A[*]), the actual value of the lower-bound is 0 (zero).

The Work Flow Language (WFL) calls programs as tasks, passes its string parameters as real arrays with lower-bounds, and passes its numeric variable and constant parameters by value as real, single-precision operands. Table 9–4 shows WFL parameters and the corresponding COBOL74 parameters.

Table 9–4. WFL and COBOL74 Parameters

WFL Parameter	COBOL74 Parameter
STRING or STRING constant	DISPLAY, COMP or BINARY group item
REAL, INTEGER, REAL constant or INTEGER constant	77-level BINARY or REAL item (single precision)

Format 2 (Extension to ANSI X3.23-1974 COBOL)

```
CALL section-name [USING actual-parameter-list]
```

Explanation of Format 2

Format 2 invokes as a procedure an externally compiled program that must have been bound into the calling program. The procedure is bound into the calling object code and is physically part of the program itself.

The actual parameter-list must consist of data items, control items, and expressions optionally separated by commas.

Table 9–5 shows the formal parameters for bound and host programs that can be declared in COBOL74, along with the corresponding declarations in ALGOL and the permissible kinds of actual parameters that can be passed.

Note that if the formal parameter in COBOL74 or COBOL68 is a 77-level BINARY item, the actual parameter can be the value of an arithmetic expression. A single item of any numeric type can be considered an arithmetic expression for this purpose.

Table 9–5. Parameters for Bound and Host Programs

COBOL74 Formal Parameter	ALGOL Formal Parameter	Permissible Actual Parameter
BINARY, 77 1-11 digits (RECEIVED BY CONTENT)	INTEGER	Arithmetic-expression
REAL, 77 (RECEIVED BY CONTENT)	REAL	Arithmetic-expression
BINARY, 77 12-23 digits or DOUBLE, 77 (RECEIVED BY CONTENT)	DOUBLE	Arithmetic-expression
BINARY, 77 1-11 digits (RECEIVED BY REFERENCE)	INTEGER	BINARY, 77 1-11 digits
REAL, 77 (RECEIVED BY REFERENCE)	REAL	REAL, 77
BINARY, 77 12-23 digits (RECEIVED BY REFERENCE)	DOUBLE	BINARY, 77 12-23 digits
DOUBLE, 77 (RECEIVED BY REFERENCE)	DOUBLE	DOUBLE, 77
EVENT or LOCK, 77	EVENT	EVENT or LOCK, 77
TASK, 77 or 01	TASK	TASK, 77 or 01
BINARY, 01	INTEGER ARRAY	BINARY, COMP, or DISPLAY, 01
REAL, 01	REAL ARRAY	REAL, 01
DOUBLE, 01	REAL ARRAY	REAL or DOUBLE, 01
COMP or DISPLAY, 01	EBCDIC ARRAY	BINARY, COMP, or DISPLAY, 01
EVENT or LOCK, 01 group	EVENT ARRAY	EVENT or LOCK, 01 group
TASK, 01 group	TASK ARRAY	TASK, 01 group
FILE	FILE	FILE

Format 3 (Extension to ANSI X3.23-1974 COBOL)

The execution of a CALL SYSTEM DUMP statement causes control to pass to the DUMP routine in the operating environment.

```
CALL SYSTEM DUMP
```

Explanation of Format 3

The CALL SYSTEM DUMP statement causes the operating system to dump data from the memory area of the program.

Format 4 (Extension to ANSI X3.23-1974 COBOL)

```
                          ┌              ┐
                          │identifier-1  │
CALL SYSTEM WFL USING     {              }
                          │literal-1     │
                          └              ┘
```

Explanation of Format 4

The CALL SYSTEM WFL statement causes an independent task to be initiated by submitting a Work Flow Language (WFL) job to the WFL compiler. After initiating the task, the executing program does not wait for the initiated task to be completed but immediately proceeds to execute the next statement.

The CALL SYSTEM WFL statement requires one parameter that can be a nonnumeric literal or the name of a 01-level data item with the USAGE IS DISPLAY phrase. The contents of the parameter must be a complete WFL job. (Refer to the *WFL Reference Manual* for information on WFL jobs.) The syntax of the WFL job deck is not checked by the COBOL compiler; thus any error detected by the WFL compiler has no direct effect on the calling program.

CAUSE (Extension to ANSI X3.23-1974 COBOL)

The CAUSE statement is normally used for communication between processes in an asynchronous processing environment.

```
CAUSE [AND RESET] event-identifier-1 [,event-identifier-2]...
```

Explanation of Format

The CAUSE statement initiates the events specified by the event-identifiers. If any process is in a Suspended condition because the program encountered a *WAIT event-identifier* statement that specified one of these event-identifiers, it resumes processing. If any process has an interrupt procedure attached to one of these events, the interrupt mechanism is initiated.

When the AND RESET phrase is specified, all the event-identifiers are SET and then immediately RESET. This action indicates to any process that has an interrupt procedure attached to one of the event-identifiers that the event has been caused.

Event-identifiers must be one of the following:

- Properly qualified and subscripted data-names with the USAGE EVENT phrase specified
- File or task attributes of type EVENT

For More Information

- For information about executing an interrupt procedure that has been attached to an EVENT item, refer to “ALLOW (Extension to ANSI X3.23-1974 COBOL)” in this section.
- For information about associating an interrupt procedure with an EVENT item, refer to “ATTACH (Extension to ANSI X3.23-1974 COBOL)” in this section.
- For information about dissociating a procedure from a task item or an EVENT item, refer to “DETACH (Extension to ANSI X3.23-1974 COBOL)” in this section.
- For information about preventing execution of an interrupt procedure that has been attached to an event, refer to “DISALLOW” in this section.
- For information about suspending the execution of an object program, refer to “WAIT (Extension to ANSI X3.23-1974 COBOL)” in this section.

CHANGE (Extension to ANSI X3.23-1974 COBOL)

The CHANGE statement modifies a file attribute or a task attribute.

Those attributes that cannot be changed at any time cannot be specified in a CHANGE statement.

Attempts to change attributes to illegal values are ignored.

Certain attributes cannot be changed while the file is in open mode. Attempts to change these attributes while the file is open are ignored.

Certain file attributes are also used by the compiler to implement various constructs in the declaration of and access to files within the program. When a file attribute can be set or declared using standard COBOL syntax, it is always preferable to use the standard syntax because changing the attribute can lead to unexpected results in cases when the attribute is also used or altered by the compiler.

The CHANGE statement has the following three formats:

Format	Explanation
1	Changes a numeric or alphanumeric file attribute in relation to the STATIONLIST file attribute for remote files
2	Changes any file attribute
3	Changes any task attribute

Format 1

```
CHANGE attribute STATIONLIST of file-name
──────

┌         ┐ ┌              ┐
│UP BY    │ │literal-1     │
└── ──    ┘ └              ┘
┌         ┐ ┌              ┐
│DOWN BY  │ │identifier-1  │
└──── ──  ┘ └              ┘
```

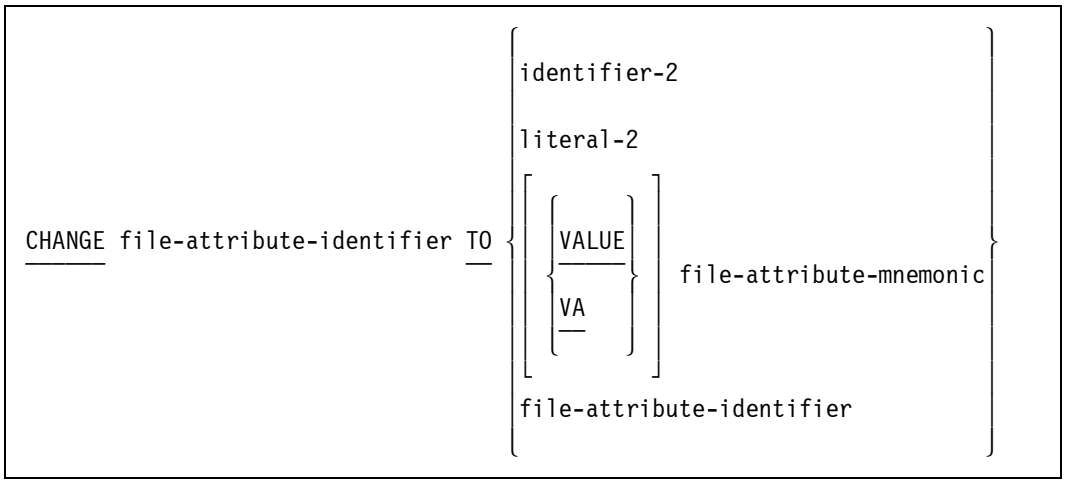
Explanation of Format 1

Format 1 is used to alter the STATIONLIST file attribute in relation to remote files. The STATIONLIST attribute can be used only with remote files.

Literal-1 must be a nonnumeric literal and can end with a period, and identifier-1 must be a nonnumeric DISPLAY data item and must end with a period.

The UP BY phrase adds to, and the DOWN BY phrase subtracts from, the list of stations associated with an open remote file.

Format 2



Explanation of Format 2

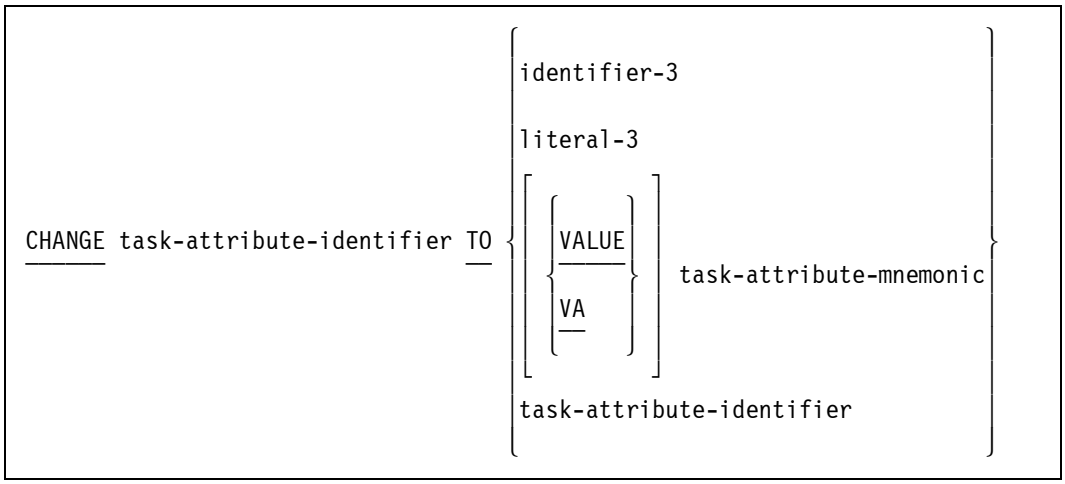
The identifier-2, literal-2, file-attribute-mnemonic, or file-attribute-identifier must be consistent with the type of the file-attribute-identifier to be changed.

If a numeric file-attribute-identifier is specified, literal-2 must be a numeric literal, and identifier-2 must be a numeric data item that represents an integer.

If an alphanumeric file-attribute-identifier is specified, literal-2 must be a nonnumeric literal and can end with a period, and identifier-2 must be a nonnumeric DISPLAY data item and must end with a period.

If there is a data-name with the same name as a mnemonic-attribute-value, the value assigned to the attribute is determined by using the optional word VALUE. If the word VALUE is present, the attribute is set to the value of the mnemonic. If the word VALUE is omitted, the attribute is set to the current value of data-name.

Format 3



Explanation of Format 3

The identifier-3, literal-3, task-attribute-mnemonic, or task-attribute-identifier must be consistent with the type of the task-attribute-identifier to be changed. Each of these variables can end with a period. However, task attributes of attribute list type (for example, LIBRARY) should not end with a period.

Type POINTER task attributes accept an alphanumeric item.

Boolean or INTEGER task attributes accept a numeric item or a literal or the value associated with a mnemonic. If the value is not in the permissible range for the attribute specified, an error occurs either at compilation time or at execution time.

A task-attribute-mnemonic is a name associated with a constant value for an attribute that has a set number of predetermined possible values.

If there is a data-name with the same name as a task-attribute-mnemonic, the value assigned to the attribute is determined by using the optional word VALUE. If the word VALUE is present, the attribute is set to the value of the mnemonic. If the word VALUE is omitted, the attribute is set to the current value of data-name.

For More Information

- For a description of the format of file-attribute-identifiers, refer to "File-Attribute Identifiers" in Section 3, "File and Task Concepts."
- For a description of the format of task-attribute identifiers, refer to "Task-Attribute Identifiers (Extension to ANSI X3.23-1974 COBOL)" in Section 3, "File and Task Concepts."
- For information on file attributes, refer to the *File Attributes Reference Manual.*
- For information on task attributes, refer to the *Task Attributes Reference Manual.*

CLOSE

The CLOSE statement ends the processing of a file or of a reel of a multivolume tape file. Unlike the OPEN statement, you do not have to specify the open mode of the file (for example, input). However, you do have to consider the device associated with the file. Some close options are not permitted for a particular device, and some close options cause different actions with different devices. The CLOSE statement does not affect either the contents or the availability of the record area of the file; this behavior is an extension to ANSI X3.23-1974 COBOL.

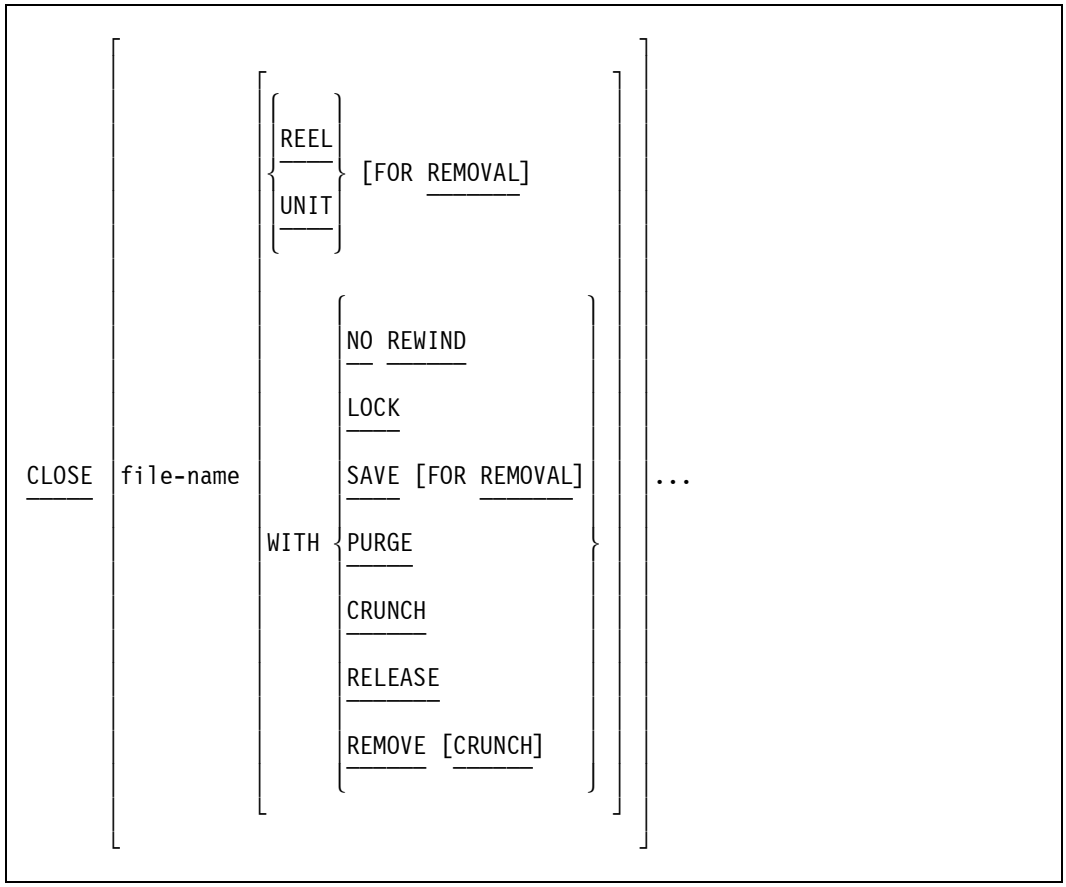
Format	Explanation
1	Closes a file for sequential I/O applications
2	Closes a file for relative and indexed I/O applications
3	Closes a port file

Note: *After an OPEN statement establishes the association between a physical file and the logical description of that file in the program, the association remains in effect until a CLOSE statement severs it through an explicit disposition that has that effect, such as CLOSE WITH SAVE or CLOSE WITH RELEASE..*

For the disk files, a CLOSE statement with no explicit disposition does not cause the file to be entered into the disk directory. An explicit disposition that requests the entering of the file into the disk directory must be used if that action is desired (Extension to ANSI X3.23-1974 COBOL).

Format 1: Sequential I/O

Format 1 is for files with sequential organization. The format is as follows:



Explanation of Format

The following Unisys extension qualifiers are syntactic extensions to standard COBOL:

- CRUNCH
- PURGE
- RELEASE
- REMOVE
- REMOVE CRUNCH
- SAVE
- SAVE FOR REMOVAL

The explanation of the format notation is divided into the following three categories of devices:

- Devices other than tape

- Single-reel tape
- Multiple-reel tape

Devices Other Than Tape

For a file associated with a disk, printer, reader, or remote device, the concept of a reel has no meaning. If you specify the REEL or UNIT option for these files, the system ignores the CLOSE statement and does not close the file. If you specify the NO REWIND option, the system ignores the option, but closes the file.

For these files, the CLOSE statement affects only the association between the logical file and the physical file, not the physical device. The operating system controls the assignment and disposition of the physical devices. (This is an extension to ANSI X3.23-1974 COBOL.)

The SAVE FOR REMOVAL phrase is intended for use only with single-reel or multireel tape files. The compiler issues a syntax error if the SAVE FOR REMOVAL phrase appears in a CLOSE statement associated with a disk, printer, reader, or remote device.

For each CLOSE statement, there is a list of actions called dispositions that apply when you use that statement. Refer to Table 9–6 later in this section for the details about each close file disposition.

Explanation of Format for Devices Other Than Tape

The CLOSE statement closes the file and applies the retain file disposition.

The file-name names an open file that you want to close. The file must be named and assigned to a device in the SELECT clause and described in the INPUT-OUTPUT and FILE SECTIONs of your program. You can close files of differing organizations and access types in one CLOSE statement.

The LOCK phrase closes the file and applies the following dispositions:

- Lock file
- Release file
- Release device

The SAVE phrase closes the file and applies the following dispositions:

- Save file
- Release file

The PURGE phrase closes the file and applies the purge file disposition.

The CRUNCH phrase closes the file and applies the following dispositions:

- Crunch file

- Save file
- Release file

The RELEASE phrase closes the file and applies the following dispositions:

- Release file
- Release device

The REMOVE phrase closes the file and applies the following dispositions:

- Save file with remove
- Release file

The REMOVE CRUNCH phrase closes the file and applies the following dispositions:

- Crunch file
- Save file with remove
- Release file

Single-Reel Tape

A single-reel tape file is a sequential file that is entirely contained on one reel.

For each close option, there is a list of actions called dispositions that apply when you use that option. Refer to Table 9–6 later in this section for the details about each close disposition.

Explanation of Format for Single-Reel Tape

The CLOSE phrase closes the file and applies the following dispositions:

- Rewind reel
- Retain file

The file-name names an open file that you want to close. The file must be named and assigned to a tape in the SELECT clause and described in the INPUT-OUTPUT and FILE SECTIONs of your program. You can close files of differing organizations and access types in one CLOSE statement. If an output tape is to contain more than one file, the file is closed but the tape is not rewound. This process enables you to write subsequent output files to the same tape.

The NO REWIND phrase closes the file and applies the following dispositions:

- No rewind of current reel
- Retain file

The LOCK phrase closes the file and applies the following dispositions:

- Lock file
- Rewind reel
- Release file
- Release device

The SAVE FOR REMOVAL phrase closes the file and applies the following dispositions:

- Rewind reel
- Release device
- Release file
- Save reel

The PURGE phrase closes the file and applies the following dispositions:

- Rewind reel
- Purge file

The RELEASE phrase closes the file and applies the following dispositions:

- Rewind reel
- Release file
- Release device

Multiple-Reel Tape

A multiple-reel tape file is a sequential file that is contained on more than one reel. If you are processing very large multiple-reel files, the following close options might be useful. These options enable you to access the next reel of tape without requiring and intervening OPEN statement.

For each close option, there is a list of actions called dispositions that apply when you use that option. Refer to Table 9–6 later in this section for the details about each close disposition.

Explanation of Format for Multiple-Reel Tape

The CLOSE statement closes the file and applies the following dispositions:

- Rewind reel
- Previous reels unaffected
- Retain file

The file-name names an open file that you want to close. The file must be named and assigned to a tape in the SELECT clause and described in the INPUT-OUTPUT and FILE

SECTIONs of your program. You can close files of differing organizations in one CLOSE statement.

The REEL or UNIT phrases do not close the file and apply the following dispositions:

- Close reel
- Rewind reel

The REEL and UNIT phrases are synonymous and interchangeable.

The REEL FOR REMOVAL phrase does not close the file and applies the following dispositions:

- Close reel
- No rewind of current reel

The NO REWIND phrase closes the file and applies the following dispositions:

- Retain file
- No rewind of current reel
- Previous reels unaffected

The LOCK phrase closes the file and applies the following dispositions:

- Lock file
- Rewind reel
- Release file
- Previous reels unaffected
- Release device

The SAVE FOR REMOVAL phrase closes the file and applies the following dispositions:

- Rewind reel
- Previous reels unaffected
- Lock file
- Release file
- Release device
- Save reel unit

The PURGE phrase closes the file and applies the following dispositions:

- Rewind reel
- Previous reels unaffected
- Purge file

The RELEASE phrase closes the file and applies the following dispositions:

- Rewind reel
- Previous reels unaffected
- Release file
- Release device

Close File Dispositions

A close file disposition is an action taken by the system during the closing procedures for a file. For a specified close option, the dispositions that occur depend on the device with which the file is associated. For example, a simple CLOSE statement with no options specified also rewinds a tape file.

A file must be in open mode to be closed. If a file is open when a program ends abnormally, executes a STOP RUN statement, or performs a CANCEL statement for the program that uses the file, the program closes the file using the dispositions of a CLOSE statement with no options specified.

The system ignores the NO REWIND, FOR REMOVAL, SAVE, PURGE, CRUNCH, RELEASE, and REMOVE phrases if they do not apply to the device associated with the file.

Once a file is closed, the program cannot perform any statement that refers to the file until the file is reopened. Because the SORT and MERGE statements have their own routines, this restriction does not apply when the program performs a SORT or a MERGE statement with the USING or GIVING options specified.

If the program attempts to close an optional input file that is not present, the program does not perform the standard end-of-file processing for that file. You can designate an optional file with the OPTIONAL phrase of the FILE-CONTROL paragraph in the ENVIRONMENT DIVISION.

When the program closes a file and releases it to the system, the program also checks the value of the EXCLUSIVE file attribute. If the value of the attribute is TRUE, the program sets it to FALSE. The CLOSE statement also changes the value of the FILEUSE attribute to I-O. This change might affect the results of any subsequent access to the RESIDENT, PRESENT, or AVAILABLE file attribute.

Table 9–6 lists each close disposition in alphabetical order and explains its actions. If the disposition depends on whether or not the file is an input, an output, or an I/O file, the differences between the open modes are described. Otherwise, the disposition applies to input, output, and I/O files.

Extensions to ANSI X3.23-1974 COBOL

The following dispositions are extensions to ANSI X3.23-1974 COBOL:

- Crunch file
- Purge file
- Release device
- Release file
- Retain file
- Save file
- Save file with remove

Table 9–6. Close-File Dispositions for Sequential I/0

Disposition	Explanation
Close reel	For input files, the following operations occur: • Reels are swapped. • Standard beginning-reel label procedure is performed. The next READ statement performed for that file makes the next data record on the new reel available. For output files, the following operations occur: • Standard ending-reel label procedure is performed. • Reels are swapped. • Standard beginning-reel label procedure is performed. The next WRITE statement performed on the file directs the next data record to the next reel of the file.
Crunch file	Releases as available to the system any unused portions of storage areas allocated for the file. This disposition is valid only for disk files. The file cannot subsequently be extended by opening the file with the OPEN EXTEND statement.
Lock file	Locks the logical file so that it cannot be reopened during execution of the program. If the file is assigned to disk, it becomes a permanent file before being made unavailable. If the file is assigned to tape, the physical unit is unreadied.
No rewind of current reel	Leaves the current reel in its current position.
Previous reels unaffected	Rewinds and locks all reels in the file prior to the current reel (except reels controlled by a prior CLOSE REEL statement). If the file is an input file, it remains open and cannot be opened in the program. If the current reel is not the last one in the input file, the program does not process the remaining reels.

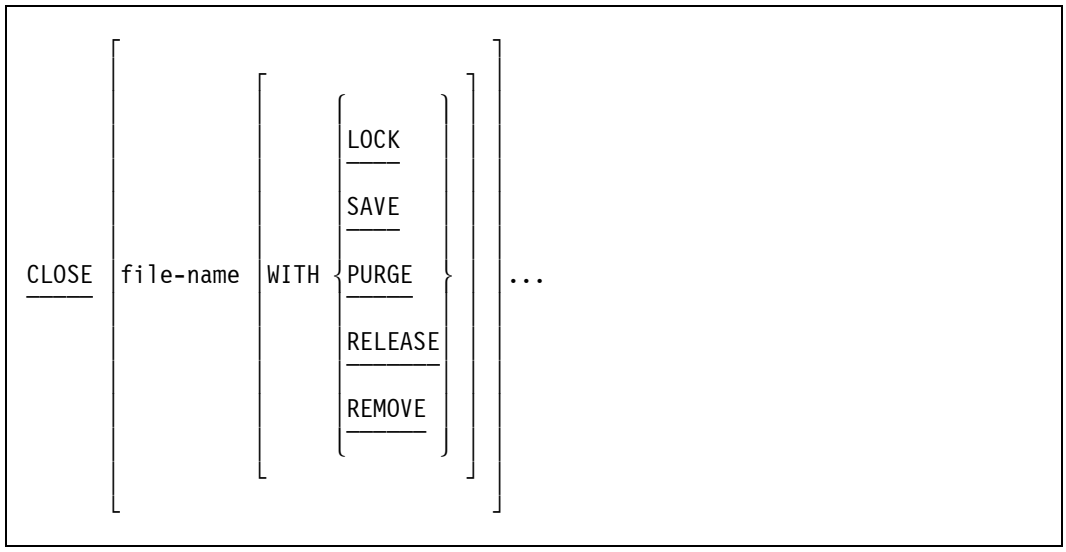
Table 9–6. Close-File Dispositions for Sequential I/0

Disposition	Explanation
Purge file	If the file is a tape file, the system rewinds the reel. If the reel has a write ring, the system writes a scratch label on it and releases the device as available to the system. If the file is a permanent disk file, the system removes the file-name from its directory and releases the disk area occupied by the file as available to the system.
Release device	Releases the device so that it is available to the system if the device to which the file was assigned can be controlled by the object program.
Release file	Severs the association between the logical file and the physical file. As the file is closed, the system checks the value of the EXCLUSIVE file attribute. If the value of the attribute is TRUE, it is assigned the value FALSE during the close process. The program releases to the system the areas of memory allocated for buffers.
Remove file	Rewinds the current reel and releases it to the system. However, you can access the reel again in its proper order of reels within the file if you perform a CLOSE statement without the REEL phrase, followed by an OPEN statement for the file.
Retain file	Retains the association between the logical file and the physical file. When you reopen the file, the operating system does not search for the physical file.
Rewind reel	Positions the current reel or analogous device at its physical beginning.
Save file	Suspends the program if a permanent file with the same title already exists and the system option 5 (AUTORM) is reset. To restart the program, the operator must enter either the RM (Remove) or OF (Optional file) system command. The new file becomes permanent after the operator enters the RM or OF command when the system option 5 (AUTORM) is set, or if no old file with the same title exists. This disposition is valid only for disk files. The file can be reopened during the execution of the program.
Save file with remove	Makes the file permanent. If a permanent file with the same title already exists, the system removes it without regard for system option 5 (AUTORM). This disposition is valid only for disk files. The program can reopen the file while the program is executing.
Save reel unit	Unreadies the physical tape unit containing the reel.

Format 2: Relative and Indexed I/O

Indexed and relative files must be associated with a disk device. If the program closes and releases the file to the system, the program checks the value of the EXCLUSIVE file attribute. If the value is TRUE, it is set to FALSE during the close process.

To close a relative or an indexed file, use the syntax shown in Format 2.



Explanation of Format

The CLOSE statement without an explicit file disposition causes the following actions to be performed:

- The file is closed.
- The association between the logical file and the physical file is retained; that is, if the file is subsequently reopened in the same program, the system does not search for the physical file.
- The file is not entered into the disk directory. If the file is to be entered into the disk directory, an explicit disposition that requests that action must appear in the CLOSE statement (extension to ANSI X3.23-1974 COBOL).

The file-name names identifies an open relative or indexed file that you want to close. The file must be named and assigned to DISK in the SELECT clause and described in the INPUT-OUTPUT and FILE SECTIONs of your program. You can close files of differing organizations and access types in one CLOSE statement.

The LOCK phrase closes the logical file and locks it so that the file cannot be reopened during the execution of the program. However, the system makes the physical file permanent and removes any existing file with the same name. The program severs the association between the logical file and the physical file and releases to the system the areas of memory allocated for buffers (extension to ANSI X3.23-1974 COBOL).

The SAVE phrase closes the logical file and suspends the program if a permanent file with the same title exists. The operator must enter either the RM (Remove) or the OF (Optional file) system command to restart the program. The new file becomes permanent after the operator enters the RM or OF system command, if system option 5 (AUTORM) is set, or if no file with the same title exists. You can reopen the file during the execution of the program. The program severs the association between the logical file and the physical file and releases to the system the areas of memory allocated for buffers (extension to ANSI X3.23-1974 COBOL).

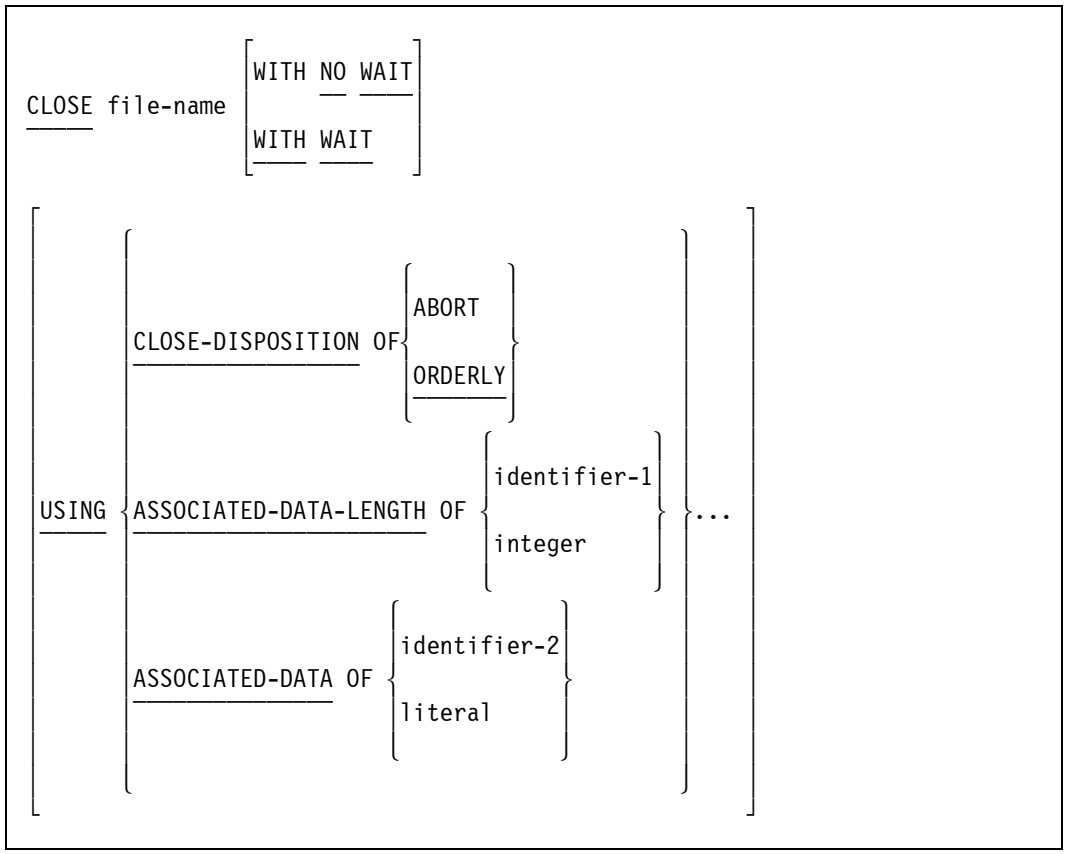
The PURGE phrase closes the logical file. If the file is permanent, the system removes the file-name from the system directory and releases to the system the areas occupied by the file.

The RELEASE phrase closes the file.

The REMOVE phrase closes the logical file and makes the physical file permanent. If a permanent file with the same title already exists, the system removes it without regard for system option 5 (AUTORM). You can reopen the file during execution of the program (extension to ANSI X3.23-1974 COBOL).

Format 3: Port Files (Extension to ANSI X3.23-1974 COBOL)

You can close a port file by performing a CLOSE statement using the syntax shown in Format 3.



Explanation of Format

The file-name names a port file.

The NO WAIT phrase closes the dialogue and returns control to the program without waiting for the correspondent endpoint to close its subfile. The close operation continues in parallel with the execution of the program. The subfile is not closed completely until the correspondent endpoint closes its subfile.

The WAIT option closes the dialogue and suspends the program until the correspondent endpoint has closed its subfile too. If you do not specify a WITH option, the program defaults to the WITH WAIT option.

Although the compiler treats the word *WITH* as optional in some cases, you should include the word *WITH* when using the WITH WAIT option to clarify possible ambiguities between this context and that of the WAIT verb.

The USING option permits each of the USING clauses to be included once only.

The CLOSE-DISPOSITION ABORT phrase closes the dialogue immediately, without ensuring that your program received all the data. It is your responsibility to guarantee that all messages are processed so there is no loss of data. The compiler will default to the ABORT phrase if the type of CLOSE-DISPOSITION is not specified.

The CLOSE-DISPOSITION ORDERLY phrase closes the dialogue in an orderly manner and ensures that no data is lost. This disposition is available only with some services.

The ASSOCIATED-DATA-LENGTH phrase specifies the number of characters to be sent. If you do not specify the length of the associated data and the associated data is a data item, your program uses the actual length of the data. If you specify the length of the associated data, the length value must be less than or equal to the actual length of the data. An error results if the length specified is not a single-precision integer value. To use the ASSOCIATED-DATA-LENGTH phrase, you must specify the ASSOCIATED-DATA phrase with either an identifier or an undigit literal, but not a nonnumeric literal.

The ASSOCIATED-DATA phrase is used on some types of networks to transfer data to the correspondent endpoint along with the request to close the dialogue.

The integer must be a numeric literal.

Identifier-1 must name an elementary integer data item.

Identifier-2 can be a group data item or an alphanumeric elementary data item.

The literal can be a nonnumeric literal or an undigit literal.

General Rules

Besides specifying the close options, you might need to designate the subfiles to close. If you are using subfiles, perform the following steps:

1. Specify the total number of subfiles in your program by using the *CHANGE ATTRIBUTE MAXSUBFILES TO VALUE attribute-value* statement.
2. Specify the subfiles to be opened by using the *SELECT port-file ASSIGN TO PORT; ACTUAL KEY IS subfile-num* clause.
3. Declare subfile-num and attribute-value in the WORKING-STORAGE SECTION.

The ACTUAL KEY clause specifies the subfile to close. Table 9–7 shows the subfiles that are closed with differing ACTUAL KEY clause values.

Table 9–7. Designating Subfiles to Close

Actual Key Value	Explanation
0 or none	Closes every open subfile
Nonzero	Closes the specified subfile
Greater than the MAXSUBFILES value or negative number	Returns a BADSUBFILEINDEX run-time error in the SUBFILERROR attribute

Examples

In the following examples, it is assumed that earlier in the program the port files were declared using an ACTUAL KEY clause and the subfile index was set to a particular subfile.

Example 9–6 causes the dialogue established on the port file for PORTFILE1 to be closed. The program is suspended until the correspondent endpoint is also closed.

```
CLOSE PORTFILE1 WITH WAIT.
```

Example 9–6. Coding a CLOSE WITH WAIT Statement

Example 9–7 closes the dialogue and returns control to the program without waiting for the correspondent endpoint to close its subfile. The subfile on PORTFILE1 is not completely closed until the correspondent endpoint closes its subfile. The ORDERLY option ensures that no data is lost during the close operation. Any messages that are received are saved in the READ queue.

```
CLOSE PORTFILE1 WITH NO WAIT USING CLOSE-DISPOSITION OF ORDERLY.
```

Example 9–7. Coding a CLOSE WITH NO WAIT Statement

Example 9–8 requests that PORTFILE1 be closed. Since the CLOSE-DISPOSITION phrase is not specified, a CLOSE ABORT statement is assumed. Since the WAIT or NO WAIT are not specified, WAIT is assumed, and control is not returned to the program until the close operation is complete. The information "MYDATA" is sent to the other program with this message:

```
CLOSE PORTFILE1 USING ASSOCIATED-DATA OF "MYDATA".
```

Example 9–8. Coding a CLOSE...ASSOCIATED-DATA Statement

Example 9–9 requests that PORTFILE1 be closed. When this message to close is sent, 10 characters of data are sent at the same time to the correspondent endpoint. The data sent begins at the location pointed to by the ALPHANUM-ITEM item.

```
CLOSE PORTFILE1
      USING ASSOCIATED-DATA OF ALPHANUM-ITEM
            ASSOCIATED-DATA-LENGTH OF 10.
```

Example 9–9. Coding a CLOSE...ASSOCIATED-DATA-LENGTH Statement

Example 9–10 requests that the multiple port files, PORTFILE1 and PORTFILE2, be closed. The ABORT option closes the dialogue, but does not ensure that your program received all the data.

```
CLOSE PORTFILE1 WITH WAIT
      USING CLOSE-DISPOSITION OF ABORT
            ASSOCIATED-DATA-LENGTH OF 10
            ASSOCIATED-DATA OF ALPHANUM-ITEM
      PORTFILE2 WITH NO WAIT
      USING CLOSE-DISPOSITION OF ORDERLY
            ASSOCIATED-DATA-LENGTH OF 10
            ASSOCIATED-DATA OF ALPHANUM-ITEM.
```

Example 9–10. Closing Multiple Port Files

For More Information

- For step-by-step information on coding port file applications, refer to the *I/O Subsystem Programming Guide.*
- For information on file attributes, refer to the *File Attributes Reference Manual.*

Handling of Error Conditions

If the program does not contain a FILE STATUS clause in the SELECT statement for the file that is the object of the CLOSE verb, and does not contain a USE AFTER STANDARD ERROR PROCEDURE ON <filename> in DECLARATIVES for the file, then the compiler generates object code that unconditionally terminates the program if an error condition arises during execution. The program is terminated even in those cases in which execution should continue (some cases in Format 3, for example). To be able to continue after such an error and to provide a way for the system to inform the program that an error has occurred, the program must contain either a FILE STATUS clause in the SELECT statement for the file, or a USE procedure in DECLARATIVES referencing the file, or both.

For more information, refer to "Handling I/O Exception Conditions" in Section 8, "PROCEDURE DIVISION Concepts."

I/O Status Value

The system returns a value that indicates the result of a CLOSE statement. You can access this value by including a *SELECT file-name FILE STATUS IS data-name* clause in your program. The operating system moves a value into the designated data-name storage area after the program performs the CLOSE statement. You can then use an IF statement to test the value of the data-name and take the desired action depending on the result. If you choose not to code an action for the I/O result, the system provides default actions for the results.

Table 9–8 shows the I/O status values and their meanings.

Table 9–8. I/O Status Values for CLOSE Statement Completion

Value	Explanation
00	Control was returned to the program after the CLOSE statement completed correctly. In the case of a port file, the close operation might be pending.
82	An error was detected while the file was being closed.

Note: *The I/O status value "82" is an extension to ANSI X3.23-1974 COBOL.*

COMPUTE

The COMPUTE statement calculates the value of the arithmetic expression following the equal sign (=) and assigns the value to one or more data items preceding the equal sign.

This statement is useful for performing complex arithmetic operations.

The general format of this statement is as follows:

```
COMPUTE identifier-1 [ROUNDED] [,identifier-2 [ROUNDED]]...

= arithmetic-expression [;ON SIZE ERROR imperative-statement]
```

Explanation of Format

If more than one identifier is specified to the left of the equal sign, the compiler computes the value of the arithmetic expression and stores the new value in identifier-1, identifier-2, and so forth in turn.

Identifiers that appear only to the left of the equal sign must refer either to an elementary numeric item or to an elementary numeric-edited item.

If the arithmetic expression consists of a single identifier or literal, the compiler sets the values of identifier-1, identifier-2, and so on to the value of the single identifier or literal.

Imperative-statement can be the NEXT SENTENCE phrase.

When a sending item and a receiving item in the same COMPUTE statement share a part, but not all, of their storage areas, the result of the execution of the statement is undefined.

Rounding Error

A rounding error can occur if identifier-1, identifier-2, and so on contain different numbers of digits after the decimal point, and if an intermediate result of the computation exceeds the maximum integer.

For More Information

- For a description of the features common to the arithmetic statements, refer to “Common Rules for Arithmetic Statements” in Section 8, “PROCEDURE DIVISION Concepts.”
- For information on producing multiple results with one arithmetic statement, refer to “Calculating Multiple Results with One Arithmetic Statement” in Section 8, “PROCEDURE DIVISION Concepts.”

- For information about the rounding of arithmetic result fields, refer to "ROUNDED Phrase" in Section 8, "PROCEDURE DIVISION Concepts."
- For information on Size Error conditions, refer to "SIZE ERROR Phrase" in Section 8, "PROCEDURE DIVISION Concepts."

CONTINUE (Extension to ANSI X3.23-1974 COBOL)

The CONTINUE statement passes control to a synchronous process that has been previously called and exited with the EXIT PROGRAM or EXIT PROGRAM RETURN HERE statement. This statement enables the called process to restart without repassing parameters or to resume at a point other than its first executable statement.

The general format of this statement is as follows:

```
CONTINUE task-identifier
```

Explanation of Format

The task-identifier must be the same as in a previously executed CALL statement.

For information about transferring control to a separate task or procedure, refer to "CALL" earlier in this section.

COPY

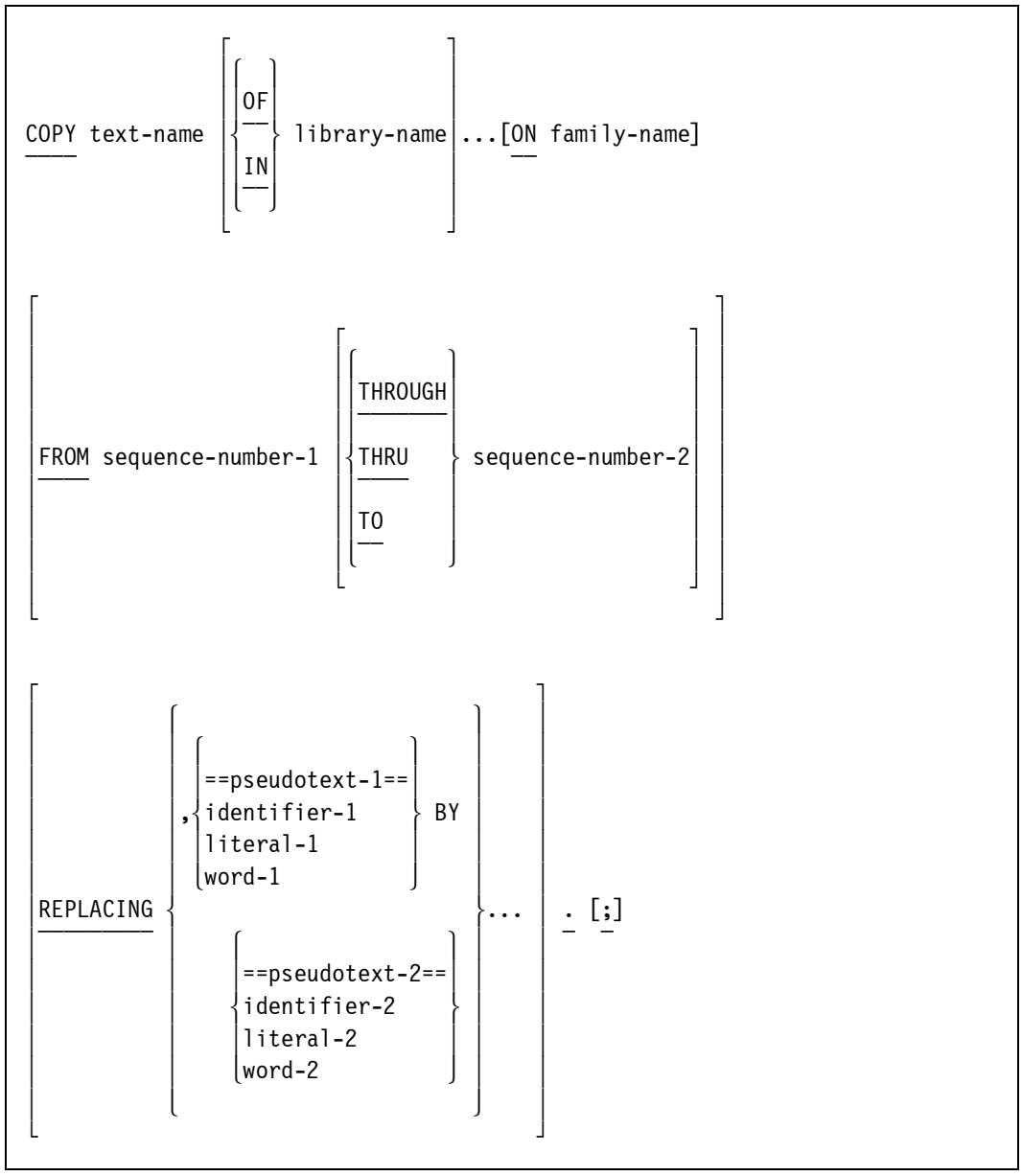
This statement incorporates text from a library program into the program that contains the COPY statement. This statement is useful for eliminating redundant programming.

You can use a COPY statement in a program anywhere a character string or a separator is allowed, but a COPY statement must not occur within another COPY statement.

The text produced from processing of a COPY statement must not contain a COPY statement.

Text in the copied library must conform to the rules for COBOL reference format.

When the compiler processes a COPY statement, it copies the library text associated with the text-name into the source program and logically replaces the entire COPY statement, beginning with the reserved word COPY and ending with the period, inclusive.The compilation of a source program containing COPY statements is logically equivalent to processing all COPY statements prior to the processing of the resultant source program.



Explanation of Format

The COPY statement must be preceded by a separator and ended by a period. (The use of a separator other than a space immediately before a COPY statement is an extension to ANSI X3.23-1974 COBOL). To comply with language standards, at least one space must be used as a separator immediately before a COPY statement.

text-name

Text-name specifies the external identification of a file in the COBOL library.

library-name

Library-name specifies the external identification of a directory-ID associated with the name of the COBOL library. Successive library-names specify parent directory-IDs; thus, a complete series of library-names represents the entire name.

ON family-name (Extension to ANSI X3.23-1974 COBOL)

Family-name specifies the name of the family in which the library file resides.

FROM sequence number (Extension to ANSI X3.23-1974 COBOL)

THRU, THROUGH, and TO are synonyms.

If the FROM phrase is specified, copying starts at the sequence number specified. If the THRU phrase is specified, copying continues until that sequence number has been copied. If the THRU phrase is not specified, copying continues to the end of the file.

REPLACING

If the REPLACING phrase is not specified, the library text is copied unchanged. If the REPLACING phrase is specified, the library text is copied, and each properly matched occurrence of pseudotext-1, identifier-1, word-1, or literal-1 in the library text is replaced by the corresponding pseudotext-2, identifier-2, word-2, or literal-2. The REPLACING phrase does not alter the source library itself, but changes the library text according to specifications in the calling program. After the REPLACING phrase is executed, the source listing contains the original library text. The source listing does not contain the results of the REPLACING phrase.

Pseudotext is a sequence of text-words or comments bounded by two consecutive equal signs (==). This sequence specifically excludes partial words. Allowable separators in the pseudotext are commas (,), semicolons (;), and spaces; a nonnumeric literal can contain quotation marks ("). Pseudotext-1 must not be null and cannot consist solely of spaces or comment lines. Pseudotext-2 can be null.

Character strings within pseudotext-1 and pseudotext-2 can be continued. However, both characters of a pseudotext delimiter must be on the same line.

Word-1 or word-2 can be any single COBOL word, including a COBOL reserved word.

For purposes of matching, identifier-1, word-1, and literal-1 are treated as pseudotext containing only identifier-1, word-1, or literal-1, respectively.

The comparison operation to determine text replacement occurs in the following manner:

- Any separator comma, semicolon, or space preceding the leftmost library text-word is copied into the source program. Starting with the leftmost library text-word and the first pseudotext-1, identifier-1, word-1, or literal-1 specified in the REPLACING phrase, the entire REPLACING phrase operand that precedes the reserved word BY is compared to an equivalent number of contiguous library text-words.
- Pseudotext-1, identifier-1, word-1, or literal-1 match the library text only if the ordered sequence of text-words that forms pseudotext-1, identifier-1, word-1, or literal-1 is equal, character for character, to the ordered sequence of library text-words.

  For purposes of matching, each occurrence of a separator comma or semicolon in pseudotext-1 or in the library text is considered to be a single space, except when pseudotext-1 consists solely of either a comma or a semicolon. In that case, pseudotext-1 participates in the match as a text-word. Each sequence of one or more space separators is considered to be a single space.
- If no match occurs, the comparison is repeated with each successive pseudotext-1, identifier-1, word-1, or literal-1, if any, in the REPLACING phrase until either a match is found or no REPLACING operand remains.

  When all REPLACING phrase operands have been compared and no match has occurred, the leftmost library text-word is copied into the source program. The next successive library text-word is then considered to be the leftmost library text-word, and the comparison cycle starts again with the first pseudotext-1, identifier-1, word-1, or literal-1 specified in the REPLACING phrase.

  Whenever a match occurs between pseudotext-1, identifier-1, word-1, or literal-1 and the library text, the corresponding pseudotext-2, identifier-2, word-2, or literal-2 is placed in the source program. The library text-word immediately following the rightmost text-word that participates in the match is then considered to be the leftmost library text-word. The comparison cycle starts again with the first pseudotext-1, identifier-1, word-1, or literal-1 specified in the REPLACING phrase.
- The comparison operation continues until the rightmost text-word in the library text has either participated in a match or has been considered as a leftmost library text-word and participated in a complete comparison cycle.

For matching, a comment line occurring in the library text and in pseudotext-1 is interpreted as a single space. Comment lines appearing in pseudotext-2 and library text are copied into the source program unchanged.

Debugging lines are permitted in library text and pseudotext-2. Debugging lines are not permitted in pseudotext-1; text-words in a debugging line participate in the matching rules as if the letter D did not appear in the indicator area. If a COPY statement is specified on a debugging line, the text that results from the processing of the COPY statement appears as if it were specified on debugging lines, with the following exception: comment lines in library text appear as comment lines in the resultant source program.

After replacement, text-words are placed in the source program for compilation according to the rules for reference format.

semicolon (;) (Extension to ANSI X3.23-1974 COBOL)

The semicolon (;) that follows the ending period can be used to control the behavior of compiler control records (CCRs) and the format of listings. This semicolon should always be separated from the ending period of the COPY statement by at least one space.

If a CCR immediately follows a COPY statement, the compiler option changes might occur before the compiler processes the included source information. This situation can be avoided by using the semicolon after the ending period. The semicolon ensures that the compiler processes the included source information before the option changes.

When a compilation listing is produced, a comment immediately following a COPY statement might be printed after the COPY statement but before the information included as a result of the COPY statement. If a semicolon is placed after the ending period but before the comment entry, the comment is printed after the included source information.

Use the optional semicolon with caution. In some cases, the compiler may recognize the optional semicolon. In other cases, the compiler may prohibit the use of the semicolon. In the latter cases, the semicolon may not produce the desired listing format and even produce syntax errors. In such cases, use the semicolon as a tool in determining whether errors can be eliminated.

In general, the semicolon can produce undesirable listing formats in the following cases:

- Multiple COPY statements follow each other with no intervening syntax.
- COPY statements have semicolons.
- The last element in the library that is the subject of a COPY statement is a PICTURE string that ends with one or more 9s followed by a period terminating the DATA declaration.

If the last statement of a COBOL74 program is a COPY statement, do not use a semicolon with that statement. The last syntax element of a COBOL74 program must always be a period that terminates the last statement or paragraph-name of the program.

Examples

The following examples represent different ways of incorporating (SYSTEM)A/B/C ON SYSPACK into the program.

```
COPY "(SYSTEM)A/B/C ON SYSPACK.".
COPY C OF B OF A OF (SYSTEM) ON SYSPACK.
```

DELETE

The DELETE statement logically removes a record from a mass-storage file.

```
DELETE file-name RECORD [;INVALID KEY imperative-statement]
```

Explanation of Format

The file-name must be a relative or an indexed file.

The imperative-statement can be the NEXT SENTENCE phrase.

For a file in the sequential access mode, the last I/O statement executed prior to execution of the DELETE statement must have been a successfully executed READ statement. The record accessed by the READ statement is deleted from the file.

For a file in random or dynamic access mode, the record to be deleted is identified by the contents of the relative or prime record key data item associated with the file-name. If the file does not contain the record specified by the key, an Invalid Key condition exists.

The file must be open in the I-O mode when this statement is executed.

After the successful execution of a DELETE statement, the identified record that is logically removed from the file can no longer be accessed.

The execution of a DELETE statement does not affect the contents of the record area associated with the file-name, nor does it affect the current record pointer.

However, the execution of the DELETE statement causes the appropriate FILE STATUS data item, if any, to be updated.

For information on using the INVALID KEY phrase, refer to “Handling I/O Exception Conditions” in Section 8, “PROCEDURE DIVISION Concepts.”

DETACH (Extension to ANSI X3.23-1974 COBOL)

The DETACH statement dissociates a procedure from a task item or an EVENT item.

```
DETACH identifier-1 [,identifier-2]...
```

Explanation of Format

Identifiers used in this statement must be defined as either elementary task items or section-names in the DECLARATIVES SECTION with a USE AS INTERRUPT clause.

If a task item is being detached, it must previously have been implicitly attached by the execution of a CALL, PROCESS, or RUN statement. The successful execution of a statement terminates the task that was attached to the task-identifier of the task and was running, because the STATUS attribute was set to TERMINATED. After execution of the DETACH statement, the task-identifier should be tested for STATUS of TERMINATED prior to the next use of that task-identifier in a CALL, PROCESS, or RUN statement, because execution of the program that contained the DETACH statement continues asynchronously while the detachment is performed.

Execution of a *DETACH section-name* statement severs the association of that interrupt procedure with its currently attached event. Detaching an interrupt procedure that is not attached to an event does not cause an error. Executions of the interrupt procedure, which might have been queued at the time of the detachment, do not occur.

The Allowed or Disallowed condition of the interrupt procedure is not changed by a DETACH statement. If the interrupt is subsequently attached, the condition is the same as it was before the DETACH statement.

For More Information

- For information about executing an interrupt procedure that has been attached to an EVENT item, refer to "ALLOW (Extension to ANSI X3.23-1974 COBOL)" earlier in this section.
- For information about associating an interrupt procedure with an EVENT item, refer to "ATTACH (Extension to ANSI X3.23-1974 COBOL)" earlier in this section.
- For information about preventing execution of an interrupt procedure that has been attached to an event, refer to "DISALLOW" earlier in this section.
- For information about specifying interrupt procedures, refer to Format 4 of "USE" later in this section.

DISALLOW

The DISALLOW statement prevents execution of an interrupt procedure that has been attached to an event.

Format	Explanation
1	Enables you to specify the interrupt procedures that are not to be executed
2	Causes a General Disallow Interrupt condition

Format 1

```
DISALLOW section-name-1 [,section-name-2]...
```

Explanation of Format 1

The section-names specify the interrupt procedures that are not to be executed when their attached events occur. The sections named must be defined in the DECLARATIVES SECTION with the USE AS INTERRUPT clause in their headers. The section-names must be allowed when the DISALLOW statement is executed.

Format 2

```
DISALLOW INTERRUPT
```

Explanation of Format 2

The DISALLOW INTERRUPT statement causes a General Disallow Interrupt condition. During the time the General Disallow Interrupt condition is in effect, execution of an interrupt procedure that is attached to an event is queued when the event is caused. Later execution of an ALLOW INTERRUPT statement immediately executes any queued procedures that are not currently disallowed because of a specific *DISALLOW section-name* statement.

General Rules

After execution of a *DISALLOW section-name* statement, the system queues executions of the specified interrupt procedures because of the events being caused. Their execution can later be caused by a corresponding *ALLOW section-name* statement unless a General Disallow Interrupt condition is in effect. The current state of the General Disallow Interrupt condition does not affect whether an interrupt is specifically allowed or disallowed.

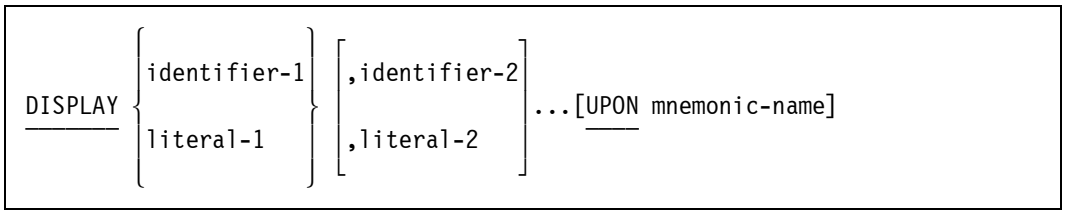
For More Information

- For information about executing an interrupt procedure that has been attached to an EVENT item, refer to "ALLOW (Extension to ANSI X3.23-1974 COBOL)" in this section.
- For information about associating an interrupt procedure with an EVENT item, refer to "ATTACH (Extension to ANSI X3.23-1974 COBOL)" in this section.
- For information about preventing execution of an interrupt procedure that has been attached to an event, refer to "DISALLOW" in this section.
- For information about dissociating a procedure from a task item or an EVENT item, refer to "DETACH (Extension to ANSI X3.23-1974 COBOL)" in this section.
- For information about specifying interrupt procedures, refer to Format 4 of "USE" in this section.

DISPLAY

The DISPLAY statement transfers data to an operator display terminal (ODT).

The general format of this statement is as follows:



Explanation of Format

The DISPLAY statement causes the contents of each operand to be transferred in the order listed.

If a figurative constant is specified as one of the operands, only a single occurrence of the figurative constant is displayed.

Each literal can be any figurative constant except ALL.

When a DISPLAY statement contains more than one operand, the size of the sending item is the sum of the sizes associated with the operands, and the values of the operands are transferred in the sequence in which the operands are encountered. In an extension to ANSI X3.23-1974 COBOL, if the data transferred does not fit on one line, carriage returns and line feeds are supplied so that the data is extended to other lines of print.

If the UPON phrase is not used, the device used is the ODT.

The mnemonic-name is associated with a hardware device in the SPECIAL-NAMES paragraph in the ENVIRONMENT DIVISION and must be associated with the hardware-name ODT.

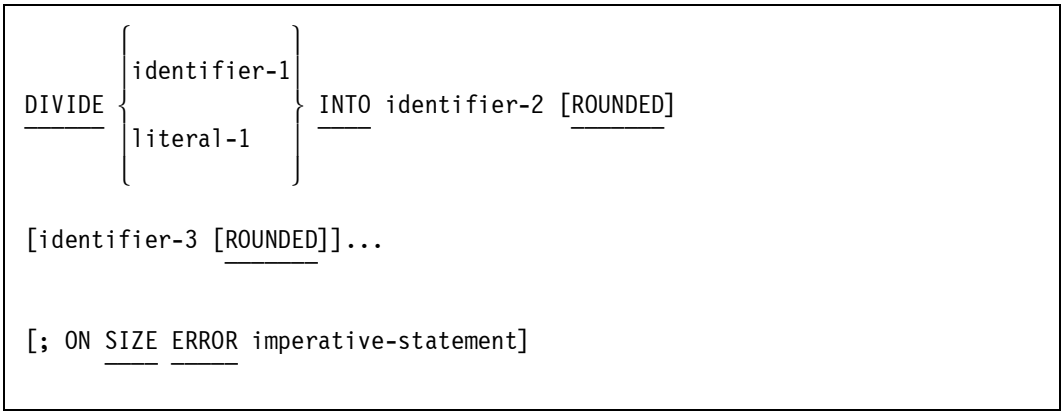
The operating system, when processing a DISPLAY statement, stops at the first null (hex 00). The operating system initializes data areas to all nulls. Therefore, if a FILLER item without a VALUE clause is present in a group item being displayed, the display ends with this item. (This is an extension to ANSI X3.23-1974 COBOL.)

DIVIDE

The DIVIDE statement divides one numeric data item into others and sets the values of data items equal to the quotient and remainder.

Format	Explanation
1	The DIVIDE...INTO format enables you to divide a numeric operand into another numeric operand and to store the result in the second operand.
2	The DIVIDE...INTO...GIVING format enables you to divide a numeric operand into another numeric operand and to specify a place to store the result.
3	The DIVIDE...BY...GIVING format enables you to divide a numeric operand by another numeric operand and to specify a place to store the result.
4	The DIVIDE...INTO...GIVING...REMAINDER format enables you to divide a numeric operand into another numeric operand and to specify a place to store the result and the remainder.
5	The DIVIDE...BY...GIVING...REMAINDER format enables you to divide a numeric operand by another numeric operand and to specify a place to store the result.

Format 1



Explanation of Format 1

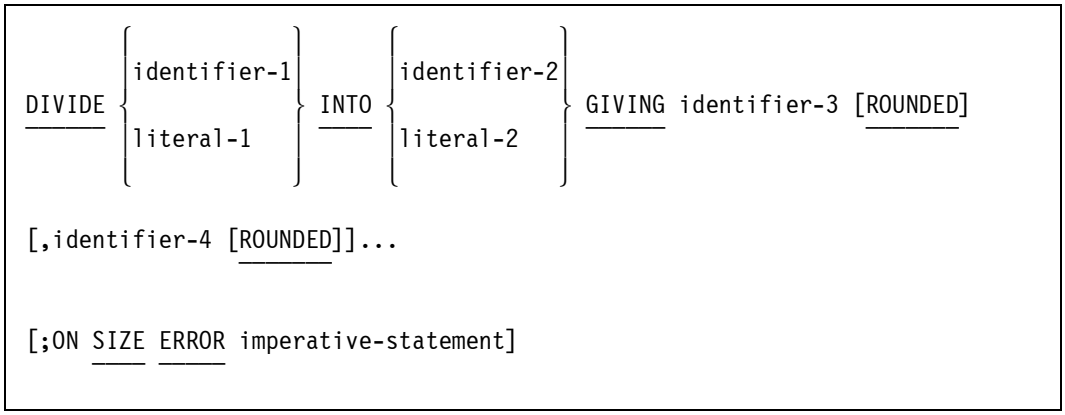
When Format 1 is used, the value of identifier-1 or literal-1 is divided into the value of identifier-2. The value of the dividend (identifier-2) is replaced by this quotient; the process is then repeated for identifier-1 or literal-1 and identifier-3, and so on.

Each identifier must refer to an elementary numeric item.

Each literal must be a numeric literal.

The imperative-statement can be the NEXT SENTENCE phrase.

Format 2



Explanation of Format 2

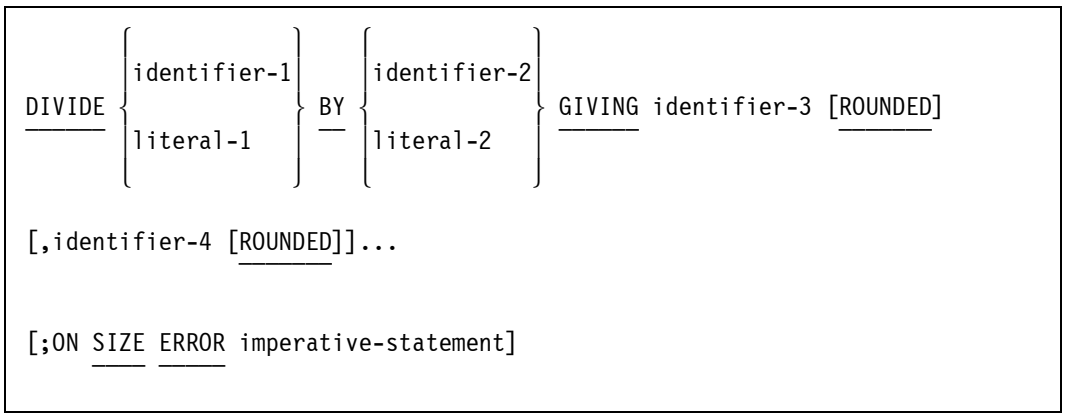
When Format 2 is used, the value of identifier-1 or literal-1 is divided into identifier-2 or literal-2, and the result is stored in identifier-3, identifier-4, and so on.

Each identifier must refer to an elementary numeric item, except that any identifier associated with the GIVING phrase must refer to either an elementary numeric item or an elementary numeric-edited item.

Each literal must be a numeric literal.

The imperative-statement can be the NEXT SENTENCE phrase.

Format 3



Explanation of Format 3

When Format 3 is used, the value of identifier-1 or literal-1 is divided by the value of identifier-2 or literal-2, and the result is stored in identifier-3, identifier-4, and so on.

Each identifier must refer to an elementary numeric item, except that any identifier associated with the GIVING phrase must refer to either an elementary numeric item or an elementary numeric-edited item.

Each literal must be a numeric literal.

The imperative-statement can be the NEXT SENTENCE phrase.

Format 4

```
        ┌              ┐      ┌              ┐
        │identifier-1  │      │identifier-2  │
DIVIDE  ┤              ├ INTO ┤              ├ GIVING identifier-3 [ROUNDED]
        │literal-1     │      │literal-2     │
        └              ┘      └              ┘

REMAINDER identifier-4 [;ON SIZE ERROR imperative-statement]
```

Format 5

```
        ┌              ┐    ┌              ┐
        │identifier-1  │    │identifier-2  │
DIVIDE  ┤              ├ BY ┤              ├ GIVING identifier-3 [ROUNDED]
        │literal-1     │    │literal-2     │
        └              ┘    └              ┘

REMAINDER identifier-4 [;ON SIZE ERROR imperative-statement]
```

Explanation of Format 4 and 5

Formats 4 and 5 are used when a remainder from the division operation is desired, namely identifier-4. The remainder in COBOL is defined as the result of subtracting the product of the quotient (identifier-3) and the divisor from the dividend. If identifier-3 is defined as a numeric-edited item, the quotient used to calculate the remainder is an intermediate field that contains the unedited quotient. If the word ROUNDED is used, the quotient used to calculate the remainder is an intermediate field that contains the quotient of the DIVIDE statement in truncated rather than rounded form.

In Formats 4 and 5, the accuracy of the REMAINDER data item (identifier-4) is defined by the preceding calculation. Appropriate decimal alignment and truncation (not rounding) are performed for the content of the data item referenced by identifier-4, as needed.

Each identifier must refer to an elementary numeric item, except that any identifier associated with the GIVING or REMAINDER phrase must refer to either an elementary numeric item or an elementary numeric-edited item.

Each literal must be a numeric literal.

The imperative-statement can be the NEXT SENTENCE phrase.

When the ON SIZE ERROR phrase is used in Formats 4 and 5, the following rules apply:

- If the Size Error condition occurs on the quotient, no remainder calculation is meaningful. Thus, the contents of the data items referenced by both identifier-3 and identifier-4 remain unchanged.
- If the Size Error condition occurs on the remainder, the contents of the data item referenced by identifier-4 remain unchanged.

General Rules

The following information applies to all formats.

When a sending item and a receiving item in the same DIVIDE statement share a part, but not all, of their storage areas, the result of the statement execution is undefined.

For More Information

- For a description of the features common to the arithmetic statements, refer to "Common Rules for Arithmetic Statements" in Section 8, "PROCEDURE DIVISION Concepts."
- For information on producing multiple results with one arithmetic statement, refer to "Calculating Multiple Results with One Arithmetic Statement" in Section 8, "PROCEDURE DIVISION Concepts."
- For information about rounding of arithmetic result fields, refer to "ROUNDED Phrase" in Section 8, "PROCEDURE DIVISION Concepts."
- For information on Size Error conditions, refer to "SIZE ERROR Phrase" in Section 8, "PROCEDURE DIVISION Concepts."

EXECUTE (Extension to ANSI X3.23-1974 COBOL)

The EXECUTE statement is synonymous with the RUN statement.

For information about this statement, refer to "RUN (Extension to ANSI X3.23-1974 COBOL)" in this section.

EXIT

The EXIT statement allows you to end a series, section of paragraphs, exit from a bound procedure or called program, or bypass statements.

Format	Explanation
1	Ends a series of sections or paragraphs referenced by a PERFORM statement
2	Enables you to exit from a bound procedure
3	Enables you to exit from a called program
4	Enables you to bypass statements

Format 1

```
EXIT.
```

Explanation of Format 1

This statement provides a method for documenting the logical endpoint for a series of sections or paragraphs that can be executed under the control of a PERFORM statement.

EXIT must appear in a sentence by itself and must be the only sentence in the paragraph.

A simple EXIT statement only assigns a procedure-name to a given point in a program and has no effect on execution of the program.

Format 2 (Extension to ANSI X3.23-1974 COBOL)

```
EXIT PROCEDURE.
```

Explanation of Format 2

This statement provides a mechanism for returning from a bound procedure.

If a Format 2 statement appears in a consecutive sequence of imperative statements within a sentence, it must appear as the last statement in that sequence. If the program in which it appears is not a bound procedure, then the EXIT PROCEDURE statement is synonymous with the STOP RUN statement.

The EXIT PROCEDURE statement should be used only for procedures compiled at level 3 or higher. If the procedure has been processed or called as a coroutine, end-of-task (EOT) occurs for that stack. If the procedure has been called as a procedure, normal exit routines transfer control back to the statement following the procedure invocation in the calling program.

An implicit EXIT PROCEDURE statement is compiled for all procedures compiled at level 3 or higher. The EXIT PROCEDURE statement need not be used when it would be the final statement in the procedure.

Format 3 (Extension to ANSI X3.23-1974 COBOL)

```
EXIT PROGRAM [RETURN HERE].
```

Explanation of Format 3

An EXIT PROGRAM statement is designed to be used to return from a program that is under control of a CALL statement.

If an EXIT PROGRAM statement is executed while a program is under control of a tasking CALL statement, then control returns to the statement following the CALL statement in the calling program. If a subsequent CONTINUE statement is executed for the same program, control passes to the first logically executable statement in the called program.

The EXIT PROGRAM statement and RETURN HERE option provide a mechanism for passing control between a dependent task and its initiating program.

If a Format 3 statement without the RETURN HERE phrase appears in a consecutive sequence of imperative statements within a sentence, it must be the last statement in that sequence.

An EXIT PROGRAM RETURN HERE statement cannot appear in a bound procedure.

When the EXIT PROGRAM RETURN HERE statement is used and a CONTINUE statement is executed on the same program, control passes to the statement immediately following the EXIT statement.

Format 4 (Extension to ANSI X3.23-1974 COBOL)

```
EXIT PERFORM.
```

Explanation of Format 4

The EXIT PERFORM statement provides a way to bypass the remainder of a PERFORM statement range.

If the program is under control of a PERFORM statement when the EXIT PERFORM statement is encountered, any remaining statements in the PERFORM statement range are bypassed. If an EXIT PERFORM statement is executed when no PERFORM statement is active, control passes to the next statement.

For information on returning control from a library to a calling program, refer to "Exiting a Library" in Section 15, "Libraries."

GO TO

The GO TO statement explicitly transfers control to another procedure.

Format	Explanation
1	The GO TO format transfers control from one part of the PROCEDURE DIVISION to another.
2	The GO TO...DEPENDING ON format transfers control from one part of the PROCEDURE DIVISION to another, depending on the value of a specified integer identifier.

Format 1

```
GO TO [procedure-name-1]
```

Explanation of Format 1

When a GO TO statement executes, control transfers to procedure-name-1 or to another procedure-name if the GO TO statement was modified by an ALTER statement. If procedure-name-1 is not specified, an ALTER statement referring to this GO TO statement must be executed before the execution of this GO TO statement. Otherwise, the program ends abnormally. When a paragraph is referenced by an ALTER statement, that paragraph can consist of only a paragraph header followed by a Format 1 GO TO statement.

A Format 1 GO TO statement without procedure-name-1 can appear only in a single-statement paragraph.

If a GO TO statement appears in a consecutive sequence of imperative statements within a sentence, it must be the last statement in that sequence.

Format 2

```
GO TO procedure-name-1, procedure-name-2 [,procedure-name-n]...

DEPENDING ON identifier
```

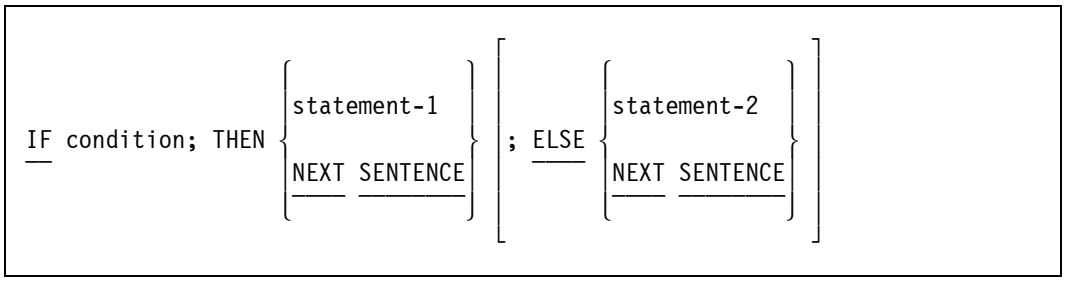
Explanation of Format 2

When a GO TO statement executes, control transfers to procedure-name-1, procedure-name-2, and so on, depending on whether the value of the identifier is 1, 2, or another integer value. If the value of the identifier is anything other than a positive or unsigned integer, no transfer occurs and control passes to the next statement.

The identifier is the name of an elementary numeric item that describes an integer. It cannot be a formula or another expression.

IF

The IF statement evaluates a condition. The subsequent action of the object program depends on whether the value of the condition is TRUE or FALSE.



Explanation of Format

Statement-1 and statement-2 represent either an imperative statement or a conditional statement; either type of statement can be followed by a conditional statement.

When an IF statement is executed, the following transfers of control occur:

- If the condition is TRUE, statement-1 is executed if specified. If statement-1 contains a procedure branching statement or conditional statement, control is explicitly transferred under the rules of that statement. If statement-1 does not contain a procedure branching or conditional statement, the specified ELSE phrase is ignored and control passes to the next executable sentence.
- If the condition is TRUE and the NEXT SENTENCE phrase is specified instead of statement-1, the specified ELSE phrase is ignored and control passes to the next executable sentence.
- If the condition is FALSE, statement-1 or its alternative, NEXT SENTENCE, is ignored; statement-2, if specified, is executed. If statement-2 contains a procedure branching or conditional statement, control is explicitly transferred under the rules of that statement. If statement-2 does not contain a procedure branching or conditional statement, control passes to the next executable sentence. If the *ELSE statement-2* phrase is not specified, statement-1 is ignored and control passes to the next executable sentence.

- If the condition is FALSE and the ELSE NEXT SENTENCE phrase is specified, statement-1 is ignored, if specified, and control passes to the next executable sentence.

Statement-1, statement-2, or both can each contain an IF statement. In this case, the IF statement is nested.

IF statements within IF statements can be considered paired IF and ELSE combinations, proceeding from left to right. Thus, any ELSE statement encountered applies to the immediately-preceding IF statement that has not been already paired with an ELSE statement.

For information about conditions, refer to "Conditional Statements and Sentences" in Section 8, "PROCEDURE DIVISION Concepts."

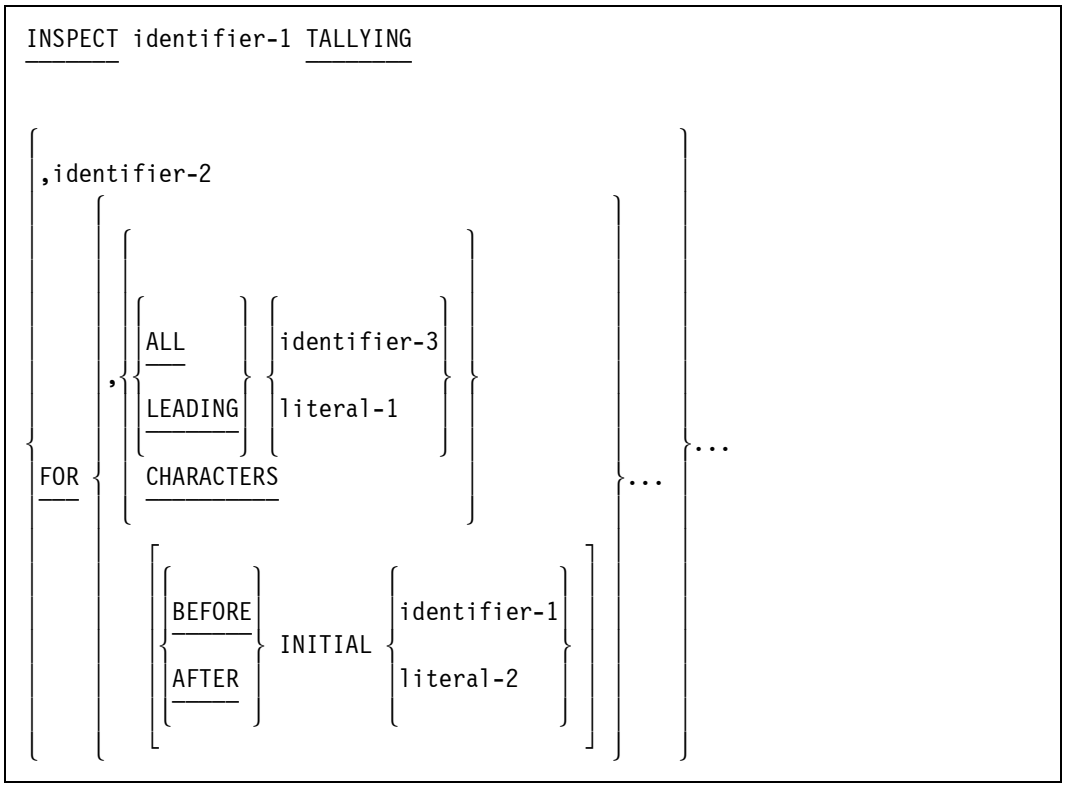
INSPECT

The INSPECT statement tallies, replaces, or tallies and replaces occurrences of single characters or groups of characters in a data item.

When a sending item and a receiving item in the same INSPECT statement share a part, but not all, of their storage areas, the result of the execution of the statement is undefined.

Format	Explanation
1	The INSPECT...TALLYING format tallies single characters or groups of characters.
2	The INSPECT...REPLACING format replaces single characters or groups of characters.
3	The INSPECT...TALLYING and REPLACING format tallies and replaces single characters or groups of characters; this format combines Formats 1 and 2.

Format 1



Explanation of Format 1

Note: *Unpredictable results can occur if identifier-1 is longer than 4095 bytes or if any one of identifier-3 through identifier-n is longer than 255 bytes. This is a permanent restriction. Avoid programs that violate it.*

identifier-1

Identifier-1 must reference either a group item or any category of elementary data item described implicitly or explicitly as USAGE IS DISPLAY.

identifier-2

Identifier-2 must reference an elementary numeric data item.

The contents of the data item referenced by identifier-2 are not initialized by execution of the INSPECT statement.

ALL

If the ALL phrase is specified, the contents of the data item referenced by identifier-2 are incremented by one for each occurrence of literal-1 matched within the contents of the data item referenced by identifier-1.

LEADING

If the LEADING phrase is specified, the contents of the data item referenced by identifier-2 are incremented by one for each contiguous occurrence of literal-1 matched within the contents of the data item referenced by identifier-1. The leftmost such occurrence must be at the point where comparison began in the first comparison cycle in which literal-1 was eligible to participate.

CHARACTERS

If the CHARACTERS phrase is specified, the contents of the data item referenced by identifier-2 are incremented by one for each character matched within the contents of the data item referenced by identifier-1.

identifier-3

Identifier-3 through identifier-n must reference elementary alphabetic, alphanumeric, or numeric items described implicitly or explicitly as USAGE IS DISPLAY.

literal-1, literal-2

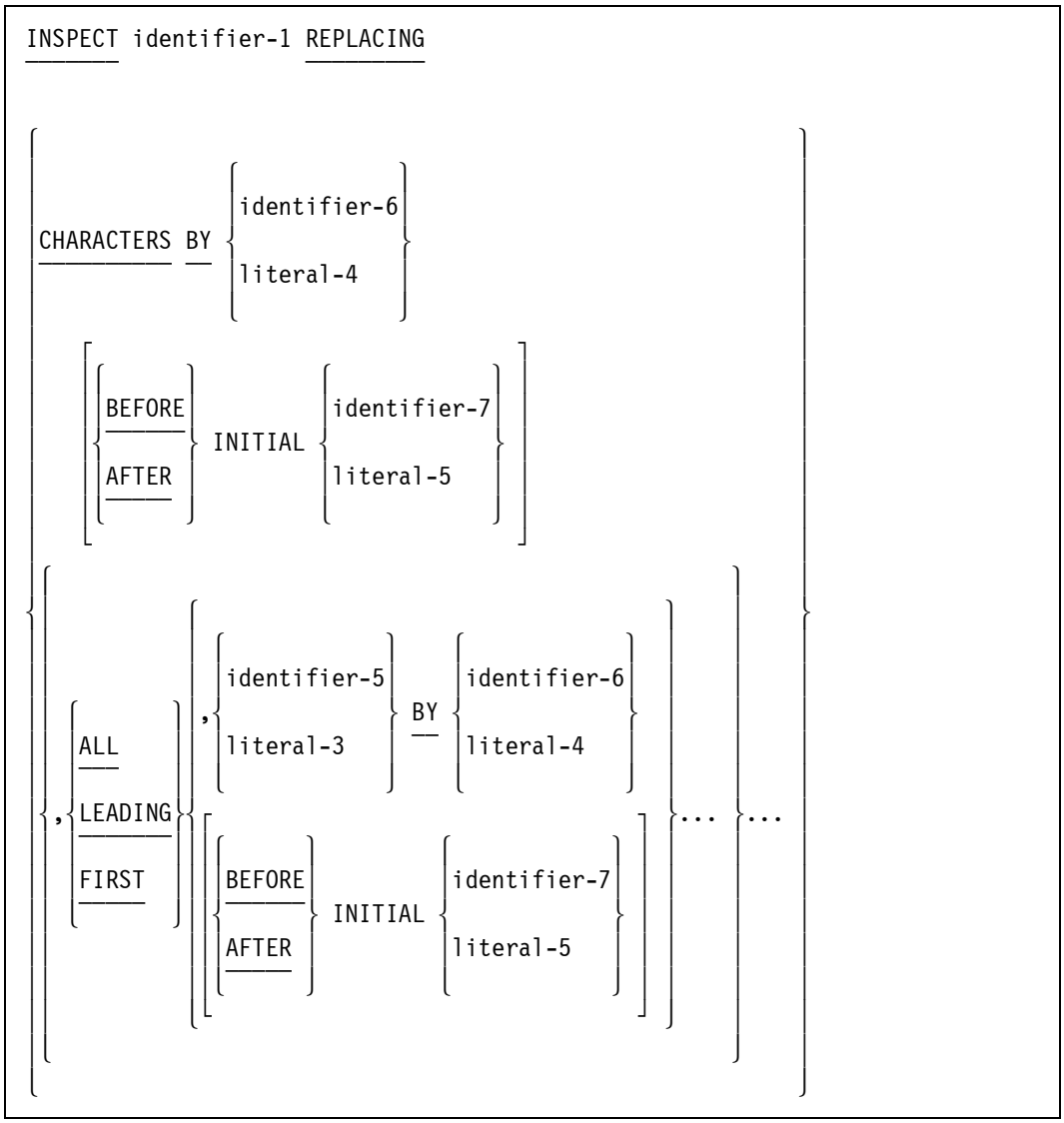
If either literal-1 or literal-2 is a figurative constant, the figurative constant refers to an implicit, one-character data item.

Each literal must be nonnumeric and can be any figurative constant except ALL.

For More Information

For information on the BEFORE and AFTER phrases, see "Establishing Boundaries for the BEFORE or AFTER Phrase" in this section.

Format 2



Explanation of Format 2

Note: *Unpredictable results can occur if identifier-1 is longer than 4095 bytes or if any one of identifier-3 through identifier-n is longer than 255 bytes. This is a permanent restriction. Avoid programs that violate it.*

identifier-1

Identifier-1 must reference either a group item or any category of elementary data item described implicitly or explicitly as USAGE IS DISPLAY.

identifier-5

Identifier-5 through identifier-n must reference elementary alphabetic, alphanumeric, or numeric items described implicitly or explicitly as USAGE IS DISPLAY.

CHARACTERS

When the CHARACTERS phrase is used, literal-4, literal-5, or the size of the data item referenced by identifier-6 or identifier-7 must be 1 character long. Each character matched in the contents of the data item referenced by identifier-1 is replaced by literal-4.

literal-4 and identifier-6

The size of the data referenced by literal-4 or identifier-6 must equal the size of the data referenced by literal-3 or identifier-5. When a figurative constant is used as literal-4, the size of the figurative constant equals the size of literal-3 or the size of the data item referenced by identifier-5.

literal-3

When a figurative constant is used as literal-3, the data referenced by literal-4 or identifier-6 must be 1 character long.

Each literal must be nonnumeric and can be any figurative constant except ALL.

ALL

When the adjective ALL is specified, each occurrence of literal-3 matched in the contents of the data item referenced by identifier-1 is replaced by literal-4.

LEADING

When the adjective LEADING is specified, each contiguous occurrence of literal-3 matched in the contents of the data item referenced by identifier-1 is replaced by literal-4, provided that the leftmost occurrence is at the point where comparison began in the first comparison cycle in which literal-3 was eligible to participate.

FIRST

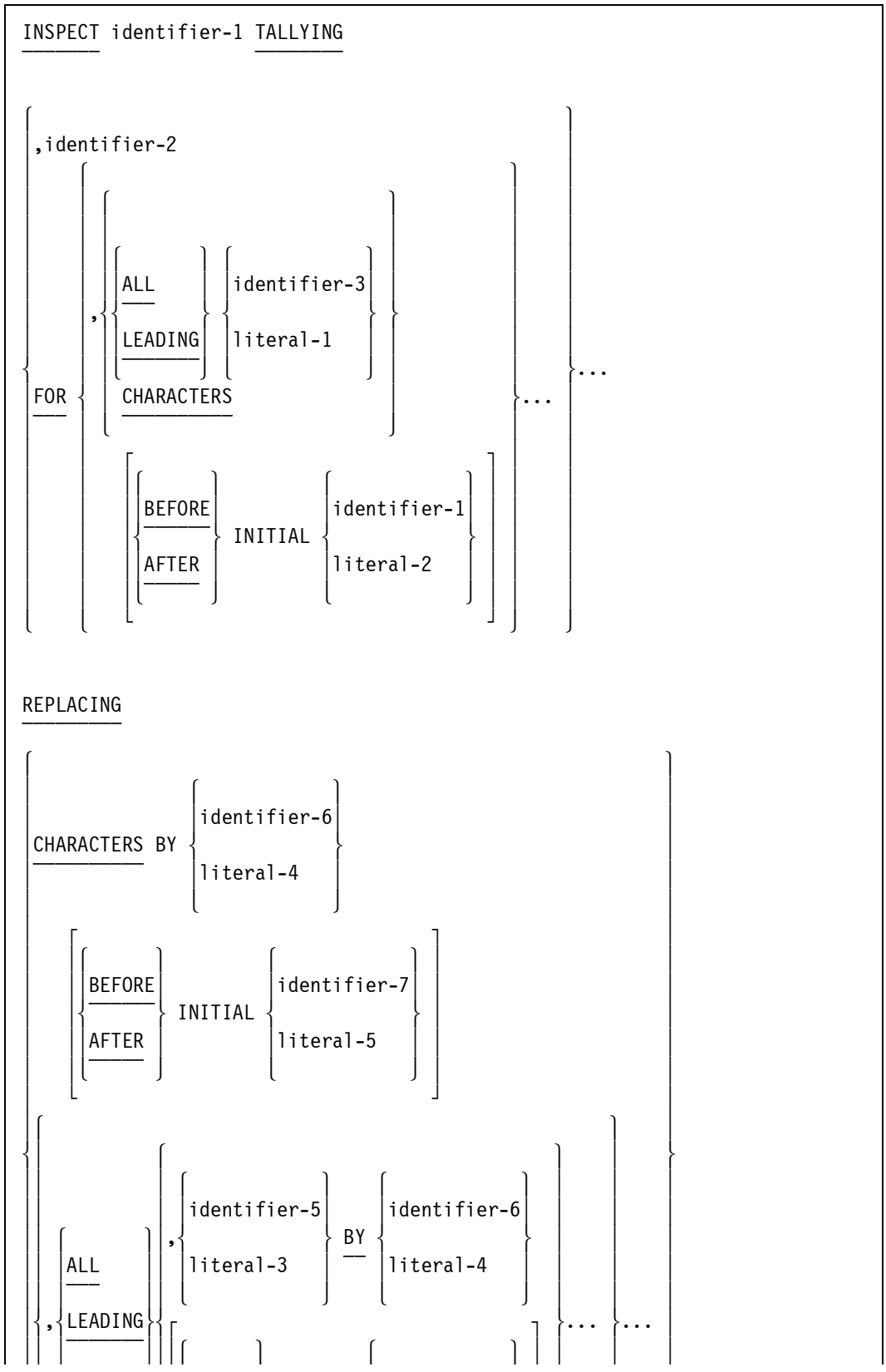
When the adjective FIRST is specified, the leftmost occurrence of literal-3 matched in the contents of the data item referenced by identifier-1 is replaced by literal-4.

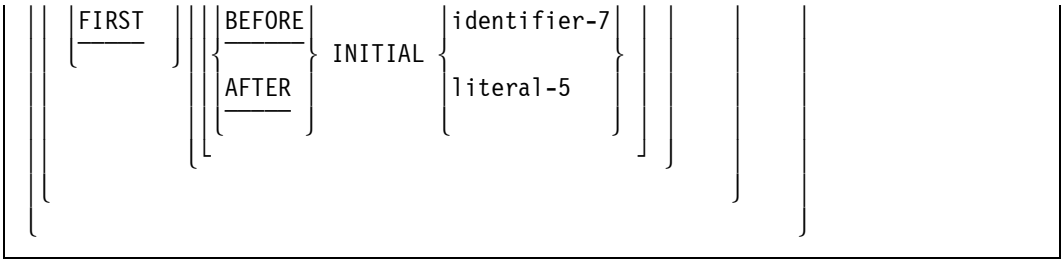
BY

The required words ALL, LEADING, and FIRST are adjectives that apply to each succeeding BY phrase until the next adjective appears.

Format 3

A Format 3 INSPECT statement is interpreted and executed as if two successive INSPECT statements specifying the same identifier-1 were written. One statement is a Format 1 statement with TALLYING phrases identical to those specified in the Format 3 statement; the other statement is a Format 2 statement with REPLACING phrases identical to those specified in the Format 3 statement. The rules for matching and counting apply to the Format 1 statement; the rules for matching and replacing apply to the Format 2 statement.


INSPECT identifier-1 TALLYING
,identifier-2
FOR
,
ALL
LEADING
identifier-3
literal-1
CHARACTERS
BEFORE
AFTER
INITIAL
identifier-1
literal-2
REPLACING
CHARACTERS BY
identifier-6
literal-4
BEFORE
AFTER
INITIAL
identifier-7
literal-5
,
ALL
LEADING
identifier-5
literal-3
BY
identifier-6
literal-4



Explanation of Format 3

Note: *Unpredictable results can occur if identifier-1 is longer than 4095 bytes or if any one of identifier-3 through identifier-n is longer than 255 bytes. This is a permanent restriction. Avoid programs that violate it.*

Identifier-1 must reference either a group item or any category of elementary data item described implicitly or explicitly as USAGE IS DISPLAY.

Identifier-2 must reference an elementary numeric data item.

Identifier-3 through identifier-n must reference elementary alphabetic, alphanumeric, or numeric items described implicitly or explicitly as USAGE IS DISPLAY.

If either literal-1 or literal-2 is a figurative constant, the figurative constant refers to an implicit, 1-character data item.

The size of the data referenced by identifier-6 or literal-4 must equal the size of the data referenced by identifier-5 or literal-3. When a figurative constant is used as literal-4, the size of the figurative constant equals the size of literal-3 or the size of the data item referenced by identifier-5.

When the CHARACTERS phrase is used, literal-4, literal-5, or the size of the data item referenced by identifier-6 or identifier-7 must be 1-character long.

When a figurative constant is used as literal-3, the data referenced by literal-4 or identifier-6 must be one character long.

Each literal must be nonnumeric and can be any figurative constant except ALL.

Note that with undigit literals each pair of hexadecimal digits corresponds to one 8-bit character. (Undigit literals are an extension to ANSI X3.23-1974 COBOL.)

Inspection

The process of inspection has the following three stages:

- The comparison cycle
- The mechanism for tallying, replacing, or both tallying and replacing
- The establishment of boundaries for the BEFORE or AFTER phrase

Inspection begins at the leftmost character position of the data item referenced by identifier-1, regardless of its class, and proceeds from left to right to the rightmost character position.

Comparison Cycle

All references to identifier-1 also apply to identifier-3, identifier-4, identifier-5, identifier-6, and identifier-7.

All references to literal-1, literal-2, literal-3, literal-4, and literal-5 also apply to the contents of the data item referenced by identifier-3, identifier-4, identifier-5, identifier-6, and identifier-7, respectively.

The INSPECT statement treats the contents of the data item referenced by identifier-1 as follows:

- If identifier-1 is described as alphanumeric, the INSPECT statement treats the contents of each such identifier as a character string.
- If identifier-1 is described as alphanumeric-edited, numeric-edited, or unsigned numeric, the INSPECT statement inspects the data item as if it were redefined as alphanumeric and as if the INSPECT statement were written to reference the redefined data item.
- If identifier-1 is described as signed numeric, the INSPECT statement inspects the data item as if it had been moved to an unsigned numeric data item of the same length, and as if the INSPECT statement were written to reference the redefined data item.

Tallying and Replacing

During inspection of the contents of the data item referenced by identifier-1, each properly matched occurrence of literal-1 is tallied (Formats 1 and 3), and/or each properly matched occurrence of literal-3 is replaced by literal-4 (Formats 2 and 3).

The comparison cycle, which determines the occurrences of literal-1 to be tallied and the occurrences of literal-3 to be replaced, occurs as follows:

- The operands of the TALLYING and REPLACING phrases are considered in the order in which they are specified in the INSPECT statement from left to right. The first literal-1/literal-3 is compared with an equal number of contiguous characters, starting with the leftmost character position in the data item referenced by identifier-1. Literal-1/literal-3 and the portion of the contents of identifier-1 match only if they are equal on a character-for-character basis.
- If no match occurs in the comparison of the first literal-1/literal-3, the comparison is repeated with each successive literal-1/literal-3, if any, until either a match is found or no subsequent literal-1/literal-3 remains. When no subsequent literal-1/literal-3 remains, the character position in identifier-1 immediately to the right of the leftmost character position in the last comparison cycle is considered as the leftmost character position, and the comparison cycle begins again with the first literal-1/literal-3.

- Whenever a match occurs, tallying, replacing, or both occurs. The character position in identifier-1 immediately to the right of the rightmost character position that participated in the match is now considered to be the leftmost character position of the data item referenced by identifier-1, and the comparison cycle starts again with the first literal-1/literal-3.
- The comparison operation continues until the rightmost character position of identifier-1 has participated in a match or has been considered as the leftmost character position. When this situation occurs, inspection is terminated.
- If the CHARACTERS phrase is specified, an implied, 1-character operand participates in the cycle previously described , except that no comparison with the contents of identifier-1 takes place. This implied character is always considered to match the leftmost character of the contents of the data item referenced by identifier-1.

For More Information

For a description of move operations, refer to "MOVE" in this section.

Establishing Boundaries for the BEFORE or AFTER Phrase

The comparison operation defined earlier is performed by the BEFORE and AFTER phrases as follows:

- If the BEFORE or AFTER phrase is not specified, literal-1/literal-3 or the implied operand of the CHARACTERS phrase participates in the comparison cycle.
- If the BEFORE phrase is specified, literal-1/literal-3 or the implied operand of the CHARACTERS phrase participates only in comparison cycles that involve the contents of identifier-1 from its leftmost character position up to, but not including, the first occurrence of literal-2/literal-5 within the contents of identifier-1. The position of the first occurrence is determined before the beginning of the first cycle of the comparison operation.

  If, on any comparison cycle, literal-1/literal-3 or the implied operand of the CHARACTERS phrase is not eligible to participate, this variable does not match the contents of identifier-1. If literal-2/literal-5 does not occur within the contents of identifier-1, then literal-1/literal-3 or the implied operand of the CHARACTERS phrase participates in the comparison operation as if the BEFORE phrase had not been specified.
- If the AFTER phrase is specified, literal-1/literal-3 or the implied operand of the CHARACTERS phrase can participate only in comparison cycles that involve the contents of identifier-1 beginning with the character position immediately to the right of the rightmost character position of the first occurrence of literal-2/literal-5 within the contents of identifier-1. The position of this first occurrence is determined before the beginning of the first cycle of the comparison operation.

  If, on any comparison cycle, literal-1/literal-3 or the implied operand of the CHARACTERS phrase is not eligible to participate, it is considered not to match the contents of identifier-1. If literal-2/literal-5 does not occur within the contents of identifier-1, literal-1/literal-3 or the implied operand of the CHARACTERS phrase is never eligible to participate in the comparison operation.

Examples of the INSPECT Statement

These examples illustrate the use of the INSPECT statement.

Example 9–11 counts the number of leading zeros in the specified word.

```
INSPECT word TALLYING count FOR LEADING "0".
```

Word	Count
00009876	4
12345	0

Example 9–11. INSPECT TALLYING Statement with LEADING Option

Example 9–12 counts the number of characters before the first Z character.

```
INSPECT word TALLYING count FOR CHARACTERS BEFORE INITIAL "Z".
```

Word	Count
ALPHA	5
ABCDEFZ	6

Example 9–12. INSPECT TALLYING Statement with BEFORE INITIAL Option

Example 9–13 counts the number of characters that occur before the first A character and counts the number of A characters followed immediately by an L character.

```
INSPECT word TALLYING count FOR LEADING "L" BEFORE INITIAL
"A", count-1 FOR LEADING "A" BEFORE INITIAL "L".
```

Word	First Count	Second Count
LARGE	1	0
ANALYST	0	1

Example 9–13. INSPECT TALLYING Statement with LEADING BEFORE Option

Example 9–14 counts the number of L letters, and replaces an A with an E if the A occurs after the first L.

```
INSPECT word TALLYING count FOR ALL "L", REPLACING LEADING
"A" BY "E" AFTER INITIAL "L".
```

Word Before	Count	Word After
DOLLAR	2	DOLLAR
SALAMI	1	SALEMI
LATTER	1	LETTER

Example 9–14. INSPECT TALLYING Statement with FOR ALL REPLACING Option

Example 9–15 replaces all occurrences of the character A with the character G until all occurrences of the character X are inspected.

```
INSPECT word REPLACING ALL "A" BY "G" BEFORE INITIAL "X".
```

Word Before	Word After
ARXAX	GRXAX
HANDAX	HGNDGX

Example 9–15. INSPECT REPLACING ALL Statement with BEFORE INITIAL Option

Example 9–16 counts the number of characters after the first J character, and replaces all occurrences of the letter A by the letter B.

```
INSPECT word TALLYING count FOR CHARACTERS AFTER INITIAL "J"
REPLACING ALL "A" BY "B".
```

Before	Count	Word After
ADJECTIVE	6	BDJECTIVE
JACK	3	JBCK
JUJMAB	5	JUJMBB

Example 9–16. INSPECT TALLYING REPLACING Statement

Example 9–17 replaces X with Y, B with Z, and W with Q for all character after the first R character.

```
INSPECT word REPLACING ALL "X" BY "Y", "B" BY "Z", "W" BY "Q"
AFTER INITIAL "R".
```

Word Before	Word After
RXXBQMY	RYYZQMY
YZACDWBR	YZACDWZR
RAWRXEB	RAQRYEZ

Example 9–17. INSPECT...REPLACING Statement with Literals

Example 9–18 replaces all character with the letter B until an A character is inspected.

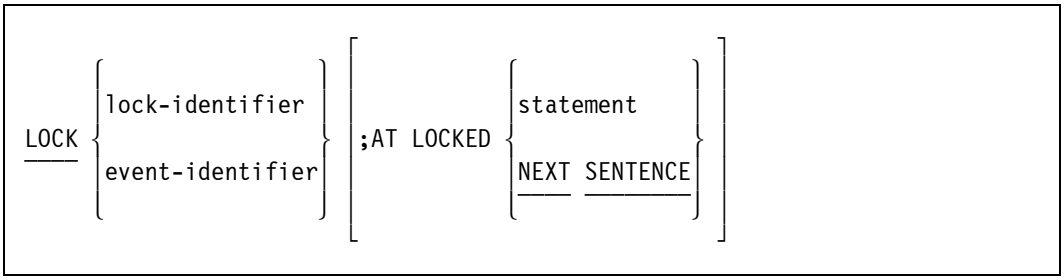
```
INSPECT word REPLACING CHARACTERS BY "B" BEFORE INITIAL "A".
```

Word Before	Word After
12XZABCD	BBBBABCD

Example 9–18. INSPECT...REPLACING CHARACTERS Statement

LOCK (Extension to ANSI X3.23-1974 COBOL)

The LOCK statement is used in an asynchronous processing environment. This statement enables one process to deny related processes access to a particular common storage area until the process has unlocked that area. It also permits a process to test a common storage area for a locked condition.



Explanation of Format

Lock-identifiers must be declared with USAGE IS LOCK.

Event-identifiers must be declared with USAGE IS EVENT or must be event-valued task attributes.

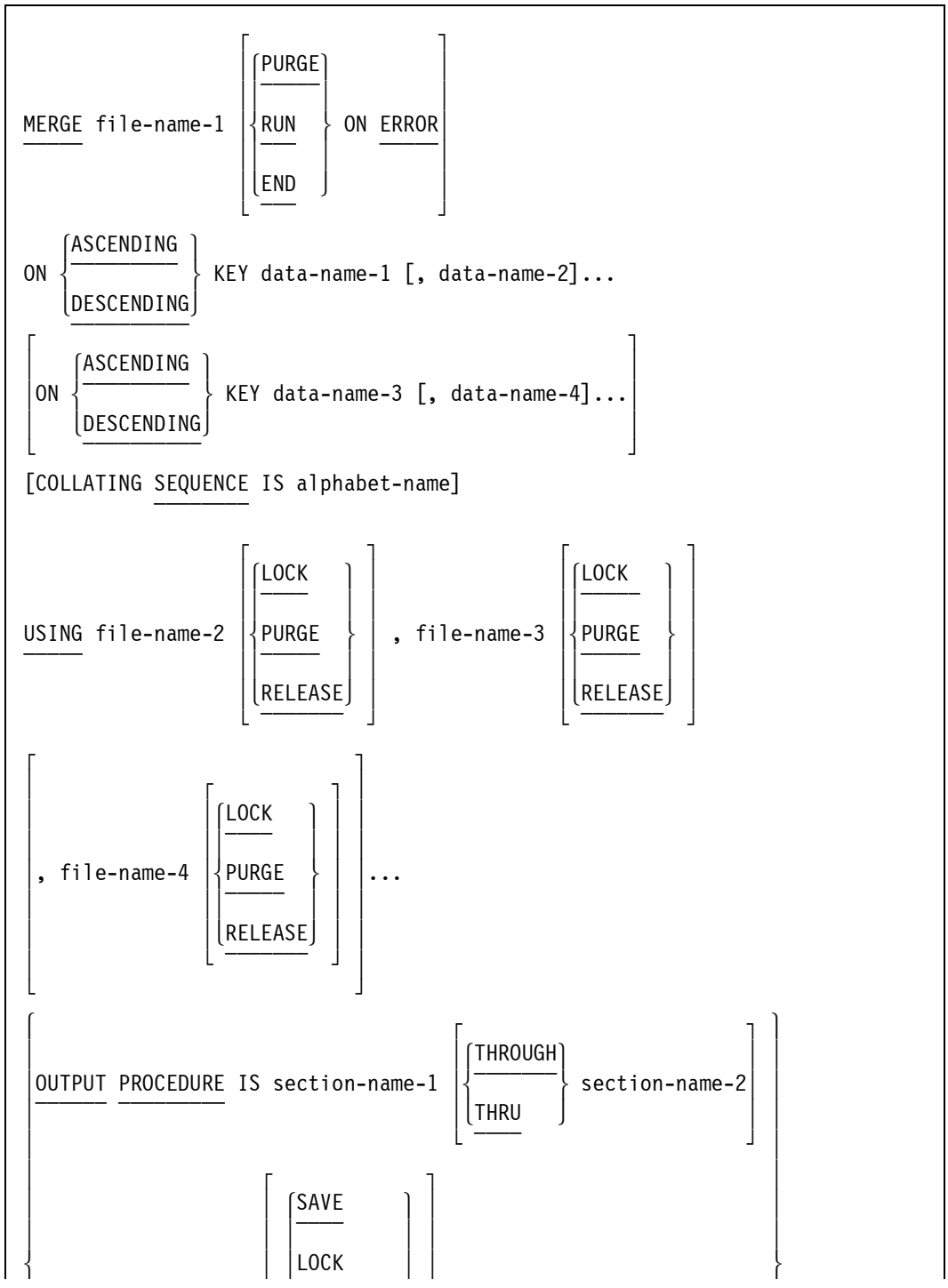
If the AT LOCKED option is specified and the lock-name or event-name is already locked when the LOCK statement is executed, control passes to the statement following the AT LOCKED phrase. If AT LOCKED is not specified, the LOCK statement continues to try the operation until the LOCK has been successfully completed, that is, until the lock-name or event-name has been unlocked from other processes.

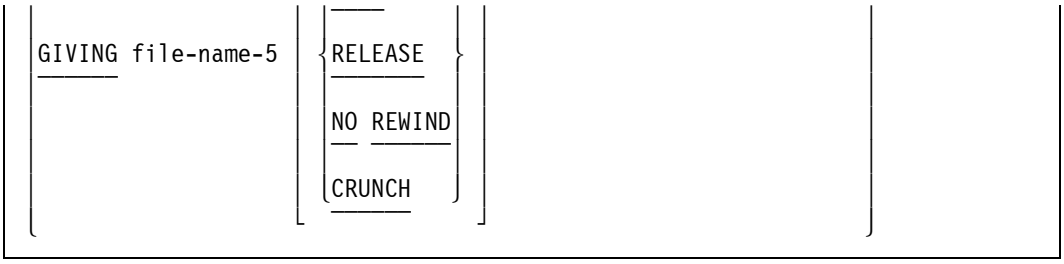
The locking or unlocking of lock-identifier or event-identifier invokes the PROCURE and LIBERATE functions of the operating system and provides priority handling, thus eliminating contention problems.

For information about releasing restrictions on common resources, refer to “UNLOCK (Extension to ANSI X3.23-1974 COBOL)” in this section.

MERGE

The MERGE statement combines two or more files by using a set of specified keys and, during this process, makes records available in merged order in an output procedure or an output file.





Explanation of Format

MERGE

The MERGE statement merges all records contained on file-name-2, file-name-3, and file-name-4. The files referenced in the MERGE statement must not be open when the MERGE statement is executed; they are automatically opened and closed by the merge operation. The terminating function for all files is performed as if a CLOSE statement without optional phrases were executed for each file.

MERGE statements can appear anywhere except in the DECLARATIVES SECTION of the PROCEDURE DIVISION or in an INPUT PROCEDURE or an OUTPUT PROCEDURE associated with a SORT or MERGE statement.

file-names

File-name-1 must be described in a sort-merge file-description entry in the DATA DIVISION. File-name-2, file-name-3, file-name-4, and file-name-5 must be described in a file-description entry that is not a sort-merge file-description entry in the DATA DIVISION. The size of the largest logical records described for file-name-2, file-name-3, file-name-4, and file-name-5 must be equal to the size of the largest logical records described for file-name-1.

When the records in the files referenced by file-name-2, file-name-3, and so on are not ordered as described in the ASCENDING or DESCENDING KEY clauses, the merge operation takes place as previously described except that all improperly ordered data records are placed in the output file or returned to the OUTPUT PROCEDURE immediately after they are read from their respective input files. As a result, when such a condition exists, the output from the MERGE statement is not in a strict ASCENDING or DESCENDING KEY order. (This paragraph is an extension to ANSI X3.23-1974 COBOL.)

No more than one file-name from a multiple-file reel can appear in the MERGE statement.

File-names must not be repeated in the MERGE statement.

ON ERROR (Extension to ANSI X3.23-1974 COBOL)

The ON ERROR option enables you to have control over irrecoverable parity errors when an OUTPUT PROCEDURE is not present in a program. The PURGE option causes all records in a block containing an irrecoverable parity error to be dropped; processing is continued after a system message giving the relative position in the file of the bad block is printed. The RUN option causes the bad block to be used by the program and provides the same message as that defined for the PURGE option. The END option causes termination and is assumed if no option is specified.

ASCENDING or DESCENDING

The data-names following the word KEY are listed from left to right in the MERGE statement in order of decreasing significance and without regard to their division into KEY phrases. In the format, data-name-1 is the major key, data-name-2 is the next most significant key, and so forth.

- When the ASCENDING phrase is specified, the merged sequence is from the lowest value of the contents of the data items identified by the KEY data-names to the highest value, according to the rules for comparison of operands in a relation condition.
- When the DESCENDING phrase is specified, the merged sequence is from the highest value of the contents of the data items identified by the KEY data-names to the lowest value, according to the rules for comparison of operands in a relation condition.

data-names

Data-name-1, data-name-2, data-name-3, and data-name-4 are KEY data-names and are subject to the following rules:

- The data items identified by KEY data-names must be described in records associated with file-name-1.
- KEY data-names can be qualified.
- The data items identified by KEY data-names must not be variable-length items.
- If file-name-1 has more than one record description, all the data items identified by KEY data-names can be described in one of the record descriptions or in any combination of record descriptions. The KEY data-names in each record description need not be described again.
- None of the data items identified by KEY data-names can be described by an entry that either contains an OCCURS clause or is subordinate to an entry that contains an OCCURS clause.

COLLATING SEQUENCE

The collating sequence that applies to the comparison of the specified nonnumeric KEY data items is determined in the following order of precedence:

1. The collating sequence established by the COLLATING SEQUENCE phrase, if specified, in that MERGE statement
2. The collating sequence established as the program-collating sequence

USING (Extension to ANSI X3.23-1974 COBOL)

As many as eight file-names can be specified in the USING clause.

The LOCK, PURGE, and RELEASE options can be used to specify the type of file close operation for the USING files, such as file-name-2, file-name-3, and file-name-4.

If no close action is specified on the file, the association between the file itself and its description in the program is maintained like any other file. You should, therefore, provide a disposition, such as RELEASE, on the input files.

The behavior of the PURGE and RELEASE options is similar to the corresponding options of the CLOSE statement. The behavior of the LOCK option is similar to the LOCK option of the CLOSE statement, except that under **some** circumstances, subsequent reopening of the file during execution of the program is **not prevented**.

When the LOCK option fails to prevent the subsequent reopening of the file, Unisys suggests the use of the USING phrase without any option, followed by an OPEN ... INPUT and a CLOSE ... LOCK either in an OUTPUT PROCEDURE specified in the MERGE statement or immediately after the MERGE statement.

The LOCK option prevents the subsequent reopening of the file. If the file needs to be reopened (for example, when one tape file is input to the MERGE and the same FD is to be used for a different tape file later in the program), Unisys suggests the use of the USING phrase without any option. The USING phrase must be followed by OPEN ... INPUT and CLOSE ... WITH SAVE FOR REMOVAL statements either in an OUTPUT PROCEDURE specified in the MERGE statement or immediately after the MERGE statement.

OUTPUT PROCEDURE

The OUTPUT PROCEDURE clause must consist of one or more paragraphs or sections that appear contiguously in a source program and do not form a part of any other procedure. To make merged records available for processing, the OUTPUT PROCEDURE clause must include execution of at least one RETURN statement. Control must not be passed to the OUTPUT PROCEDURE clause except when a related SORT or MERGE statement is being executed. The OUTPUT PROCEDURE clause can consist of any procedures needed to select, modify, or copy the records being returned one at a time in merged order from file-name-1. The restrictions on the procedural statements in the OUTPUT PROCEDURE clause are as follows:

- The OUTPUT PROCEDURE clause must not contain any transfers of control to points outside the output procedure. ALTER, GO TO, and PERFORM statements in the OUTPUT PROCEDURE clause cannot refer to procedure-names outside the output procedure. COBOL statements are allowed that cause an implied transfer of control to declaratives.
- The OUTPUT PROCEDURE clause must not contain any SORT or MERGE statements.
- The remainder of the PROCEDURE DIVISION must not contain any transfers of control to points inside the output procedures. ALTER, GO TO, and PERFORM statements in the remainder of the PROCEDURE DIVISION cannot refer to procedure-names in the output procedures.

If an OUTPUT PROCEDURE clause is specified, control passes to it during execution of the MERGE statement. The compiler inserts a return mechanism at the end of the output procedure. When control passes to the last statement in the output procedure, the return mechanism ends the merge operation and passes control to the next executable statement after the MERGE statement. Before entering the output procedure, the merge procedure reaches a point at which it can select the next record in merged order when requested. The RETURN statements in the output procedure are the requests for the next record.

GIVING

When GIVING is specified, all merged records in file-name-1 are written on file-name-5.

section-name-1 and section-name-2

Section-name-1 and section-name-2 are the names of OUTPUT PROCEDURE clauses.

THRU and THROUGH

The words THRU and THROUGH are equivalent. They indicate a range of OUTPUT PROCEDURE clauses.

SAVE, LOCK, RELEASE, NO REWIND, and CRUNCH

The SAVE, LOCK, RELEASE, NO REWIND, and CRUNCH options can be used to specify the type of file close operation for the GIVING file-name-5 phrase.

For information on closing files, refer to "CLOSE" in this section.

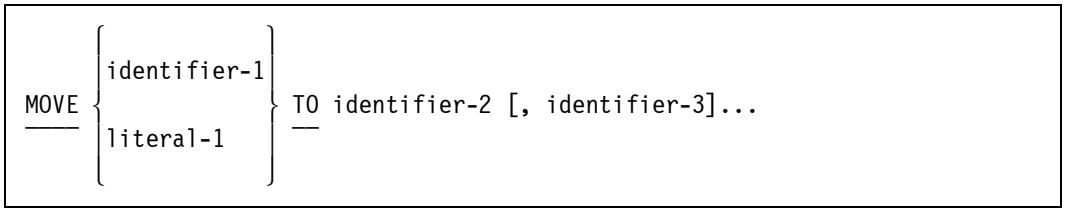
MOVE

The MOVE statement transfers data to one or more receiving data items.

When a sending item and a receiving item in the same MOVE statement share a part, but not all, of their storage areas, the result of the statement execution is undefined.

Format	Explanation
1	This format transfers data to one or more data areas.
2	The MOVE CORRESPONDING format transfers selected items in identifier-1 to selected items in identifier-2. This format transfers items having the same name as an item in the receiving field to that corresponding field.
3	This format transfers selected bit ranges between two BINARY data items.

Format 1



Explanation of Format 1

Identifier-1 and literal-1 represent the sending area.

The data designated by literal-1 or identifier-1 are moved first to identifier-2, then to identifier-3, and so on. The rules referring to identifier-2 also apply to the other receiving areas. Any subscripting or indexing associated with identifier-2, identifier-3, and so forth is evaluated immediately before the data is moved to the respective data item.

An index data item cannot appear as an operand of a MOVE statement.

Any MOVE statement in which the sending and receiving items are both elementary items is considered an elementary move operation. Every elementary item belongs to one of the following categories: numeric, alphabetic, alphanumeric, numeric-edited, alphanumeric-edited, Kanji, or Kanji-edited. These categories are determined by the PICTURE clause. Numeric literals belong to the category numeric; nonnumeric literals belong to the category alphanumeric. The figurative constant ZERO belongs to the category numeric. The figurative constant SPACE belongs to the category alphabetic. All other figurative constants belong to the category alphanumeric.

The following rules apply to an elementary move between these categories:

- The figurative constant SPACE or a numeric-edited, an alphanumeric-edited, or an alphabetic data item must not be moved to a numeric or a numeric-edited data item.
- A numeric literal, the figurative constant ZERO, a numeric data item, or a numeric-edited data item must not be moved to an alphabetic data item.
- A noninteger numeric literal or a noninteger numeric data item must not be moved to an alphanumeric or an alphanumeric-edited data item.
- An undigit literal containing nonnumeric information must not be moved to a numeric, numeric-edited, or alphanumeric-edited data item. Otherwise, subsequent operations on these items are unpredictable. (Undigit literals are an extension to ANSI X3.23-1974 COBOL.)
- When moving the figurative constant ZERO to a BINARY item, the result is a numeric zero (all bits off). When moving the figurative constant ZERO to a COMPUTATIONAL or DISPLAY item, the result is EBCDIC zeros.
- A Kanji data item or a Kanji-edited data item can be moved only to a Kanji data item or a Kanji-edited data item, respectively. All other items cannot be moved to a Kanji data item or a Kanji-edited data item.

All other elementary moves are legal and are performed according to the rules given in the following paragraphs.

Any necessary conversion of data from one form of internal representation to another takes place during legal elementary moves along with any editing specified for the receiving data item, as follows:

- When an alphanumeric-edited or an alphanumeric item is a receiving item, alignment by decimal point and any necessary zero-filling takes place according to the data alignment rules. If the size of the sending item is greater than the size of the receiving item, the excess characters are right-truncated after the receiving item is filled.

  If the sending item is described as signed numeric, the operational sign is not moved. If the operational sign occupies a separate character position, that character is not moved, and the size of the sending item is considered to be one less than its actual size (in terms of standard data format characters).
- When a receiving field is described as alphabetic, justification and any necessary space-filling take place according to data-alignment rules. If the size of the sending item is greater than the size of the receiving item, the excess characters are right-truncated after the receiving item is filled.
- If the receiving item has an associated TYPE clause, the clause contains the data formatted using the type, convention, and language declared for the item. If the item does not have an associated convention or language declared, the system determines the convention or language based on a default hierarchy.

  The sending item must be a nonnumeric literal or an identifier with the USAGE IS DISPLAY clause.
- When a numeric or a numeric-edited item is the receiving item, alignment by decimal point and any necessary zero-filling take place according to data alignment rules, except where zeros are replaced because of editing requirements.
  - When a signed numeric item is the receiving item, the sign of the sending item is placed in the receiving item. Conversion of the representation of the sign takes place as necessary. If the sending item is unsigned, a positive sign is generated for the receiving item.
  - When an unsigned numeric item is the receiving item, the absolute value of the sending item is moved and no operational sign is generated for the receiving item.
  - When an alphanumeric data item is the sending item, data is moved as if the sending item were described as an unsigned numeric integer. If the alphanumeric item contains characters other than the digits 0 through 9, the result in the receiving field is unpredictable.
  - If any digits of a sending COMPUTATIONAL numeric item are greater than 9, they can be changed when moved from the source. If you wish to preserve these values, you must use nonnumeric moves.

Any move operation that is not an elementary move is treated like an alphanumeric-to-alphanumeric elementary move. In such a move, the receiving area is filled without regard to the individual elementary or group items contained in either the sending or the receiving area. This rule applies regardless of the declared or implied usage of either the source or the destination field. A move operation in which either the source field or the destination field is a group item is thus unconditionally treated as a transfer of EBCDIC characters with EBCDIC space-fill if the source is shorter than the destination.

It is important to note that when the source field is a group item and the destination field is an elementary COMPUTATIONAL item, or vice versa, and the source field is shorter than the destination field, the system performs space-fill by inserting single-digit zeros as needed, such that the EBCDIC spaces are always aligned on byte boundaries.

In nonelementary moves, USAGE REAL elementary items are treated like 6-character DISPLAY items, USAGE DOUBLE elementary items are treated like 12-character DISPLAY items, USAGE BINARY items not longer than 11 digits are treated like 6-character DISPLAY items, and USAGE BINARY items longer than 11 digits are treated like 12-character DISPLAY items.

Table 9–9 summarizes the legality of the various types of MOVE statements. For example, a numeric noninteger can be moved to a numeric integer, a numeric noninteger, or a numeric-edited field.

Table 9–9. Comparison of Sending and Receiving Items in MOVE Statements

	Category of Receiving Data Item			
Category of Sending Data Item	**Alphabetic**	**Alphanumeric Edited and Alphanumeric**	**Numeric Integer, Numeric Noninteger, and Numeric-Edited**	**Kanji and Kanji-Edited**
Alphabetic	Yes	Yes	No	No
Alphanumeric	Yes	Yes	Yes	No
Alphanumeric-edited	Yes	Yes	No	No
Numeric integer	No	Yes	Yes	No
Numeric noninteger	No	No	Yes	No
Numeric-edited	No	Yes	No	No
Kanji or Kanji-edited	No	No	No	Yes

For More Information

- Refer to "Aligning Data" in Section 6, "Data Concepts."
- Refer to "SIGN Clause" in Section 7, "DATA DIVISION."

Format 2

```
MOVE { CORRESPONDING } identifier-1 TO identifier-2
     { CORR         }
```

Explanation of Format 2

CORR is an abbreviation for CORRESPONDING.

When the CORRESPONDING phrase is used, both identifiers must be group items. Selected items in identifier-1 are moved to selected items in identifier-2, according to the rules for the CORRESPONDING phrase in Section 8. The results are the same as if each pair of corresponding identifiers were referenced in separate MOVE statements.

An index data item cannot appear as an operand of a MOVE statement.

For More Information

Refer to "CORRESPONDING Phrase" in Section 8, "PROCEDURE DIVISION Concepts."

Format 3 (Extension to ANSI X3.23-1974 COBOL)

```
MOVE identifier-4 TO identifier-5

   { literal-2      }
[  {               }
   { arithmetic-expression-1 }

   { literal-3      }     { literal-4      }
:  {               } : [  {               } ]
   { arithmetic-expression-2 }     { arithmetic-expression-3 }
```

Explanation of Format 3

Identifier-4 represents the sending area and identifier-5 represents the receiving area. An index data item cannot appear as an operand of a MOVE statement.

Format 3 allows bit manipulation and character manipulation. The brackets and the colons are required. The declarations of identifier-4 and identifier-5 must be single-precision word-oriented items (either USAGE BINARY with fewer than 12 digits in the PICTURE clause, or USAGE REAL).

A Format 3 MOVE statement moves bits from identifier-4 into identifier-5 with only the indicated bits of identifier-5 being changed. Literal-2 or arithmetic-expression-1 represents the location in identifier-4 at which the transfer begins (that is, the source bit position). Literal-3 or arithmetic-expression-2 represents the location in identifier-5 at which the transfer begins (that is, the destination bit position). Literal-4 or arithmetic-expression-3 represents the number of bits to be transferred.

The bits in a word-oriented numeric item are numbered 47 through 0, from left to right. Therefore, only 0 through 47 are valid numbers for source and destination bit positions.

The compiler ensures that unsigned USAGE BINARY items never have negative values when they are stored in memory. The compiler unconditionally resets the sign bit of an unsigned binary item before storing it. As a result, bit 46 of such an item is always reset in memory even if it is explicitly set by a Format 3 MOVE statement. It is therefore better to ensure that the program declares items referred to in Format 3 MOVE statements either as signed USAGE BINARY (an S appears in the PICTURE clause) or as USAGE REAL items.

Examples of Format 3

The following statements unpack a BINARY word that contains two 20-bit fields:

```
MOVE A-AND-B-BOTH TO A-ONLY [39:19:20].
MOVE A-AND-B-BOTH TO B-ONLY [19:19:20].
```

The following two statements repack the fields:

```
MOVE B-ONLY TO A-AND-B-BOTH.
MOVE A-ONLY TO A-AND-B-BOTH [19:39:20].
```

MULTIPLY

This statement multiplies numeric data items and sets their values equal to the results.

Format	Multiplies the value of . . .
1	Identifier-1 by the value of identifier-2, identifier-3, and so on. The product is stored in identifier-2, identifier-3, and so on.
2	Identifier-1 or literal-1 by the value of identifier-2 or literal-2. The product is stored in identifier-3, identifier-4, and so on.

Format 1

```
                 ┌              ┐
                 │ identifier-1 │
MULTIPLY         │              │ BY identifier-2 [ROUNDED]
                 │ literal-1    │
                 └              ┘

[, identifier-3 [ROUNDED]]...


[;ON SIZE ERROR imperative-statement]
```

Explanation of Format 1

The value of identifier-1 or literal-1 is multiplied by the value of identifier-2. The value of the identifier-2 is replaced by this product; this process is then repeated for identifier-1 or literal-1 and for identifier-3, and so forth. Each literal must be a numeric literal. Each identifier must refer to a numeric elementary item.

The imperative-statement can be the NEXT SENTENCE phrase.

The composite of operands (which is the hypothetical data item resulting from the superimposition of all receiving data items of a given statement aligned by their decimal points) must not contain more than 18 digits.

Format 2

```
                 ┌              ┐       ┌              ┐
                 │ identifier-1 │       │ identifier-2 │
MULTIPLY         │              │ BY    │              │
                 │ literal-1    │       │ literal-2    │
                 └              ┘       └              ┘

GIVING identifier-3 [ROUNDED] [, identifier-4 [ROUNDED]]...


[;ON SIZE ERROR imperative-statement]
```

Explanation of Format 2

When Format 2 is used, the value of identifier-1 or literal-1 is multiplied by the value of identifier-2 or literal-2; the result is stored in identifier-3, identifier-4, and so forth.

Each literal must be a numeric literal.

Each identifier following the word GIVING must refer either to an elementary numeric item or to an elementary numeric-edited item.

The composite of operands (which is the hypothetical data item resulting from the superimposition of all receiving data items of a given statement aligned by their decimal points) must not contain more than 18 digits.

The imperative-statement can be the NEXT SENTENCE phrase.

General Rules

The following information applies to all the formats.

When a sending item and a receiving item in the same MULTIPLY statement share a part, but not all, of their storage areas, the result of the statement execution is undefined.

Note that unpredictable results occur in a Format 1 MULTIPLY statement if the same operand appears more than once in the list of multiplicands that follows the word BY in the statement. For example, assuming X contains the value 9 and Y contains the value 2, the value of X is 18 rather than 72 after execution of the following statement:

```
MULTIPLY Y BY X, X, X
```

OPEN

The OPEN statement associates the logical file description in the program with its physical representation in the operating environment, and makes the file available for processing. The OPEN statement also specifies what kinds of I/O operations are allowed for that file. You might designate, for example, an input file to be read by the program and an output file to which records are written by the program.

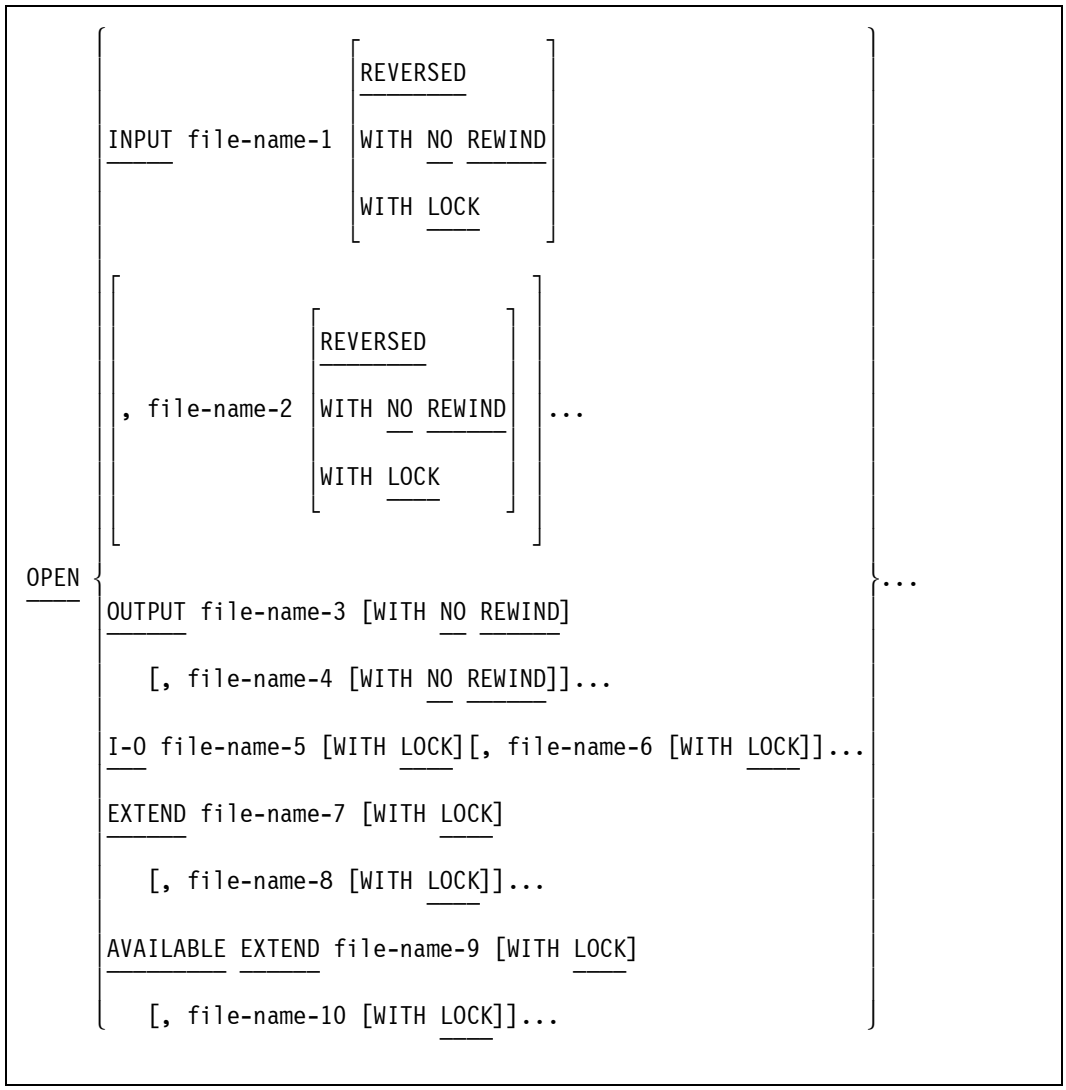
Before your program executes a READ, SEEK, START, or WRITE statement against a file, it must execute an OPEN statement against the file so that the file is in open mode. The results of using means other than the OPEN verb (for example, file attributes) to cause the files to be open are unpredictable.

Note: *After an OPEN statement establishes the association between a physical file and the logical description of that file in the program, the association remains in effect until a CLOSE statement severs it through an explicit disposition that has that effect, such as CLOSE WITH SAVE or CLOSE WITH RELEASE.*

Format	Explanation
1	Opens a file for sequential, relative, and indexed I/O applications
2	Opens a port file

Format 1: Sequential, Relative, and Indexed I/O

Format 1 is used for files with sequential, relative, and indexed organization. A sequential I/O operation is performed when a program acts on a sequential file, a relative I/O operation is performed when a program acts on a relative file, and an indexed I/O operation is performed when a program acts on an indexed file. For example, a program randomly accessing a sequential file performs a sequential I/O operation, and a program sequentially accessing a relative file performs a relative I/O operation.



Note: *The AVAILABLE EXTEND, EXTEND, NO REWIND, and REVERSED phrases apply only to sequential files.*

Explanation of Format 1

The INPUT phrase opens the file for reading only. The OPEN INPUT statement makes the file available for reading by positioning the current-record pointer at the first record in the file. The first read operation of an open file causes an At End condition if the file has no records or if it is an optional file that is not present.

The OUTPUT phrase opens the file for writing only. A successful OPEN OUTPUT statement creates a new file, provided you have not previously closed and retained a file with the same name.

The I-O phrase opens the file for both reading and writing. Because this phrase implies the existence of the file, you cannot use it to create a file. The OPEN I-O statement positions the current-record pointer at the first record present in the file. The first read operation of the file causes an At End condition if the file has no records.

The EXTEND phrase opens a sequential file so that new records can be written to the end. The OPEN EXTEND statement positions the current-record pointer at the end of the file. Subsequent records that are written are added as if the file had been opened with the OPEN OUTPUT statement. You cannot use the EXTEND phrase with multiple-reel files.

The AVAILABLE EXTEND phrase opens a sequential file so that new records can be written to the end of the file. If the file cannot be opened, the program is not suspended and operator intervention is not required. The program can determine the cause of the open failure by examining the value of the AVAILABLE file attribute.

File-name-1 through file-name-10 designate files that must be named in the SELECT clause and described in the INPUT-OUTPUT SECTION and FILE SECTION of your program. You can open files of differing organizations and access types with one OPEN statement. The REVERSED option designates that the sequential file is read in reverse order, beginning with the last record first. An OPEN INPUT REVERSED statement positions the current-record pointer at the end of the file. This statement is meaningful for sequential single-reel files, but is ignored if the statement does not apply to the storage medium on which the file resides.

The NO REWIND option opens the sequential file without repositioning the current-record pointer. The OPEN INPUT WITH NO REWIND or OPEN OUTPUT WITH NO REWIND statements are meaningful for sequential files, but are ignored if these statements do not apply to the storage medium on which the file resides. The WITH NO REWIND phrase is useful for opening the next file on a multiple-file tape.

The LOCK phrase opens the file if it is available. The WITH LOCK phrase is meaningful for mass-storage files; it is ignored for all other files.

The LOCK operation works as follows:

- If another program has already opened the file, the system suspends your program until the file is exclusively available.
- If no program has opened the file, the file is opened.

The LOCK phrase is controlled by the EXCLUSIVE file attribute. When you open and lock a file using an OPEN WITH LOCK statement, the value of the EXCLUSIVE attribute becomes TRUE when the file is opened. If you close the file without retaining it, the value of the EXCLUSIVE attribute becomes FALSE when the file is closed. This value ensures that other programs can access that file.

You should avoid directly using the EXCLUSIVE file attribute because you might interfere with the lock mechanism and cause unexpected results in your program.

General Rules

You can open and close a file many times in a program by using any of the syntax options. To reopen a file after you open it initially, you must first close the file without specifying the REEL, UNIT, or LOCK phrase.

Because the sort module has opening routines for files associated with the USING and the GIVING phrases of the SORT and MERGE statements, your program must not open these files with an OPEN statement.

Example

Many programs open all files at the beginning of the PROCEDURE DIVISION. Although you can use a separate OPEN statement for each file, typically you open many files with one OPEN statement.

A sample of this type of OPEN statement is shown in Example 9–19. This example does the following operations:

- Creates a file called GUEST-FOLIO
- Opens two existing files, ROOM-STATUS and RESERVATIONS, for update; allows no other programs to access these files while they are open
- Opens the file ROOM-AVAIL for reading in reverse order
- Opens the file CREDIT-CHARGE so that records can be added at the end of the file, and allows no other programs to access the file while it is open

```
PROCEDURE DIVISION.
OPEN-PARA.
  OPEN OUTPUT GUEST-FOLIO,
       I-O ROOM-STATUS WITH LOCK, RESERVATIONS WITH LOCK,
       INPUT ROOM-AVAIL REVERSED,
       AVAILABLE EXTEND CREDIT-CHARGE WITH LOCK.
```

Example 9–19. Coding an OPEN Statement

For More Information

- For information about handling an At End condition that occurs during a read operation, refer to "READ" in this section.
- For information about the meanings of status code values, refer to "I/O Status" in Section 5, "ENVIRONMENT DIVISION."
- For information about the AVAILABLE and EXCLUSIVE attributes, refer to the *File Attributes Reference Manual.*

Open Modes

A successfully opened file is considered to be in an open mode. The open modes include INPUT, OUTPUT, I-O, and EXTEND. These modes correspond to the phrases of the same name allowed for the OPEN statement. The type of open mode, the file organization, and the method of file access determine the I/O statements that are permitted in your program.

Table 9–10 shows the I/O statements you can perform with each open mode and the method of file access for files with sequential organization.

Table 9–10. I/O Statements Allowed for Open Files with Sequential Organization

Open Mode	Access	Statements Allowed
INPUT	S, R	READ
OUTPUT	S, R	WRITE
I-O	S	READ, REWRITE
	R	READ, WRITE, REWRITE
EXTEND	S	WRITE

Legend

S Sequential
R Random

Table 9–11 shows the I/O statements you can use with each open mode and the method of file access for files with relative or indexed organization.

Table 9–11. I/O Statements Allowed for Open Relative or Indexed Files

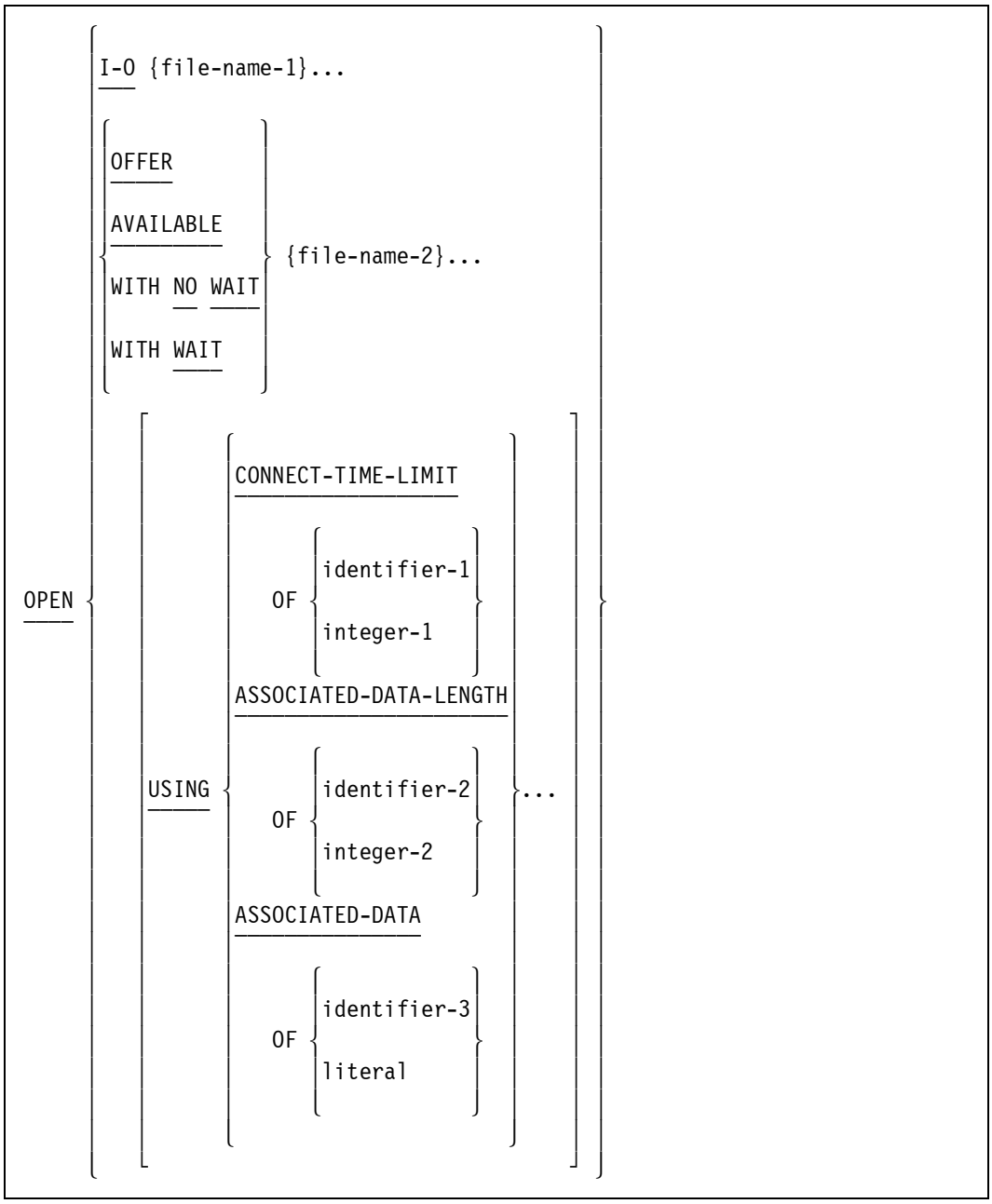
Open Mode	Access	Statements Allowed
INPUT	S, D	READ, START
	R	READ
OUTPUT	S, R, D	WRITE
I-O	S	READ, REWRITE, START, DELETE
	R	READ, WRITE, REWRITE, DELETE
	D	READ, WRITE, REWRITE, START, DELETE

Legend

S Sequential
R Random
D Dynamic

Format 2: Opening Port Files (Extension to ANSI X3.23-1974 COBOL)

Before a dialogue can be established between two programs that communicate through a port file, each program must open a subfile. The system establishes the logical communication path between the two port files, provided that the connection descriptions of the two files match. Because the two programs run independently, one program cannot affect when the subfile of the other program is open. The OPEN statement for port files enables a program to request that a dialogue be established with a correspondent endpoint without the programming determining whether the subfile of the correspondent endpoint is open.



Explanation of Format 2

The I-O phrase opens the file for both input and output operations. No further options can be specified. The WAIT option is the default control option.

File-name-1 can designate the port file to open.

File-name-2 must designate the port file to open

The OFFER phrase opens the port file and returns control to the program after you determine the availability of the host by using the YOURHOST file attribute. If the

program specifies subfiles, the OPEN OFFER statement returns control to the program after the availability of the correspondent endpoint is determined for all subfiles. The dialogue is then established in parallel with the processing of the program.

The AVAILABLE phrase opens a subfile that matches a subfile that has already been offered. If no subfile has been offered, the subfile remains closed and is not considered for subsequent matching. This phrase is the same as an OPEN WAIT statement with the AVAILABLEONLY file attribute set to TRUE.

The NO WAIT phrase opens the port file and returns control to the program as soon as the OPEN NO WAIT statement is checked for correctness. The dialogue is established while the program continues processing.

Note: *Your program might be significantly delayed while the availability of the correspondent endpoint is determined. If this delay is undesirable, use the OPEN NO WAIT statement.*

The WAIT phrase opens the port file and suspends the program until either the dialogue is established or the open operation fails. If the program specifies subfiles, the OPEN WAIT statement suspends the program until the open operation on each subfile succeeds or fails.

The CONNECT-TIME-LIMIT option indicates the time in minutes that the system allows for establishing a dialogue. The open operation fails if a correspondent endpoint is not found in that amount of time. If you do not use the CONNECT-TIME-LIMIT phrase or if you specify a value of 0 (zero), the system allows an indefinite amount of time to establish a dialogue. An error results or if you specify a negative or noninteger value.

The ASSOCIATED-DATA-LENGTH option specifies the number of characters of associated data to be sent. If you do not specify the length of the associated data and the associated data is a data item, your program uses the actual length of the data. If you do specify the length of the associated data, the length value must be less than or equal to the actual length of the data. An error results if the length specified is not a single-precision integer value.

The ASSOCIATED-DATA option transfers data to the correspondent endpoint along with the open request for some types of networks.

Identifier-1 through identifier-3 designate elementary integer data items. Identifier-3 can also designate a group data item.

Integer-1 and integer-2 must be integer numeric literals.

This literal specifies either a nonnumeric or hex literal.

General Rules

In addition to deciding the method of opening a subfile, you need to designate the subfiles to open. If you want to designate more than one subfile, make sure your program includes code to perform the following tasks:

1. Specify the total number of subfiles in your program by using the *CHANGE ATTRIBUTE MAXSUBFILES TO VALUE attribute-value* statement.

2. Specify the subfiles to open by using the *SELECT port-file ASSIGN TO PORT; ACTUAL KEY IS subfile-num* clause.
3. Declare subfile-num and attribute-value in the WORKING-STORAGE SECTION.

Table 9–12 shows how the ACTUAL KEY clause value decides the subfile that is opened.

Table 9–12. Designating Subfiles to Open

Actual Key Value	Explanation
0	Opens every closed subfile
Nonzero	Opens the specified subfile
None	Opens a single subfile
Greater than the MAXSUBFILES value or a negative number	Returns a BADSUBFILEINDEX run-time error in the SUBFILERROR attribute

Examples

In Examples 9–20 through 9–22, it is assumed that earlier in the program the port files were declared using an ACTUAL KEY clause to specify a subfile index and that the subfile index was set to a particular subfile.

Example 9–20 opens port file PORTFILE1 and offers it for matching. The program is suspended until a matching subfile is found or until 10 minutes have elapsed.

```
OPEN WAIT PORTFILE1
     USING CONNECT-TIME-LIMIT OF 10.
```

Example 9–20. Coding an OPEN WAIT Statement

Example 9–21 opens PORTFILE1 with the control option set to OFFER. Control is returned to the program when it is determined that the host can be reached. When a matching subfile is found, the information "MYDATA" is sent to the other process as associated data during the OPEN process.

```
OPEN OFFER PORTFILE1
           ASSOCIATED-DATA OF "MYDATA".
```

Example 9–21. Coding an OPEN OFFER Statement

Example 9–22 offers port files PORTFILE1 and PORTFILE2 for matching. The NO WAIT option indicates that control is returned to the program as soon as the open process begins. Fourteen characters of information are sent to the other process as associated data with the connect request, beginning at the location pointed to by GROUP-ITEM.

```
OPEN NO WAIT PORTFILE1 PORTFILE2
     USING ASSOCIATED-DATA OF GROUP-ITEM
           ASSOCIATED-DATA-LENGTH OF 14.
```

Example 9–22. Coding an OPEN NO WAIT Statement

For More Information

- For information about another type of open operation used to establish a dialogue, refer to "AWAIT-OPEN (Extension to ANSI X3.23-1974 COBOL)" in this section.
- For step-by-step information on coding port file applications, refer to the *I/O Subsystem Programming Guide.*
- For references to file attributes, refer to the *File Attributes Reference Manual.*

Handling of Error Conditions

If the program does not contain a FILE STATUS clause in the SELECT statement for the file that is the object of the OPEN verb, and does not contain a USE AFTER STANDARD ERROR PROCEDURE ON <filename> in DECLARATIVES for the file, then the compiler generates object code that unconditionally terminates the program if an error condition arises during execution. The program is terminated even in those cases in which execution should continue (OPEN AVAILABLE [EXTEND], for example). To be able to continue after such an error and to provide a way for the system to inform the program that an error has occurred, the program must contain either a FILE STATUS clause in the SELECT statement for the file, or a USE procedure in DECLARATIVES referencing the file, or both.

For more information, refer to "Handling I/O Exception Conditions" in Section 8, "PROCEDURE DIVISION Concepts."

I/O Status Value

The system returns a value indicating the result of an OPEN statement. You can access this value by including *SELECT file-name FILE STATUS IS data-name* in the program. The system moves a value into the designated data-name storage area after the program performs the OPEN. You can then use an IF statement to test the value of the data-name and take the desired action depending on the result. If you choose to omit an action for the result of an open operation, the system provides a default action for each result.

Table 9–13 shows the I/O status values and their meanings.

Table 9–13. I/O Status Values for OPEN Statement

Value	Explanation
00	Control was returned to the program after the OPEN statement completed correctly. In the case of a port file, the open operation might be pending.
81	An error was detected while the file was being opened.

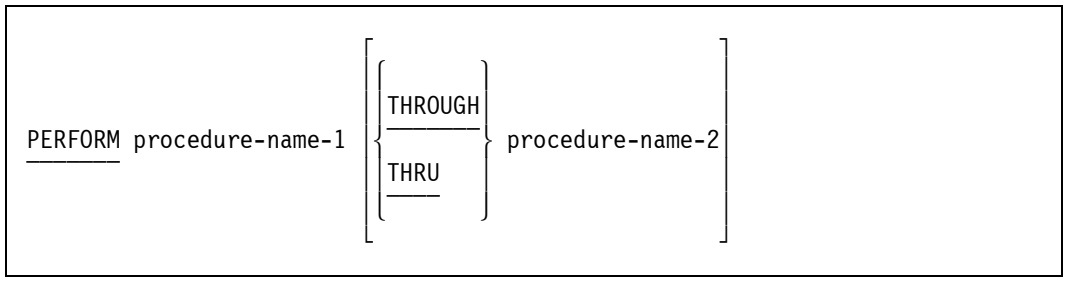
Note: *The I/O status value "81" is an extension to ANSI X3.23-1974 COBOL.*

PERFORM

The PERFORM statement transfers control explicitly to one or more procedures and returns control implicitly whenever execution of the specified procedure is complete.

Format	Explanation
1	Executes the specified procedures and then transfers control to the next instruction after the PERFORM statement.
2	Executes procedures a specified number of times.
3	Executes procedures until a specified condition is TRUE.
4	Executes procedures repetitively, increasing or decreasing the value of a counter data item once for each repetition until one or more conditions are satisfied. The conditions that end the repetitions are tested before the procedures are executed.

Format 1



Explanation of Format 1

Format 1 is the basic PERFORM statement. A procedure referenced by this type of PERFORM statement is executed once, and control then passes to the next executable statement following the PERFORM statement.

procedure-name-1 THROUGH procedure-name-2

When the PERFORM statement is executed, the program transfers control from the PERFORM statement to the first statement of the procedure named by procedure-name-1. This transfer occurs only once for each execution of a PERFORM statement. The program transfers control from the procedure to the next executable statement following the PERFORM statement, as follows:

- If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, the program returns control after the last statement of procedure-name-1.
- If procedure-name-1 is a section-name and procedure-name-2 is not specified, the program returns control after the last statement of the last paragraph in procedure-name-1.

- If procedure-name-2 is specified and is a paragraph-name, the program returns control after the last statement of the paragraph.
- If procedure-name-2 is specified and is a section-name, then the program returns control after the last statement of the last paragraph in the section.

When procedure-name-1 and procedure-name-2 are both specified and one is the name of a procedure in the DECLARATIVES SECTION of the program, then both must be procedure-names in the same DECLARATIVES SECTION.

No particular sequential relationship needs to exist between procedure-name-1 and procedure-name-2. More than one logical path of program control through the performed range of procedures can exist. A common, though not required, method of documenting the final paragraph of a performed range of procedures is through the use of the EXIT statement.

An implicit return mechanism is established at the end of a performed range of procedures and is activated by the execution of a PERFORM statement. When program control reaches an active return mechanism, control returns to the activating PERFORM statement. A return mechanism permanently deactivates itself by transferring program control back to a PERFORM statement; an active return mechanism is temporarily deactivated by the execution of a PERFORM statement. Program control passes through a nonactive return mechanism to the next executable statement following the PERFORM range. This rule applies to all formats.

A procedure executed under the control of a PERFORM statement can execute PERFORM statements. The range of procedures executed under the control of the nested PERFORM statement need not be declared totally within, or disjoint from, the range of procedures executed by the first PERFORM statement. The permanent deactivation of an active return mechanism causes the last return mechanism temporarily deactivated to become active again, allowing overlapping PERFORM ranges (two or more PERFORM ranges that have a common exit point) to logically execute in the same way as disjoint PERFORM ranges.

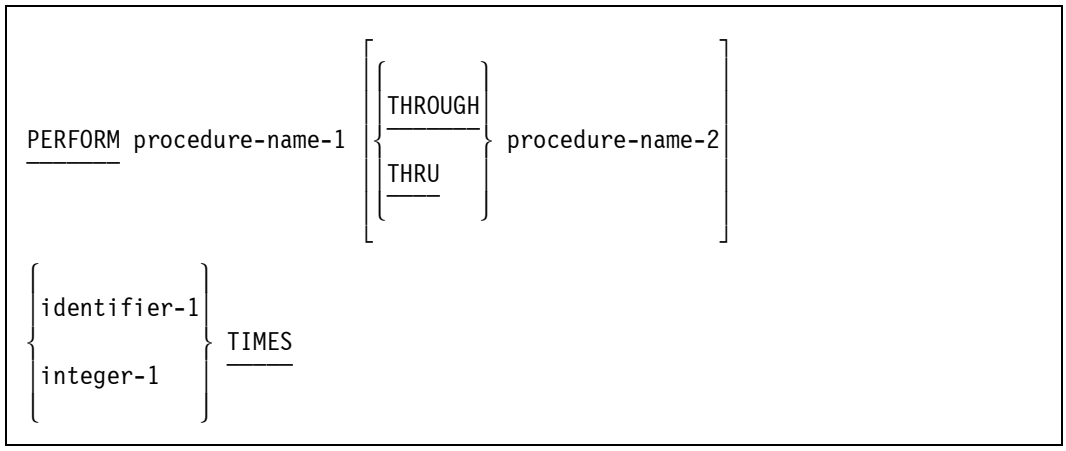
Transfer of program control by means of a GO statement from a range of procedures being executed under control of a PERFORM statement does not cause the return mechanism to be deactivated. Subsequent transfer of program control back into the PERFORM range causes control to return to the PERFORM statement, provided that the return mechanism is still active. Repeated branching out of a PERFORM range without allowing control to reach the ending paragraph can cause the program to end with a STACK OVERFLOW fault.

Note: *A RELEASE statement in a SORT INPUT procedure and a RETURN statement in a SORT OUTPUT procedure will conditionally cause the program to terminate with an INVALID INDEX error if either statement is executed when the PERFORM statement nesting level is greater than 32. This is a permanent nesting-level restriction.*

When procedure-name-1 and procedure-name-2 are both specified and one is the name of a procedure in the DECLARATIVES SECTION of the program, then both must be procedure-names in the same DECLARATIVES SECTION.

The words THRU and THROUGH are equivalent.

Format 2



Explanation of Format 2

Format 2 is the PERFORM...TIMES variation. The procedures are performed the number of times specified by integer-1 or by the initial value of the data item referenced by identifier-1 for that execution. If, at the time of execution of a PERFORM statement, the value of the data item referenced by identifier-1 is equal to 0 or is negative, then control passes to the next executable statement following the PERFORM statement. After the procedures have been executed the specified number of times, control is transferred to the next executable statement following the PERFORM statement.

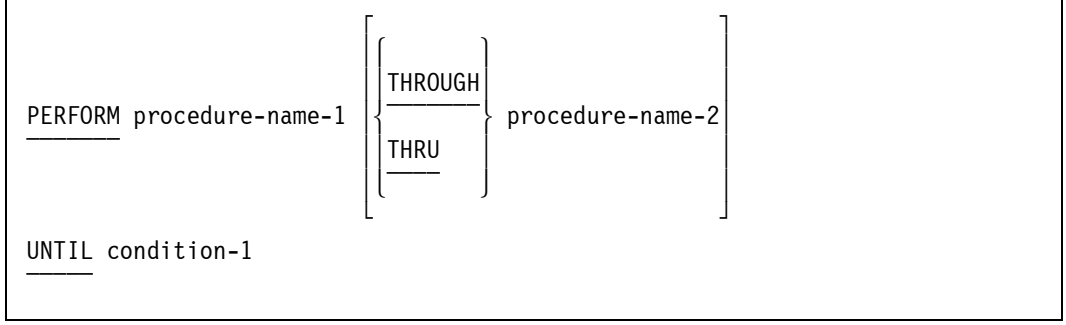
procedure-name-1 THROUGH procedure-name-2

Refer to the discussion of procedure-name-1 THROUGH procedure-name-2 in Format 1 of the PERFORM statement for information.

TIMES

Each identifier represents an elementary numeric item described in the DATA DIVISION. Identifier-1 must be described as a numeric integer. During execution of the PERFORM statement, references to identifier-1 cannot alter the number of times the procedures are to be executed as specified by the initial value of identifier-1.

Format 3



Explanation of Format 3

Format 3 is the PERFORM...UNTIL variation. The specified procedures are performed until the condition specified by the UNTIL phrase is TRUE. When the condition is TRUE, control is transferred to the next executable statement after the PERFORM statement. If the condition is TRUE when the PERFORM statement is entered, no control transfer to procedure-name-1 takes place, and control is passed to the next executable statement following the PERFORM statement.

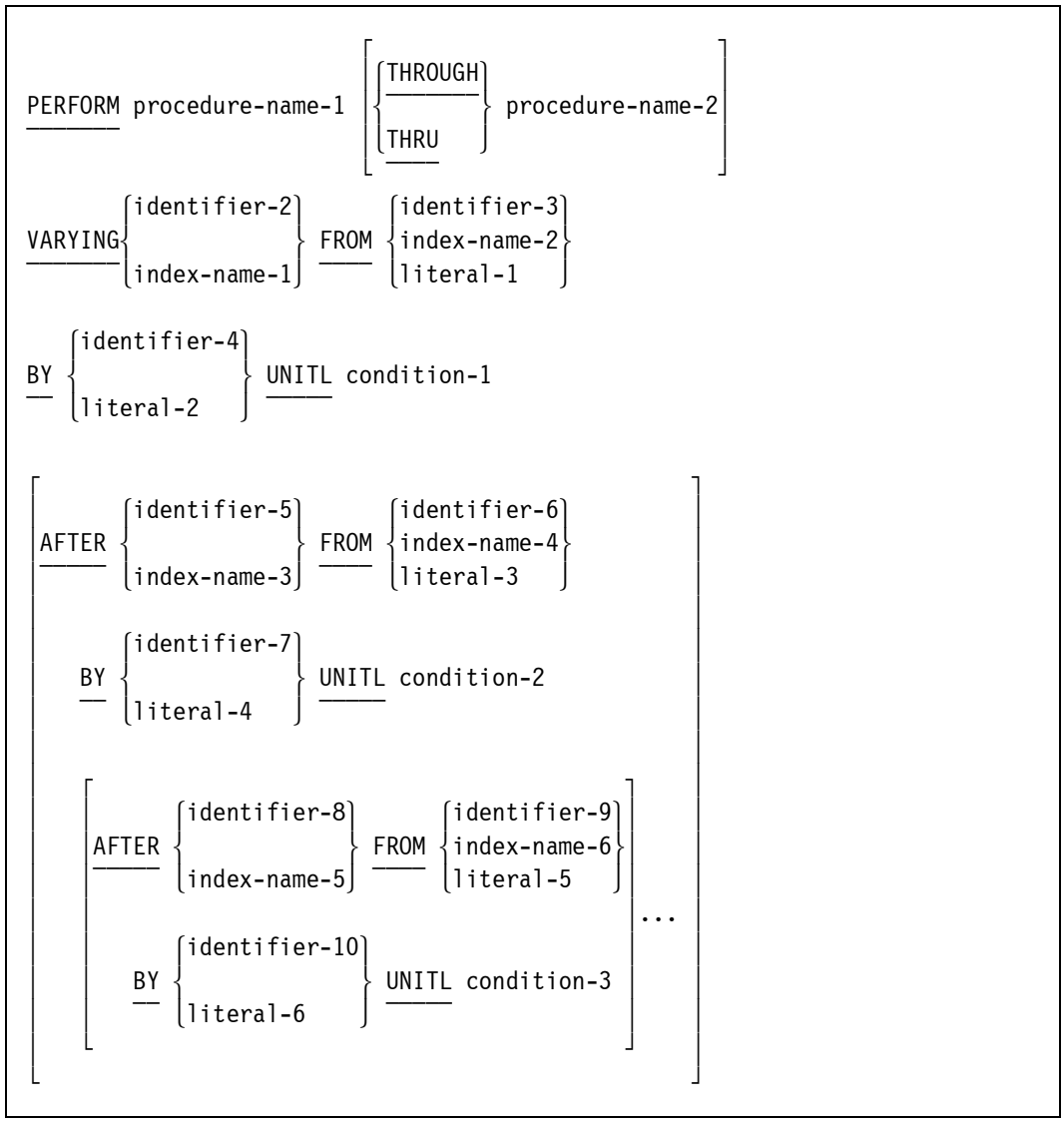
procedure-name-1 THROUGH procedure-name-2

Refer to the discussion of procedure-name-1 THROUGH procedure-name-2 in Format 1.

UNTIL

Condition-1 can be any conditional expression described according to the rules for conditional expressions.

Format 4



Explanation of Format 4 (PERFORM...VARYING)

The values referenced by one or more identifiers or index-names are incremented or decremented in an orderly fashion during execution of a PERFORM statement

In the following discussion, every reference to an identifier as the object of the VARYING, AFTER, and FROM (current value) phrases also refers to index-names.

Condition-1, condition-2, and condition-3 can be any conditional expression described according to the rules for conditional expressions.

Each identifier represents an elementary numeric item described in the DATA DIVISION. Each literal represents a numeric literal.

procedure-name-1 THROUGH procedure-name-2

See procedure-name-1 THROUGH procedure-name-2 in Format 1 for information.

VARYING

This phrase specifies a data item that tracks the iterations of the PERFORM statement. If an index-name is specified in the VARYING phrase, the following conditions apply:

- The identifier in the associated FROM and BY phrases must be an integer data item.
- The literal in the associated FROM phrase must be a positive integer.
- The literal in the associated BY phrase must be a nonzero integer.

If an index-name is specified in the VARYING or the AFTER phrase and an identifier is specified in the associated FROM phrase, then the data item referenced by the identifier must have a positive value.

FROM

This phrase specifies the initial value of the tracking data item in the VARYING phrase. If an index-name is specified in the FROM phrase, the following conditions apply:

- The identifier in the associated VARYING or AFTER phrase must be an integer data item.
- The identifier in the associated BY phrase must be an integer data item.
- The literal in the associated BY phrase must be a nonzero integer.

If an index-name appears in a VARYING phrase, an AFTER phrase, or in both phrases, then the index-name is initialized and subsequently incremented (as described in the following paragraphs) according to the rules of the SET statement.

When an index-name appears in the FROM phrase and an identifier appears in an associated VARYING or AFTER phrase, the identifier is initialized according to the rules of the SET statement. Subsequent incrementation is explained in the flowchart descriptions in the text that follows.

BY

The BY phrase specifies the value by which the counter data item is incremented.

UNTIL

This phrase provides the test condition that, if TRUE, ends the iterations of the PERFORM statement.

AFTER...UNTIL

The AFTER phrase specifies the next condition to be evaluated. If an index-name is specified in the AFTER phrase, the following conditions apply:

- The identifier in the associated FROM and BY phrases must be an integer data item.

- The literal in the associated FROM phrase must be a positive integer.
- The literal in the associated BY phrase must be a nonzero integer.

Flowcharts

Figure 9–1 shows the flowchart for the VARYING phrase with one condition.

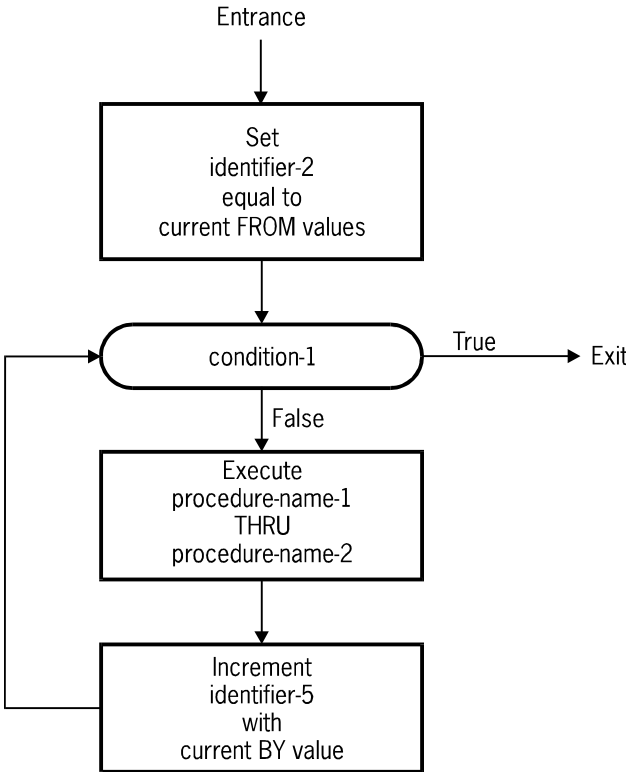


Figure 9–1. VARYING Phrase of a PERFORM Statement with One Condition

When the PERFORM statement has one condition, the following events occurs:

1. The program sets identifier-2 to the value established in the FROM phrase.
2. The program evaluates condition-1.
3. If condition-1 is TRUE when it is evaluated, the program transfers control to the next executable statement after the PERFORM statement.
4. If condition-1 is FALSE, the program executes procedure-name-1 and procedure-name-2.
5. The program increments the value specified in the VARYING phrase by the value designated in the BY phrase.

Figure 9–2 shows the flowchart for the VARYING phrase with two conditions.

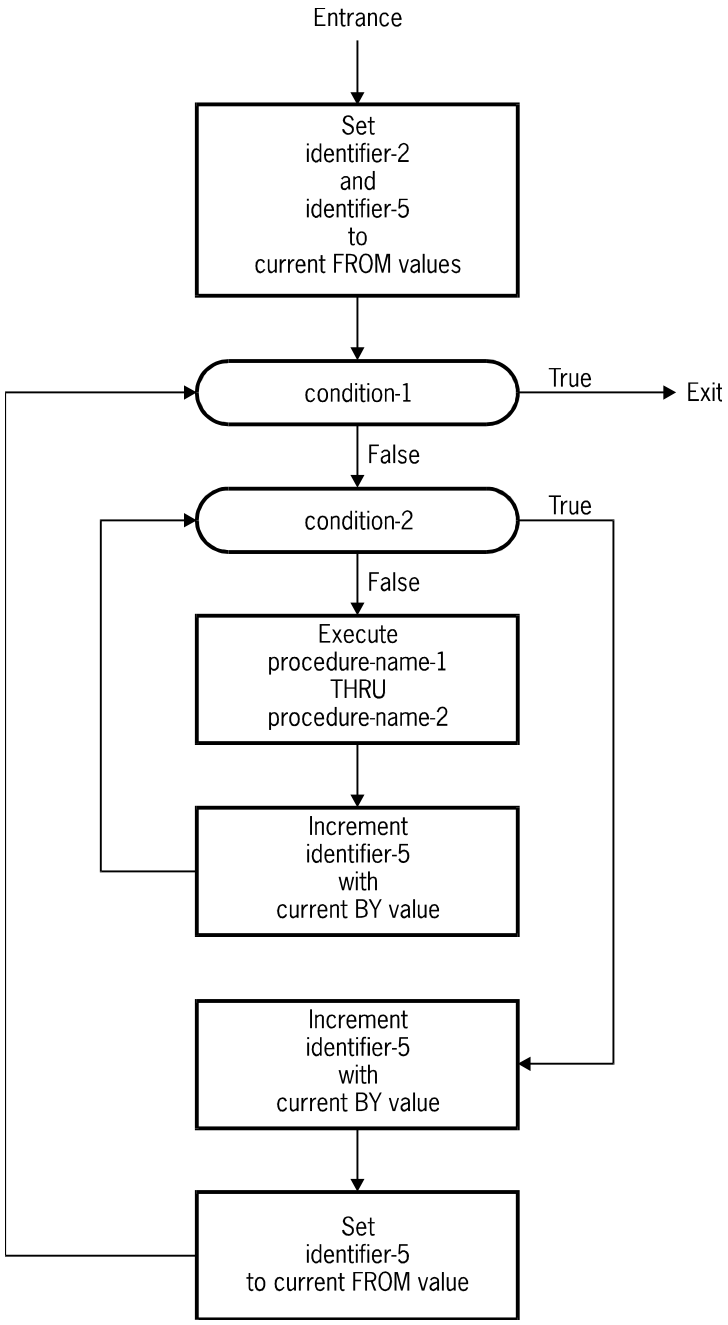


Figure 9–2. VARYING Phrase of a PERFORM Statement with Two Conditions

When the PERFORM statement has two conditions, the following sequence of events occurs:

1. The program sets identifier-2 and any occurrences of identifier-5 to the value of the literal or the current value of the identifier in the associated FROM phrase.
2. The program evaluates condition-1.
3. If any condition other than the last one is FALSE when it is evaluated, the program evaluates the next condition immediately.

   If the last condition is FALSE when it is evaluated, the program executes procedure-name-1 through procedure-name-2 once. Then, the program adds the last BY value to the associated identifier-2 or identifier-5, and evaluates the condition again.

   If condition-1 is TRUE when it is evaluated, the program transfers control to the next executable statement after the PERFORM statement. If condition-1 is TRUE the first time it is evaluated, the program does not evaluate the remaining conditions and does not execute procedure-name-1 through procedure-name-2.

   If condition-2 is TRUE when it is evaluated, the program increments the identifier of the immediately preceding AFTER or VARYING phrase by the associated BY value, sets the associated identifier-5 to the value of literal-3 or to the current value of identifier-6 in the associated FROM phrase, and evaluates the condition specified in that preceding AFTER or VARYING phrase.

For More Information

For information about identifying conditions, refer to “Conditional Expressions” in Section 8, “PROCEDURE DIVISION Concepts.”

PROCESS (Extension to ANSI X3.23-1974 COBOL)

The PROCESS statement initiates the parallel execution of another task.

```
PROCESS task-identifier WITH section-name [USING actual-parameter-list]
```

Explanation of Format

The PROCESS statement creates a dependent process as a separate task. Unlike the separate task started by a RUN statement, the task initiated by a PROCESS statement depends on the initiator. If the initiator ends before the termination of the dependent process, a critical block exit occurs.

A dependent process must be bound to a host program if the dependent process contains data areas described with the GLOBAL or OWN clauses. A dependent process runs in its own stack.

The syntax of the PROCESS statement is the same as the CALL statement format that initiates a task. The difference is that the CALL statement initiates serial (or synchronous) processing of another program, while the PROCESS statement initiates parallel (or asynchronous) processing.

The actual-parameter-list must consist of a series of data items, task items, and expressions, optionally separated by commas. Arithmetic expressions can be passed as variables, and certain kinds of variables can be passed (received) by reference. As a general rule, the kind of actual parameter must not conflict with the corresponding formal parameter.

For More Information

For an additional explanation of the syntax, refer to “CALL” in this section.

READ

The READ statement reads one record from a file on an input device and stores it in a record storage area associated with the file-name. Before your program executes a READ statement against a file, it must execute an OPEN statement against the file specifying a mode of INPUT or I-O so that the file is open when the program executes the READ statement. The results of using means other than the OPEN verb (for example, file attributes) to cause the file to be open are unpredictable. If the read operation is successful, the data is available in the record storage area.

For sequential access, the READ statement makes the next logical record from a file available. For random access, the READ statement makes a specified record available from a mass-storage file.

The execution of the READ statement causes the value of any FILE STATUS data item associated with the file-name to be updated.

When the logical records of a file are described with more than one record description, these records automatically share the same record storage area. This sharing is equivalent to an implicit redefinition of the area. The contents of any data items that are beyond the range of the current data record are undefined after the READ statement is executed.

Format	Explanation
1	Reads the next record of any file in sequential access mode, regardless of organization.
2	Reads a record in random or dynamic access mode when the record is retrieved randomly.
3	Reads relative or indexed files in dynamic access mode.
4	Reads indexed files in random access mode, or reads files in dynamic access mode when records are retrieved randomly.
5	Reads a formlibrary. Refer to Volume 2 for more information.

Format 1: Sequential Access

The READ statement make the next record available as follows:

- If the open operation positions the current-record pointer, the record indicated by the current-record pointer is made available.
- If a read operation positions the current-record pointer, the current-record pointer is updated to point to the next record in the file and that record is made available.

If the position of the current-record pointer for that file is undefined for a read operation, the read operation is unsuccessful.

If the end of a reel or a unit is recognized during execution of a READ statement and the logical end of the file has not been reached, the following operations are executed:

- For an indexed file being sequentially accessed, records with the same duplicate value in an alternate record key (the key of reference) are made available in the same order they would be in if they were executed with WRITE or REWRITE statements that create such duplicate values.
- If the ACTUAL KEY or RELATIVE KEY phrase is specified for a sequential or a relative organization file with sequential access mode, the execution of a READ statement updates the contents of the ACTUAL KEY or RELATIVE KEY data item to the ordinal number of the logical record accessed. However, if an At End condition is reached, the contents of the ACTUAL KEY phrase are not changed.

The format for sequential access is as follows:

```
READ file-name RECORD [WITH NO WAIT] [INTO identifier]

[;AT END imperative-statement]
```

Explanation of Format 1

Format 1 is used for all files in sequential access mode, regardless of organization.

WITH NO WAIT (Extension to ANSI X3.23-1974 COBOL)

The WITH NO WAIT phrase can be specified only for port files.

A READ statement causes the program to wait until a logical record is available. For port files, you can prevent the possibility of this suspension by specifying the WITH NO WAIT phrase. A status key value of 94 indicates that no logical record was available for the read operation.

If you declare an ACTUAL KEY clause for a port file, you are responsible for updating the ACTUAL KEY value with an appropriate subfile index. If the ACTUAL KEY value is not 0 (zero), a read operation from the specified subfile is performed. If the ACTUAL KEY value is 0 (zero), a nonselective read operation is performed and the ACTUAL KEY value is updated to indicate the subfile index of the subfile that was read.

For a nonselective read operation, the first logical record to arrive at a subfile in the port file is returned as the data for the READ statement. The subfile to be read is determined by the operating system, and no specific selection algorithm is guaranteed. However, no subfile is read continuously at the expense of the other subfiles.

If you do not declare an ACTUAL KEY clause for the port file, the file must contain a single subfile and that subfile is read.

INTO

If the INTO phrase is specified, the record being read is moved from the record area to the area specified by the identifier. The move takes place according to the rules specified for the MOVE statement. The sending area is considered to be a group item equal in size to the maximum record size for this file. The implied MOVE statement does not occur if the read operation was unsuccessful. Any subscripting or indexing associated with the identifier is evaluated after the record is read and immediately before the record is moved to the data item.

When the INTO phrase is used, the record being read is available in both the input record area and the data area associated with the identifier. The record storage area associated with the identifier and the record storage area associated with file-name must not be the same record storage area.

AT END

When the At End condition occurs, the read operation is unsuccessful.

The imperative-statement can be the NEXT SENTENCE phrase.

A READ statement with an AT END phrase transfers control to the imperative-statement in the AT END phrase if the READ statement is executed after an At End condition has been recognized but before a successful CLOSE statement and a successful OPEN statement have been executed for the file. (The ability to read a record after the At End condition has been reached and the repeated execution of the imperative-statement in the AT END phrase of such READ statements are extensions to ANSI X3.23-1974 COBOL).

If an optional file is not present when the file is opened, the read operation is unsuccessful. The At End condition occurs at execution of the first READ statement for the file. The standard end-of-file procedures are not performed.

For more information on using the AT END phrase, refer to “Handling I/O Exception Conditions” in Section 8, “PROCEDURE DIVISION Concepts.”

Format 2: Random Access of Relative or Indexed Files

For random access of relative files, the execution of a Format 2 READ statement accesses the record specified by the contents of the ACTUAL KEY data item or RELATIVE KEY data item. If, on execution of a Format 2 READ statement, the contents of the ACTUAL KEY data item or RELATIVE KEY data item are less than one or greater than the ordinal number of the last record written to the file, the READ statement is unsuccessful and the Invalid Key condition results.

For random access of indexed files, the prime record key is established as the key of reference. The prime record key for the file is the data item specified in the RECORD KEY clause of the file-control-entry for a file.

The format for random access of relative files or indexed files is as follows:

```
READ file-name RECORD [WITH NO WAIT] [INTO identifier]

[;INVALID KEY imperative-statement]
```

Explanation of Format 2

Format 2 is used for files in random access or dynamic access mode when records are to be retrieved randomly.

WITH NO WAIT (Extension to ANSI X3.23-1974 COBOL)

The WITH NO WAIT phrase can be specified only for port files.

A READ statement causes the program to wait until a logical record is available. For port files, you can prevent the possibility of this suspension by specifying the WITH NO WAIT phrase. A status key value of 94 indicates that no logical record was available for the read operation.

If you declare an ACTUAL KEY clause for a port file, you are responsible for updating the ACTUAL KEY value with an appropriate subfile index. If the ACTUAL KEY value is not 0 (zero), a read operation from the specified subfile is performed. If the ACTUAL KEY value is 0 (zero), a nonselective read operation is performed and the ACTUAL KEY value is updated to indicate the subfile index of the subfile that was read.

For a nonselective read operation, the first logical record to arrive at a subfile in the port file is returned as the data for the READ statement. The subfile to be read is determined by the operating system, and no specific selection algorithm is guaranteed. However, no subfile is read continuously at the expense of the other subfiles.

If you do not declare an ACTUAL KEY clause for the port file, the file must contain a single subfile and that subfile is read.

INTO

If the INTO phrase is specified, the record being read is moved from the record area to the area specified by the identifier. The move takes place according to the rules specified for the MOVE statement. The sending area is considered to be a group item equal in size to the maximum record size for this file. The implied MOVE statement does not occur if the read operation was unsuccessful. Any subscripting or indexing associated with the identifier is evaluated after the record is read and immediately before the record is moved to the data item.

When the INTO phrase is used, the record being read is available in both the input record area and the data area associated with the identifier. The record storage area associated with the identifier and the record storage area associated with file-name must not be the same storage area.

INVALID KEY

When the Invalid Key condition occurs, the read operation is unsuccessful.

The imperative-statement can be the NEXT SENTENCE phrase.

For more information on using the INVALID KEY phrase, refer to "Handling I/O Exception Conditions" in Section 8, "PROCEDURE DIVISION Concepts."

Format 3: Dynamic Access of Relative or Indexed I/O Files

The format for dynamic access of relative or indexed files is as follows:

```
READ file-name [NEXT] RECORD [INTO identifier]

[;AT END imperative-statement]
```

Explanation of Format 3

Format 3 is used for relative or indexed I/O files in dynamic access mode.

NEXT

For indexed files, a READ NEXT statement causes the next logical record to be retrieved from the file.

For relative files, a READ NEXT statement updates the contents of the RELATIVE KEY data item so that it contains the ordinal record number of the record made available.

To differentiate more clearly between a Format 3 READ statement that lacks a NEXT phrase and a Format 4 READ statement that lacks a KEY phrase, Unisys recommends that the optional NEXT phrase always be specified for Format 3 READ statements against dynamic files. A Format 3 READ statement always accesses the file sequentially from its current record position; specifying NEXT [RECORD] ensures that the READ is clearly identifiable as sequential.

INTO

If the INTO phrase is specified, the record being read is moved from the record area to the area specified by the identifier. The move takes place according to the rules specified for the MOVE statement. The sending area is considered to be a group item equal in size to the maximum record size for this file. The implied MOVE statement does not occur if the read operation was unsuccessful. Any subscripting or indexing associated with the identifier is evaluated after the record is read and immediately before the record is moved to the data item.

When the INTO phrase is used, the record being read is available in both the input record area and the data area associated with the identifier. The record storage area associated with the identifier and the record storage area associated with file-name must not be the same storage area.

AT END

When the At End condition occurs, the read operation is unsuccessful.

The imperative-statement can be the NEXT SENTENCE phrase.

A READ statement with an AT END phrase transfers control to the imperative-statement in the AT END phrase if the READ statement is executed after an At End condition has been recognized but before a successful CLOSE statement and a successful OPEN statement have been executed for the file. (The ability to read a record after the At End condition has been reached and the repeated execution of the imperative-statement in the AT END phrase of such READ statements are extensions to ANSI X3.23-1974 COBOL).

If an optional file is not present when the file is opened, the read operation is unsuccessful. The At End condition occurs at execution of the first READ statement for the file. The standard end-of-file procedures are not performed.

For more information on using the AT END phrase, refer to "Handling I/O Exception Conditions" in Section 8, "PROCEDURE DIVISION Concepts."

Format 4: Random Access of Indexed Files

The format for random access of indexed files is as follows:

```
READ file-name RECORD [INTO identifier] [;KEY IS data-name]

[;INVALID KEY imperative-statement]
```

Explanation of Format 4

Format 4 is used for indexed I/O files in random access mode or for files in dynamic access mode when records are to be retrieved randomly.

INTO

If the INTO phrase is specified, the record being read is moved from the record area to the area specified by the identifier. The move takes place according to the rules specified for the MOVE statement. The sending area is considered to be a group item equal in size to the maximum record size for this file. The implied MOVE statement does not occur if the read operation was unsuccessful. Any subscripting or indexing associated with the identifier is evaluated after the record is read and immediately before the record is moved to the data item.

When the INTO phrase is used, the record being read is available in both the input record area and the data area associated with the identifier. The record storage area associated with the identifier and the record storage area associated with file-name must not be the same storage area.

KEY

Data-name is established as the key of reference for this retrieval. If the dynamic access mode is specified, this key of reference is also used by any subsequent executions of Format 4 READ statements for the file until a different key of reference is established for the file.

The data-name must be the name of a data item specified as a record key associated with the file-name. The data-name can be qualified.

Execution of the 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 until the first record with a matching value is found. The current-record pointer indicates this record, which is then made available. If no record can be so identified, the Invalid Key condition exists and execution of the READ statement is unsuccessful.

To differentiate more clearly between a Format 4 READ statement that lacks a KEY phrase and a Format 3 READ statement that lacks a NEXT phrase, Unisys recommends that the optional KEY phrase always be specified for Format 4 READ statements against dynamic files. A Format 4 READ statement can always be expected to change the key of reference and to access the file randomly according to that key; specifying KEY [IS] ensures that the READ is clearly identifiable as random.

INVALID KEY

When the Invalid Key condition occurs, the read operation is unsuccessful. The imperative-statement can be the NEXT SENTENCE phrase.

For More Information

- For information about naming a file, identifying its medium, specifying its organization, and so on, refer to “FILE-CONTROL Paragraph” in Section 5, “ENVIRONMENT DIVISION.”
- For information about identifying the status of I/O operations, refer to “I/O Status” in Section 5, “ENVIRONMENT DIVISION.”

- For information about opening a file, refer to “OPEN” in this section.
- For more information on using the INVALID KEY phrase, refer to “Handling I/O Exception Conditions” in Section 8, “PROCEDURE DIVISION Concepts.”

RELEASE

The RELEASE statement transfers records to the initial phase of a sort operation.

```
RELEASE record-name [FROM identifier]
```

Explanation of Format

The RELEASE statement releases the record named by the record-name to the initial phase of a sort operation.

A RELEASE statement can be used only in an INPUT PROCEDURE associated with a SORT statement for a file whose sort-merge file-description entry contains the record-name.

The record-name must be the name of a logical record in the associated sort-merge file-description entry and can be qualified. The record-name and the identifier must not refer to the same storage area.

If the FROM phrase is used, the contents of the identifier data area are moved to record-name. The contents of the record-name are released to the sort file. Moving takes place according to the rules specified for the MOVE statement.

In an extension to ANSI X3.23-1974 COBOL, the execution of a RELEASE statement does not affect the contents or the accessibility of the record area. If the sort-merge file is named in a SAME RECORD AREA clause, the logical record is also available as a record of other files referenced in the same SAME RECORD AREA clause. When control passes from the INPUT PROCEDURE, the file consists of all records that were placed in it by the execution of RELEASE statements.

Note: *If a RELEASE statement is executed when there are more than 32 PERFORM statements active, the RELEASE statement unconditionally causes the program to terminate with an INVALID INDEX error. This is a permanent restriction.*

For information about the SORT statement, refer to “SORT” in this section.

RESET (Extension to ANSI X3.23-1974 COBOL)

The RESET statement is used to control communication between processes in an asynchronous processing environment.

```
RESET event-identifier-1 [, event-identifier-2]...
```

Explanation of Format

The RESET statement causes the event specified by event-identifier-1, event-identifier-2, and so on to be turned off. If the event is then tested, the condition returns the value FALSE.

Event-identifiers must be one of the following:

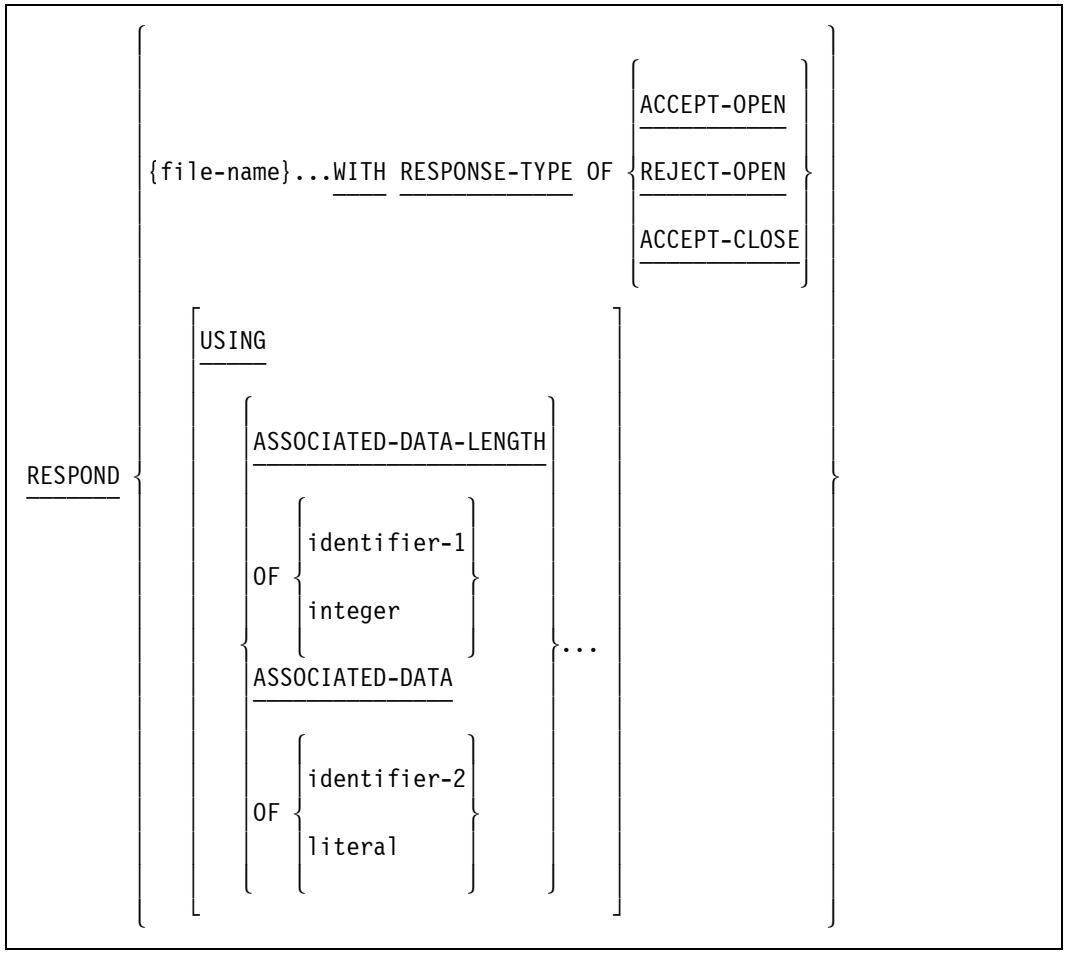
- Properly qualified and subscripted data-names with event usage specified
- File or task attributes of type EVENT

For information about Inter-Program Communication, refer to "CAUSE (Extension to ANSI X3.23-1974 COBOL)" in this section.

RESPOND (Extension to ANSI X3.23-1974 COBOL)

The RESPOND statement is used only with port files. This statement enables a program to accept or reject a request for a dialogue to be established or terminated.

The program is prompted for a response through the CHANGEEVENT file attribute. The FILESTATE file attribute indicates the type of response for which the program is being prompted, for example, OPENRESPONSEPLEASE or CLOSERESPONSEPLEASE.



Explanation of Format

The file-name identifies one or more port files.

The WITH RESPONSE-TYPE phrase specifies the type of response. The response type must be consistent with the current state of the subfile.

The ACCEPT-OPEN phrase accepts the request for a dialogue to be started.

The REJECT-OPEN phrase rejects the request for a dialogue to be started.

The ACCEPT-CLOSE phrase accepts the request for a dialogue to be ended.

The USING option permits each of the USING clauses to be included one time only.

The ASSOCIATED-DATA-LENGTH phrase specifies the number of characters to be sent. If you do not specify the length of the associated data and the associated data is a data item, your program uses the actual length of the data. If you do specify the length of the associated data, the length value must be less than or equal to the actual length of the data. An error results if the length specified is not a single-precision integer value. To use the ASSOCIATED-DATA-LENGTH phrase, you must specify the ASSOCIATED-DATA phrase with either an identifier or an undigit literal, but not a nonnumeric literal.

The ASSOCIATED-DATA phrase transfers data to the corresponding subfile along with the request to close the dialogue.

The integer must be a numeric literal.

The identifier must name an elementary integer data item.

The identifier can be a group data item or an alphanumeric elementary data item.

The literal can be a nonnumeric or undigit literal.

General Rules

You must use the RESERVE NETWORK clause in the SPECIAL-NAMES paragraph for the compiler to recognize the words RESPOND and RESPONSE-TYPE as keywords.

The ACTUAL KEY clause specifies the subfile that is responding. If you do not code an ACTUAL KEY clause, the port file must contain a single subfile. If you are using multiple subfiles, do the following:

1. Specify the total number of subfiles in your program by using the *CHANGE ATTRIBUTE MAXSUBFILES TO VALUE attribute-value* statement.
2. Specify the subfile that is to respond by using the *SELECT port-file ASSIGN TO PORT; ACTUAL KEY IS subfile-num* 93 clause.
3. Declare subfile-num and attribute-value in the WORKING-STORAGE SECTION.

Table 9–14 shows the effects of the ACTUAL KEY clause value on the RESPOND verb.

Table 9–14. Designating Subfiles to Respond

Actual Key Value	Explanation
0 (zero) or none	Returns a BADSUBFILEINDEX run-time error in the SUBFILEERROR file attribute. You are not allowed to respond to all subfiles at once.
Nonzero	Responds with the specified subfile.
Greater than the MAXSUBFILES value or a negative number	Returns a BADSUBFILEINDEX run-time error in the SUBFILERROR file attribute.

The system returns a value that indicates the result of a RESPOND statement. You can access this value by including a *SELECT file-name FILE STATUS IS data-name* clause in your program. The operating system moves a value into the data-name storage area after the program performs the RESPOND statement. You can then use an IF statement to test the value of the data-name and take the desired action depending on the result. If you choose not to code actions for the result of a RESPOND statement, the system provides a default action for each result.

Table 9–15 shows the I/O status values and their meanings.

Table 9–15. Values for RESPOND Statement Completion

Value	Explanation
00	Control was returned to the program after the RESPOND statement completed correctly. In the case of a port file, the close operation might be pending.
84	An error was detected while the file was being closed.

Note: *The I/O status value "84" is an extension to ANSI X3.23-1974 COBOL.*

Examples

In the following examples, it is assumed that earlier in the program the port files were declared using an ACTUAL KEY clause to specify a subfile index and that the subfile index was set to a particular index.

Example 9–23 accepts a request for an orderly close operation from the correspondent endpoint for the subfile of port file PORTFILE1.

```
RESPOND PORTFILE1 WITH RESPONSE-TYPE OF ACCEPT-CLOSE.
```

Example 9–23. Coding a RESPOND Statement for an Orderly Close Operation

Example 9–24 accepts a request for a dialogue to be started on the subfile of port file PORTFILE1.

```
RESPOND PORTFILE1 WITH RESPONSE-TYPE OF ACCEPT-OPEN.
```

Example 9–24. Coding a RESPOND Statement That Requests a Dialogue

Example 9–25 rejects an open dialogue request for the subfile on port file PORTFILE1. The associated data "MYDATA" is sent with the response.

```
RESPOND PORTFILE2 WITH RESPONSE-TYPE OF REJECT-OPEN
        USING ASSOCIATED-DATA OF "MYDATA".
```

Example 9–25. Coding a RESPOND Statement That Rejects an Open Request

Example 9–26 accepts an open dialogue request for the subfile on port file PORTFILE1. Twelve characters of associated data are sent with the response, beginning with the location pointed to by the ALPHANUM-ITEM item.

```
RESPOND PORTFILE1 WITH RESPONSE-TYPE OF ACCEPT-OPEN
        USING ASSOCIATED-DATA-LENGTH 12
              ASSOCIATED-DATA OF ALPHANUM-ITEM.
```

Example 9–26. Coding a RESPOND Statement That Uses Associated Data

Example 9–27 accepts open dialogue requests for the subfiles on the port files PORTFILE1 and PORTFILE2.

```
RESPOND PORTFILE1 PORTFILE2 WITH RESPONSE-TYPE OF ACCEPT-OPEN
         USING ASSOCIATED-DATA OF "MYDATA"
         USING ASSOCIATED-DATA-LENGTH NUMERIC-ITEM.
```

Example 9–27. Coding a RESPOND Statement with Multiple Files

For More Information

- For information about the RESERVE clause in the SPECIAL-NAMES paragraph, refer to Section 5, “ENVIRONMENT DIVISION.”
- For step-by-step information on coding port file applications, refer to the *I/O Subsystem Programming Guide.*
- For references to file attributes, refer to the *File Attributes Reference Manual.*

RETURN

The RETURN statement causes the next record (in the order specified by the KEY clause in the SORT or the MERGE statement) to be transferred to the record area.

The RETURN statement obtains either sorted records from the final phase of a sort operation or merged records during a merge operation.

```
RETURN file-name RECORD [INTO identifier]

; AT END imperative-statement
```

Explanation of Format

file-name

The file-name must be described by a sort-merge file-description entry in the DATA DIVISION.

A RETURN statement can be used only in the range of an output procedure associated with a SORT or a MERGE statement for the file-name.

INTO

If the INTO phrase is specified, the current record is moved from the input area to the area specified by the identifier. The move takes place according to the rules for the MOVE statement without the CORRESPONDING phrase. The sending area is considered to be a group item with a fixed size equal to the maximum record size. The implied move does not occur if an At End condition exists. Any subscripting or indexing associated with the identifier is evaluated after the record is returned and immediately before it is moved to the data item. When the INTO phrase is used, the data is available in both the input record area and the data area associated with the identifier.

The INTO phrase must not be used when the input file contains logical records of various sizes as indicated by the record descriptions. The storage area associated with the identifier and the record area associated with the file-name must not be the same storage area.

AT END

If no next logical record exists for the file when a RETURN statement is executed, the At End condition occurs.

When the At End condition occurs, no transfer of data to the record area takes place and the contents of the record area are undisturbed. After the At End condition occurs, the contents of the record area are still accessible. (This is an extension to ANSI X3.23-1974 COBOL.) After execution of the imperative-statement in the AT END phrase, no RETURN statement can be executed as part of the current output procedure.

The imperative-statement can be the NEXT SENTENCE phrase.

General Rules

When the logical records of a file are described with more than one record description, these records automatically share the same storage area. This sharing is equivalent to an implicit redefinition of the area. The contents of any data items beyond the range of the current data record are undefined when the execution of the RETURN statement is complete.

If records smaller than the maximum record size are released, the contents of the record area beyond the end of the released record are unpredictable when subsequently returned. In that case, you must account for the size of the logical record returned. (This is an extension to ANSI X3.23-1974 COBOL.)

Note: *If a RETURN statement is executed when there are more than 32 PERFORM statements active, the RETURN statement unconditionally causes the program to terminate with an INVALID INDEX error. This is a permanent restriction.*

REWRITE

The REWRITE statement replaces a logical record that exists in a mass-storage file. This statement causes the value of the FILE STATUS data item associated with the file, if any, to be updated.

```
REWRITE record-name [SYNCHRONIZED] [FROM identifier]

[;INVALID KEY imperative-statement]
```

Explanation of Format

record-name and SYNCHRONIZED

The record-name is the name of a logical record in the FILE SECTION of the DATA DIVISION and can be qualified. The file associated with the record-name must be open in the I-O mode when this statement is executed. The number of character positions in the record referenced by the record-name must equal the number of character positions in the record being replaced.

Synchronization of all output records can be designated with the SYNCHRONIZE file attribute. Synchronization means that output must be written to the physical file before the program initiating the output can resume execution, thereby ensuring synchronization between logical and physical files. The SYNCHRONIZED clause enables you to override the synchronization specified by the file attribute for a specific output record. Synchronization is available for use by tape files and disk files with sequential organization only.

FROM

The FROM option makes the REWRITE statement operate like a MOVE statement followed by a WRITE statement. The move takes place according to the rules of the MOVE statement without the CORRESPONDING option.

The record-name and the identifier must not refer to the same storage area.

INVALID KEY

Refer to “Indexed I/O” later in this section for information about the Invalid Key condition.

The imperative-statement can be the NEXT SENTENCE phrase.

The execution of the REWRITE statement does not affect the contents or accessibility of the record area. (This is an extension to ANSI X3.23-1974 COBOL.)

The execution of a REWRITE statement with the FROM phrase is equivalent to the execution of the statement *MOVE identifier TO record-name* followed by the execution of the same REWRITE statement without the FROM phrase.

Sequential I/O

For files in the sequential access mode, the last I/O statement executed for the associated file before the execution of the REWRITE statement must have been a successfully executed READ statement. the I/O subsystem logically replaces the record that was accessed by the READ statement.

For sequential files in the random or the dynamic access mode, the record to be rewritten is specified by the value of the actual key or the relative key. No prior reading of this record is required.

Relative I/O

For relative files in random or dynamic access mode, the record to be rewritten is specified by the value of the actual key or the relative key. No prior reading of this record is required.

Indexed I/O

The following rules apply to indexed I/O only:

- For a file in the sequential access mode, the record to be replaced is specified by the value contained in the prime record key. When the REWRITE statement is executed, the value contained in the prime-record-key data item of the record to be replaced must equal the value of the prime record key of the last record read from this file.
- For a file in the random or the dynamic access mode, the record to be replaced is specified by the prime-record-key data item.
- The contents of alternate-record-key data items of the record being rewritten can differ from those in the record being replaced. The system uses the contents of the record key data items during execution so that subsequent access of the record can be made based on any of those specified record keys.
- The Invalid Key condition exists under the following circumstances:
  - The access mode is sequential, and the value contained in the prime-record-key item of the record to be replaced does not equal the value of the prime record key of the last record read from this file.
  - The value contained in the prime-record-key data item does not equal that of any record stored in the file.
  - The value contained in an alternate-record-key data item for which a DUPLICATES clause is omitted equals that of a record already stored in the file.

For More Information

- For details on the status of I/O operations, see "I/O Status" in Section 5, "ENVIRONMENT DIVISION."
- For information about opening a file in I-O mode, refer to "OPEN" in this section.
- For information on the INVALID KEY phrase, see "Handling I/O Exception Conditions" in Section 8, "PROCEDURE DIVISION Concepts."

RUN (Extension to ANSI X3.23-1974 COBOL)

The RUN statement enables a program to initiate another program as an independent, asynchronous task. Once a program is initiated by executing a RUN statement, it executes independently of the initiating program.

```
RUN task-identifier WITH section-name

[USING arithmetic-expression-1 [, arithmetic-expression-2]...]
```

Explanation of Format

The section-name must be the name of a USE procedure declared as EXTERNAL. Parameters must be passed by value, that is, contain a RECEIVED BY CONTENT clause. Only arithmetic values can be passed or received because only formal parameters with arithmetic properties can be described in the syntax.

The formal parameters to which the values of the arithmetic expressions are passed must be described as single-precision or double-precision 77-level items and must have a RECEIVED BY CONTENT clause. The compiler makes adjustments, if necessary, to truncate double-precision values to single-precision values or extend single-precision values to double-precision values to ensure that the value passed has the same precision as the corresponding formal parameter. All values are passed with a scale of 0, regardless of the scale of the corresponding formal parameter, and can be passed as normalized values.

The RUN statement creates an independent process, which does not share the resources of the initiator and can continue running after the termination of the initiator.

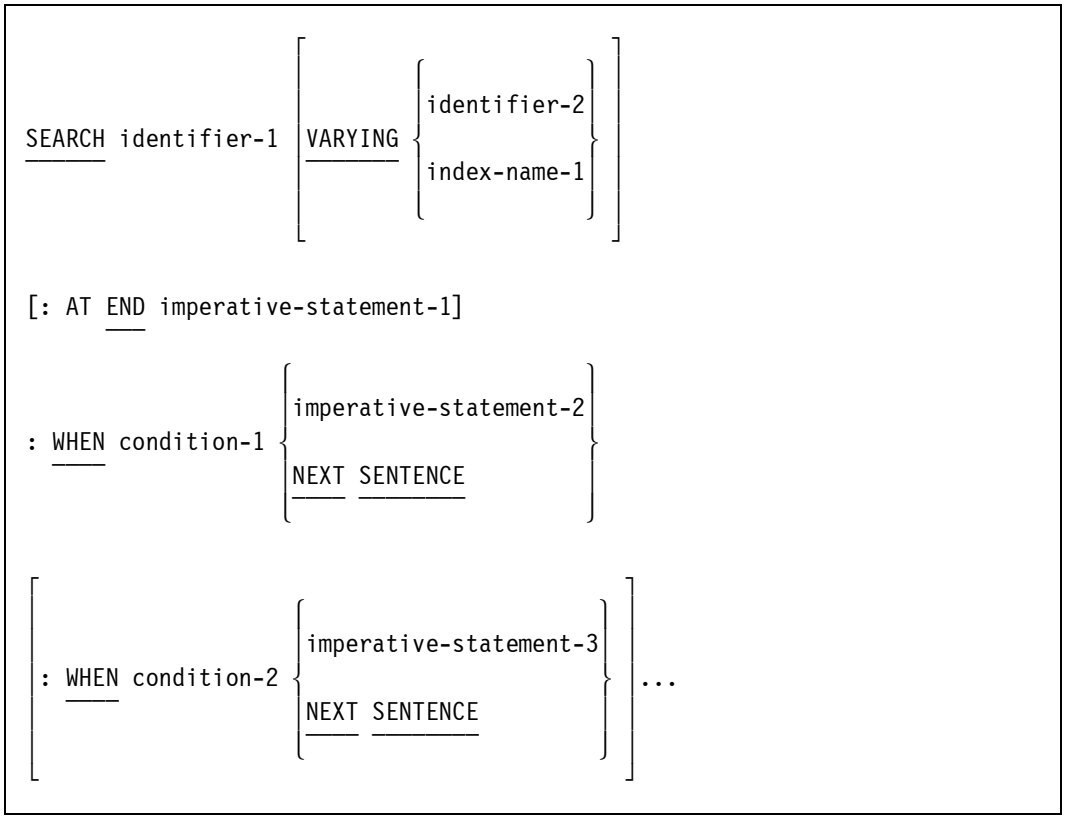
An independent process must not be bound to a host program, nor can any of its data be declared with the GLOBAL or OWN phrases. Independent procedures (all the procedures of an independent process) must be compiled at the 02-level.

SEARCH

The SEARCH statement searches a table for a table element that satisfies the specified condition, and adjusts the associated index-name to indicate that table element. A SEARCH statement must be performed for each level of a table to be searched.

Format	Explanation
1	Performs a serial search on an unordered table. An unordered table is not arranged in a particular order.
2	Performs a binary search on an ordered table. An ordered table is arranged in ascending or descending order.

Format 1



Explanation of Format 1

Format 1 of the SEARCH statement serially searches a table for a table element that satisfies the WHEN conditions. The search begins at the current index setting.

identifier-1

Identifier-1 must not be subscripted or indexed, but its description must contain an OCCURS clause and an INDEXED BY clause.

VARYING

The VARYING option increments an index.

Identifier-2, when specified, must be described as USAGE IS INDEX or as a numeric elementary item without positions to the right of the assumed decimal point.

If the *VARYING index-name-1* phrase is specified and if index-name-1 appears in the INDEXED BY phrase of identifier-1, that index-name-1 is used for the search. If this is not the case, or if the *VARYING identifier-2* phrase is specified, the first (or only) index-name given in the INDEXED BY phrase of identifier-1 is used for the search. In addition, the following operations occur:

- If the *VARYING index-name-1* phrase is used, and if index-name-1 appears in the INDEXED BY phrase of another table entry, the occurrence number represented by index-name-1 is incremented by the same amount (and at the same time) as the occurrence number represented by the index-name associated with identifier-1.
- If the *VARYING identifier-2* phrase is specified and if identifier-2 is an index data item, then the data item referenced by identifier-2 is incremented by the same amount (and at the same time) as the index associated with identifier-1. If identifier-2 is not an index data item, the data item referenced by identifier-2 is incremented by the value 1 at the same time the index referenced by the index-name associated with identifier-1 is incremented.

AT END

An AT END phrase can be used if the search ends and none of the conditions are satisfied.

WHEN

The WHEN phrase determines the ending condition for the search operation. You can specify a number of conditions for ending the search by using multiple WHEN phrases. If the SEARCH statement is executed conditionally, fewer WHEN phrases may be permitted depending on the level to which the SEARCH statement is nested.

Condition-1, condition-2, and so on can be any conditional expression.

Figure 9–3 shows a flowchart of the Format 1 SEARCH statement containing two WHEN phrases. The following list describes the flow of control shown in Figure 9–3.

- If, at the start of the execution of the SEARCH statement, the index-name associated with identifier-1 contains a value that corresponds to an occurrence number greater than the highest permissible occurrence number for identifier-1, the search ends immediately. If the AT END phrase is specified, the imperative-statement-1 is executed. If the AT END phrase is not specified, control passes to the next executable sentence.
- If, at the start of the execution of the SEARCH statement, the index-name associated with identifier-1 contains a value that corresponds to an occurrence number not greater than the highest permissible occurrence number for identifier-1, then the SEARCH statement operates by evaluating the conditions in their written order. The SEARCH statement makes use of index settings, wherever specified, to determine the occurrence of the items to be tested.

- If none of the conditions is satisfied, the index-name for identifier-1 is incremented to refer to the next occurrence. The process is then repeated using the new index-name settings. If the new value of the index-name settings for identifier-1 corresponds to a table element outside the permissible range of occurrence values, the search ends as indicated in the first bulleted item of this list.
- If one of the conditions is satisfied on evaluation, the search ends immediately and the imperative-statement associated with that condition is executed. The index-name remains set at the occurrence that caused the condition to be satisfied.

For More Information

The number of occurrences of identifier-1, the last of which is the highest permissible value, is discussed in Section 7, "DATA DIVISION."

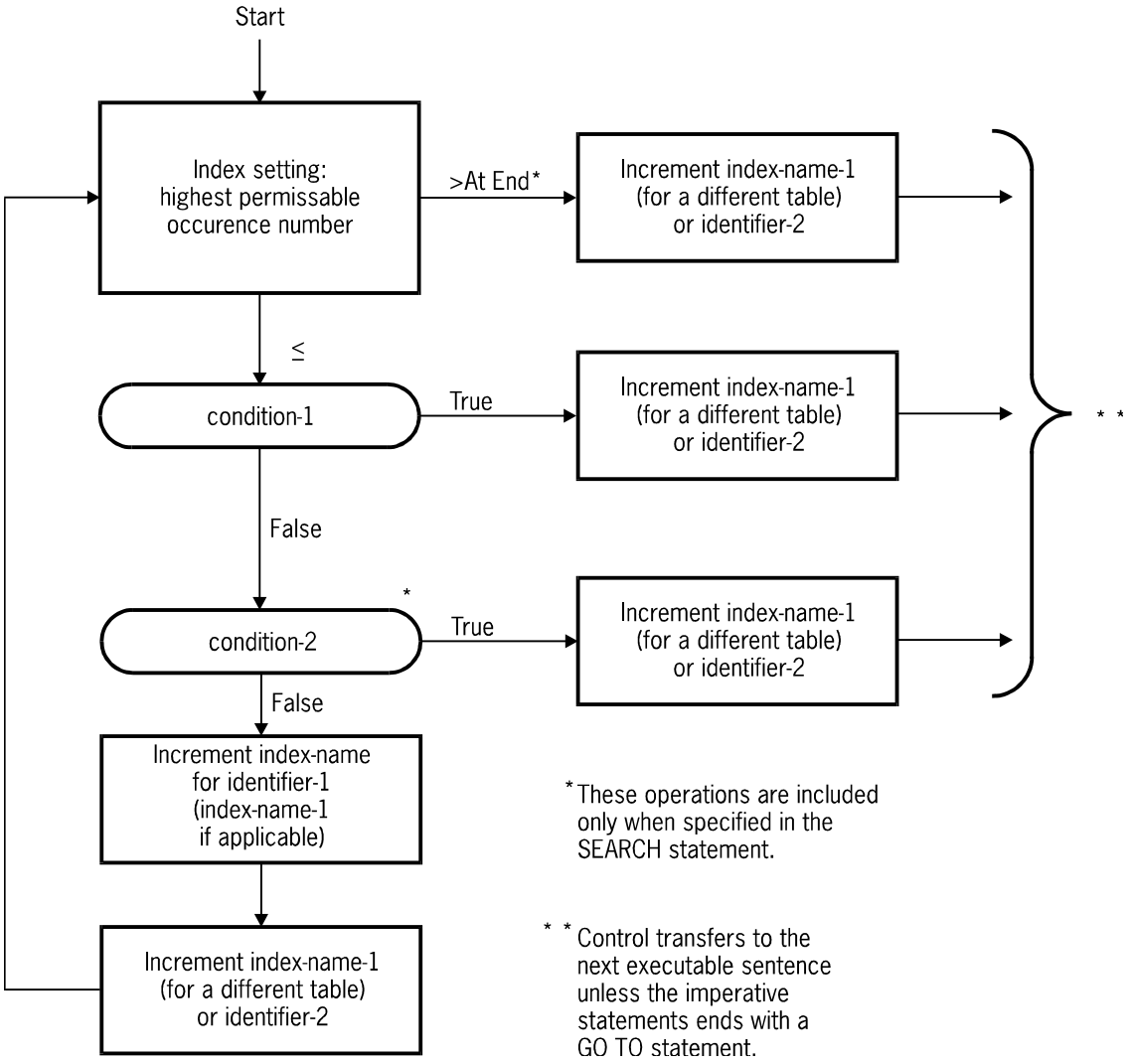
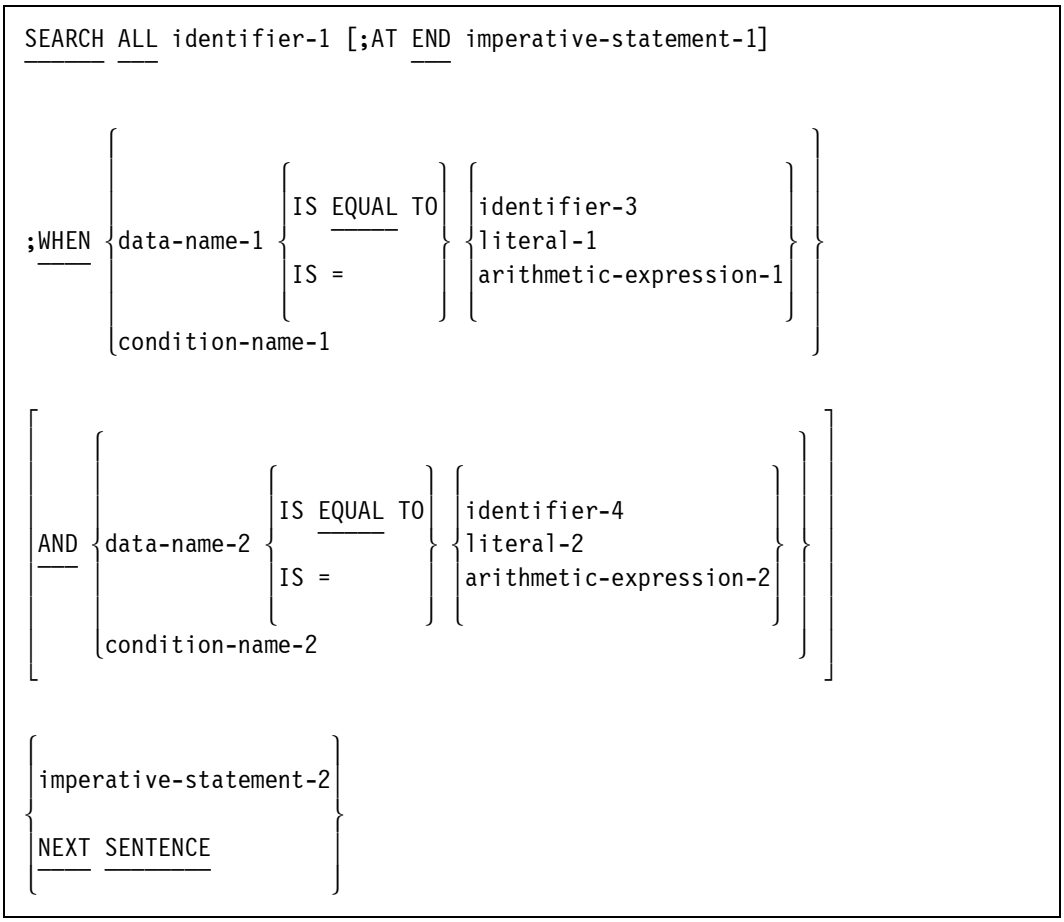


Figure 9–3. SEARCH Statement Containing Two WHEN Phrases

Format 2



Note: *The required relational character = is not underlined to avoid confusion with other symbols.*

Explanation of Format 2

Format 2 of the SEARCH statement searches an ordered, one-dimensional array. The data description of the table referred to in the SEARCH ALL statement must contain a Format 2 OCCURS clause with the INDEXED BY and KEY IS phrases.

Identifier-1 must not be subscripted or indexed.

The index-name used for the search operation is the first (or only) index-name that appears in the INDEXED BY phrase of identifier-1. Any other index-names for identifier-1 remain unchanged.

All referenced condition-names must be defined as having only a single value. The data-name associated with a condition-name must appear in the KEY clause of identifier-1. Data-name-1 and data-name-2 can be qualified. In addition, data-name-1 and data-name-2 must be indexed by the first index-name associated with identifier-1 along with other indexes or literals (as required) and must be referenced in the KEY clause of identifier-1.

Identifier-3, identifier-4, or identifiers specified in arithmetic-expression-1 or arithmetic-expression-2 must not be referenced in the KEY clause of identifier-1, nor must they be indexed by the first index-name associated with identifier-1.

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 results of the SEARCH ALL statement are predictable only in the following cases:

- The data in the table is ordered in the same manner as described in the ASCENDING/DESCENDING KEY clause associated with the description of identifier-1.
- The contents of the keys referenced in the WHEN clause are sufficient to identify a unique table element.

If the table is not ordered as specified, unpredictable results can be expected.

When duplicates occur, the index indicates the occurrence of the duplicate closest to the beginning of the table.

If any of the conditions specified in the WHEN clause cannot be satisfied for any setting of the index in the permitted range, control is passed to imperative-statement-1 of the AT END phrase, when specified, or to the next executable sentence when this phrase is not specified. In either case, the final setting of the index is the value corresponding to an occurrence number that is one greater than the last element of the table.

General Rules for Format 1 and Format 2

If all the conditions can be satisfied, the index indicates an occurrence that allows the conditions to be satisfied and control passes to imperative-statement-2.

After execution of imperative-statement-1, imperative-statement-2, or imperative-statement-3 that does not end with a GO TO statement, control passes to the next executable sentence. In Format 1, if the VARYING phrase is not used, the index-name used for the search operation is the first (or only) index-name that appears in the INDEXED BY phrase of identifier-1. Any other index-names for identifier-1 remain unchanged.

If identifier-1 is a data item subordinate to a data item containing an OCCURS clause (providing for a multidimensional table), an index-name must be associated with each dimension of the table through the INDEXED BY phrase of the OCCURS clause. Only the setting of the index-name associated with identifier-1 (and the data item identifier-2 or index-name-1, if present) is modified by the execution of the SEARCH statement. To search an entire multidimensional table, a SEARCH statement must be executed several times. Before each execution of a SEARCH statement, SET statements need to be executed whenever index-names need to be adjusted to appropriate settings.

For More Information

- For information about conditional expressions, refer to “Conditional Expressions” in Section 8, “PROCEDURE DIVISION Concepts.”
- For information about defining repeated data items, refer to “OCCURS Clause” in Section 7, “DATA DIVISION.”

SEEK (Extension to ANSI X3.23-1974 COBOL)

The SEEK statement repositions a mass-storage file for subsequent sequential access.

```
SEEK file-name RECORD
```

Explanation of Format

For files specified with sequential access, the SEEK statement repositions the file so that a succeeding READ or WRITE statement access the record sought.

The value of the ACTUAL KEY clause specified for the file-name is used as the ordinal number of the record sought. If the actual key contains a negative number or 0 (zero), the first record of the file is sought. The ACTUAL KEY clause must be specified in the FILE-CONTROL entry for the file-name.

The file-name must be the name of a mass-storage file of sequential organization.

The execution of a SEEK statement does not cause the contents of the STATUS KEY data item to be updated.

Before your program executes a SEEK statement against a file, it must execute an OPEN statement against the file so that the file is open when the program executes the SEEK statement. The results of using means other than the OPEN verb (for example, file attributes) to cause the files to be open are unpredictable.

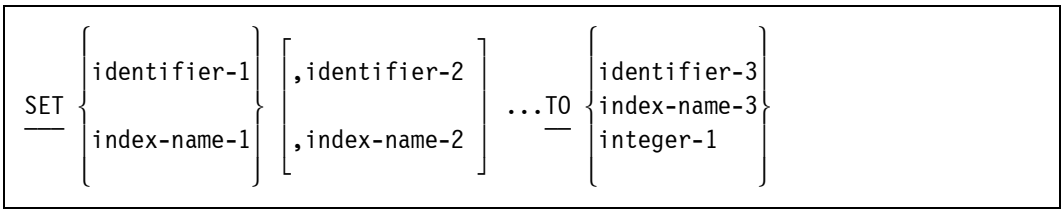
SET

The SET statement establishes reference points for table handling operations by setting index-names associated with table elements. The SET statement can also be used to modify a file or a task attribute.

When a sending item and a receiving item in the same SET statement share a part, but not all, of their storage area, the result of the statement execution is undefined.

Format	Explanation
1	Establishes an index value for table look-up operations
2	Increments or decrements an index value
3	Establishes the value of a table attribute
4	Establishes the value of a file attribute

Format 1



Explanation of Format 1

All references to index-name-1 and identifier-1 also apply to index-name-2 and identifier-2, respectively.

If index-name-1 is specified, the value of the index after the execution of the SET statement should correspond to an occurrence number of an element in the associated table. If index-name-3 is specified, the value of the index before execution of the SET statement should correspond to an occurrence number of an element in the associated table.

Index-name-1 can be set to any value, with the following restrictions:

- If overflow occurs, the value in the index-name is left-truncated according to the arithmetic rules for a Size Error condition without a SIZE ERROR phrase.
- When a statement is executed using the index-name to refer to a table element, the value contained in the index or produced by relative indexing must fall in the range specified by the OCCURS clause that defines the table. Otherwise, an abnormal termination of the program occurs.

Identifier-1 and identifier-3 must name either index data items or elementary items described as integers outside of the DATA-BASE section. For use of SET with elementary items in a data base, refer to Volume 2.

Integer-1 can be signed and can be 0 (zero).

Table 9–16 shows the validity of various operand combinations of the SET statement.

Table 9–16. Validity of Operands for the SET Statement

Sending Item	Receiving Item Integer Data Item	Receiving Item Index-Name	Receiving Item Index Data Item
Integer literal	Not valid	Valid	Not valid
Integer data item	Not valid	Valid	Not valid
Index-name	Valid	Valid	Valid
Index data item	Not valid	Valid	Valid

In Format 1, the following actions occur:

- Index-name-1 is set to a value that causes it to refer to the table element whose occurrence number corresponds to the table element referenced by index-name-3, identifier-3, or integer-1. If identifier-3 is an index data item or if index-name-3 is related to the same table as index-name-1, no conversion takes place.
- If identifier-1 is an index data item, it can be set equal to either the contents of index-name-3 or to the contents of identifier-3, where identifier-3 is also an index data item. No conversion takes place in either case.
- If identifier-1 is not an index data item, it can be set only to an occurrence number that corresponds to the value of index-name-3. Neither identifier-3 nor integer-1 can be used in this case.
- These actions can be repeated for index-name-2, identifier-2, and so on, if these variables are specified. The value of index-name-3 or identifier-3 is used each time as it was at the beginning of the statement execution. Any subscripting or indexing associated with identifier-1 and so on is evaluated immediately before the value of the respective data item is changed.

Format 2

```
SET index-name-4 [, index-name-5]...{          } {            }
                                    {UP BY   } {identifier-4}
                                    {DOWN BY } {integer-2   }
                                    {        } {            }
```

Explanation of Format 2

All references to index-name-4 also apply to index-name-5.

Index-name-4 can be set to any value, with the following restrictions:

- If overflow occurs, the value in the index-name is left-truncated according to the arithmetic rules for a Size Error condition without a SIZE ERROR phrase.
- At execution of a statement using the index-name to refer to a table element, the value contained in the index or produced by relative indexing should fall in the range specified by the OCCURS clause that defines the table. A program can be abnormally terminated if the index address an item beyond the limits of the 01-level record in which the table resides. Data can be retrieved from or modified within areas outside of the table (but still in the 01-level record in which the table resides) with unexpected results.
- When integer-1 in a *SET index-name-1 TO integer-1* statement exceeds the limits of the OCCURS clause to which index-name-1 refers, object code is generated for the SET statement that unconditionally terminates the program abnormally.

  In all other cases of the SET statement, the value to which index-name is set is not checked at execution time against the OCCURS clause for the table to which the index-name refers.

  You must ensure that the value to which index-name is set is appropriate, or you must ensure that the table associated with the index-name is correct. The use of inappropriate index values to access parts of the 01-level record outside the bounds of the table to which the index refers is strongly discouraged.

The contents of index-name-4 are incremented using the UP BY phrase or decremented using the DOWN BY phrase by a value corresponding to the number of occurrences represented by the value of integer-2 or identifier-4. Thereafter, the incremental process is repeated for index-name-5, and so on. For each repetition, the value of identifier-4 is used as it was at the beginning of the statement execution.

If index-name-4 is specified, the value of the index both before and after the execution of the SET statement should correspond to an occurrence number of an element in the associated table.

Index-names are considered to be related to a given table and are defined by specification in the INDEXED BY clause.

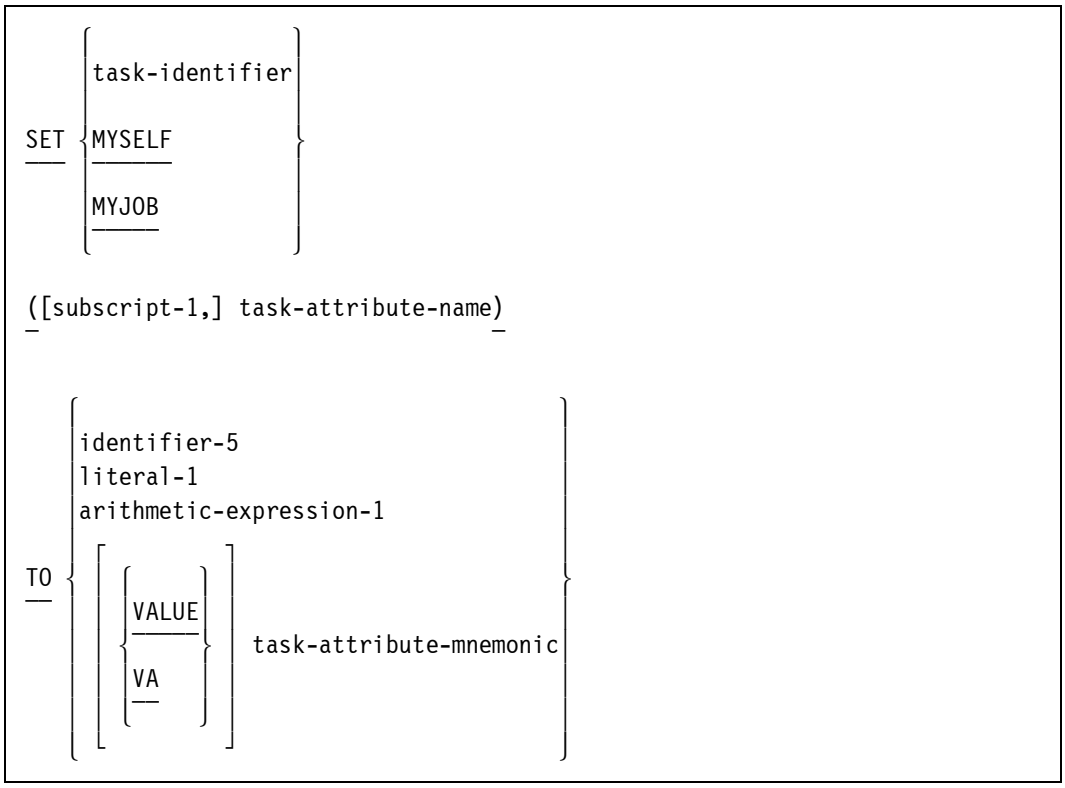
Identifier-4 must be described as an elementary numeric integer.

Integer-2 can be signed.

For More Information

- For information on handling Size Error conditions, refer to "SIZE ERROR Phrase" in Section 8, "PROCEDURE DIVISION Concepts."
- For information on handling tables, refer to "Tables" in Section 6, "Data Concepts."
- For information about the validity of various operand combinations, refer to Table 9–16, "Validity of Operands for the SET Statement."

Format 3



Explanation of Format 3

Note: *Format 3 is considered an obsolete element of COBOL74. The CHANGE statement is the preferred syntax.*

The task-attribute-name must be a system-name that defines a task attribute. Subscript-1 specifies the member of the task array that is affected, and can be an arithmetic expression. Subscript-1 is required when the task-identifier requires a subscript.

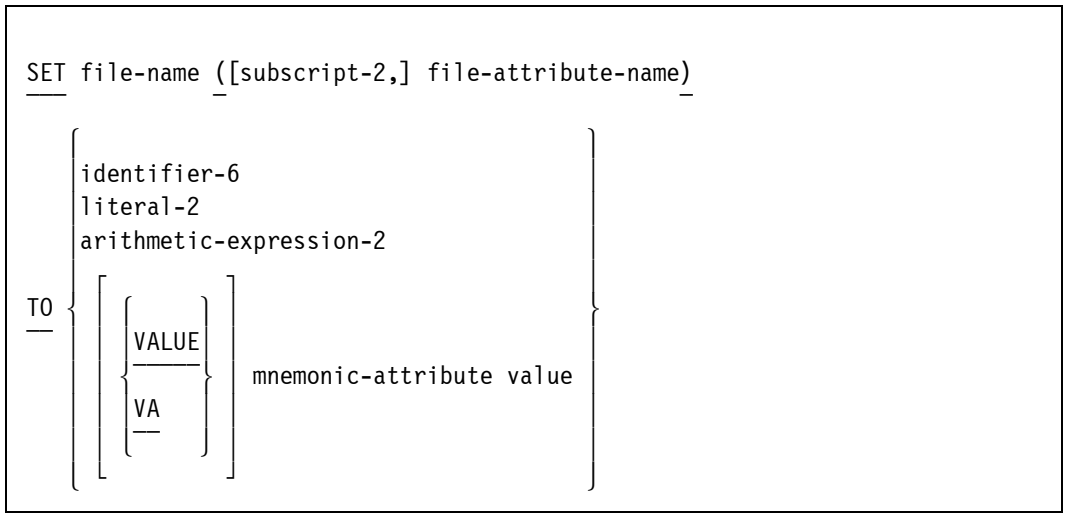
Identifier-5 or literal-1 or the task-attribute-mnemonic must be consistent with the type of the task-attribute-name.

A task-attribute-mnemonic is a name associated with a constant value for an attribute that has a set number of predetermined possible values.

The choice of identifier, literal, arithmetic-expression, or task-attribute-mnemonic depends on the attribute being set and its declared value. (This is an extension to ANSI X3.23-1974 COBOL).

If a data-name has the same name as a task-attribute-mnemonic, the value assigned to the attribute is determined by the use of the optional word VALUE. If the word VALUE is present, the attribute is set to the value of the mnemonic. If the word VALUE is omitted, the attribute is set to the current value of the data-name.

Format 4



Explanation of Format 4

Note: *Format 4 is considered an obsolete element of COBOL74. The CHANGE statement is the preferred syntax.*

The file-name qualifies the file-attribute-name enclosed the required parentheses.

Subscript-2 can be used only with a port file. The subscript can be an arithmetic-expression, and the value of the expression specifies the subfile of the file that is affected.

The choice of literal-2, arithmetic-expression-2, identifier-6, or the mnemonic-attribute-value depends on the type of the attribute and the static or dynamic nature of the value to which the attribute is to be set.

Mnemonic-attribute-values can be used as data-names or procedure-names in the program if they are not COBOL reserved words. (This is an extension to ANSI X3.23-1974 COBOL.)

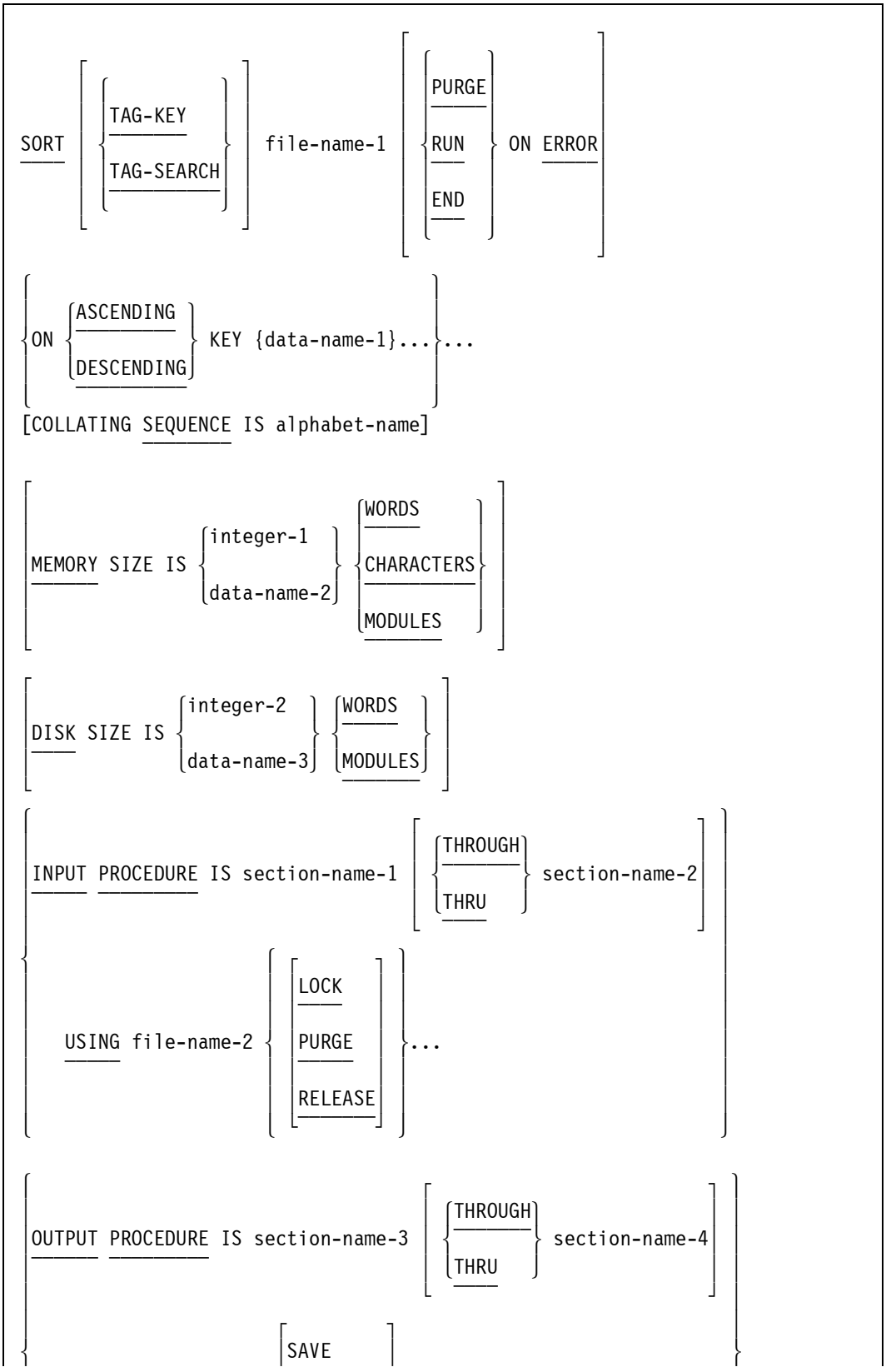
Boolean or integer task attributes accept a numeric item or literal or the value associated with a mnemonic. If the value is not in the permissible range for the attribute specified, an error occurs at compilation time or at execution time. Task or file attributes of type POINTER accept alphanumeric items.

If a data-name has the same name as a mnemonic-attribute-value, the value assigned to the attribute is determined by the use of the optional word VALUE. If the word VALUE is present, the attribute is set to the value of the mnemonic. If the word VALUE is omitted, the attribute is set to the current value of the data-name.

SORT

The SORT statement creates a sort file by executing input procedures or by transferring records from another file. Records are sorted by the sort file using a set of specified keys. In the final phase of the sort operation, each record is made available in sorted order to output procedures or to an output file. The SORT statement invokes the SORT utility of the operating system to perform the actual sort.

SORT statements can appear anywhere except in the DECLARATIVES SECTION of the PROCEDURE DIVISION or in an INPUT PROCEDURE or an OUTPUT PROCEDURE clause associated with a SORT or a MERGE statement.


SORT
TAG-KEY
TAG-SEARCH
file-name-1
PURGE
RUN
END
ON ERROR
ON
ASCENDING
DESCENDING
KEY {data-name-1}...
COLLATING SEQUENCE IS alphabet-name
MEMORY SIZE IS
integer-1
data-name-2
WORDS
CHARACTERS
MODULES
DISK SIZE IS
integer-2
data-name-3
WORDS
MODULES
INPUT PROCEDURE IS section-name-1
THROUGH
THRU
section-name-2
USING file-name-2
LOCK
PURGE
RELEASE
OUTPUT PROCEDURE IS section-name-3
THROUGH
THRU
section-name-4
SAVE

Explanation of Format

TAG-KEY (Extension to ANSI X3.23-1974 COBOL)

When the TAG-KEY option is used, sorting is performed on keys rather than on the entire record. The record numbers are placed in the sorted order in the GIVING file, which is restricted to records with a size of 8 DISPLAY digits. The TAG-KEY option prohibits the use of input and output procedures.

TAG-SEARCH (Extension to ANSI X3.23-1974 COBOL)

When the TAG-SEARCH option is used, sorting is performed on keys rather than on the entire record. The records are then ordered in the GIVING file according to the sorted order of the record numbers. The TAG-SEARCH option prohibits the use of input and output procedures.

Note: *The file-name specified in the GIVING clause of a SORT TAG-SEARCH statement must not be the same as that specified in the USING clause.*

The TAG-SEARCH option is not supported for tape input files or for multiple-file input.

file-name-1

File-name-1 must be described in a sort-merge file-description entry in the DATA DIVISION.

ON ERROR (Extension to ANSI X3.23-1974 COBOL)

The ON ERROR option enables you to limit irrecoverable parity errors when input and output procedures are not present in a program.

The PURGE phrase causes all records in a block that contains irrecoverable parity errors to be dropped; processing is continued after an ODT message is printed that gives the relative position in the file of the bad block.

The RUN phrase causes the bad block to be used by the program and provides the same message as defined for the PURGE phrase.

The END phrase causes program termination and is the default if no other phrase is specified.

ASCENDING

When the ASCENDING phrase is specified, the sorted sequence is from the lowest value of the contents of the data items identified by the KEY data-names to the highest value, according to the rules for comparison of operands in a relation condition.

DESCENDING

When the DESCENDING phrase is specified, the sorted sequence is from the highest value of the contents of the data items identified by the KEY data-names to the lowest value, according to the rules for comparison of operands in a relation condition.

data-name-1, data-name-2, and data-name 3

Data-name-1, data-name-2, and data-name-3 are KEY data-names and are subject to the following rules:

- The data items identified by KEY data-names must be described in records associated with file-name-1.
- KEY data-names can be qualified.
- The data items identified by KEY data-names must not be variable-length items.
- If file-name-1 has more than one record description, then all the data items identified by the KEY data-names can be described in one of the record descriptions or in any combination of record descriptions. The KEY data-names in each record description need not be described again.
- None of the data items identified by KEY data-names can be described by an entry that either contains an OCCURS clause or is subordinate to an entry that contains an OCCURS clause.
- The data-names are listed from left to right in order of decreasing significance without regard to their division into KEY phrases. In the format, data-name-1 is the major key, data-name-2 is the next most significant key, and so on.

COLLATING SEQUENCE

The collating sequence that applies to the comparison of the nonnumeric key data items specified is determined in the following order of precedence:

1. The collating sequence established by the COLLATING SEQUENCE phrase, if specified, in the SORT statement
2. The collating sequence established as the program-collating sequence

MEMORY SIZE (Extension to ANSI X3.23-1974 COBOL)

The MEMORY SIZE option is a guideline for allocating the SORT memory area and takes precedence over the same clause in the OBJECT-COMPUTER paragraph. This option can be allocated as MODULES, WORDS, or CHARACTERS. If the MEMORY SIZE option is not specified, either in the OBJECT-COMPUTER paragraph or in the SORT statement, a default value of 12,000 words is assumed. If the number of records to be sorted varies from run to run, the memory size can be allocated by specifying data-name-5.

DISK SIZE (Extension to ANSI X3.23-1974 COBOL)

The DISK SIZE option is a guideline for allocating the SORT disk area and takes precedence over the same clause in the OBJECT-COMPUTER paragraph. This option can be allocated as WORDS or MODULES. If the DISK SIZE option is not specified, either in the OBJECT-COMPUTER paragraph or in the SORT statement, a default value of 900,000 words is assumed. One module of disk is equivalent to 1.8 million words of disk. If the number of records to be sorted varies from run to run, the disk size can be allocated by specifying data-name-6.

INPUT PROCEDURE

The input procedure must consist of one or more procedures that appear contiguously in a source program and do not form a part of any output procedure. To transfer records to the file referenced by file-name-1, the INPUT PROCEDURE clause must include at least one RELEASE statement. Control must not be passed to the input procedure except when a related SORT statement is being executed. The input procedure can include any sections needed to select, create, or modify records. The restrictions on the procedural statements in the INPUT PROCEDURE clause are as follows:

- The INPUT PROCEDURE clause must not contain any SORT or MERGE statements.
- The remainder of the PROCEDURE DIVISION must not contain any transfers of control to points inside the input procedure; ALTER, GO TO, and PERFORM statements in the remainder of the PROCEDURE DIVISION must not refer to section-names in the INPUT PROCEDURE clause. Also, while the input procedure can perform a program portion outside of itself, control must always return to the input procedure.

If an input procedure is specified, control is passed to the input procedure before file-name-1 is sequenced by the SORT statement. The compiler inserts a one-time-only return mechanism at the end of the last section in the input procedure; an EXIT statement cannot be used to return control to the SORT statement. Control then passes to the last statement in the INPUT PROCEDURE clause and the records that have been released to file-name-1 are sorted.

In the input procedure, excessive use of GO TO statements to transfer control out of blocks being executed by PERFORM statements can cause the program to fault with an Invalid Index condition at the RETURN statement.

The words THRU and THROUGH are equivalent.

Section-name-1 represents the name of an input procedure.

USING

If the USING phrase is specified, all records in file-name-2 are transferred automatically to file-name-1. At execution of the SORT statement, file-name-2 must not be open. The SORT statement automatically does the following:

- Starts the processing of file-name-2
- Makes the logical records for file-name-2 available

- Ends the processing of file-name-2

These implicit functions are performed so that any associated USE procedures are executed. Also, the SORT statement automatically and implicitly moves the records from the file area of file-name-2 to the file area of file-name-1, and releases records to the initial phase of the sort operation.

If no close action is specified on the file, then the association between the file and its description in the program is maintained like any other file. If both USING and GIVING clauses are present and they specify the same file name, but the USING clause has no close disposition, then the sorted records are written into the original input file. This occurs even when the user requests (through the GIVING close disposition) for the sorted results to be discarded. As a result, you must provide a disposition, such as RELEASE, on the input file.

file-name-2 and file-name-3

File-name-2 and file-name-3 must be described in a file-description entry (not a sort-merge file-description entry) in the DATA DIVISION. The actual size of the logical records described for file-name-2 and file-name-3 must equal the actual size of the logical records described for file-name-1.

LOCK, PURGE, and RELEASE (Extension to ANSI X3.23-1974 COBOL)

The LOCK, PURGE, and RELEASE options can be used to specify the type of file close operation for the USING files, such as file-name-2 and file-name-3. If no close action is specified on the file, the association between the file itself and its description in the program is maintained like any other file. You should, therefore, provide a disposition, such as RELEASE, on the input files. The TAG-SEARCH option prohibits the use of the close options LOCK and PURGE on the USING file.

The behavior of the PURGE and RELEASE options is similar to the corresponding options of the CLOSE statement. The behavior of the LOCK option is similar to the LOCK option of the CLOSE statement, except that under **some** circumstances subsequent reopening of the file during execution of the program is **not prevented**.

When the LOCK option fails to prevent the subsequent reopening of the file, Unisys suggests the use of the USING phrase without any option, followed by an OPEN ... INPUT and a CLOSE ... LOCK either in an OUTPUT PROCEDURE specified in the SORT statement or immediately after the SORT statement.

The LOCK option prevents the subsequent reopening of the file. If the file needs to be reopened (for example, when one tape file is input to the SORT and the same FD is to be used for a different tape file later in the program), Unisys suggests the use of the USING phrase without any options. The USING phrase must be followed by OPEN ... INPUT and CLOSE ... WITH SAVE FOR REMOVAL statements either in an OUTPUT PROCEDURE specified in the SORT statement or immediately after the SORT statement.

OUTPUT PROCEDURE

The output procedure must consist of one or more procedures that appear contiguously in a source program and do not form part of any input procedure. To make sorted records available for processing, the OUTPUT PROCEDURE clause must include at least one RETURN statement. Control must not be passed to the output procedure, except when a related SORT statement is being executed. The output procedure can consist of any procedures needed to select, modify, or copy the records that are being returned, one at a time and in sorted order, from the sort file. The restrictions on the procedural statements in the OUTPUT PROCEDURE clause are as follows:

- The OUTPUT PROCEDURE clause must not contain any SORT or MERGE statement.
- The remainder of the PROCEDURE DIVISION must not contain any transfers of control to points inside the output procedure; ALTER, GO TO, and PERFORM statements in the remainder of the PROCEDURE DIVISION cannot refer to section-names in the OUTPUT PROCEDURE clause. Also, while an output procedure can perform a program portion outside of itself, control must always return to the output procedure.

If an output procedure is specified, control passes to the output procedure after file-name-1 has been sequenced by the SORT statement. The compiler inserts a return mechanism at the end of the last section in the output procedure. When control passes the last statement in the OUTPUT PROCEDURE clause, the return mechanism provides for termination of the sort operation and then passes control to the next executable statement after the SORT statement. Before entering the output procedure, the sort procedure reaches a point at which it can select the next record in sorted order, when requested. The RETURN statements in the OUTPUT PROCEDURE clause are the requests for the next record.

In the output procedure, excessive use of GO TO statements to transfer control out of blocks being executed by PERFORM statements can cause the program to fault with an Invalid Index condition at the RETURN statement.

Section-name-3 represents the name of an output procedure.

GIVING

If the GIVING phrase is specified, all sorted records in file-name-1 are automatically written on file-name-3 as the implied output procedure for this SORT statement. At execution of the SORT statement, file-name-3 must not be open.

The SORT statement automatically does the following:

- Starts the processing of file-name-3
- Releases the logical records to file-name-3
- Ends the processing of file-name-3

The terminating function is performed as if a CLOSE statement without optional phrases had been executed for the file. These implicit functions are performed so that any associated USE procedures are executed. Also, the SORT statement automatically and implicitly returns the sorted records from the final phase of the sort operation and moves the records from the file area for file-name-1 to the file area for file-name-3.

The SAVE, LOCK, PURGE, RELEASE, NO REWIND, and CRUNCH options can be used to specify the type of file close operation for the GIVING file.

For More Information

- For information about the SORT system utility, refer to the *System Software Utilities Operations Reference Manual.*
- For information on the SAVE, LOCK, PURGE, RELEASE, NO REWIND, and CRUNCH options for closing files, refer to "CLOSE" in this section.

Examples

In Example 9–28, the files defined by the FD entries are opened before the SORT statement is executed. The SORT statement sets the sorting mechanism in motion by transferring control to the input procedure. The files can be opened with the input procedure, but the input file must not be opened more than once or closed before input is completed.

Sorting is conducted on an ascending key. Note that the key identifiers are defined in the sort file WORK-ING, not the input file TO-BE-SORTED.

The input procedure is completed when the entire input file has been transferred to the sort file. The sort mechanism then transfers control to the output procedure, and records are written on the file AFTER-THE-SORT in the order specified by the key parameters. The sorted records are then listed after the sort operation has been completed.

```
IDENTIFICATION DIVISION.
*
*THIS SORT PROGRAM EXAMPLE SHOWS HOW THE "INPUT PROCEDURE IS"
*AND "OUTPUT PROCEDURE IS" OPTIONS WORK.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. B7900.
 OBJECT-COMPUTER. B7900.
 INPUT-OUTPUT SECTION.


FILE-CONTROL.
    SELECT TO-BE-SORTED     ASSIGN TO DISK.
    SELECT AFTER-THE-SORT   ASSIGN TO DISK.
    SELECT WORK-ING         ASSIGN TO SORT.
    SELECT FOR-THE-PRINTER  ASSIGN TO PRINTER.
 DATA DIVISION.
 FILE SECTION.
 FD  TO-BE-SORTED
     LABEL RECORDS ARE OMITTED
     DATA RECORD IS A-FILE.
 01  A-FILE.
     02  1-ACCOUNT  PICTURE IS X(4)
```

```
     02  1-TYPE PICTURE IS X.
     02  1-AREA PICTURE IS XXX.
     02  1-BODY PICTURE IS X(72).
 FD  AFTER-THE-SORT
     LABEL RECORDS ARE OMITTED
     DATA RECORD IS B-FILE.
 01  B-FILE PIC IS X(80).
 SD  WORK-ING
     DATA RECORD IS C-FILE.
```

Example 9–28. INPUT PROCEDURE IS and OUTPUT PROCEDURE IS Options

```
 01  C-FILE.
     02  3-ACCOUNT PIC IS X(4).
     02  3-TYPE PIC IS X.
     02  3-AREA PIC IS XXX.
     02  3-BODY PIC IS X(72).
FD  FOR-THE-PRINTER
     LABEL RECORDS ARE OMITTED
     DATA RECORD IS D-FILE.
 01  D-FILE PIC IS X(80).
 PROCEDURE DIVISION.
*
*FILES DEFINED BY FD ENTRIES ARE OPENED.
*
OPENING SECTION.
     OPENER.
      OPEN INPUT TO-BE-SORTED.
      OPEN OUTPUT AFTER-THE-SORT FOR-THE-PRINTER.
*
*SORTING BEGINS AS CONTROL IS PASSED TO THE INPUT PROCEDURE.
*SORTING IS CONDUCTED ON 3 ASCENDING KEYS DEFINED IN WORK-ING.
*THE INPUT PROCEDURE IS COMPLETED WHEN THE ENTIRE INPUT FILE IS
*TRANSFERRED TO THE SORT FILE.
*

*THE SORT FILE THEN TRANSFERS CONTROL TO THE OUTPUT PROCEDURE, AND
*RECORDS ARE WRITTEN ON THE FILE AFTER-THE-SORT IN THE ORDER
*SPECIFIED BY THE KEY PARAMETERS.  THE SORTED RECORDS ARE THEN
*LISTED AFTER THE SORT FUNCTION IS DONE.
*
     A-SORT SECTION.
     P1.  SORT WORK-ING ON ASCENDING KEY 3-ACCOUNT 3-TYPE 3-AREA
            INPUT PROCEDURE IS B-SORT THROUGH B1-SORT
            OUTPUT PROCEDURE IS C-SORT THROUGH C1-SORT.
            GO TO OK-PRINT.
     B-SORT SECTION.
     P2.  READ TO-BE-SORTED AT END GO TO B1-SORT.
```

```
     RELEASE C-FILE FROM A-FILE.
     GO TO B-SORT.
B1-SORT SECTION.
P3. CLOSE TO-BE-SORTED.
C-SORT SECTION.
P4.  RETURN WORK-ING RECORD INTO B-FILE AT END GO TO C1-SORT.
       WRITE B-FILE.
       GO TO C-SORT.
C1-SORT SECTION.
P5.  CLOSE AFTER-THE-SORT.
OK-PRINT SECTION.
P6.  OPEN INPUT AFTER-THE-SORT.
A-LOOP SECTION.
```

Example 9-28. INPUT PROCEDURE IS and OUTPUT PROCEDURE IS Options

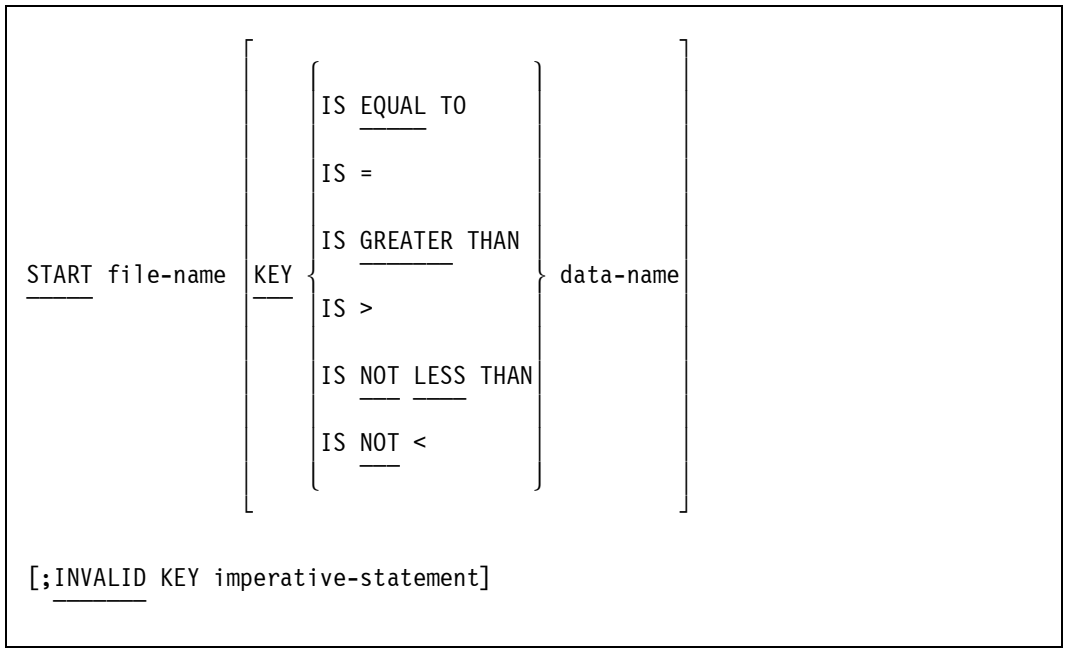
```
P7.  READ AFTER-THE-SORT AT END GO TO A-1-MOVE.
     PERFORM A-MOVE.
     WRITE D-FILE.
     GO TO A-LOOP.    A-MOVE SECTION.
P8. MOVE B-FILE TO D-FILE.
A-1-MOVE SECTION.
P9.  EXIT.
ALL-DONE SECTION.
PA.  CLOSE AFTER-THE-SORT FOR-THE-PRINTER.
     STOP RUN.
```

Example 9–28. INPUT PROCEDURE IS and OUTPUT PROCEDURE IS Options

START

The START statement positions records logically in a relative or an indexed file for sequential record retrieval.

Before your program executes a START statement against a file, it must execute an OPEN statement against the file so that the file is open when the program executes the START statement. The results of using means other than the OPEN verb (for example, file attributes) to cause the file to be open are unpredictable.



Note: *The required relational characters >, <, and = are not underlined to avoid confusion with other symbols.*

Explanation of Format

file-name

The file-name must be the name of a file with sequential or dynamic access.

The file-name must be open in the INPUT or I-O mode when the START statement is executed.

Execution of the START statement causes the value of the FILE STATUS data item, if any, associated with the file-name to be updated.

KEY

For relative I/O, any specified data-name must be the data item specified in the RELATIVE KEY phrase of the associated file-control entry.

For indexed I/O, if the KEY phrase is specified, the data-name can reference either of the following:

- A data item specified as a record key associated with the file-name
- Any alphanumeric data item subordinate to the data item specified as a record key, and associated with a file-name with a leftmost character position corresponding to the leftmost character position of that record key data item

If the KEY phrase is not specified, the relational operator IS EQUAL TO is implied.

If execution of the START statement is unsuccessful, the reference key is undefined.

The data-name can be qualified.

imperative-statement

The imperative-statement can be the NEXT SENTENCE phrase.

For More Information

- For information about the status of I/O operations, refer to "I/O Status" in Section 5, "ENVIRONMENT DIVISION."
- For information about using the INVALID KEY phrase, refer to "READ" earlier in this section and "Handling I/O Exception Conditions" in Section 8, "PROCEDURE DIVISION Concepts."

Relative I/O Comparison

The type of comparison specified by the relational operator in the KEY phrase occurs between a key associated with a record in the file referenced by the file-name and the data item referenced by the RELATIVE KEY clause associated with the file-name. The following occurs during a relative I/O comparison:

- The current-record pointer is positioned to the first logical record in the file with a key that satisfies the comparison.
- If the comparison is not satisfied by any record in the file, then an Invalid Key condition exists, execution of the START statement is unsuccessful, and the position of the current-record pointer is undefined.

Indexed I/O Comparison

The type of comparison specified by the relational operator in the KEY phrase occurs between a key associated with a record in the file referenced by the file-name and a data item referenced by the data-name. If the file-name references an indexed file and the operands are of unequal size, comparison proceeds as if the longer operand were truncated on the right so that its length equals that of the shorter operand. All other nonnumeric comparison rules apply, except that the presence of the PROGRAM COLLATING SEQUENCE clause does not affect the comparison.

The following occurs in an indexed I/O comparison:

- The current record pointer is positioned to the first logical record currently existing in the file that has a key that satisfies the comparison.
- If the comparison is not satisfied by any record in the file, then an Invalid Key condition exists, a value of 23 is returned to the status key, execution of the START statement is unsuccessful, and the position of the current record pointer is undefined.

If the KEY phrase is not specified, the comparison previously described uses the data item referenced in the RECORD KEY clause associated with the file-name.

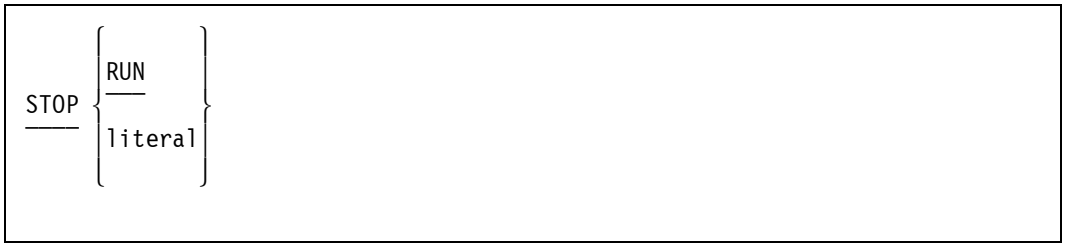
Reference Key Use

When the START statement has successfully executed, a reference key is established and used in subsequent Format 3 READ statements as follows:

- If the KEY phrase is not specified, the prime record key specified for the file-name becomes the reference key.
- If the KEY phrase is specified and the data-name is specified as a record key for file-name, that record key becomes the reference key.
- If the KEY phrase is specified and the data-name is not specified as a record key for the file-name, the record key with the leftmost character position corresponding to the leftmost character position of the data item specified by the data-name becomes the reference key.

STOP

The STOP statement permanently ends or temporarily suspends the object program.



Explanation of Format

STOP RUN

When the STOP RUN statement is executed, all open files are closed, all storage areas are returned to the system, and the program is terminated normally. If the program is linked to other programs as a run-unit through the standard COBOL Inter-Program Communication (IPC) mechanism, or if the program is operating as a library, these programs are also ended.

STOP literal

If the *STOP literal* statement is specified, the literal is displayed to the operator. When the operator responds with an OK response for the program, the object program continues with the next executable statement.

If the program is run from CANDE, the literal is also displayed on the remote terminal. The OK response can also be entered from the terminal. For more information, refer to the *CANDE Operations Reference Manual.*

The literal can be numeric or nonnumeric, and can be any figurative constant that does not include the optional word ALL.

If the literal is numeric, it must be an unsigned integer.

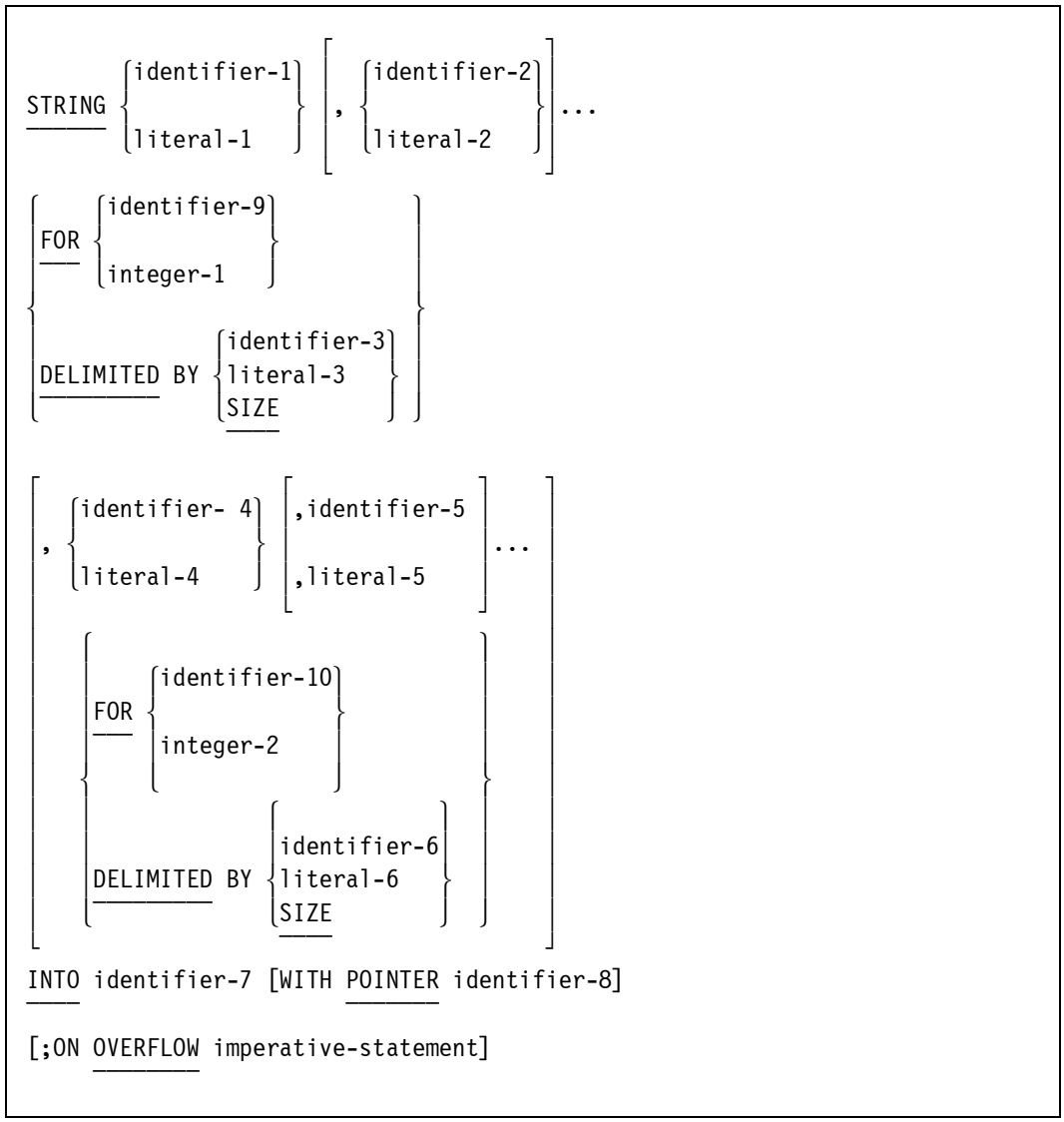
A *STOP literal* statement in a called program that is part of an IPC run-unit is ignored at execution time.

For More Information

Execution of the STOP RUN statement in a called program is described under “CALL” in this section.

STRING

The STRING statement joins several data items to form one data item. This process is termed concatenation.



Explanation of Format

All references to identifier-1, identifier-2, identifier-3, identifier-9, literal-1, literal-2, literal-3, and integer-1 apply equally to identifier-4, identifier-5, identifier-6, identifier-10, literal-4, literal-5, literal-6, and integer-2, respectively. (Identifier-9, integer-1, identifier-10, and integer-2 are extensions to ANSI X3.23-1974 COBOL.)

All literals must be described as nonnumeric literals, and all identifiers except identifier-8, identifier-9, and identifier-10 must be described implicitly or explicitly as USAGE IS DISPLAY.

When a sending item and a receiving item in the same STRING statement share a part, but not all, of their storage areas, the result of the execution of the statement is undefined.

identifier-1 or literal-1

Identifier-1 or literal-1 and any repetition represent the sending items.

Characters from literal-1, literal-2 or from the contents of identifier-1, identifier-2 are transferred to identifier-7 under the rules for alphanumeric-to-alphanumeric moves, except that no space-filling is provided.

Literal-1 can be any figurative constant that does not include the optional word ALL.

If identifier-1 or any repetition is an elementary numeric data item, then the data item must be described as an integer.

When a figurative constant is specified as literal-1, it refers to an implicit 1-character data item with USAGE IS DISPLAY.

FOR (Extension to ANSI X3.23-1974 COBOL)

If the FOR phrase is specified, the contents of identifier-1, identifier-2 or the contents of literal-1, literal-2 are transferred to identifier-7 in the sequence specified in the STRING statement. This sequence begins with the leftmost character and continuing from left to right until the end of identifier-7 is reached, or until the number of characters specified by integer-1 or by the contents of identifier-9 are moved.

Identifier-9 and identifier-10 must represent data items that are elementary numeric integers.

Integer-1 and identifier-9 indicate the number of characters to be moved.

DELIMITED BY identifier-3 or literal-3

This phrase specifies the characters that delimit the end of the data item to be joined.

The contents of identifier-1, identifier-2 or the contents of literal-1, literal-2 are transferred to identifier-7 in the sequence specified in the STRING statement. This sequence begins with the leftmost character and continuing from left to right until the end of the data item is reached, or until the characters specified by literal-3 or by the contents of identifier-3 are encountered. The characters specified by literal-3 or identifier-3 are not transferred.

When a figurative constant is used as a delimiter, it stands for a single-character, nonnumeric literal.

When a figurative constant is specified as literal-3, it refers to an implicit 1-character data item with display usage. Literal-3 can be any figurative constant that does not include the optional word ALL.

Literal-3 or identifier-3 indicates the characters delimiting the move.

DELIMITED BY SIZE

If the SIZE phrase is used, the complete data item defined by identifiers or literals is moved.

The contents of literal-1, literal-2 or the contents of identifier-1, identifier-2 are transferred (in the sequence specified in the STRING statement) to identifier-7 until all the data has been transferred or until the end of identifier-7 is reached.

INTO

Identifier-7 represents the receiving item.

When characters are transferred to identifier-7, they are moved in the following two ways:

- As if the characters were moved one at a time from the source into the character position of the data item referenced by identifier-7 and designated by the value associated with identifier-8.
- As if identifier-8 were increased by one prior to the movement of the next character. The value associated with identifier-8 is changed during execution of the STRING statement only as specified in the previous paragraph.

At the end of the execution of the STRING statement, only the portion of identifier-7 that was referenced during the STRING statement's execution is changed.

Identifier-7 must represent an alphanumeric data item without editing symbols or the JUSTIFIED clause.

POINTER

This option indicates the starting position for data to be moved to the receiving field.

The initial value of identifier-8 must not be less than 1.

Identifier-8 must represent an elementary-numeric-integer data item large enough to contain a value equal to the size of the area referenced by identifier-7 plus 1.

If, at any time during or after the STRING-statement initialization but before the completion of the STRING statement execution, the value of identifier-8 is less than 1 or exceeds the number of character positions in identifier-7, no further data is transferred to identifier-7. The imperative-statement in the ON OVERFLOW phrase is then executed, if specified.

If the POINTER phrase is not specified, the following overflow rules apply as if identifier-8 were specified with an initial value of 1.

ON OVERFLOW

An overflow condition occurs when the receiving field is full and the sending field still has characters to be moved, or when the pointer does not specify a valid value.

If the ON OVERFLOW phrase is not specified when the overflow conditions are encountered, control is transferred to the next executable statement.

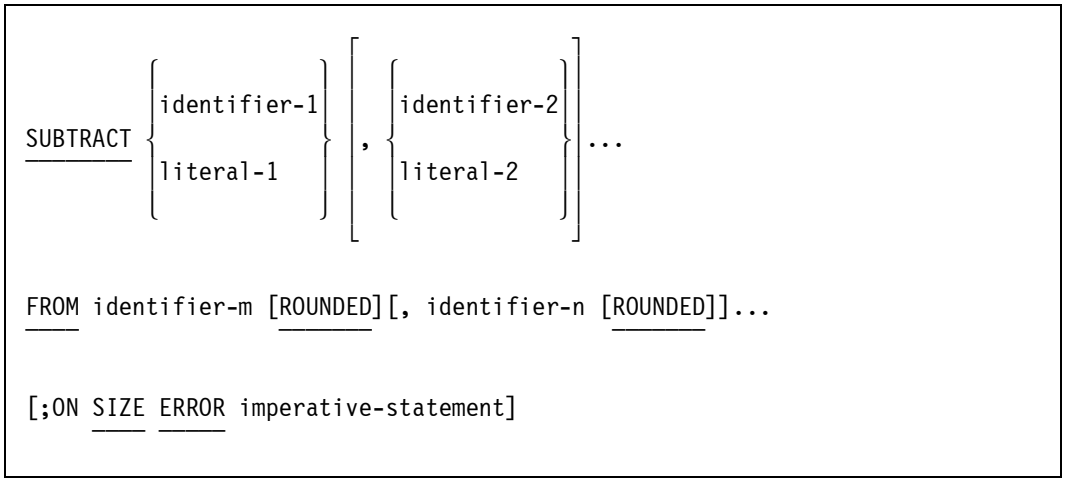
The imperative-statement can be the NEXT SENTENCE phrase.

SUBTRACT

The SUBTRACT statement subtracts one numeric data item or the sum of two or more numeric data items from one or more items, and sets the values of one or more items equal to the results.

Format	Explanation
1	Adds the operands preceding the word FROM and subtracts the total from the current value of identifier-m; the results of the subtraction are stored in identifier-n. This process is repeated for each operand following the word FROM.
2	Adds the operands preceding the word FROM and subtracts the total from identifier-m. The result is stored in identifier-n.
3	Subtracts corresponding items.

Format 1



Explanation of Format 1

In Format 1, all literals or identifiers preceding the word FROM are added together. This total is subtracted from the current value of identifier-m. The result is immediately stored in identifier-m; the process is then repeated for each operand following the word FROM.

Each identifier must refer to a numeric elementary item.

Each literal must be a numeric literal.

The imperative-statement can be the NEXT SENTENCE phrase.

Format 2

```
                 ⎧identifier-1⎫ ⎡, identifier-2⎤          ⎧identifier-m⎫
SUBTRACT         ⎨            ⎬ ⎢              ⎥ ...FROM  ⎨           ⎬
                 ⎩literal-1   ⎭ ⎣, literal-2   ⎦          ⎩literal-m   ⎭

GIVING identifier-n [ROUNDED][, identifier-o [ROUNDED]]...


[;ON SIZE ERROR imperative-statement]
```

Explanation of Format 2

In Format 2, all literals or identifiers preceding the word FROM are added together, the sum is subtracted from literal-m or identifier-m, and the result of the subtraction is stored as the new value of identifier-n, identifier-o, and so on.

Each identifier must refer to a numeric elementary item.

Each identifier following the word GIVING must refer to either an elementary numeric item or an elementary numeric-edited item.

Each literal must be a numeric literal.

Format 3

```
                 ⎧               ⎫
                 ⎪CORRESPONDING  ⎪
SUBTRACT         ⎨               ⎬ identifier-1
                 ⎪CORR           ⎪
                 ⎩               ⎭

FROM identifier-2 [ROUNDED]


[;ON SIZE ERROR imperative-statement]
```

Explanation of Format 3

If Format 3 is used, data items in identifier-1 are subtracted from, and stored into, corresponding data items in identifier-2.

Each identifier must refer to a group item.

CORR is an abbreviation for CORRESPONDING.

The imperative-statement can be the NEXT SENTENCE phrase.

General Rules

When a sending item and a receiving item in the same SUBTRACT statement share a part, but not all, of their storage areas, the result of the execution of the statement is undefined.

Note that unpredictable results occur in a Format 1 SUBTRACT statement if the same operand appears more than once in the list of operands that follows the word FROM in the statement. For example, assuming X contains the value 9 and Y contains the value 2, the value of X is 7 instead of 3 after execution of the following statement:

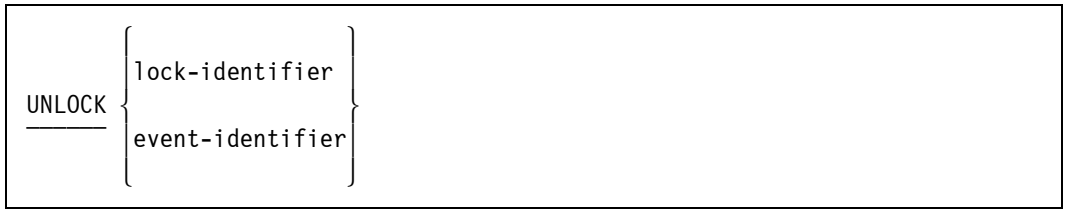
```
SUBTRACT Y FROM X, X, X
```

For More Information

- For information on the format rules for arithmetic statements, refer to “Common Rules for Arithmetic Statements” in Section 8, “PROCEDURE DIVISION Concepts.”
- For information about corresponding fields in group items, refer to “CORRESPONDING Phrase” in Section 8, “PROCEDURE DIVISION Concepts.”
- For information about storing the result of the subtraction in more than one field, refer to “Calculating Multiple Results with One Arithmetic Statement” in Section 8, “PROCEDURE DIVISION Concepts.”
- For information on truncation and rounding of results, refer to “ROUNDED Phrase” in Section 8, “PROCEDURE DIVISION Concepts.”
- For information on handling Size Error conditions, refer to “SIZE ERROR Phrase” in Section 8, “PROCEDURE DIVISION Concepts.”

UNLOCK (Extension to ANSI X3.23-1974 COBOL)

The UNLOCK statement is used in an asynchronous processing environment in conjunction with the LOCK statement.



Explanation of Format

The UNLOCK statement acts as the logical opposite of the LOCK statement by releasing the imposed common resource restriction.

The system intrinsic LIBERATE function is invoked by the UNLOCK statement.

Lock-identifiers must be declared with the phrase USAGE IS LOCK. Event-identifiers must be declared with the phrase USAGE IS EVENT or event-valued task attributes.

For information about locking common storage areas and testing for a locked condition, refer to “LOCK” in this section.

UNSTRING

The UNSTRING statement separates contiguous data in a sending field and places it in multiple receiving fields.

When a sending item and a receiving item in the same UNSTRING statement share a part, but not all, of their storage areas, the result of the execution of the statement is undefined.

Format	Explanation
1	Separates one field into a number of fields. A delimiter specifies the location at which to separate the field.
2	Specifies a number of consecutive characters in a sending field to be moved to a receiving field.

Format 1

```
                      [                         {identifier-2}         ]
                      [ DELIMITED BY [ALL]      {literal-1   }         ]
UNSTRING identifier-1 [                                                ]
                      [    [ , OR [ALL] {identifier-3} ]               ]
                      [    [            {literal-2   } ] ...           ]

INTO identifier-4 [, DELIMITER IN identifier-5][, COUNT IN identifier-6]

[,identifier-7 [, DELIMITER IN identifier-8][, COUNT IN identifier-9]]...

[, WITH POINTER identifier-10][, TALLYING IN identifier-11]

[;ON OVERFLOW imperative-statement]
```

Explanation of Format 1

Each literal must be a nonnumeric literal and can be any figurative constant that does not include the optional word ALL. Identifier-1, identifier-2, identifier-3, identifier-5, and identifier-8 must be described as alphanumeric data items.

Identifier-6, identifier-9, identifier-10, and identifier-11 must be described as elementary-numeric-integer data items.

All references in this discussion to identifier-2, literal-1, identifier-4, identifier-5, and identifier-6 apply equally to identifier-3, literal-2, identifier-7, identifier-8, and identifier-9, respectively.

identifier-1

Identifier-1 represents the sending area.

DELIMITED BY

Literal-1 or identifier-2 specifies a delimiter.

ALL

When the optional ALL phrase is specified, one or more contiguous occurrences of the delimiter are considered to be one delimiter (for matching purposes), but only one of these occurrences is moved to the receiving area for delimiters.

literal-1 or identifier-2

Each literal-1 or identifier-2 represents one delimiter. When a delimiter contains two or more characters, all of the characters must be present in contiguous positions of the sending item, in the order given, to be recognized as a delimiter.

When the program encounters two contiguous delimiters, the current receiving area is either space- or zero-filled, according to the description of the receiving area.

OR

When two or more delimiters are specified in the DELIMITED BY phrase, an Or condition exists between them. Each delimiter is compared to the sending field. If a match occurs, multiple characters in the sending field are considered to be a single delimiter. No characters in the sending field can be considered a part of more than one delimiter.

Each delimiter is applied to the sending field in the sequence specified in the UNSTRING statement.

When a figurative constant is used as a delimiter, it stands for a single-character, nonnumeric literal.

INTO

Identifier-4 represents the data receiving area and is described as USAGE IS DISPLAY.

Identifier-4 can be described as one of the following:

- Alphabetic (except that the symbol B cannot be used in the PICTURE character string)
- Alphanumeric (except that the symbol P cannot be used in the PICTURE character string)

- Numeric (except that the symbol P cannot be used in the PICTURE character string)

DELIMITER IN

If several delimiters are used to split a field, the DELIMITED IN option can be used to identify the delimiter that unstrung the field.

Identifier-5 represents the receiving area for delimiters.

The DELIMITER IN phrase can be specified only if the DELIMITED BY phrase is specified.

COUNT IN

This phrase counts the number of characters in each field.

Identifier-6 represents the number of characters in identifier-1 that are isolated by the delimiters for the move to identifier-4. This value does not include the number of delimiter characters.

The COUNT IN phrase can be specified only if the DELIMITED BY phrase is specified.

POINTER

Identifier-10 must be described as an elementary-numeric-integer data item.

The initialization of the contents of the data items associated with the POINTER phrase is your responsibility.

The contents of identifier-10 are incremented by 1 for each character examined in identifier-1. When execution of an UNSTRING statement with a POINTER phrase is completed, identifier-10 contains a value equal to its initial value plus the number of characters examined in identifier-1.

TALLYING

Identifier-11 is a counter that records the number of data items acted on during the execution of an UNSTRING statement. When the execution of an UNSTRING statement with a TALLYING phrase is completed, identifier-11 contains a value equal to its initial value plus the number of data receiving items acted upon.

Identifier-11 must be described as an elementary-numeric-integer data item.

The initialization of the contents of the data items associated with the TALLYING phrase is your responsibility.

ON OVERFLOW

Either of the following situations causes an overflow condition:

- An UNSTRING statement is initiated, and the value in identifier-10 is less than 1 or greater than the size of identifier-1.
- During execution of an UNSTRING statement, all data receiving areas have been acted upon and identifier-1 contains characters that have not been examined.

When an overflow condition exists, the unstring operation ends. If an ON OVERFLOW phrase is specified, the imperative-statement included in the ON OVERFLOW phrase is executed. If the ON OVERFLOW phrase is not specified, control is transferred to the next executable statement.

The imperative-statement can be the NEXT SENTENCE phrase.

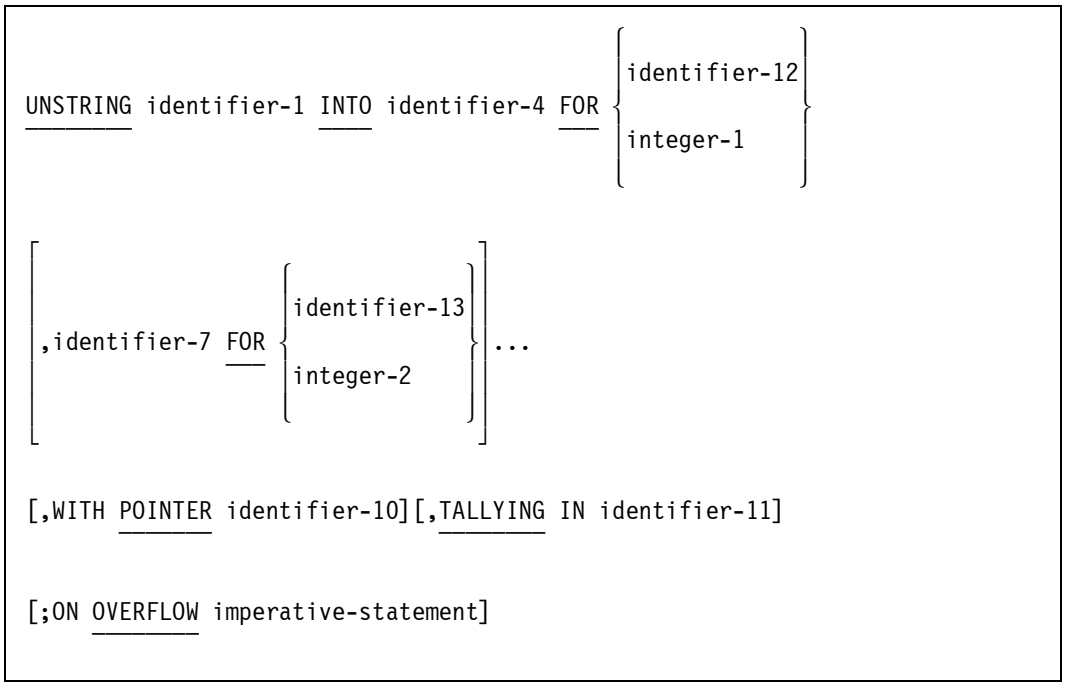
Rules for Data Transfer

When the UNSTRING statement is initiated, the current receiving area is identifier-4. Data is transferred from identifier-1 to identifier-4 according to the following rules:

- If the POINTER phrase is specified, the string of characters referenced by identifier-1 is examined beginning with the relative character position indicated by the value of identifier-10. If the POINTER phrase is not specified, the string of characters is examined beginning with the leftmost character position.
- If the DELIMITED BY phrase is specified, the examination proceeds from left to right until a delimiter specified either by literal-1 or by identifier-2 is encountered. If the DELIMITED BY phrase is not specified, the number of characters examined equals the size of the current receiving area. However, if the sign of the receiving item is defined as occupying a separate character position, the number of characters examined is one less than the size of the current receiving area.

  If the end of identifier-1 is encountered before the delimiting condition is met, the examination terminates with the last character examined.
- The characters thus examined, except any delimiting characters, are treated as elementary alphanumeric data items and are moved into the current receiving area according to the rules for the MOVE statement.
- If the DELIMITER IN phrase is specified, the delimiting characters are treated as elementary alphanumeric data items and are moved into identifier-5 according to the rules for the MOVE statement. If the delimiting condition is the end of identifier-1, then identifier-5 is space-filled.
- If the COUNT IN phrase is specified, a value equal to the number of characters thus examined, except any delimiting characters, is moved into identifier-6 according to the rules for a MOVE statement with elementary items.
- If the DELIMITED BY phrase is specified, the string of characters is further examined beginning with the first character to the right of the delimiter. If the DELIMITED BY phrase is not specified, the string of characters is further examined beginning with the character to the right of the last character transferred.
- After data is transferred to identifier-4, the current receiving area is identifier-7. The action described in the preceding paragraphs is repeated until the characters in identifier-1 are depleted or until no receiving areas remain.

Format 2 (Extension to ANSI X3.23-1974 COBOL)



Explanation of Format 2

All references to identifier-4, identifier-12, and integer-1 apply equally to identifier-7, identifier-13, and integer-2, respectively.

identifier-1

Identifier-1 represents the sending area and must be described as an alphanumeric data item.

identifier-4

Identifier-4 represents the data receiving area and must be described as USAGE IS DISPLAY.

Identifier-4 can be described as alphabetic (except that the symbol B cannot be used in the PICTURE character string), alphanumeric, or numeric (except that the symbol P cannot be used in the PICTURE character string).

The imperative-statement can be the NEXT SENTENCE phrase.

FOR

Integer-1 or identifier-12 specifies the number of characters in identifier-1 that are moved to identifier-4. If the number of characters remaining in identifier-1 is less than the number of characters specified by integer-1 or referenced by identifier-12, then the short field is transferred. Any characters examined, except delimiting characters, are treated as elementary alphanumeric data items and are moved into the current receiving area according to the rules for the MOVE statement.

Identifier-12 must be described as an elementary-numeric-integer data item (except that the symbol P cannot be used in the PICTURE character string).

POINTER

When the UNSTRING statement is initiated, the current receiving area is identifier-4. Data is transferred from identifier-1 to identifier-4 according to the following rules:

- If the POINTER phrase is specified, the string of characters referenced by identifier-1 is examined beginning with the relative character position indicated by the value of identifier-10. If the POINTER phrase is not specified, the string of characters is examined beginning with the leftmost character position.
- The characters thus examined, except any delimiting characters, are treated as elementary alphanumeric data items and are moved into the current receiving area according to the rules for the MOVE statement.

Identifier-10 must be described as an elementary-numeric-integer data item.

The initialization of the contents of the data items associated with the POINTER phrase is your responsibility.

The contents of identifier-10 are incremented by one for each character examined in identifier-1. When execution of an UNSTRING statement with a POINTER phrase is completed, identifier-10 contains a value equal to the initial value plus the number of characters examined in identifier-1.

TALLYING

Identifier-11 is a counter that records the number of data items acted on during the execution of an UNSTRING statement. When the execution of an UNSTRING statement with a TALLYING phrase is completed, identifier-11 contains a value equal to its initial value plus the number of data receiving items acted upon.

Identifier-11 must be described as an elementary-numeric-integer data item.

The initialization of the contents of the data items associated with the TALLYING phrase is your responsibility.

ON OVERFLOW

Each of the following situations causes an overflow condition:

- An UNSTRING statement is initiated, and the value in identifier-10 is less than 1 or greater than the size of identifier-1.
- During execution of an UNSTRING statement, all data receiving areas have been acted upon and identifier-1 contains characters that have not been examined.
- An UNSTRING statement is initiated, and the value in identifier-12 is less than 1 or greater than the size of identifier-1. (This is an extension to ANSI X3.23-1974 COBOL.)
- During execution of an UNSTRING statement, all data receiving areas have been acted upon and the number of characters acted upon is less than the value of identifier-12 or integer-1. (This is an extension to ANSI X3.23-1974 COBOL.)

When an overflow condition exists, the unstring operation ends. If an ON OVERFLOW phrase is specified, the imperative-statement included in the ON OVERFLOW phrase is executed. If the ON OVERFLOW phrase is not specified, control is transferred to the next executable statement.

The imperative-statement can be the NEXT SENTENCE phrase.

For information about the rules of move operations, "MOVE" in this section.

Examples of Format 1

Example 9–29 causes the contents of identifier-1 to be moved to identifier-4.

```
UNSTRING identifier-1 INTO identifier-4.
```

Variable	Before	After
identifier-1	123ABC	123ABC
identifier-4	3456789	123ABC

Example 9–29. Coding a Simple UNSTRING Statement

Example 9–30 causes the first three characters of identifier-1 to be moved to identifier-4. The delimiter G ends the unstring operation.

```
UNSTRING identifier-1 DELIMITED BY literal-1 INTO identifier-4.
```

Variable	Before	After
literal-1	G	G
identifier-1	STAGE	STAGE
identifier-4	CLUMP	STA

Example 9–30. UNSTRING Statement Using the DELIMITED BY Option

Example 9–31 causes the characters before the delimiter S to be moved to identifier-4. Identifier-7 becomes the receiving field, and the remaining characters are moved to this field.

```
UNSTRING identifier-1 DELIMITED BY ALL literal-1 INTO identifier-4,
identifier-7.
```

Variable	Before	After
identifier-1	MISSIN	MISSIN
literal-1	S	S
identifier-4	WWXXYYZZ	MI
identifier-7	ABC123	IN

Example 9–31. UNSTRING Statement Using the DELIMITED BY ALL Option

At the end of the unstring operation in Example 9–32, the pointer points one character beyond the last character processed. Therefore, the pointer, which is identifier-10, equals 9 because it points to the ninth character in the string.

```
UNSTRING identifier-1 INTO identifier-4 WITH POINTER identifier-10.
```

Variable	Before	After
identifier-1	BASEBALL	BASEBALL
identifier-4	FOOTBALL	BALL
identifier-10	5	9

Example 9–32. UNSTRING Statement Using the WITH POINTER Option

In Example 9–33, the sending string, ZSITXBYXRUN, cannot be entirely processed. The unstring operation ends before the remaining characters R, U, and N are encountered. This action causes an overflow condition.

The TALLYING IN phrase starts a count of the number of receiving strings filled.

```
UNSTRING identifier-1 DELIMITED BY literal-1 INTO identifier-4,
DELIMITER IN identifier-5, COUNT IN identifier-6, identifier-7,
DELIMITER IN identifier-8, COUNT IN identifier-9 WITH POINTER
 identifier-10 TALLYING IN identifier-11.
```

Variable	Before	After
literal-1	X	X
identifier-1	ZSITXBYXRUN	ZSITXBYXRUN
identifier-4	ABCDEF	SIT
identifier-5		X
identifier-6		3
identifier-7	GHIJKL	BY
identifier-8		X
identifier-9		2
identifier-10	2	9
identifier-11		2

Example 9–33. UNSTRING Statement Using Many Options

Examples of Format 2

In Example 9–34, use of the FOR phrase enables a specific number of consecutive characters in a sending string to be moved to a receiving string.

```
UNSTRING identifier-1 INTO identifier-4 FOR identifier-12,
  identifier-7 FOR identifier-13.
```

Variable	Before	After
identifier-1	XTOXBEXOFXDO	XTOXBEXOFXDO
identifier-4	SUXYZ	XTO
identifier-12	3	3
identifier-7	CDEGJKLM	XBEXOFX
identifier-13	7	7

Example 9–34. UNSTRING Statement Using the FOR Option

In Example 9–35, use of the FOR phrase enables a specific number of consecutive characters in a sending string to be moved to a receiving string. The string of characters examined begins at position 2, as indicated by the POINTER phrase.

Variable	Before	After
identifier-1	WXYZABCD	WXYZABCD
identifier-12	5	5
identifier-10	2	7
identifier-4	CARTON	XYZAB

```
UNSTRING identifier-1 INTO identifier-4 FOR identifier-12 with POINTER
identifier-10.
```

Example 9–35. Coding an UNSTRING Statement Using FOR, WITH POINTER Options

Example 9–36 is a sample program that uses the preceding example fragments from both Format 1 and 2.

```
002000 IDENTIFICATION DIVISION.
004000 ENVIRONMENT DIVISION.
006000 DATA DIVISION.
008000 WORKING-STORAGE SECTION.
010000 01  SOURCE6 PIC X(6).
012000 01  SOURCE5 PIC X(5).
014000 01  SOURCE8 PIC X(8).
016000 01  SOURCE11 PIC X(11).
018000 01  DEST PIC X(8).
020000 01  OTHERDEST PIC X(8).
022000 77  P PIC S9(11) BINARY.
024000 77  T PIC S9(11) BINARY.
026000 77  D PIC X.
028000 77  D2 PIC X.
030000 77  C PIC 9.
032000 77  C2 PIC 9.
034000 PROCEDURE DIVISION.
036000 LBL.
038000    MOVE "123ABC" TO SOURCE6.
040000    MOVE "3456789" TO DEST.
042000    DISPLAY "1. " SOURCE6 " " DEST.
044000    UNSTRING SOURCE6 INTO DEST.
046000    DISPLAY "1. " SOURCE6 " " DEST.
048000
050000    MOVE "STAGE" TO SOURCE5.
052000    MOVE "CLUMP" TO DEST.
```

Example 9–36. Sample Program Using Format 1 and 2 Examples

```
054000    DISPLAY "2. " SOURCE5 " " DEST.
056000    UNSTRING SOURCE5 DELIMITED BY "G" INTO DEST.
058000    DISPLAY "2. " SOURCE5 " " DEST.
060000
062000    MOVE "MISSIN" TO SOURCE6.
064000    MOVE "WWXXYYZZ" TO DEST.
066000    MOVE "ABC123" TO OTHERDEST.
068000    DISPLAY "3. " SOURCE6 " " DEST " " OTHERDEST.
070000    UNSTRING SOURCE6 DELIMITED BY ALL "S" INTO DEST, OTHERDEST.
072000    DISPLAY "3. " SOURCE6 " " DEST " " OTHERDEST.
074000
076000    MOVE "BASEBALL" TO SOURCE8.
078000    MOVE "FOOTBALL" TO DEST.
080000    MOVE 5 TO P.
082000    DISPLAY "4. " SOURCE8 " " DEST " " P.
084000    UNSTRING SOURCE8 INTO DEST WITH POINTER P.
086000    DISPLAY "4. " SOURCE8 " " DEST " " P.
088000
090000    MOVE "ZSITXBYXRUN" TO SOURCE11
092000    MOVE "ABCDEF" TO DEST.
094000    MOVE "GHIJKL" TO OTHERDEST.
096000    MOVE 2 TO P.
098000    MOVE 0 TO T.
100000    MOVE " " TO D, D2.102000    MOVE 0 TO C, C2.
104000    DISPLAY "5. " SOURCE11 " " DEST " " D " " C.
106000    DISPLAY "5. " OTHERDEST " " D2 " " C2 " " P " " T.
108000    UNSTRING SOURCE11 DELIMITED BY "X" INTO DEST
110000         DELIMITER IN D, COUNT IN C,
112000         OTHERDEST, DELIMITER IN D2, COUNT IN C2
114000         WITH POINTER P TALLYING IN T.
116000    DISPLAY "5. " SOURCE11 " " DEST " " D " " C.
118000    DISPLAY "5. " OTHERDEST " " D2 " " C2 " " P " " T.
120000
122000    MOVE "XTOXBEXOFXDO" TO SOURCE11.
124000    MOVE "SUXYZ" TO DEST.
126000    MOVE 3 TO C.
128000    MOVE "CDEGJKLM" TO OTHERDEST.
130000    MOVE 7 TO C2.
132000    DISPLAY "6. " SOURCE11 " " DEST " " C " " OTHERDEST " " C2.
134000    UNSTRING SOURCE11 INTO DEST FOR C, OTHERDEST FOR C2.
136000    DISPLAY "6. " SOURCE11 " " DEST " " C " " OTHERDEST " " C2.
```

Example 9–36. Sample Program Using Format 1 and 2 Examples

```
138000
140000    MOVE "WXYZABCD" TO SOURCE11.
142000    MOVE "CARTON" TO DEST.
144000    MOVE 5 TO C.
146000    MOVE 2 TO P.
148000    DISPLAY "7. " SOURCE11 " " C " " P " " DEST.
150000    UNSTRING SOURCE11 INTO DEST FOR C WITH POINTER P.
152000    DISPLAY "7. " SOURCE11 " " C " " P " " DEST.
154000
156000    STOP RUN.
```

Example 9–36. Sample Program Using Format 1 and 2 Examples

Example 9–37 shows the output from the previous program.

```
RUN UNSTRING/TEST
#RUNNING 5109
#5109 DISPLAY:1. 123ABC 3456789 .
#5109 DISPLAY:1. 123ABC 123ABC  .
#5109 DISPLAY:2. STAGE CLUMP   .
#5109 DISPLAY:2. STAGE STA     .
#5109 DISPLAY:3. MISSIN WWXXYYZZ ABC123  .
#5109 DISPLAY:3. MISSIN MI       IN      .
#5109 DISPLAY:4. BASEBALL FOOTBALL +00000000005.
#5109 DISPLAY:4. BASEBALL BALL     +00000000009.
#5109 DISPLAY:5. ZSITXBYXRUN ABCDEF     0.
#5109 DISPLAY:5. GHIJKL     0 +00000000002 +00000000000.
#5109 DISPLAY:5. ZSITXBYXRUN SIT      X 3.
#5109 DISPLAY:5. BY       X 2 +00000000009 +00000000002.
#5109 DISPLAY:6. XTOXBEXOFXD SUXYZ    3 CDEGJKLM 7.
#5109 DISPLAY:6. XTOXBEXOFXD XTO      3 XBEXOFX  7.
#5109 DISPLAY:7. WXYZABCD    5 +00000000002 CARTON  .
#5109 DISPLAY:7. WXYZABCD    5 +00000000007 XYZAB   .
```

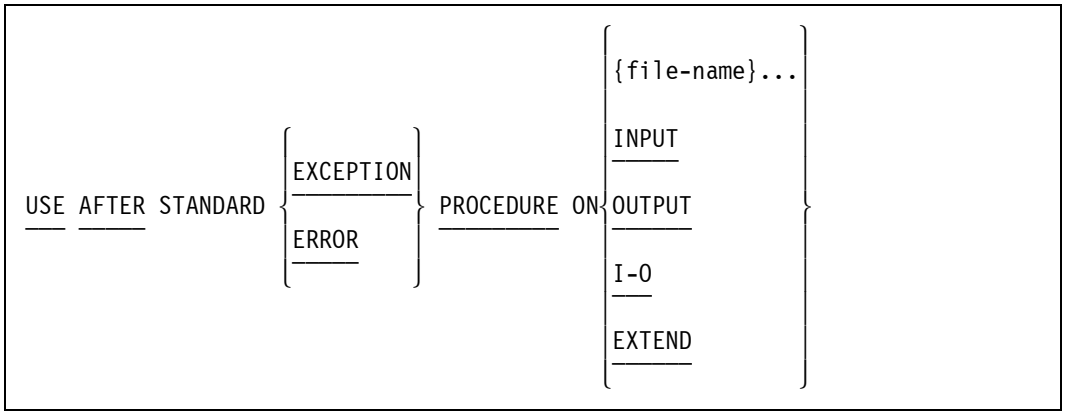
Example 9–37. Display from UNSTRING Program

USE

A USE statement, when present, must immediately follow a section header in the DECLARATIVES SECTION and must be followed by a period. The remainder of the sections consist of any number of procedural paragraphs that define the procedures to be used. The USE statement itself is never executed; it merely defines conditions that call for the execution of USE procedures.

Format	Explanation
1	Specifies supplemental procedures for I-O error and tape-label handling
2	Specifies procedures for tape-file-label handling
3	Specifies procedures to be employed in a parallel processing environment
4	Specifies interrupt procedures

Format 1



Explanation of Format 1

The USE AFTER statement specifies procedures for I/O exception handling.

After execution of a USE procedure, control is returned to the invoking routine.

The status key of a file can be used to determine the nature of the exception that occurred.

In a USE procedure, no reference can be made to any nondeclarative procedure. Conversely, in the nondeclarative portion of a program, no reference can be made to procedure-names that appear in the DECLARATIVES SECTION. The exception is that PERFORM statements in the nondeclarative portion of a program can refer to Format 1 USE statements or to procedures associated with a Format 1 USE statement.

ERROR and EXCEPTION

The words ERROR and EXCEPTION are synonymous and can be used interchangeably.

file-name

The designated procedures are executed on recognition of any error or exception for the file, including an end-of-file exception when no AT END clause has been specified.

The same file-name can appear in a different specific arrangement of the format. The appearance of a file-name in a USE statement must not cause a simultaneous request for execution of more than one USE procedure.

The files implicitly or explicitly referenced in a USE statement need not all have the same organization or access.

INPUT

The exception handling procedures are executed for any file opened in input mode, or in the process of being opened in input mode, except if the file is referenced by file-name in another USE statement specifying the same condition.

OUTPUT

The exception handling procedures are executed for any file opened in output mode, or in the process of being opened in output mode, except if the file is referenced by file-name in another USE statement specifying the same condition.

I/O

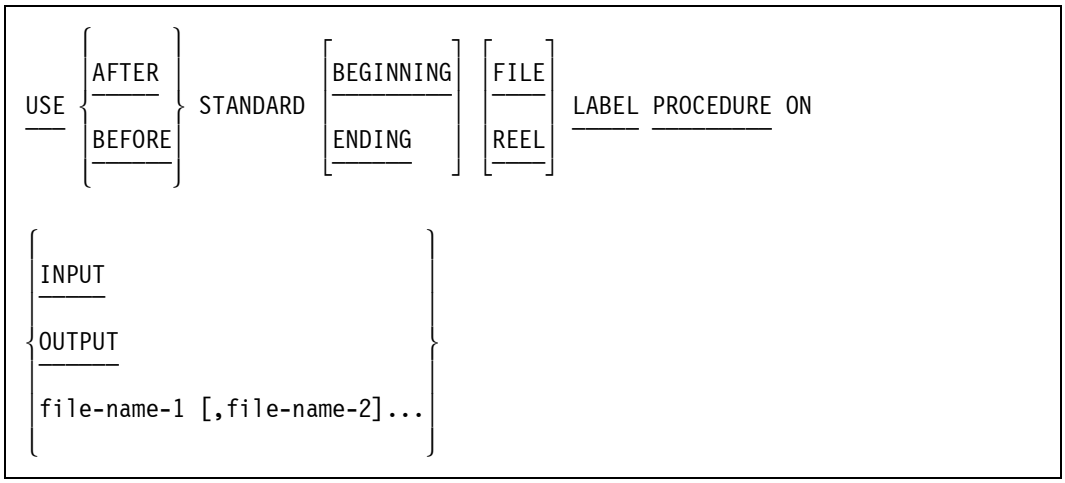
The exception handling procedures are executed for any file opened in I/O mode, or in the process of being opened in I/O mode, except if the file is referenced by file-name in another USE statement specifying the same condition.

EXTEND

The EXTEND phrase applies only to sequentially organized files. The exception handling procedures are executed for any file opened in extended mode, or in the process of being opened in extended mode, except if the file is referenced by file-name in another USE statement specifying the same condition.

For information on exception handling, refer to “Handling I/O Exception Conditions” in Section 8, “PROCEDURE DIVISION Concepts.”

Format 2



Explanation of Format 2

Format 2 enables you to define up to nine tape labels (header and trailer records) that are each 80 characters long. The first four characters of each label are used by the system for maintaining the label identity. For example, header labels contain UHLn, and trailer labels contain UTLn, where *n* can be a number from 1 through 9. You can use the remaining 76 characters for storing any desired information.

You can define the labels in the WORKING-STORAGE SECTION or in the FILE SECTION in the respective file-description (FD) entry for the file. If you use multiple labels, you must define all labels in the same area, either in the FD entry, or in the WORKING-STORAGE SECTION. Label records are made available only when both the LABEL RECORDS clause and the USE statement are specified.

AFTER or BEFORE

Theses phrases determine when the I/O subsystem executes the designated procedures.

After execution of a USE AFTER LABEL or a USE BEFORE LABEL statement, control passes to the invoking routine in the system. If the I-O status value does not indicate a critical I/O error, the system returns control to the next executable statement that follows the I/O statement whose execution caused the USE procedure to be invoked.

A USE AFTER or a USE BEFORE statement must immediately follow a section header in the DECLARATIVES SECTION and must appear in a sentence by itself. The remainder of the section must consist of any number of procedural paragraphs that define the procedures to be used. The USE AFTER LABEL or the USE BEFORE LABEL statement is never executed itself; it merely defines the condition calling for the execution of the USE procedures.

A declarative procedure must not reference any nondeclarative procedures. The procedure-names associated with the USE AFTER LABEL statement or the USE BEFORE LABEL statement can be referenced in a different DECLARATIVE SECTION, or in a nondeclarative procedure with only a PERFORM statement.

BEGINNING or ENDING

The BEGINNING phrase specifies that USE procedures are executed for header labels. The ENDING phrase specifies that USE procedures are executed for trailer labels. If the BEGINNING phrase or the ENDING phrase is not included, the USE procedure is executed for both header and trailer labels.

FILE or REEL

The REEL phrase specifies that the USE procedure is executed for reel labels. The FILE phrase specifies that the USE procedure is executed for file labels. If the FILE phrase or the REEL phrase is not included, the USE procedure is executed for both file and reel labels.

INPUT or OUTPUT

The input procedure is a label-checking procedure. When the USE BEFORE LABEL INPUT clause is specified, the designated procedure is executed before the tape header or trailer label records are read.

When the USE AFTER LABEL INPUT clause is specified, the designated procedure is executed after the tape header or trailer label records are read.

The output procedure is a label writing procedure. When the USE BEFORE LABEL OUTPUT clause is specified, the designated procedure is executed before the tape header or trailer label records are written.

When the USE AFTER LABEL OUTPUT clause is specified, the designated procedure is executed after the tape header or trailer label records are written.

The INPUT and OUTPUT phrases can each be specified only once for any USE AFTER LABEL or USE BEFORE LABEL statement. The procedures are not executed for any file whose file-description (FD) entry specifies LABEL RECORDS ARE OMITTED.

When the INPUT or OUTPUT phrase is used, references to label data items must be qualified.

file-name

When the *USE BEFORE LABEL file-name-1* clause is specified, the designated procedure is executed before the tape header or trailer records are read or written.

When the *USE AFTER LABEL file-name-1* clause is specified, the designated procedure is executed after the tape header or trailer records are read or written.

The appearance of file-name-1 in a USE AFTER LABEL or a USE BEFORE LABEL statement must not cause the simultaneous request for execution of more than one USE AFTER LABEL or USE BEFORE LABEL procedure. That is, when file-name-1 is specified explicitly, no other USE AFTER LABEL or USE BEFORE LABEL statement can apply to file-name-1. The file-description (FD) entry for file-name-1 must not specify LABEL RECORDS ARE OMITTED.

The program must not open or close the file associated with the USE AFTER LABEL or USE BEFORE LABEL statement within that procedure.

Those files referenced by file-name-1 in USE AFTER LABEL or USE BEFORE LABEL statements are not included in any input or output procedures.

The same file-name can appear in a different specific arrangement of the format. The appearance of a file-name in a USE statement must not cause a simultaneous request for execution of more than one USE procedure.

For More Information

- Further information on labels can be found in the *I/O Subsystem Programming Guide*.
- For an example of tape label access by way of USE routines, refer to Appendix D.

Format 3 (Extension to ANSI X3.23-1974 COBOL)

```
USE { [ EXTERNAL { identifier-1   } AS PROCEDURE ]
    {            { mnemonic-name }                }
    { AS GLOBAL PROCEDURE                         }

[ ;WITH { local-storage-name-1 } [ , { local-storage-name-2 } ]... ]
        { file-name-1          }     { file-name-2          }

[ ;USING identifier-2 [,identifier-3]... ]
```

Explanation of Format 3

Format 3 enables untyped procedures or subroutines to be declared global in the same way that they can be declared external. The GLOBAL phrase identifies a procedure that exists in the host program and is to be called in a bound procedure.

There must be no paragraphs in a section with a USE EXTERNAL or USE AS GLOBAL PROCEDURE phrase.

The use of the STATISTICS option is incompatible with programs that use the USE EXTERNAL PROCEDURE statement. If STATISTICS is TRUE when the USE EXTERNAL

PROCEDURE is encountered during compilation, then a warning message is issued and the STATISTICS option is set to FALSE.

EXTERNAL

The EXTERNAL phrase identifies a separately compiled program that is to be used as a task when the section is referenced. The EXTERNAL phrase can also identify a separately compiled program to be bound into the COBOL74 host program. The distinction is made by the format of the CALL statement that invokes the procedure or initiates the task.

Identifier-1 must be defined in the WORKING-STORAGE SECTION so that its value can be a program-name. If mnemonic-name is used, it must be defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION to represent a program-name.

A program with the USE EXTERNAL PROCEDURE statement is not marked as library-capable.

GLOBAL

This phrase identifies a procedure in the host program that is referenced in a bound procedure.

WITH

Local-storage-names must be defined in the LOCAL-STORAGE SECTION. A local-storage-name must be present if the USING phrase is present.

The files implicitly or explicitly referenced in a USE statement need not all have the same organization or access.

USING

The USING phrase is included in the USE statement if there is a USING phrase in the PROCEDURE DIVISION header of the referenced, separately compiled program. The number, type, and order of the operands in the two USING phrases must then be identical.

Identifier-2 and so forth must be uniquely defined as files in the FILE SECTION or as 01-level or 77-level items of the local-storage-name specified in the WITH phrase. These identifiers can describe any combination of data items, task (control point) items, EVENT items, or lock items.

Format 4 (Extension to ANSI X3.23-1974 COBOL)

```
USE AS INTERRUPT PROCEDURE
```

Explanation of Format 4

This phrase specifies a declarative as an interrupt procedure. An interrupt procedure provides a means of interrupting a process when an EVENT item attached to that procedure is caused. Statements to be executed when the event is caused and the interrupt procedure is allowed must follow the USE statement. When an interrupt procedure is being executed, all other interrupts are disallowed during the execution. That is, an interrupt procedure cannot itself be interrupted.

For More Information

- For information on handling interrupt procedures, refer to ALLOW, ATTACH , CAUSE, DETACH, DISALLOW, and RESET statements in this section.
- For information on EVENT items, see "USAGE Clause" in Section 7, "DATA DIVISION."

WAIT (Extension to ANSI X3.23-1974 COBOL)

The WAIT statement suspends the execution of the object program for a specified time or until one or more conditions are TRUE.

Format	Explanation
1	Suspends an object program for a specified number of seconds
2	Suspends an object program until an event is caused
3	Suspends an object program until execution of an interrupt procedure

Format 1

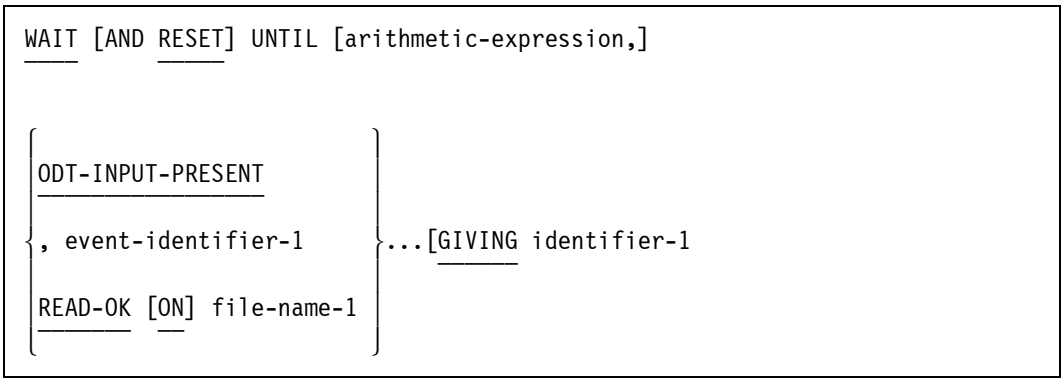
```
WAIT UNTIL arithmetic-expression
```

Explanation of Format 1

When an arithmetic-expression is specified, the execution of the object program is suspended for the number of seconds determined by the value of the expression.

The maximum WAIT time limit is 164,925 seconds (approximately 45.8 hours). If a wait time value greater than 164925 is specified in a WAIT statement, the task resumes after waiting 164,925 seconds.

Format 2



Explanation of Format 2

Format 2 causes a suspension for the number of seconds specified by the arithmetic-expression (if present) or until one of the event-identifiers has been caused.

If any event in the list is TRUE, the WAIT statement ends and control passes to the next executable statement.

If, in the initial scan of the events, no event is found to be TRUE, program execution is suspended until any event in the list becomes TRUE. When an event becomes TRUE, the WAIT statement ends and control passes and the next executable statement.

AND RESET

If the AND RESET phrase is specified, the event-identifier that causes the WAIT statement to end is reset. If the number of seconds specified by the arithmetic-expression has elapsed before an event is caused, the AND RESET phrase has no effect.

arithmetic-expression

When an arithmetic-expression is specified, the execution of the object program is suspended for the number of seconds determined by the value of the expression.

No more than one arithmetic-expression can be specified, and if specified, it must be the first item in the list.

ODT-INPUT-PRESENT

The ODT-INPUT-PRESENT clause is a special event-identifier. This event is caused whenever input is sent to the process by way of the AX (accept) system command. Using an ODT device to execute an ACCEPT statement causes the event to be reset. The event can also be reset with the AND RESET syntax.

No more than one ODT-INPUT-PRESENT clause can be specified.

Note: *No more than one ACCEPT message can be queued. This event corresponds to the system attribute ACCEPTEVENT.*

event-identifiers

Event-identifiers must be either data items with event usage or file or task attributes of type EVENT.

The EXCEPTIONEVENT attribute of a task-identifier is a special case event-identifier. The behavior of the WAIT statement is the same as that for other event-identifiers.

READ-OK

When the READ-OK option is specified, the program is suspended until there is at least one record to be read in file-name-1. For files that are not open, the event is always FALSE. This event can only be used with port files and remote files.

GIVING

When the GIVING option is specified, the data item referenced by identifier-1 is set to the position in the list of the data item that ended the WAIT statement. For example, if the second event in the list of events is TRUE, the data item referenced by identifier-1 is set to the value 2.

Identifier-1 must be described as an elementary-numeric-integer data item without the symbol P in its PICTURE character string.

For More Information

For information about the event-identifier condition, refer to "Conditional Expressions" in Section 8, "PROCEDURE DIVISION Concepts."

Format 3

```
WAIT UNTIL INTERRUPT
```

Explanation of Format 3

The INTERRUPT phrase causes execution of this task to be suspended until at least one of its interrupt procedures has been executed. After execution of the interrupt procedure, the task is again suspended. Thus, the WAIT INTERRUPT construct provides a mechanism for an interrupt-driven program. The program runs indefinitely unless one or more of the interrupt procedures contain a STOP RUN statement.

WRITE

The WRITE statement releases a logical record to a file. Also, it specifies the vertical position of lines on a logical page.

Before your program executes a WRITE statement against a file, it must execute an OPEN statement specifying a mode of OUTPUT, I-O, or EXTEND against the file so that it is open when the program executes the WRITE statement. The results of using means other than the OPEN verb (for example, file attributes) to cause the file to be open are unpredictable.

The WRITE statement does not affect the current-record pointer.

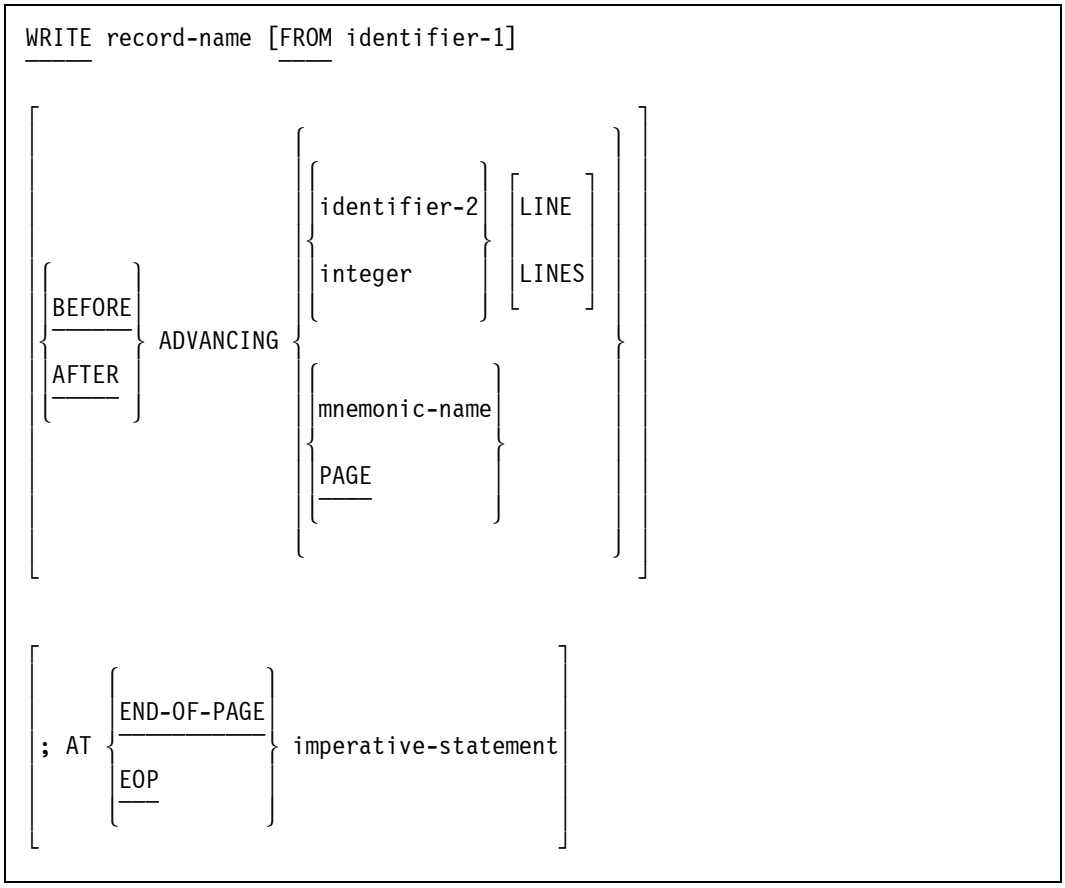
A write operation updates the value of the FILE STATUS data item associated with the file.

Format	Explanation
1	Writes records to a sequential file, and positions lines vertically on a logical page.
2	Records to sequential, relative, and indexed files. This format can be used with port files.
3	Creates Kanji records, positions lines vertically on a logical page, and writes the records to a sequential file. This format is for use with remote or printer files.
4	Creates Kanji records and writes them to sequential, relative, or indexed files.
5	Used with form libraries. Refer to Volume 2 for information on the WRITE FORM statement.

Format 1: Sequential I/O and Vertical Positioning of Lines

Format 1 is used for files of sequential organization that are not on mass-storage devices.

Format 1



Explanation of Format 1

record-name

The record-name is the name of a logical record in the FILE SECTION of the DATA DIVISION and can be qualified.

FROM

The FROM option makes the WRITE statement operate like a MOVE statement followed by a WRITE statement. The move takes place according to the rules of the MOVE statement without the CORRESPONDING option.

Record-name and identifier-1 must not reference the same storage area.

BEFORE

If the BEFORE phrase is used, the line is presented before the representation of the printed page is advanced according to rules described for the ADVANCING phrase.

AFTER

If the AFTER phrase is used, the line is presented after the representation of the printed page is advanced according to rules described for the ADVANCING phrase.

ADVANCING

The ADVANCING phrase controls the vertical positioning of each line on a representation of a printed page.

If the ADVANCING phrase is not used, automatic advancing is provided as if AFTER ADVANCING 1 LINE were specified. If the ADVANCING phrase is used, advancing is provided as follows:

- If identifier-2 is specified, the representation of the printed page is advanced a number of lines equal to the value of identifier-2. The variable identifier-2 must be the name of an elementary integer data item and can be equal to 0 (zero).
- If an integer is specified, the representation of the printed page is advanced a number of lines equal to the value of integer. The integer can be 0 (zero).
- If a mnemonic-name is specified, the representation of the printed page is advanced to the line number corresponding to the channel number. The mnemonic-name must be associated with a channel number. The mnemonic-name is defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

  The *ADVANCING mnemonic-name* phrase cannot be specified when writing a record to a file that has a file-description (FD) entry that contains the LINAGE clause.
- If the PAGE option is specified, the record is presented on the logical page either before or after the device is repositioned to the next logical page. If the record to be written is associated with a file that has a LINAGE clause in its file-description (FD) entry, the device is repositioned to the first line that can be written on the next logical page, as specified in the LINAGE clause. If the record to be written is associated with a file that does not have a LINAGE clause in its file-description (FD) entry, the device is repositioned to channel 1 or, when appropriate for the hardware device, to the next logical page.

  If the PAGE option has no meaning in conjunction with a specific device, then the page is adjusted as if BEFORE ADVANCING ONE LINE or AFTER ADVANCING 1 LINE (depending on the phrase used) were specified.

END-OF-PAGE (EOP)

The END-OF-PAGE phrase enables you to control the vertical positioning of each line on a representation of a printed page.

If the END-OF-PAGE phrase is specified, the LINAGE clause must be specified in the file-description (FD) entry for the associated file.

If the logical end of the representation of the printed page is reached during execution of a WRITE statement with the END-OF-PAGE phrase, the imperative-statement specified in the END-OF-PAGE phrase is executed. The logical end is specified in the LINAGE clause associated with record-name.

An End-Of-Page condition is reached whenever execution of a given WRITE statement with the END-OF-PAGE phrase causes printing or spacing within the footing area of a page body. This condition occurs when execution of such a WRITE statement causes the LINAGE-COUNTER, a special register, to equal or exceed the value specified by integer-6 or by the data item referenced by data-name-7 of the LINAGE clause. If the End-Of-Page condition occurs, the WRITE statement is executed and then the imperative-statement in the END-OF-PAGE phrase is executed.

An automatic Page-Overflow condition is reached whenever the execution of a WRITE statement (with or without an END-OF-PAGE phrase) cannot be fully accommodated within the current page body. This condition occurs when a WRITE statement causes the LINAGE-COUNTER register to exceed the value specified by integer-5 or data-name-6 of the LINAGE clause. If the Page-Overflow condition occurs, the record is presented on the logical page before or after (depending on the phrase used) the device is repositioned to the first line that can be written on the next logical page (as specified in the LINAGE clause). The imperative-statement in the END-OF-PAGE clause, if specified, is executed after the record is written and the device has been repositioned.

If the FOOTING phrase of the LINAGE clause is not specified, no End-Of-Page (EOP) condition distinct from the Page-Overflow condition is detected. In this case, the End-Of-Page (EOP) condition and Page-Overflow condition occur simultaneously.

If the FOOTING phrase of the LINAGE clause is specified, but execution of a given WRITE statement causes the LINAGE-COUNTER register to simultaneously exceed the value of both of the following items, then the operation proceeds as if integer-6 or data-name-7 had not been specified:

- Integer-6 or the data item referenced by data-name-7
- Integer-5 or the data item referenced by data-name-6

The words END-OF-PAGE and EOP are equivalent.

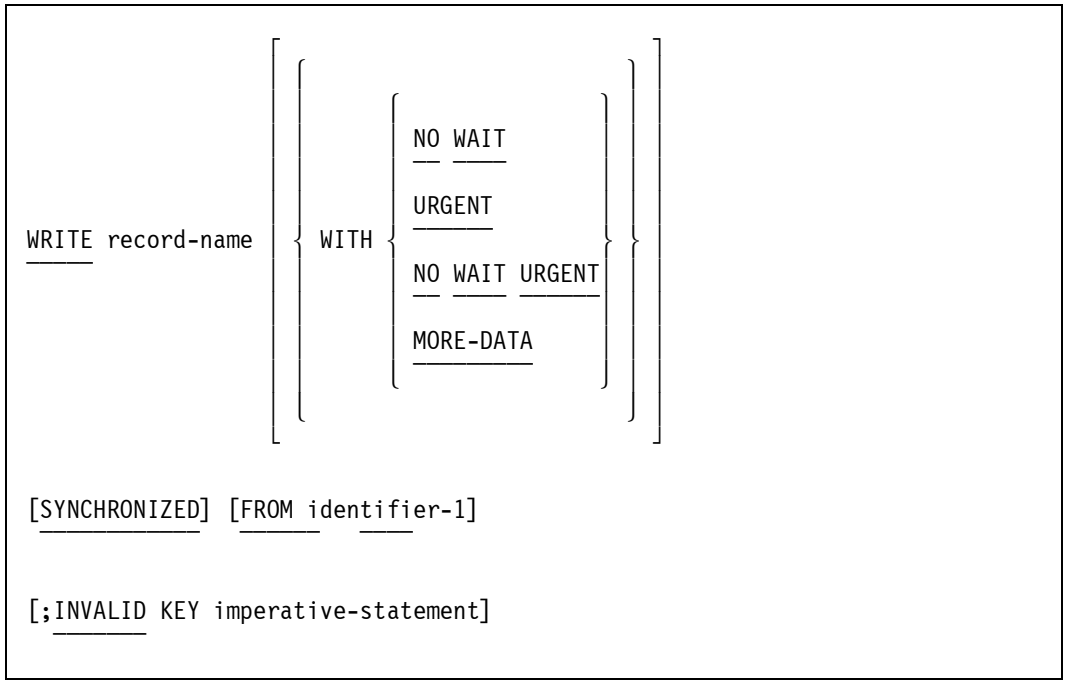
The imperative-statement can be the NEXT SENTENCE phrase.

For more information how to use the EOP phrase, refer to “Handling I/O Exception Conditions” in Section 8, “PROCEDURE DIVISION Concepts.”

Format 2: Sequential, Relative, and Indexed I/O

Format 2 is used for mass-storage files of any organization. Format 2 must be used for port files. (Port files are an extension to ANSI X3.23-1974 COBOL.)

Format 2



Explanation of Format 2

record-name

The record-name is the name of a logical record in the FILESECTION of the DATA DIVISION and can be qualified.

WITH

The WITH phrase can be specified only for port files. The NO WAIT and URGENT phrases can be included only once each, either individually or together. (These phrases are extensions to ANSI X3.23-1974 COBOL.)

The WITH URGENT phrase is meaningful only when the Transmission Control Protocol/Internet Protocol (TCP/IP) is being used. The WITH URGENT phrase sets the urgent indicator associated with the data. For more information on TCP/IP, refer to the *Distributed Systems Services (DSS) Operations Guide.*

The WITH MORE-DATA phrase enables an OSI port file that uses the segmented I/O capability to pass a message in segments, and thus override the message size limit of 64,512 words. The MORE-DATA phrase indicates that more data for the same message is forthcoming after the transmission of the initial segment.

SYNCHRONIZED

Synchronization of all output records can be designated with the SYNCHRONIZE file attribute. Synchronization means that output must be written to the physical file before the program initiating the output can resume execution, thereby ensuring synchronization between logical and physical files. The SYNCHRONIZED clause enables you to override the synchronization specified by the file attribute for a specific output record. A periodic synchronous WRITE statement that follows one or more asynchronous WRITE statements can be used as a checkpoint to ensure that all outstanding records are written to the file before the program continues execution. Synchronization is available for use by tape files and disk files with sequential organization only, and is not available for use by port files.

FROM

The results of the execution of the WRITE statement with the FROM phrase is equivalent to the execution of the statement *MOVE identifier-1 TO record-name* according to the rules specified for the MOVE statement, followed by the same WRITE statement without the FROM phrase.

Record-name and identifier-1 must not reference the same storage area.

INVALID KEY

When the Invalid Key condition occurs, any FILE STATUS data item of the file is set to a value indicating the cause of the condition.

The imperative-statement can be the NEXT SENTENCE phrase.

For More Information

- For information about the status of I/O operations, refer to "I/O Status" in Section 5, "ENVIRONMENT DIVISION."
- For information using the INVALID KEY phrase, refer to "Handling I/O Exception Conditions" in Section 8, "PROCEDURE DIVISION Concepts."

Sequential I/O

For a mass-storage file with sequential access mode, the execution of a Format 2 WRITE statement releases the record area to the next logical record in the file.

If the ACTUAL KEY phrase is specified for a mass-storage file with sequential access mode, the successful execution of a Format 2 WRITE statement updates the contents of the ACTUAL KEY data item to the ordinal number of the logical record written.

For a mass-storage file with random access mode, the execution of a Format 2 WRITE statement releases the record area to the logical record of the file specified by the contents of the ACTUAL KEY data item.

Additionally, for a mass-storage file with random access mode, an Invalid Key condition exists when the value of the ACTUAL KEY data item is less than one or greater than the ordinal number of the last logical record allowed for the file, if the maximum logical size of the file is specified.

For a mass-storage file with sequential access mode, the Invalid Key condition exists when a maximum logical size has been specified for the file and no more logical records can be written.

Relative I/O

When a file is opened in the OUTPUT mode, records can be placed in the file by one of the following methods:

- If the access mode is sequential, the WRITE statement causes a record to be released to the I/O subsystem. The first record has an ordinal record number of 1, and subsequent records released have ordinal record numbers of 2, 3, 4, and so on. If the relative key data item has been specified in the file-control entry for the associated file, the ordinal record number of the record just released is placed in the relative key data item by the I/O subsystem during execution of the WRITE statement.
- If the access mode is random or dynamic, then prior to the execution of the WRITE statement, the value of the relative key data item must be initialized in the program with the relative record number to be associated with the record in the record area. Execution of the WRITE statement releases that record in the I/O subsystem.

When a file is opened in the I-O mode and the access mode is random or dynamic, records are inserted in the associated file. The value of the RELATIVE KEY data item must be initialized by the program with the relative record number associated with the record in the record area. Execution of a WRITE statement then causes the contents of the record area to be released to the I/O subsystem.

The Invalid Key condition occurs under the following circumstances:

- The access mode is random or dynamic, and the RELATIVE KEY data item specifies a record already present in the file.
- An attempt is made to write beyond the logical boundaries of the file.

Indexed I/O

Execution of the WRITE statement causes the contents of the record area to be released. The I/O subsystem uses the contents of the record keys so that subsequent access of the record key can be made based on any of the specified record keys.

The value of the prime record key must be unique within the records in the file.

The data item specified as the prime record key must be set by the program to the desired value prior to execution of the WRITE statement.

If sequential access mode is specified for the file, records must be released to the I/O subsystem in ascending order of prime record key values.

If random or dynamic access mode is specified, records can be released to the I/O subsystem in any order.

When the ALTERNATE RECORD KEY clause is specified in the file-control entry for an indexed file, the value of the alternate record key can be nonunique only if the DUPLICATES phrase is specified for that data item. In this case, the I/O subsystem provides storage of records so that when records are accessed sequentially, the order of retrieval of those records is the order in which they are released to the I/O subsystem.

The Invalid Key condition occurs under the following circumstances:

- Sequential access mode is specified for a file opened in the OUTPUT mode, and the value of the prime record key is not greater than the value of the prime record key of the previous record.
- The file is opened in the OUTPUT or I-O mode, and the value of the prime record key equals the value of a prime record key of a record already present in the file.
- The file is opened in the OUTPUT or I-O mode, and the value of an alternate record key for which duplicates are not allowed equals the corresponding data item of a record already present in the file.

Port Files (Extension to ANSI X3.23-1974 COBOL)

A WRITE statement causes the program to wait until a buffer is available to store the record. For port files, you can prevent the possibility of this suspension by using the WITH NO WAIT phrase. A status key value of 95 indicates that no buffer was available for the logical record.

If you declare an ACTUAL KEY clause for a port file, you are responsible for updating the actual key with an appropriate value. A WRITE statement causes the actual key value to be passed to the I/O subsystem to indicate the desired subfile destination. If the actual key value is 0 (zero), the program performs a broadcast write operation. A broadcast write operation sends data to all opened subfiles of the port file.

If no ACTUAL KEY clause is declared for the file, the file must contain a single subfile that is written.

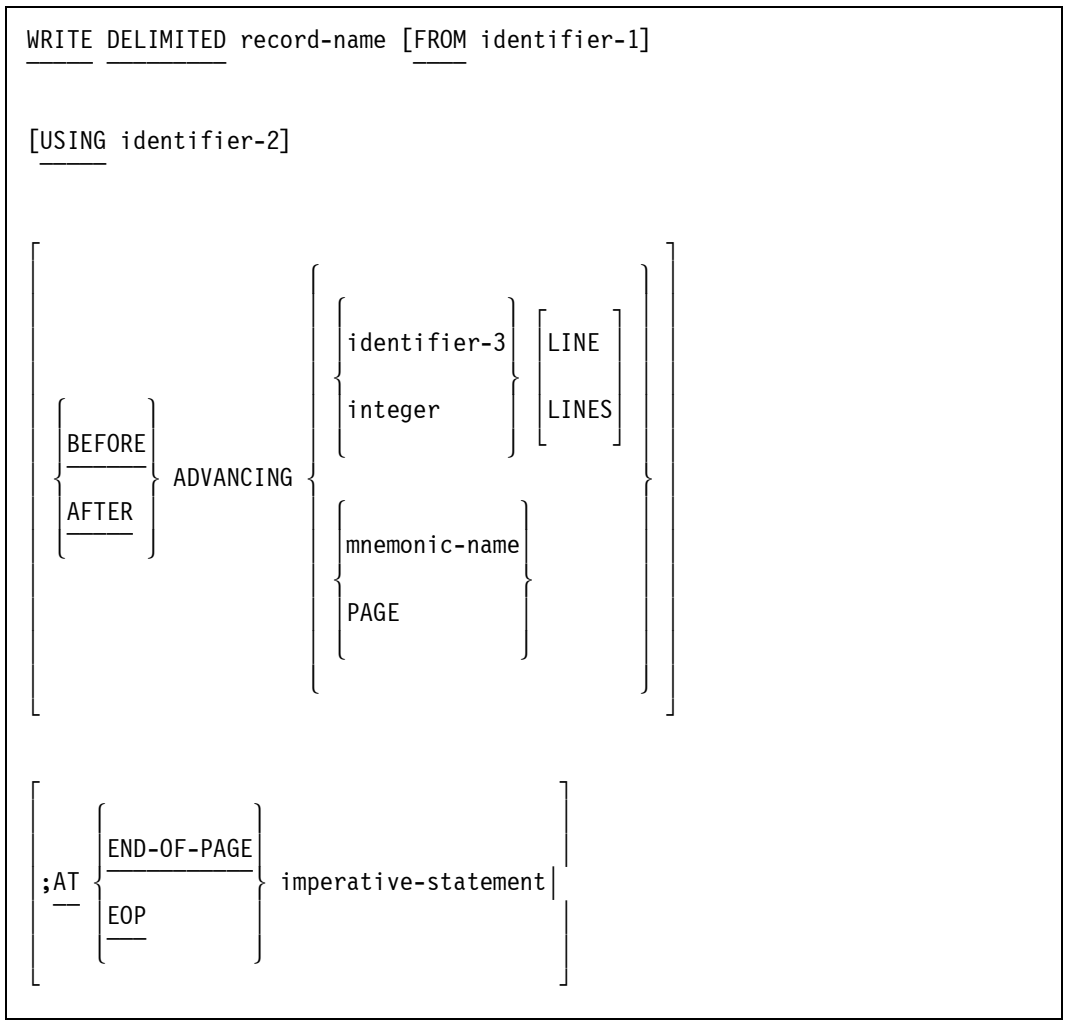
Formats 3 and 4: Kanji Delimiters

The WRITE DELIMITED statement formats a record into a scratch array by inserting start-of-double-octet (SDO) and end-of-double-octet (EDO) delimiters before and after each eligible Kanji item in the specified format record. The resulting scratch array is then transferred to the I/O subsystem.

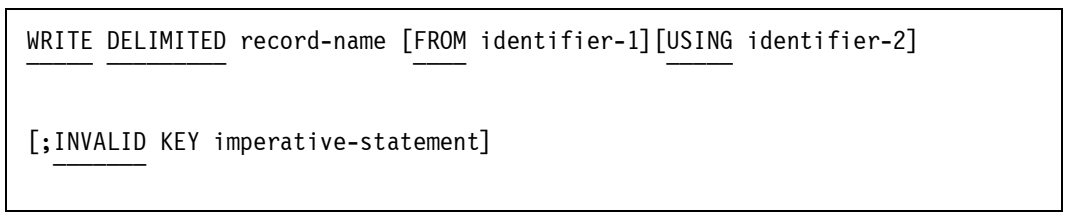
The hexadecimal values for the delimiter characters are as follows:

Delimiter	Value
SDO	2B
EDO	2C

Format 3



Format 4



Explanation of Formats 3 and 4

The records named in a WRITE DELIMITED statement are maintained according to the rules for a WRITE statement without the DELIMITED phrase.

The WRITE DELIMITED statement can involve the physical areas that match any of the four following descriptions:

- The target record is the record described in the file description (FD) whose name follows DELIMITED in the syntax.
- The source record is the record in the FROM phrase.
- The format record is the record in the USING phrase.
- The output record is the actual physical array used to contain the formatted result.

If neither the USING phrase nor the FROM phrase is specified, the target record is used as both the source record and the format record. If only the FROM phrase is used, the source record is first moved to the target record, and then the source record is used as the format record. If only the USING phrase is present, the target record is used as the source record, and the final formatting is performed using the format record.

The compiler uses the descriptions of the format record to transfer the data into the output record.

Eligible Kanji items are those elementary Kanji items that neither specify nor are subordinate to an item containing a REDEFINES or RENAMES clause.

Record-name and identifier-1 must not reference the same storage area.

The record-name is the name of a logical record in the FILE SECTION of the DATA DIVISION and can be qualified.

When a mnemonic-name is specified, it must be associated with a channel number. The mnemonic-name is defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

The integer and the value of the data item are referenced by identifier-2 and identifier-3, respectively.

Format 3 can be used only for remote or printer files.

Format 4 can be used only for remote files.

When the FROM phrase is used, identifier-1 and identifier-2 must not be the name of a RENAMES clause.

The *ADVANCING mnemonic-name* phrase cannot be specified when writing a record to a file that has a file-description entry (FD) containing the LINAGE clause.

The imperative-statement can be the NEXT SENTENCE phrase.

When an Invalid Key condition occurs, any FILE STATUS data item of the file is set to a value indicating the cause of the condition.

For more information on using the INVALID KEY and EOP phrases, refer to “Handling I/O Exceptions Conditions” in Section 8, “PROCEDURE DIVISION Concepts.”

Section 10
Segmentation

Segmentation was designed to enable you to manually divide the PROCEDURE DIVISION of a very large program into segments. The segments are stored on disk when there is not enough memory to store all the declared data items and the object program in main memory. The segments stored on disk are brought into main memory when program execution reaches that segment.

Actual COBOL74 Segmentation

The COBOL74 compiler automatically handles program segmentation. The compiler normally creates a segment at the beginning of each section or at the first paragraph encountered after 1500 words of code have been produced. The compiler does not change segments in the middle of a paragraph. Because the maximum segment size is 8192 words, you must ensure that no paragraph exceeds this limit.

If you use segment numbers in section headers, the compiler includes sections with the same segment number (other than 0) in one segment.

Standard COBOL74 Segmentation

Because many systems are incapable of automatic program-segment overlay, the standard COBOL segmentation facility is oriented toward a user-controlled grouping of sections into one of three types of segments. This segmentation is based on the value of an optional integer (or segment number) following the word SECTION in the section header.

The general format is as follows:

```
Section-name SECTION [segment-number].
```

The value of the segment-number can range from 0 through 9999, although the system allows only a physical maximum of 8192 code segments. If the segment-number is omitted from the section header, the value is assumed to be 0. Sections in the DECLARATIVES SECTION must declare segment-numbers less than 50. Sections with the same segment-numbers need not be adjacent to one another.

Segments are of three types, depending on the segment-number: fixed permanent, fixed overlayable, and independent. Segments with segment-numbers less than 50 belong to the fixed portion of the program, which is composed of two types of segments: fixed permanent segments and fixed overlayable segments. The SEGMENT-LIMIT clause in the OBJECT-COMPUTER paragraph is used to specify the cutoff point between fixed permanent and fixed overlayable segments.

The general format of this clause is as follows:

```
[, SEGMENT-LIMIT IS segment-number]
```

Explanation of Format

The distinction between fixed permanent and fixed overlayable is irrelevant in COBOL because both types of segments are always made available in their last-used state. Thus, the SEGMENT-LIMIT clause is ignored. (This is an extension to ANSI X3.23-1974 COBOL.)

The third type of segment, the independent segment, has segment-numbers ranging from 50 through 9999. An independent segment is made available in its initial state under the following circumstances:

- When control is transferred to the segment as a result of an implicit transfer of control between consecutive statements from a segment with a different segment-number
- When control is transferred to the segment as a result of an implicit transfer of control between a SORT or a MERGE statement in a segment with a different segment-number and an input or an output procedure in the independent segment
- When control is transferred explicitly to the segment with a GO or a PERFORM statement from a segment with a different segment-number

In all other cases, an independent segment is made available in its last-used state.

Although COBOL conforms to the preceding rules, the use of independent segments containing altered GO TO statements can be restricted for the following two reasons (extension to ANSI X3.23-1974 COBOL):

- Altered GO TO statements are handled using references located in the stack of the program. When an independent segment must be made available in the initial state, these references must be changed to the initial value.
- Most important, the control path of a program can be changed simply by changing a segment-number in a section header.

Section 11
Debugging

The debug module provides a means for monitoring data item values and program-control status during execution of an object program. You control the monitoring and display of information on the output device; the debug facility simply provides a convenient access to pertinent information.

The following features support the debug module:

- Compile-time switch WITH DEBUGGING MODE
- Object-time switch
- USE FOR DEBUGGING statement
- Special register DEBUG-ITEM
- Debugging lines

Compile-Time Switch

The WITH DEBUGGING MODE clause is specified in the SOURCE-COMPUTER paragraph and serves as a compile-time switch to enable the debugging statements written in the program.

When the WITH DEBUGGING MODE clause is specified, all debugging sections and debugging lines are compiled as executable procedures and statements. When the WITH DEBUGGING MODE clause is not specified, all debugging lines and debugging sections are compiled as comment lines.

If you are using a debugging line in a program and are compiling the program from CANDE, you must change the FREE compiler control option to FALSE. The FREE option is TRUE by default when compiling from CANDE. (This is an extension to ANSI X3.23-1974 COBOL.)

Object-Time Switch

The DEBUG option of the OPTION task attribute serves as an execution-time switch to enable or disable the debugging code enabled by the compile-time switch. The object program must be executed with OPTION=DEBUG to enable the debugging code. The statement *RUN USER/PROGRAM; OPTION=DEBUG* is an example of how to set the OPTION task attribute.

The debugging lines are not affected by the use of this execution-time switch. Only those items associated with the USE FOR DEBUGGING statement are affected.

ENVIRONMENT DIVISION in the Debug Module

The WITH DEBUGGING MODE clause is specified in the SOURCE-COMPUTER paragraph and activates all debugging sections and debugging lines.

The general format is as follows:

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

Explanation of Format

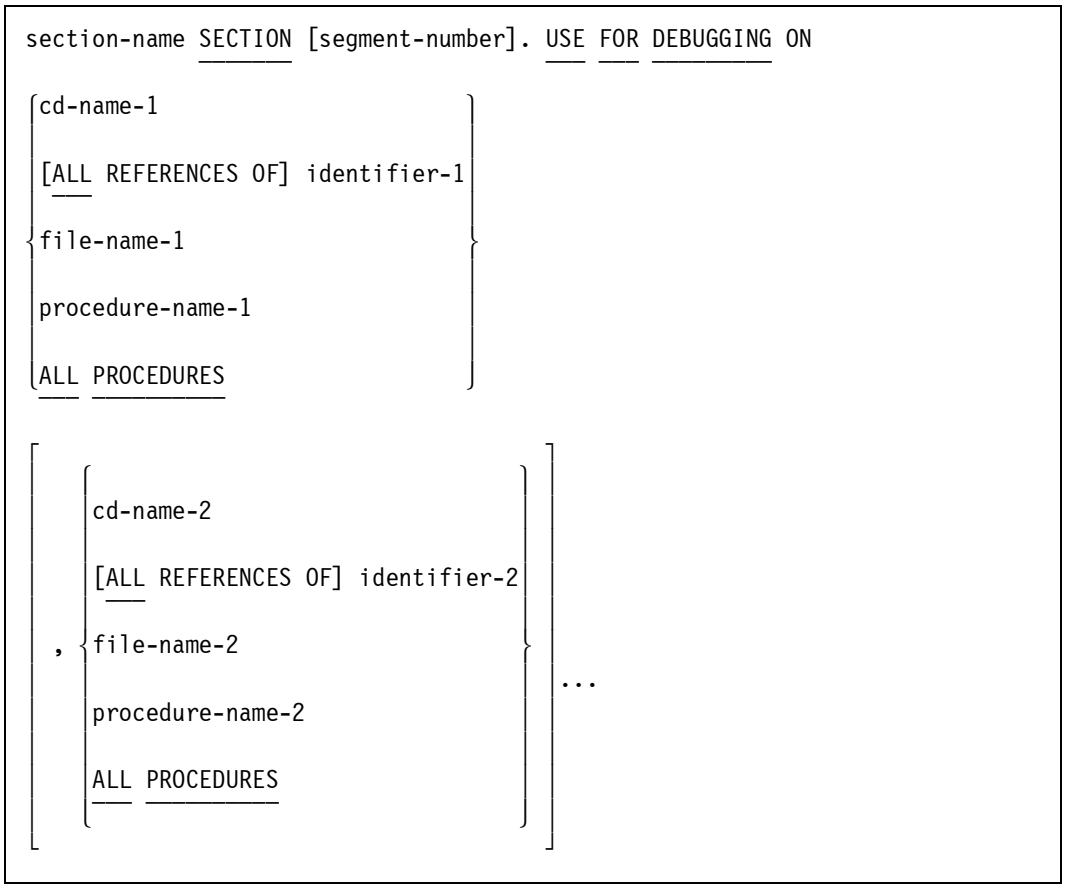
If the WITH DEBUGGING MODE clause is specified, all USE FOR DEBUGGING statements and debugging lines are compiled.

If the WITH DEBUGGING MODE clause is not specified, the compiler treats any USE FOR DEBUGGING statements or debugging lines and all associated debugging sections as comment lines.

PROCEDURE DIVISION in the Debug Module

The USE FOR DEBUGGING statement identifies the items to be monitored by the associated debugging section.

The general format is as follows:



Explanation of Format

Debugging sections, if specified, must appear together, immediately after the DECLARATIVES SECTION header.

Except in the USE FOR DEBUGGING statement itself, no reference can be made to any nondeclarative procedure in the debugging section.

Statements appearing outside the set of debugging sections must not reference procedure-names defined in the set of debugging sections.

Except for the USE FOR DEBUGGING statement itself, statements appearing in a given debugging section can reference procedure-names defined in a different USE procedure only by using a PERFORM statement.

Procedure-names defined in debugging sections must not appear in USE FOR DEBUGGING statements.

Any identifier, cd-name, file-name, or procedure-name can appear in only one USE FOR DEBUGGING statement and can 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, procedure-name-2, and so forth must not be specified in any USE FOR DEBUGGING statement.

Identifier-1, identifier-2, and so on must not refer to any data item defined in the REPORT SECTION except sum counters, and must not refer to any item in the DATA-BASE SECTION. Refer to Volume 2 for information about data management.

If the data-description entry of the data item referenced by identifier-1, identifier-2, and so forth contains an OCCURS clause or is subordinate to a data-description entry that contains an OCCURS clause, then identifier-1, identifier-2, and so forth must be specified without the subscripting or indexing normally required.

References to the special register DEBUG-ITEM are restricted to references in a debugging section.

General Rules

In the following general rules, all references to cd-name-1, identifier-1, procedure-name-1, and file-name-1 also apply to cd-name-2, identifier-2, procedure-name-2, and file-name-2, respectively.

Automatic execution of a debugging section is not caused by a statement appearing in a debugging section.

When file-name-1 is specified in a USE FOR DEBUGGING statement, that debugging section is executed at each of the following times:

- After execution of any OPEN or CLOSE statement that references file-name-1
- After execution of any READ statement (and after any other specified USE procedure) not resulting in execution of an associated AT END or INVALID KEY imperative-statement
- After execution of any DELETE or START statement that references file-name-1

When procedure-name-1 is specified in a USE FOR DEBUGGING statement, that debugging section is executed at each of the following times:

- Immediately before each execution of the named procedure
- Immediately after execution of an ALTER statement that references procedure-name-1

The ALL PROCEDURES phrase causes the effects described previously for procedure-name-1 to occur for every procedure-name in the program except those appearing in a debugging section.

When the *ALL REFERENCES OF identifier-1* phrase is specified, that debugging section is executed for every statement that explicitly references identifier-1 at each of the following times:

- In a WRITE or a REWRITE statement, immediately before execution of that WRITE or REWRITE statement and after execution of any implicit move resulting from the presence of the FROM phrase
- In a GO TO statement with a DEPENDING ON phrase, immediately before control is transferred and before the execution of any debugging section associated with the procedure-name to which control is to be transferred
- In a PERFORM statement in which a VARYING, an AFTER, or an UNTIL phrase references identifier-1, immediately after each initialization, modification, or evaluation of identifier-1
- In any other COBOL statement, immediately after execution of that statement

When identifier-1 is specified without the ALL REFERENCES OF phrase, that debugging section is executed at each of the following times:

- In a WRITE or a REWRITE statement that explicitly references identifier-1, immediately before execution of that WRITE or REWRITE statement and after execution of any implicit move resulting from the presence of the FROM phrase
- In a PERFORM statement in which a VARYING, an AFTER, or an UNTIL phrase references identifier-1, immediately after each initialization, modification, or evaluation of identifier-1
- In any other COBOL statement, immediately after execution, if it explicitly references the data item referenced by identifier-1 and causes the contents to be changed

If identifier-1 is specified in a phrase that is not executed or evaluated, the associated debugging section is not executed.

The associated debugging section is not executed for a specific operand more than once as a result of the execution of a single statement, regardless of the number of times that operand is explicitly specified. In a PERFORM statement that causes iterative execution of a referenced procedure, the associated debugging section is executed once for each iteration.

In an imperative-statement, each individual occurrence of an imperative verb identifies a separate statement for debugging purposes.

When cd-name-1 is specified in a USE FOR DEBUGGING statement, that debugging section is executed at each of the following times:

- After execution of any ENABLE, DISABLE, or SEND statement that references cd-name-1
- After execution of a RECEIVE statement referencing cd-name-1 that does not result in execution of the NO DATA imperative-statement
- After execution of an ACCEPT MESSAGE COUNT statement that references cd-name-1

A reference to file-name-1, identifier-1, procedure-name-1, or cd-name-1 as a qualifier does not constitute a reference to that item for debugging, as described in the preceding general rules.

DEBUG-ITEM Special Register

The reserved word DEBUG-ITEM is the name of a special register that is generated automatically. Only one DEBUG-ITEM reserved word is allocated for each program. The names of the subordinate data items in DEBUG-ITEM are also reserved words.

The special register DEBUG-ITEM is associated with each execution of a debugging section. This register provides information about the conditions that caused execution of a debugging section. The DEBUG-ITEM special register has the implicit description shown in Example 11–1.

Before each execution of a debugging section, the contents of the data item referenced by the reserved word DEBUG-ITEM are space-filled. The contents of data items subordinate to DEBUG-ITEM are then updated immediately before control is passed to that debugging section. The contents of any data item not specified in the following general rules remain space-filled.

Updating occurs according to the rules for the MOVE statement. The sole exception is the move to the DEBUG-CONTENTS item, which is treated as an alphanumeric-to-alphanumeric elementary move with no conversion of data from one form of internal representation to another.

```
01 DEBUG-ITEM.
    02 DEBUG-LINE     PIC X(6).
    02 FILLER         PIC X   VALUE SPACE.
    02 DEBUG-NAME     PIC X(30).
    02 FILLER         PIC X   VALUE SPACE.
    02 DEBUG-SUB-1    PIC S9999 SIGN IS LEADING SEPARATE CHARACTER.
    02 FILLER         PIC X   VALUE SPACE.
    02 DEBUG-SUB-2    PIC S9999 SIGN IS LEADING SEPARATE CHARACTER.
    02 FILLER         PIC X   VALUE SPACE.
    02 DEBUG-SUB-3    PIC S9999 SIGN IS LEADING SEPARATE CHARACTER.
    02 FILLER         PIC X   VALUE SPACE.
    02 DEBUG-CONTENTS PIC X(n).
```

Example 11–1. Implicit Description of DEBUG-ITEM Special Register

The DEBUG-LINE data item identifies the particular source statement that caused the debugging section to be executed. The DEBUG-LINE item contains the line number of the source image.

The DEBUG-NAME data item contains the first 30 characters of the name that caused the debugging section to be executed.

All qualifiers of the name are separated in the DEBUG-NAME item by the word OF. Subscripts or indexes, if any, are not entered into the DEBUG-NAME item.

If the reference to a data item that causes the debugging section to be executed is subscripted or indexed, the occurrence number of each level is entered in the DEBUG-SUB-1, DEBUG-SUB-2, and DEBUG-SUB-3 data items, as necessary.

If the data item is subscripted with more than three subscripts or indexes, only the occurrence numbers of the first three levels are entered into the DEBUG-ITEM item. (This is an extension to ANSI X3.23-1974 COBOL.)

The DEBUG-CONTENTS data item is large enough to contain the data required by the following general rules.

General Rules

If the first execution of the first nondeclarative procedure in the program causes the debugging section to be executed, the following conditions result:

- The DEBUG-LINE item identifies the first statement of that procedure.
- The DEBUG-NAME item contains the name of that procedure.
- The DEBUG-CONTENTS item contains the START PROGRAM statement.

If a reference to procedure-name-1 in an ALTER statement causes the debugging section to be executed, the following conditions result:

- The DEBUG-LINE item identifies the ALTER statement that references procedure-name-1.
- The DEBUG-NAME item contains procedure-name-1.
- The DEBUG-CONTENTS item contains the applicable procedure-name associated with the TO phrase of the ALTER statement.

If the transfer of control associated with the execution of a GO TO statement causes the debugging section to be executed, the following conditions result:

- The DEBUG-LINE item identifies the GO TO statement that transfers control to procedure-name-1 when executed.
- The DEBUG-NAME item contains procedure-name-1.

If the reference to procedure-name-1 in the INPUT or the OUTPUT phrase of a SORT or a MERGE statement causes the debugging section to be executed, the following conditions result:

- The DEBUG-LINE item identifies the SORT or the MERGE statement that references procedure-name-1.
- The DEBUG-NAME item contains procedure-name-1.
- The DEBUG-CONTENTS item contains the following:
  - The SORT INPUT statement if the reference to procedure-name-1 is in the INPUT phrase of a SORT statement
  - The SORT OUTPUT statement if the reference to procedure-name-1 is in the OUTPUT phrase of a SORT statement
  - The MERGE OUTPUT statement if the reference to procedure-name-1 is in the OUTPUT phrase of a MERGE statement

If the transfer of control from the control mechanism associated with a PERFORM statement causes the debugging section associated with procedure-name-1 to be executed, the following conditions result:

- The DEBUG-LINE item identifies the PERFORM statement that references procedure-name-1.
- The DEBUG-NAME item contains procedure-name-1.
- The DEBUG-CONTENTS item contains the PERFORM LOOP statement.

If procedure-name-1 is a USE procedure that is to be executed, the following conditions result:

- The DEBUG-LINE item identifies the statement that causes execution of the USE procedure.
- The DEBUG-NAME item contains procedure-name-1.
- The DEBUG-CONTENTS item contains the USE PROCEDURE statement.

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 result:

- The DEBUG-LINE item identifies the previous statement.
- The DEBUG-NAME item contains procedure-name-1.
- The DEBUG-CONTENTS item contains the FALL THROUGH statement.

If references to a file-name-1 or a cd-name-1 data item cause the debugging section to be executed, the following conditions result:

- The DEBUG-LINE item identifies the source statement that references a file-name-1 or a cd-name-1 data item.
- The DEBUG-NAME item contains the name of the file-name-1 or the cd-name-1 data item.
- For a READ statement, the DEBUG-CONTENTS item contains the entire record read.
- For all other references to file-name-1, the DEBUG-CONTENTS item contains spaces.
- For any reference to cd-name-1, the DEBUG-CONTENTS item contains the contents of the area associated with the cd-name.

If a reference to identifier-1 causes the debugging section to be executed, the following conditions result:

- The DEBUG-LINE item identifies the source statement that references identifier-1.
- The DEBUG-NAME item contains the name of identifier-1.
- The DEBUG-CONTENTS item contains the contents of identifier-1 at the time control passes to the debugging section.

Debugging Lines

A debugging line is any line with the letter D in the indicator area of the line. Any debugging line consisting solely of spaces from margin A to margin R is considered a blank line.

A debugging line is considered a comment line if the WITH DEBUGGING MODE clause is not specified in the SOURCE-COMPUTER paragraph. The contents of a debugging line must be such that a syntactically correct program is formed whether or not the debugging lines are considered comment lines.

Successive debugging lines are allowed.

A debugging line is permitted in the program only after the OBJECT-COMPUTER paragraph.

Debug Module Program Sample

Example 11–2 uses the debug module to produce the output shown in Example 11–1.

```
  IDENTIFICATION DIVISION.
  PROGRAM-ID.  DEBUGTEST.
*
*     This program is an example of the COBOL74 debug module.
*     Note that the WITH DEBUGGING MODE statement in the SOURCE-
*     COMPUTER paragraph instructs the compiler to compile debugging
*     statements into the object code. Setting OPTION = DEBUG
*     at run time enables the effects described in
*     the USE FOR DEBUGGING statement.

  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SOURCE-COMPUTER.  B7900 WITH DEBUGGING MODE.
  OBJECT-COMPUTER.  B7900.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
      SELECT PRINTFILE ASSIGN TO PRINTER.
D      SELECT DBGFILE ASSIGN TO PRINTER.
  DATA DIVISION.
  FILE SECTION.
  FD  PRINTFILE.
  01  PRINT-REC                   PIC X(132).
D  FD  DBGFILE.
D  01  DEBUG-REC                   PIC X(132).
  WORKING-STORAGE SECTION.
  77  ALFA                        PIC 9 VALUE 4.
  77  BRAVEO                      PIC 99 VALUE 16.
  77  CHARLIE                     PIC 99.
D  77  FLAG                        PIC 9.
D      88  NEED-IT                VALUE ZERO.
D      88  FILE-OPEN              VALUE 1.
  01  BARBARA.
      03  CHERYL.
          05  DIANNE              PIC 99 VALUE 16.
  77  ANN                         PIC 9(1) VALUE 8.
  77  ELVA                        PIC 9(18).
  77  FRAN    PIC X(32) VALUE "TEST PROGRAM FOR COBOL74 MANUAL".
  77  GOLDIE                      PIC 9 VALUE 1.
  01  HEIDI.
      03  ILENE                   PIC 9(8) OCCURS 5 TIMES.
      03  JANIS                   PIC 9.
*
  PROCEDURE DIVISION.
  DECLARATIVES.
```

Example 11–2. Debug Module Sample Program

```
  D-BUG SECTION.
       USE FOR DEBUGGING ON
          ALL REFERENCES OF CHARLIE
          ALL REFERENCES OF BRAVEO
          ALL PROCEDURES.
* RECORD DESCRIPTION FOR DEBUG-ITEM.
*
* 01  DEBUG-ITEM.
*     02  DEBUG-LINE              PIC X(6).
*     02  FILLER                  PIC X VALUE SPACE.
*     02  DEBUG-NAME              PIC X (30).
*     02  FILLER                  PIC X VALUE SPACE.
*     02 DEBUG-SUB-1              PIC S9(4) SIGN IS LEADING
*                                          SEPARATE CHARACTER.
*     02  FILLER                  PIC X VALUE SPACE.
*     02  DEBUG-SUB-2             PIC S9(4) SIGN IS LEADING
*                                          SEPARATE CHARACTER.
*     02  FILLER                  PIC X VALUE SPACE.
*     02  DEBUG-SUB-3             PIC S9(4) SIGN IS LEADING
*                                          SEPARATE CHARACTER.
*     02  FILLER                  PIC X VALUE SPACE.
*     02  DEBUG-CONTENTS          PIC X(n).
*
   DEBUG-OUT.
      IF  NEED-IT
          OPEN OUTPUT DBGFILE.
          MOVE 1 TO FLAG.
      WRITE DEBUG-REC FROM DEBUG-ITEM.
   END DECLARATIVES.
   WORK SECTION.
   START-IT.
       SUBTRACT ALFA FROM BRAVEO GIVING CHARLIE.
       DIVIDE BRAVEO BY CHARLIE GIVING ALFA.
       SUBTRACT ALFA FROM BRAVEO GIVING CHARLIE.
   FOLLOW-UP.
       SUBTRACT 4 FROM ANN.
       DIVIDE ANN INTO DIANNE GIVING JANIS.
       ACCEPT ILENE (GOLDIE) FROM TIME.
       OPEN OUTPUT PRINTFILE.
       WRITE PRINT-REC FROM FRAN.
       WRITE PRINT-REC FROM ILENE(GOLDIE).
       WRITE PRINT-REC FROM JANIS.
  CLOSE-SHOP.
      CLOSE PRINTFILE RELEASE.
D     IF FILE-OPEN
D         CLOSE DBGFILE RELEASE.
      STOP RUN.
```

Example 11–3 shows output from the debug module sample program when it is run with OPTION=DEBUG.

```
150000 WORK                                    START PROGRAM
150000 START-IT                                START PROGRAM
150000 CHARLIE                                 12
150000 BRAVEO                                  16
152000 CHARLIE                                 12
152000 BRAVEO                                  16
154000 CHARLIE                                 15
154000 BRAVEO                                  16
154000 FOLLOW-UP                               FALL THROUGH
168000 CLOSE-SHOP                              FALL THROUGH
```

Example 11–3. Debugging Output from Debug Module Sample Program

Section 12
Report Writer

Report Writer is a special-purpose language subset of COBOL that provides a convenient way to produce reports.

The file-description (FD) entry for Report Writer requires a REPORT clause to name the reports that are created. The other FD clauses that can be used are described in Section 7, "DATA DIVISION."

Most of the coding required for Report Writer is in describing the data. The REPORT SECTION of the DATA DIVISION defines the logical organization, format, contents, and structure of the report. A hierarchy of levels defines the logical organization. Each report consists of report groups; these groups are divided into sequences of items.

Using the report description (RD) in the REPORT SECTION, the compiler automatically generates PROCEDURE DIVISION code. This code moves data, constructs print lines, counts line and page numbers, produces heading and footing lines, sums information, and checks control breaks.

Example 12–1, which appears at the end of this section, shows a complete program that illustrates many of the features of Report Writer.

FILE SECTION REPORT Clause

In the FILE SECTION, Report Writer requires the REPORT clause to list the names of the reports to be produced.

The general format is as follows:

```
FILE SECTION.

                    ┌             ┐
                    │ REPORT IS   │
FD report-file      { ──────      } report-name-1 [, report-name-2]...
                    │ REPORTS ARE │
                    └ ─────── ────┘
```

Explanation of Format

report-file

Report-file identifies the file to which reports are written when a GENERATE statement is executed.

A report-file can be referenced only by the OPEN OUTPUT statement or by the CLOSE statement.

After the execution of an INITIATE statement and before the execution of a TERMINATE statement for the same report file, no other explicit I/O operation can reference that report file or the record-description (RD) entries associated with it. (This is an extension to ANSI X3.23-1974 COBOL.)

REPORT or REPORTS

The REPORT clause specifies the names of reports that make up a report file.

More than one report-name in a REPORT clause indicates that the report file contains more than one report. The maximum number of reports that can be associated with a given file is 255. A syntax error is issued if this limit is exceeded.

report-name

Each report-name specified in a REPORT clause must be the subject of a report-description (RD) entry in the REPORT SECTION. The order of appearance of the report-names is not significant. A report-name must appear in only one REPORT clause.

REPORT SECTION Report-Description Entry

The REPORT SECTION includes a report-description (RD) entry and one or more report group descriptions.

The RD entry describes the general organization of the report, including the layout of elements on a page. This entry is uniquely identified in the REPORT SECTION by the level indicator RD. The RD entry is analogous to the FD entry in the FILE SECTION.

The report group descriptions provide information about each part of the report and about sources of information for the report. The clauses that follow the report-name are optional, and the order of appearance is not significant. These clauses are discussed separately on the following pages.

You must use a period at the end of the report-description (RD) entry.

```
RD report-name

[;CODE literal-1]

[ ; { CONTROL IS   } { data-name-1 [, data-name-2]...                    } ]
  { CONTROLS ARE }

[ ;PAGE [ { LIMIT IS  } ] integer-1 [ { LINE  } ]                        ]
        [ { LIMITS ARE} ]             [ { LINES} ]
    [, HEADING integer-2]
    [, FIRST DETAIL integer-3]
    [, LAST DETAIL integer-4]
    [, FOOTING integer-5]                                                ]
```

Explanation of Format

RD

The level indicator RD identifies the beginning of a report description and must precede the report-name. The report-name must appear in one, and only one, REPORT clause.

report-name

Report-name is the highest permissible qualifier that can be specified for LINE-COUNTER, PAGE-COUNTER, and all data-names defined in the REPORT SECTION.

CODE Clause (Extension to ANSI X3.23-1974 COBOL)

The CODE clause places a nonnumeric value on each line of the report for identification purposes.

If more than one report is associated with a file and the reports are produced simultaneously, the CODE clause must be used so that the individual lines of each report can be identified by the operating system printer-backup routine.

```
CODE literal-1
____
```

Explanation of Format

CODE

When the CODE clause is specified, literal-1 is automatically placed in each record generated. The positions occupied by literal-1 are not included in the description of the print line, but are included in the size of a logical record.

If the CODE clause is specified for any report in a file, it must be specified for all reports in that file.

literal-1

Literal-1 must be a 2-character, nonnumeric literal.

Example

The following paragraphs detail the use of the CODE clause with WFL.

The backup printer file that is on disk, for the purpose of producing a report, has the following title when the CODE clause is specified:

```
BDREPORT/job-number/task-number/count-ID
```

The job-number is the mix number assigned to the job that created the report and is expressed as a 7-digit number. The task-number is the mix number assigned to the task that created the report and is expressed as a 7-digit number. The count is 001 for the first time the file is opened within the task, 002 for the second time, and so forth.

The ID is taken from the VALUE OF TITLE clause specified for the file. For example, if the job-number is 325 and the task-number is 327 and the VALUE OF TITLE clause specifies the value ABCD, then the file title assigned to a backup disk file is

```
BDREPORT/0000325/0000327/001ABCD
```

The second time the file is opened within that task, the file title is

```
BDREPORT/0000325/0000327/002ABCD
```

The backup disk files can be printed by using either of the following WFL statements:

```
PB D job-number KEY REPORT EQUAL literal-1

PB D * KEY REPORT EQUAL literal-1
```

Job-number is the 4-digit mix number of the job that created the report. The asterisk (*) indicates that the job-number to be used is the job-number of the WFL job itself. The asterisk function is useful a PB (Printer Backup) statement is included in a WFL statement that both creates and prints the report.

When the KEY clause is used in the WFL statement literal-1 must match literal-1 used in the CODE clause. For example:

```
PB D 0325 KEY REPORT EQUAL "A2"
```

The backup tape file of a Report Writer report for which a CODE clause was specified can be printed by using the following WFL statement:

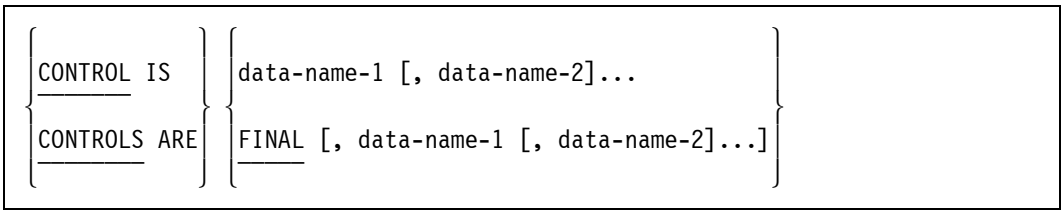
```
PB MT xxx [FILE integer] KEY REPORT EQUAL literal-1
```

In the preceding statement, *xxx* is the tape unit on which the backup tape resides, and literal-1 is used as shown in the backup disk file example.

When a printer file with an associated CODE clause is opened, the object code generated includes instructions to the operating system to set the BDNAME attribute of the program task to BDREPORT. In a program that contains multiple printer files—some of which contain CODE clauses and some of which do not—BDREPORT is used as the file title prefix for all files that are opened after a file with an associated CODE clause is opened, even if the remaining files do not have associated CODE clauses. If this is not the desired result and multiple printer files are required in the program, the printer files that do not have an associated CODE clause should be opened before the ones that do.

CONTROL Clause

The CONTROL clause specifies the names of the control data items, in order from the most significant to the least significant. The CONTROL clause is required when control headings, control footings, or both groups are used. The CONTROL clause establishes the level of the control hierarchy for the report.



Explanation of Format

The data-names specified in the CONTROL clause are the only data-names referred to by the RESET and TYPE clauses in the report-group descriptions for a report. No data-name, including the word FINAL, can be referenced by more than one CONTROL HEADING report group and one CONTROL FOOTING report group.

The data-names and the word FINAL specify the levels of the control hierarchy. FINAL, if specified, is the highest control; data-name-1 is the major control; data-name-2 is an intermediate control; and so forth. The last data-name specified is the minor control.

Data-name-1 and data-name-2 must not be defined in the REPORT SECTION. Data-name-1 and data-name-2 can be qualified but must not be subscripted or indexed.

Each data-name must identify a different data item.

Data-name-1, data-name-2, and so on must not have subordinate variable-occurrence data items. Control data items are subject to the same rules that apply to sort keys.

Causing a Control Break

A control break is a change in the value of a data item. The data item, which is referenced in the CONTROL clause, controls the hierarchical structure of the report.

The execution of the first chronological GENERATE statement for a report causes the values of all control data items associated with that report to be saved. On subsequent executions of all GENERATE statements for that report, Report Writer tests control data items for a change of value. A change of value in any control data item causes a control break to occur. The control break is associated with the highest level for which a change of value is noted.

Report Writer tests for a control break by comparing the contents of each control data item with the prior contents saved from the execution of the previous GENERATE statement for the same report. Report Writer applies the relation test as follows:

- If the control data item is a numeric data item, the program compares two numeric operands.
- If the control data item is an index data item, the program compares two index data items.
- If the control data item is a data item other than a numeric or an index data item, the program compares two nonnumeric operands.

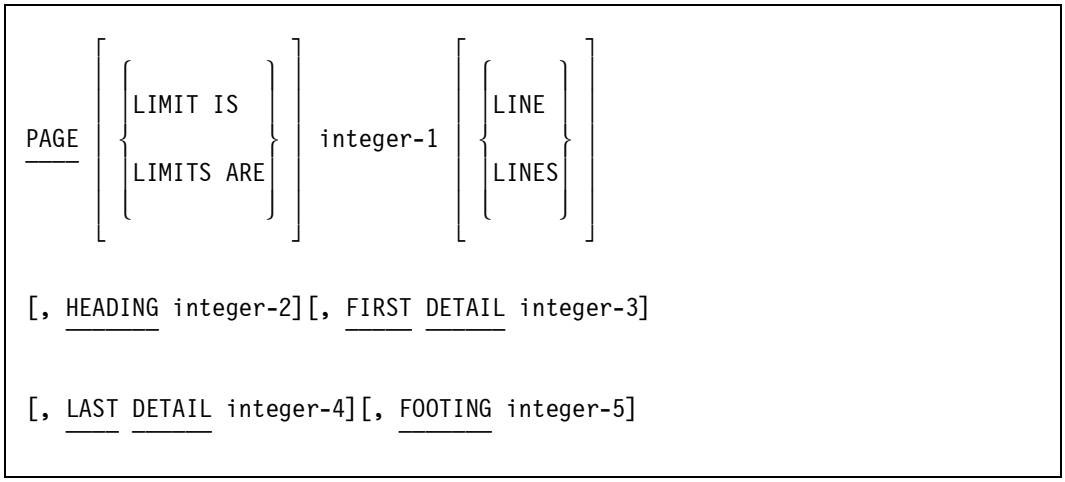
A control break for the word FINAL occurs before the first detail line is printed and whenever a TERMINATE statement is executed. A control break occurring at a particular level implies a control break for each lower level in the control hierarchy.

For example, when control headings and footings are coded, the CONTROL clause *CONTROLS ARE MAJ-KEY, INT-KEY, MIN-KEY* causes the control headings and footings to be printed in the following order on a control break on MAJ-KEY:

```
CONTROL FOOTING    (for MIN-KEY)
CONTROL FOOTING    (for INT-KEY)
CONTROL FOOTING    (for MAJ-KEY)
CONTROL HEADING    (for MAJ-KEY)
CONTROL HEADING    (for INT-KEY)
CONTROL HEADING    (for MIN-KEY)
```

PAGE Clause

The PAGE clause defines the length of a page and the vertical subdivisions within which report groups are presented. If the PAGE clause is omitted, the report consists of a single page of indefinite length.



Note: *The HEADING, FIRST DETAIL, LAST DETAIL, and FOOTING phrases can be written in any order.*

Explanation of Format

LIMIT IS integer-1 LINES

The words LIMIT IS or LIMITS ARE and LINE or LINES are optional and can be omitted.

Integer-1 defines the vertical length of a report page by specifying the number of lines available on each page.

Integer-1 must not be greater than 255. (This is an extension to ANSI X3.23-1974 COBOL.) Also, integer-1 must be greater than or equal to integer-5.

Absolute LINE NUMBER or absolute NEXT GROUP spacing must be consistent with controls specified in the PAGE LIMIT clause.

HEADING

A heading can be one of the following types:

A . . . heading	Appears at the beginning of the . . .
REPORT	Report only
PAGE	Page
CONTROL	Control group to which it belongs

The *HEADING integer-2* phrase defines the first line number on which a REPORT HEADING or PAGE HEADING report group can begin.

Integer-2 must be greater than or equal to 1. If the HEADING phrase is omitted, a value of 1 is assumed for integer-2.

The following rules indicate the vertical subdivision of the page in which each type of report group can appear when the PAGE clause is specified:

- If you want a REPORT HEADING report group to be on a page by itself, the program must define it to be within the vertical subdivision of the page that extends from the line number specified by integer-2 to the line number specified by integer-1, inclusive.
- If you want a REPORT HEADING report group that is not on a page by itself, the program must define it to be within the vertical subdivision of the page that extends from the line number specified by integer-2 to the line number specified by integer-3 minus 1, inclusive.
- If you want a PAGE HEADING report group, the program must define it to be in the vertical subdivision of the page that extends from the line number specified by integer-2 to the line number specified by integer-3 minus 1, inclusive.
- If you want a CONTROL HEADING or a DETAIL report group, the program must define it to be in the vertical subdivision of the page that extends from the line number specified by integer-3 to the line number specified by integer-4, inclusive.
- If the HEADING phrase is omitted, a value of 1 is assumed for integer-2.

FIRST DETAIL

The *FIRST DETAIL integer-3* phrase defines the first line number on which a body group can begin. REPORT HEADING and PAGE HEADING report groups cannot be on or beyond the line number specified by integer-3.

Integer-3 must be greater than or equal to integer-2.

If the FIRST DETAIL phrase is omitted, a value equal to integer-2 is given to integer-3.

LAST DETAIL

The *LAST DETAIL integer-4* phrase defines the last line number on which a CONTROL HEADING or a DETAIL report group can be presented.

Integer-4 must be greater than or equal to integer-3.

If the LAST DETAIL and FOOTING phrases are both omitted, the value of integer-1 is given to both integer-4 and integer-5.

If the LAST DETAIL phrase is specified and the FOOTING phrase is omitted, the value of integer-4 is given to integer-5.

If absolute line spacing is indicated for all report groups, integer-2 through integer-5 need not be specified. If relative line spacing is indicated for individual detail report group entries, some or all of the limits must be defined (depending on the type of report groups within the report) for control of page formatting to be maintained.

FOOTING

A footing can be one of the following types:

A . . . footing	Appears at the end of a . . .
REPORT	Report only
PAGE	Page
CONTROL	Control group to which it belongs

The *FOOTING integer-5* phrase defines the last line number on which a CONTROL FOOTING report group can be presented. PAGE FOOTING and REPORT FOOTING report groups must follow the line number specified by integer-5.

Integer-5 must be greater than or equal to integer-4.

If the FOOTING phrase is specified and the LAST DETAIL phrase is omitted, the value of integer-5 is given to integer-4.

The following rules indicate the vertical subdivision of the page in which each type of report group can appear when the PAGE clause is specified:

- If you want a CONTROL FOOTING report group, the program must define it to be in the vertical subdivision of the page that extends from the line number specified by integer-3 to the line number specified by integer-5, inclusive.
- If you want a PAGE FOOTING report group, the program must define it to be in the vertical subdivision of the page that extends from the line number specified by integer-5 plus 1 to the line number specified by integer-1, inclusive.
- If you want a REPORT FOOTING report group to be on a page by itself, the program must define it to be in the vertical subdivision of the page that extends from the line number specified by integer-2 to the line number specified by integer-1, inclusive.
- If you want a REPORT FOOTING report group that is not on a page by itself, the program must define it to be within the vertical subdivision of the page that extends from the line number specified by integer-5 plus 1 to the line number specified by integer-1, inclusive.

All report groups must be described so that they can be presented on one page. A multiline report group is never split across page boundaries.

General Rules

Figure 12–1 illustrates page format control of report groups when the PAGE LIMIT clause is specified.

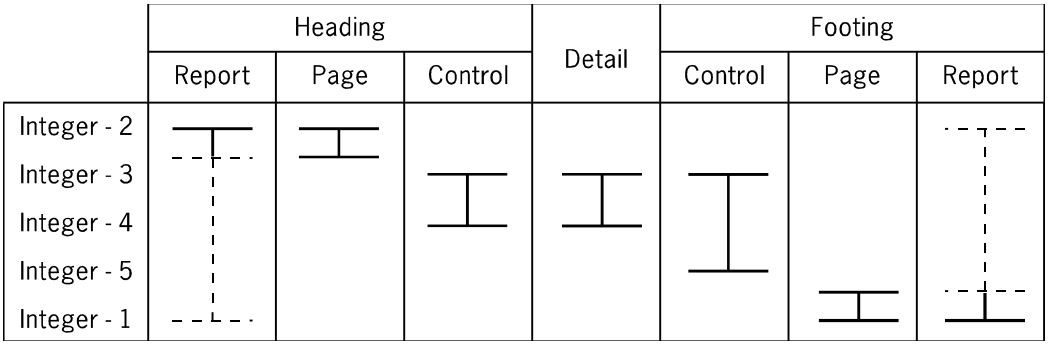


Figure 12–1. Page Format Control

The page regions established by the PAGE clause are shown in Table 12–1.

Table 12–1. Page Regions Established by the PAGE Clause

Report Groups That Can Be Presented in the Region	First Line Number of the Region	Last Line Number of the Region
REPORT HEADING described with *NEXT GROUP NEXT PAGE* phrase	integer-2	integer-1
REPORT FOOTING described with *LINE integer-1 NEXT PAGE* phrase	integer-2	integer-1
REPORT HEADING not described with *NEXT GROUP NEXT PAGE* phrase	integer-2	integer-3 minus 1
PAGE HEADING	integer-2	integer-3 minus 1
CONTROL HEADING	integer-2	integer-4
DETAIL	integer-2	integer-4
CONTROL FOOTING	integer-3	integer-5
PAGE FOOTING	integer-5 plus 1	integer-1
REPORT FOOTING not described with *LINE integer-1 NEXT PAGE* phrase	integer-5 plus 1	integer-1

Special Registers

The compiler automatically supplies the following two special registers for each report described in the REPORT SECTION:

- PAGE-COUNTER
- LINE-COUNTER

Because the special registers are automatically supplied, you do not have to code data-description entries for them.

PAGE-COUNTER

PAGE-COUNTER refers to a special register that the compiler automatically creates for each report specified in the REPORT SECTION. The purpose of PAGE-COUNTER is to provide consecutive page numbers for a report.

In the REPORT SECTION, a reference to PAGE-COUNTER can appear only in a SOURCE clause. Outside the REPORT SECTION, the special register PAGE-COUNTER can be used in any context in which a data-name of integral value can appear.

If more than one PAGE-COUNTER exists in a program, PAGE-COUNTER must be qualified by a report-name whenever it is referenced in the PROCEDURE DIVISION. In the REPORT SECTION, an unqualified reference to PAGE-COUNTER is implicitly qualified by the name of the report in which the reference is made; whenever the PAGE-COUNTER of a different report is referenced, PAGE-COUNTER must be explicitly qualified by that report-name.

Execution of an INITIATE statement resets to 1 the PAGE-COUNTER for the referenced report.

PAGE-COUNTER can be altered by PROCEDURE DIVISION statements. If you want a starting value other than 1, the program should change the contents of PAGE-COUNTER following the INITIATE statement for that report.

LINE-COUNTER

LINE-COUNTER refers to a special register that the compiler automatically creates for each report for which the PAGE LIMIT clause is specified. If more than one LINE-COUNTER exists in a program, then all references to LINE-COUNTER must be qualified. In the REPORT SECTION, an unqualified reference to LINE-COUNTER is implicitly qualified by the name of the report in which the reference is made.

In the REPORT SECTION, a reference to LINE-COUNTER can appear only in a SOURCE clause. Outside the REPORT SECTION, the special register LINE-COUNTER can be used in any context in which a data-name of integral value can appear. Changing the LINE-COUNTER by using PROCEDURE DIVISION statements can cause page-format control to become unpredictable.

Execution of an INITIATE statement resets to 0 (zero) the LINE-COUNTER for the referenced report. LINE-COUNTER is also reset to 0 each time a page is advanced for the associated report.

After a report group is printed, LINE-COUNTER contains the line number that corresponds to the last line of the report group that was printed, unless the report group designates the NEXT GROUP clause or the specific line number. In that case, LINE-COUNTER contains the value 0 (zero).

For further information on line number positioning, refer to "LINE NUMBER Clause" and "NEXT GROUP Clause" in this section.

REPORT SECTION Report-Group Descriptions

One or more report groups follow each report-description (RD) entry. Each group describes one or more print lines related to a specific function in producing a report. A report group is described by a hierarchic data structure similar to record descriptions in the other sections of the DATA DIVISION.

The report-group descriptions can appear only in the REPORT SECTION.

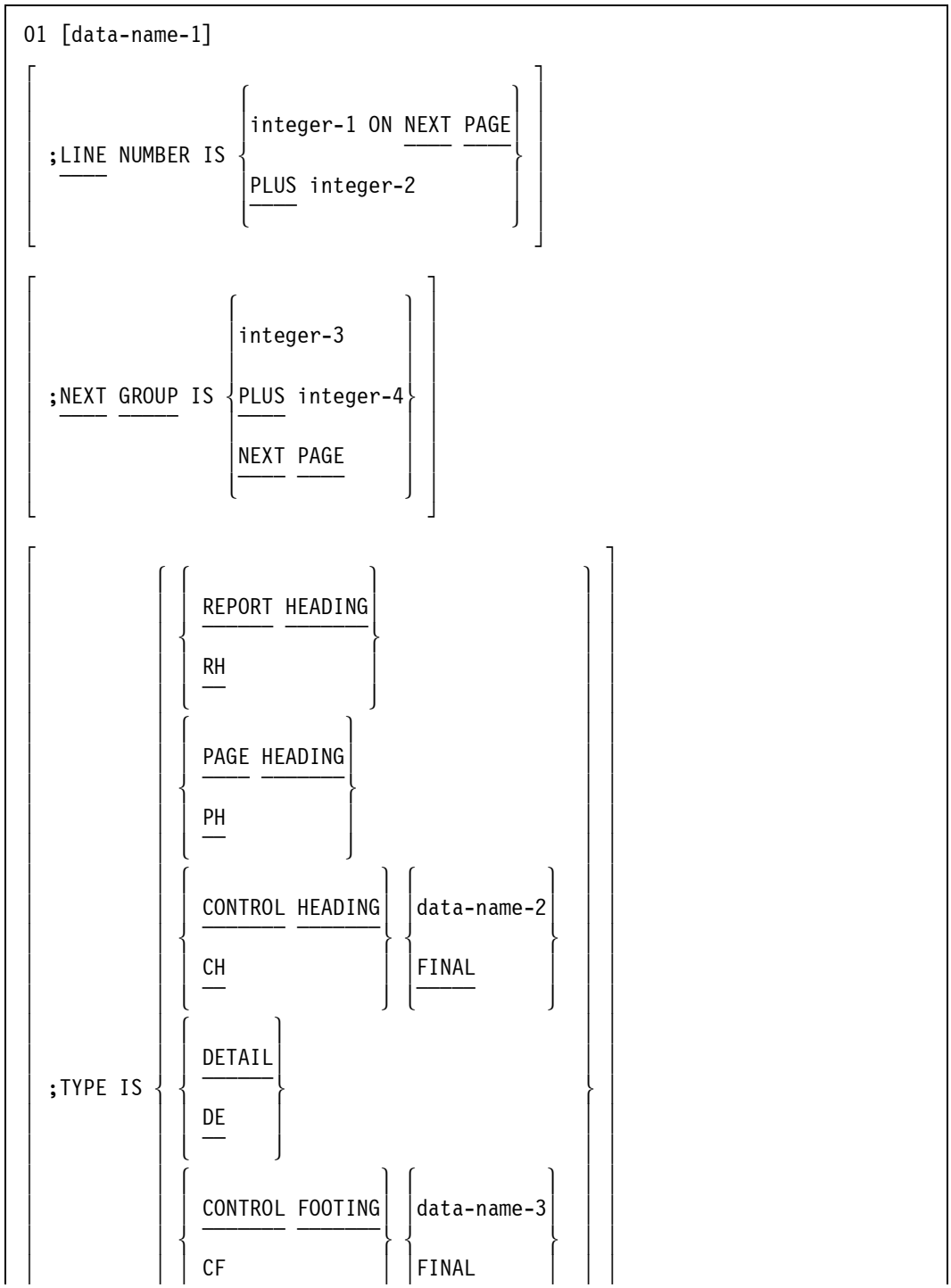
Except for the DATA-NAME clause (which, when present, must immediately follow the level-number), clauses can be written in any sequence.

The description of a report group can consist of one, two, or three hierarchic levels.

Format	Describes . . .
1	The 01-level of the report group.
2	A single line of the report group. This format must be followed immediately by Format 3 entries that describe the items to be printed on the line.
3	The items to be printed on a line, or describes a line that contains only one item to be printed.

Format 1: Report-Group Descriptions

Format 1 describes the 01-level of the report group. The first entry of a report group must be a Format 1 entry, which is as follows:



```
        ⎧PAGE FOOTING⎫
        ⎨            ⎬
        ⎩PF          ⎭

        ⎧REPORT FOOTING⎫
        ⎨              ⎬
        ⎩RF            ⎭

[;USAGE IS DISPLAY].
```

Explanation of Format 1

In this format, the integers must be greater than 0.

data-name-1

Data-name-1 of a Format 1 entry can be referenced only by a GENERATE statement, the UPON phrase of a SUM clause, a USE BEFORE REPORTING statement, or as a sum-counter qualifier. Data-name-1 is required in a Format 1 entry only in the following cases:

- When a DETAIL report group is referenced by a GENERATE statement
- When a DETAIL report group is referenced by the UPON phrase of a SUM clause
- When a report group is referenced in a USE BEFORE REPORTING sentence
- When the name of a CONTROL FOOTING report group is used to qualify a reference to a sum counter

LINE NUMBER Clause

The LINE NUMBER clause specifies information about the vertical positioning of the report group.

A LINE NUMBER clause must be specified to establish each print line of a report group.

Every entry that defines a printable item must either contain a LINE NUMBER clause or be subordinate to an entry that contains a LINE NUMBER clause.

An entry that contains a LINE NUMBER clause must not have a subordinate entry that also contains a LINE NUMBER clause.

Explanation of LINE NUMBER Clause Format

integer-1

Integer-1 specifies the absolute line number on which the print line is printed.

Integer-1 must not exceed 3 significant digits in length.

Integer-1 cannot be specified in a way that would cause any line of a report group to be presented outside the vertical page subdivision which is defined for that report-group type by the PAGE clause.

integer-2

Integer-2 specifies a relative line number. If a relative LINE NUMBER clause is specified, the line number on which the print line is printed is determined by the sum of the line number on which the previous print line of the report group was printed and integer-2 of the relative LINE NUMBER clause.

Integer-2 must not exceed 3 significant digits in length.

Integer-2 cannot be specified in a way that would cause any line of a report group to be presented outside the vertical page subdivision designated for that report group type as defined by the PAGE clause.

NEXT PAGE

The NEXT PAGE phrase specifies that the report group is to be presented beginning on the indicated line number on a new page.

A NEXT PAGE phrase can appear only once. If present, this phrase must be in the first LINE NUMBER clause. A LINE NUMBER clause with the NEXT PAGE phrase can appear only in the description of body groups and in a REPORT FOOTING report group.

General Rules

All absolute LINE NUMBER clauses must precede all relative LINE NUMBER clauses.

Successive absolute LINE NUMBER clauses must specify integers in ascending order. The integers need not be consecutive.

If the PAGE clause is omitted, only relative LINE NUMBER clauses can be specified in any report-group description entry in the report.

The vertical positioning specified by a LINE NUMBER clause occurs before the line established by that LINE NUMBER clause is printed.

The first LINE NUMBER clause specified within a PAGE FOOTING report group must be an absolute LINE NUMBER clause.

NEXT GROUP Clause

The NEXT GROUP clause specifies information for vertical positioning of a page following the presentation of the last line of a report group. A report-group description entry must not contain a NEXT GROUP clause unless the description of that report group contains at least one LINE NUMBER clause.

Integer-3 and integer-4 must not exceed the value 255. (This is an extension to ANSI X3.23-1974 COBOL.)

If the PAGE clause is omitted from the report-description entry, only a relative NEXT GROUP clause can be specified in any report-group description entry in that report.

The NEXT PAGE phrase of the NEXT GROUP clause must not be specified in a PAGE FOOTING report group.

The NEXT GROUP clause must not be specified in a REPORT FOOTING report group or in a PAGE HEADING report group.

General Rules

Integer-3 specifies the absolute line on which the next group is printed. The PLUS phrase designates the number of lines after the current line on which the next group is printed. The NEXT PAGE phrase prints the next group on the next page.

Any positioning of the page specified by the NEXT GROUP clause takes place after the report group in which the clause appears is printed.

The vertical positioning information supplied by the NEXT GROUP clause is interpreted along with information from the TYPE and PAGE clauses and the value in the LINE-COUNTER special register. This information is used to determine a new value for LINE-COUNTER.

The NEXT GROUP clause is ignored when it is specified on a CONTROL FOOTING report group that is at a level other than the highest level at which a control break is detected.

The NEXT GROUP clause of a body group refers to the next body group to be printed and, therefore, can affect the location at which the next body group is printed. The NEXT GROUP clause of a REPORT HEADING report group can affect the location at which the PAGE HEADING report group is printed. The NEXT GROUP clause of a PAGE FOOTING report group can affect the location at which the REPORT FOOTING report group is printed.

TYPE Clause

The TYPE clause specifies the particular type of report group described by this report-group description and indicates the time at which the report group is to be processed. This clause is a required entry in Format 1.

REPORT HEADING, PAGE HEADING, CONTROL HEADING FINAL, CONTROL FOOTING FINAL, PAGE FOOTING, and REPORT FOOTING report groups can appear no more than once in the description of a report.

Explanation of TYPE Clause Format

data-name-2, data-name-3, and FINAL

If present, these data items must be specified in the CONTROL clause of the corresponding report-description entry. At most, one CONTROL HEADING report group and one CONTROL FOOTING report group can be specified for each data-name or FINAL option in the CONTROL clause of the report-description entry. However, neither a CONTROL HEADING report group nor a CONTROL FOOTING report group is required for a data-name or a FINAL option specified in the CONTROL clause of the report-description entry.

In CONTROL FOOTING, PAGE HEADING, PAGE FOOTING, and REPORT FOOTING report groups, the SOURCE clauses and the USE statements must not reference any of the following:

- Group data items containing a control data item
- Data items subordinate to a control data item
- A redefinition or renaming of any part of a control data item

In PAGE HEADING and PAGE FOOTING report groups, the SOURCE clauses and the USE statements must not reference control data-names.

REPORT HEADING or RH

The REPORT HEADING phrase specifies a report group that is processed only once for each report as the first report group of that report. The REPORT HEADING report group is processed during the execution of the first chronological GENERATE statement for that report.

PAGE HEADING or PH

The PAGE HEADING phrase specifies a report group that is processed as the first report group on each page of that report, except under the following conditions:

- A PAGE HEADING report group is not processed on a page that is to contain only a REPORT HEADING report group or only a REPORT FOOTING report group.
- A PAGE HEADING report group is processed as the second report group on a page when it is preceded by a REPORT HEADING report group that is not to be printed on a page by itself.
- PAGE HEADING report groups can be specified only if a PAGE clause is specified in the corresponding report-description entry.

CONTROL HEADING or CH

The CONTROL HEADING phrase specifies a report group that is processed at the beginning of a control group for a designated control data-name or, in the case of the FINAL option, is processed during execution of the first chronological GENERATE statement for that report. If a control break is detected during the execution of any GENERATE statement, any CONTROL HEADING report groups associated with the highest level of the control break and lower levels are processed.

DETAIL or DE

The DETAIL report groups are processed as a direct result of GENERATE statements. If a report group is other than type DETAIL, processing is an automatic function.

The DETAIL phrase specifies a report group that is processed when a corresponding GENERATE statement is executed.

CONTROL FOOTING or CC

The CONTROL FOOTING phrase specifies a report group that is processed at the end of a control group for a designated control data-name. In the case of the FINAL option, the CONTROL FOOTING report group is processed only once for each report, as the last body group of that report. During execution of any GENERATE statement in which a control break is detected, any CONTROL FOOTING report group associated with the highest level of the control break or with more minor levels is printed. All CONTROL FOOTING report groups are printed during execution of the TERMINATE statement if at least one GENERATE statement has been executed for the report.

PAGE FOOTING or PF

The PAGE FOOTING phrase specifies a report group that is processed as the last report group on each page, except under the following conditions:

- A PAGE FOOTING report group is not processed on a page that is to contain only a REPORT HEADING report group or only a REPORT FOOTING report group.
- A PAGE FOOTING report group is processed as the second-to-last report group on a page when it is followed by a REPORT FOOTING report group that is not to be processed on a page by itself.
- PAGE FOOTING report groups can be specified only if a PAGE clause is specified in the corresponding report-description entry.

REPORT FOOTING or RF

The REPORT FOOTING phrase specifies a report group that is processed only once for each report as the last report group of that report. The REPORT FOOTING report group is processed during the execution of a corresponding TERMINATE statement if at least one GENERATE statement has been executed for the report.

USAGE Clause (Report Writer)

A USAGE clause specifies the format of a data item in computer storage.

The USAGE clause can be used at either the elementary level or 01-level; however, the USAGE clause of all report groups and their elementary items must be the same as the USAGE clause of the file on which the report is written.

For More Information

For a detailed description of the USAGE clause, refer to “USAGE Clause” in Section 7, “DATA DIVISION.”

Processing Report Groups

The sequence of steps executed when REPORT HEADING, PAGE HEADING, CONTROL HEADING, PAGE FOOTING, or REPORT FOOTING report groups are processed as discussed in the following list:

1. If a USE BEFORE REPORTING procedure is present that references the data-name of the report group, the USE procedure is executed.
2. If the report group is not printable, no further processing is done for the report group.
3. Otherwise, the print lines are formatted and printed according to the rules for the given type of report group.

Processing a CONTROL FOOTING Report Group

The sequence of steps executed when a CONTROL FOOTING report group is processed is described in the following list:

The GENERATE statement rules specify that when a control break occurs, the CONTROL FOOTING report groups, beginning at the minor level and proceeding upwards, are processed through the level at which the highest control break was detected. Although no CONTROL FOOTING report group has been defined for a given control data-name, step 5 in the following procedure is executed if a RESET phrase within the report description (RD) specifies that control data-name.

1. Sum counters are crossfooted; all sum counters defined in this report group that are operands of SUM clauses in the same report group are added to the sum counters.
2. Sum counters are rolled forward; that is, all sum counters defined in this report group that are operands of SUM clauses in higher-level CONTROL FOOTING report groups are added to the higher-level sum counters.
3. If a USE BEFORE REPORTING procedure references the data-name of the report group, the USE procedure is executed.
4. If the report group is not printable, step 5 is executed next; otherwise, the print lines are formatted and the report group is printed according to the rules for CONTROL FOOTING report groups.
5. Sum counters that are to be reset when this level is processed in the control hierarchy are reset.

Processing a DETAIL Report Group

The following five steps describe the detail-related processing that is executed in response to a *GENERATE report-name* statement when the description of the report includes exactly one DETAIL report group. These steps are performed as if a *GENERATE data-name* statement were being executed.

When the description of a report includes no DETAIL report groups, the detail-related processing in response to a *GENERATE report-name* statement is executed as described in step 1. This step is performed as if the description of the report included exactly one DETAIL report group and a *GENERATE data-name* statement were being executed.

1. Any subtotaling designated for the DETAIL report group is performed.
2. If a USE BEFORE REPORTING procedure refers to the data-name of the report group, the USE procedure is executed.
3. If the report group is not printable, no further processing is done for the report group.
4. If the DETAIL report group is processed as a consequence of a *GENERATE report-name* statement, no further processing is done for the report group.
5. If these conditions are not applicable, the print lines are formatted, and the report group is printed according to the rules for DETAIL report groups.

Processing After Printing a Body Group

When a CONTROL HEADING, a CONTROL FOOTING, or a DETAIL report is processed, interruption of a previously described processing of that body group might be necessary after determining that the body group is to be printed. A page advance is executed (and PAGE FOOTING and PAGE HEADING report groups are processed) before the body group is actually printed.

During control-break processing, the values of control data items used to detect a given control break are known as prior values. The following rules apply to prior values:

- During control-break processing of a CONTROL FOOTING report group, any references to control data items in a USE procedure or a SOURCE clause associated with that CONTROL FOOTING report group are supplied with prior values.
- When a TERMINATE statement is executed, the prior value for the control data item are made available to a SOURCE clause or USE procedure references in CONTROL FOOTING and REPORT FOOTING report groups as if a control break were detected in the highest control data-name.
- All other data item references in report groups and USE procedures access the current values contained in the data items when the report group is processed.

Format: 2 Report-Group Descriptions

A Format 2 entry describes a single line of the report group and must be followed immediately by Format 3 entries describing the printable items for the line.

```
level-number [data-name-1]

[                                          ]
[                  {                       }]
[ ;LINE NUMBER IS  {integer-1 [ON NEXT PAGE]}]
[                  {                       }]
[                  {PLUS integer-2         }]
[                  {                       }]
[                                          ]

[ ;[USAGE IS] DISPLAY].
```

Explanation of Format 2

The level-number in Format 2 can be any integer between 02 and 48, inclusive. Data-name-1 is optional. If present, it can be used only to qualify a sum-counter reference. A Format 2 entry must contain at least one of the optional clauses.

LINE NUMBER

The LINE NUMBER clause specifies vertical positioning information for its report group. A LINE NUMBER clause must be used to establish each print line of a report group.

USAGE (Report Writer)

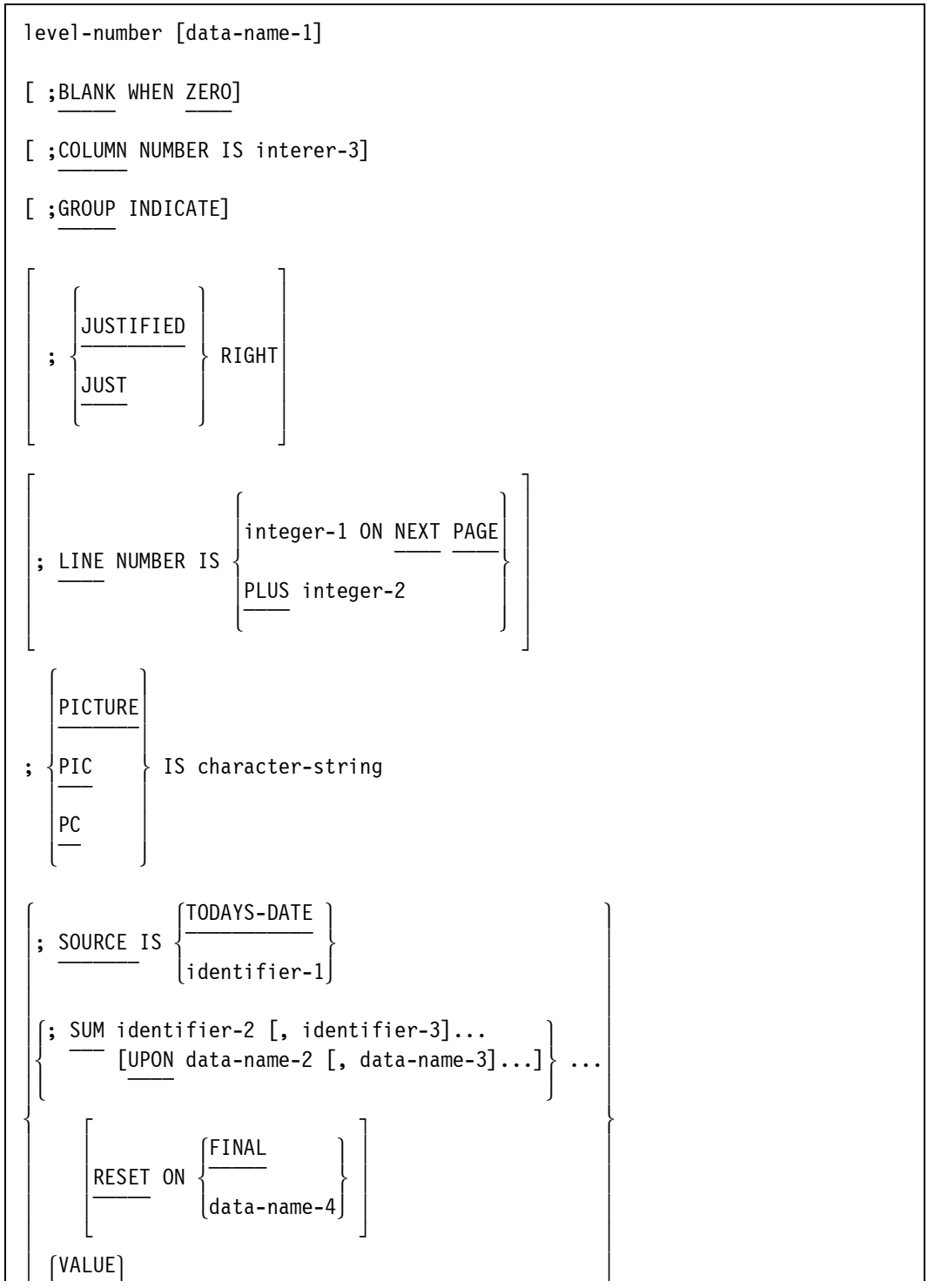
A USAGE clause specifies the format of a data item in computer storage. The USAGE clause can be used at the elementary item level. The usage of all elementary items must be the same as the usage of the file on which the report is written.

For More Information

- For details on the USAGE clause, see "USAGE Clause" in Section 7, "DATA DIVISION."
- For details about the LINE NUMBER clause, refer to "Format 1: Report-Group Descriptions" in this section.

Format 3: Report-Group Description

Format 3 entries, besides describing a single, printable item of a line, can also be used to describe a line that contains only one printable item.



```
[ ;{VALUE} IS literal-1                                    ]
[  {VA   }                                                 ]

[ ;[USAGE IS] DISPLAY].
```

Explanation of Format 3

In this format, the integers must be greater than 0.

level-number

The level-number in Format 3 can be any integer between 02 and 49, inclusive.

data-name-1

Data-name-1 is optional and can be referenced only if the entry defines a sum counter.

BLANK WHEN ZERO Clause

The BLANK WHEN ZERO clause controls printing of spaces when the value of a data item is 0 (zero). The rules for using this clause in Report Writer are the same as those described for the clause in Section 7, "DATA DIVISION."

COLUMN NUMBER Clause

The COLUMN NUMBER clause identifies an item to be printed and specifies the horizontal position of the item on the print line.

The absence of a COLUMN NUMBER clause indicates that the entry is not to be printed.

The COLUMN NUMBER clause indicates that the object of a SOURCE clause, the object of a VALUE clause, or the sum counter defined by a SUM clause is to be printed with the leftmost character position indicated by integer-3.

The COLUMN NUMBER clause can be specified only at the elementary level. When this clause is used, it must appear in, or be subordinate to, an entry that contains a LINE NUMBER clause.

Integer-3 specifies the leftmost character position of the printable item. It is important to ensure that integer-3 is greater than 0, that printable items on a given print line are defined in ascending column-number order and do not overlap, and that each character occupies a unique position.

Space characters are automatically provided for all positions of a print line that are not occupied by printable items.

The first or leftmost character of a print line is column number 1.

An entry that contains a COLUMN NUMBER clause but no LINE NUMBER clause must be subordinate to an entry that contains a LINE NUMBER clause.

An entry that contains a VALUE clause must also have a COLUMN NUMBER clause.

GROUP INDICATE Clause

The GROUP INDICATE clause causes an associated elementary item to be produced at the top of a page or when a control break occurs.

If a GROUP INDICATE clause is specified, it causes the SOURCE or VALUE clauses to be ignored and spaces to be provided except in the following cases:

- On the first printing of the DETAIL report group in the report
- On the first printing of the DETAIL report group after a page advance
- On the first printing of the DETAIL report group after every control break

If the report-description entry specifies neither a PAGE clause nor a CONTROL clause, then a GROUP INDICATE printable item is printed the first time the DETAIL report group is printed after the INITIATE statement is executed. Thereafter, spaces are supplied for indicated items with SOURCE or VALUE clauses.

The GROUP INDICATE clause can appear only in a DETAIL report group at the elementary item level within an entry that defines a printable item.

A GROUP INDICATE clause can appear only in a DETAIL report group.

JUSTIFIED Clause

The JUSTIFIED clause causes alphanumeric data to be right-justified in a receiving field. The rules for using the JUSTIFIED clause in Report Writer are the same as those described for the clause in Section 7, “DATA DIVISION.”

LINE NUMBER Clause

The LINE NUMBER clause specifies vertical positioning information for its report group. A LINE NUMBER clause must be used to establish each print line of a report group.

A LINE-NUMBER clause must not be the only clause specified.

Refer to the description of the LINE NUMBER clause in Format 1 earlier in this section for details about this clause.

PICTURE Clause

The PICTURE clause specifies the output format of an item and is required in Format 3.

The rules for using the PICTURE clause in Report Writer are the same as those described for the clause in Section 6, “Data Concepts” and in Section 7, “DATA DIVISION.”

SOURCE Clause

The SOURCE clause specifies the data item to be printed or designates a data item to be summed in a CONTROL FOOTING report group.

Identifier-1 specifies the sending data item of the implicit MOVE statement that is executed to move identifier-1 to the printable item. Identifier-1 must be defined so that it conforms to the rules for sending items in the MOVE statement. In an extension to ANSI X3.23-1974 COBOL, identifier-1 can be any special register, attribute, intrinsic function, or identifier.

The print lines of a report group are formatted immediately before presentation of the report group. At that time, the implicit MOVE statements specified by SOURCE clauses are executed.

Identifier-1 can be defined in any section of the DATA DIVISION. If identifier-1 is a REPORT SECTION item, it can be only PAGE-COUNTER, LINE-COUNTER, or a sum counter of the report in which the SOURCE clause appears.

SUM Clause

The SUM clause establishes a sum counter and names the data items to be summed.

A SUM clause can appear only in a CONTROL FOOTING report group.

Explanation of Sum Clause Format

identifier-2 and identifier-3

Identifier-2 and identifier-3 must be defined as numeric data items. When defined in the REPORT SECTION, identifier-2 and identifier-3 must be the names of sum counters. If the UPON phrase is omitted, any identifiers in the associated SUM clause that are themselves sum counters must be defined either in the same report group that contains the SUM clause or in a report group at a lower level in the control hierarchy of the report. If the UPON phrase is specified, any identifiers in the associated SUM clause must not be sum counters.

UPON

The UPON phrase selectively subtotals the DETAIL report groups named in the phrase.

Data-name-2 and data-name-3 must be the names of DETAIL report groups described in the same report as the CONTROL FOOTING report group in which the SUM clause appears. Data-name-2 and data-name-3 can be qualified by a report-name.

For more information on the CONTROL FOOTING report group, see “CONTROL Clause” earlier in this section.

RESET ON data-name-4

Data-name-4 must be one of the data-names specified in the CONTROL clause for this report. Data-name-4 must not be a lower-level control than the associated control for the report group in which the RESET phrase appears.

For more information on the CONTROL clause, see "CONTROL Clause" earlier in this section.

RESET ON FINAL

The FINAL option, if specified in the RESET phrase, must also appear in the CONTROL clause for the report.

VALUE Clause

The VALUE clause defines the value of REPORT SECTION printable items.

An entry that contains a VALUE clause must also have a COLUMN NUMBER clause.

Only Format 1 of the VALUE clause is permitted in the REPORT SECTION.

In an extension to ANSI X3.23-1974 COBOL, VA is an abbreviation for VALUE.

For a more detailed description of the VALUE clause, refer to "VALUE Clause" in Section 7, "DATA DIVISION."

USAGE Clause (Report Writer)

A USAGE clause specifies the format of a data item in computer storage.

The USAGE clause can be used at the elementary item level. The usage of all elementary items must be the same as the usage of the file on which the report is written.

For a detailed description of the USAGE clause, refer to "USAGE Clause" in Section 7, "DATA DIVISION."

Summary of RD Entries

Table 12–2 shows all the possible combinations of clauses in Format 3. Each column represents an allowable combination of clauses. For example, the first column under "Permitted Combinations of Clauses" indicates that when combining the required PICTURE and SUM clauses, the LINE clause is allowed but not required. All other remaining clauses are invalid for this combination. The PICTURE clause is always required.

Table 12–2. Permissible Combinations in Format 3 Report-Group Description Entries

Clause	Permitted Combinations of Clauses				
PICTURE	Required	Required	Required	Required	Required
COLUMN	Invalid	Required	Allowed	Allowed	Required
SOURCE	Invalid	Invalid	Required	Required	Invalid
SUM	Required	Required	Invalid	Invalid	Invalid
VALUE	Invalid	Invalid	Invalid	Invalid	Required
JUST	Invalid	Invalid	Allowed	Invalid	Allowed
BLANK ZERO	Invalid	Allowed	Invalid	Allowed	Invalid
GROUP	Invalid	Invalid	Allowed	Allowed	Allowed
USAGE	Invalid	Allowed	Allowed	Allowed	Allowed
LINE	Allowed	Allowed	Allowed	Allowed	Allowed

Understanding Sum Counters

The SUM clause establishes a sum counter. The sum counter is a numeric data item with an operational sign. At run time, each of the values identifier-1, identifier-2, and so forth is added directly into the sum counter. This addition is performed under the rules of the ADD statement.

The size of the sum counter is equal to the number of receiving character positions specified by the PICTURE clause that accompanies the SUM clause in the description of the elementary item.

Only one sum counter exists for an elementary report entry, regardless of the number of SUM clauses specified in the elementary report entry.

If the elementary report entry for a printable item contains a SUM clause, the sum counter serves as a source data item. The data contained in the sum counter is moved, according to the rules of the MOVE statement, to the printable item for printing.

If the data-name appears as the subject of an elementary report entry that contains a SUM clause, the data-name is the name of the sum counter, not the name of the printable item that the entry can also define.

The highest permissible qualifier of a sum counter is the report-name.

PROCEDURE DIVISION statements can alter the contents of sum counters.

If two or more identifiers specify the same addend, the addend is added into the sum counter as many times as the addend is referenced in the SUM clause.

Two or more of the data-names can specify the same DETAIL report group. When a *GENERATE data-name* statement for such a DETAIL report group is given, the incrementing occurs as many times as data-name appears in the UPON phrase.

In the absence of an explicit RESET phrase, the compiler sets the sum counter to the value 0 (zero) when Report Writer processes the CONTROL FOOTING report group within which the sum counter is defined. If an explicit RESET phrase is specified, the compiler sets the sum counter to 0 (zero) when Report Writer processes the designated level of the control hierarchy. The compiler initially sets the sum counter to 0 (zero) during execution of the INITIATE statement for the report containing the sum counter.

Incrementing Sum Counters

Report Writer adds the identifiers into the sum counters during execution of the GENERATE and TERMINATE statements. Report Writer adds each identifier to be added into the sum counter at a time that depends on the characteristics of the identifier. The characteristics of the identifiers to be added and the time of addition are described for each of the following three categories of sum-counter incrementing:

Category	Identifier Characteristics and Time of Addition
Subtotaling	Subtotaling is the process of accumulating identifiers to be added into a sum counter. Subtotaling occurs only during execution of GENERATE statements, after Report Writer processes any control break, but before it processes the DETAIL report group. If the SUM clause contains the UPON phrase, Report Writer subtotals the identifiers to be added when it executes a GENERATE statement for the designated DETAIL report group. If the SUM clause does not contain the UPON phrase, Report Writer subtotals the identifiers that are not sum counters when it executes any GENERATE data-name statement for the report in which the SUM clause appears.
Rolling Forward	Rolling forward is the process of accumulating sum counters defined in a lower-level CONTROL FOOTING report group into the sum counter. The program rolls forward a sum counter in a lower-level CONTROL FOOTING report group when a control break occurs and when the lower-level CONTROL FOOTING report group is processed.

Category	Identifier Characteristics and Time of Addition
Crossfooting	Crossfooting is the process of accumulating a sum counter defined in the same CONTROL FOOTING report group into the sum counter. Crossfooting occurs when a control break occurs and when the CONTROL FOOTING report group is processed. The program performs the crossfooting according to the sequence in which sum counters are defined in the CONTROL FOOTING report group; that is, the program completes all crossfooting into the first sum counter defined in the CONTROL FOOTING report group. Then the program completes all crossfooting into the second sum counter defined in the CONTROL FOOTING report group. The program repeats this procedure until it completes all the crossfooting operations.

PROCEDURE DIVISION Statements

The following statements apply in the PROCEDURE DIVISION only with Report Writer:

- INITIATE
- GENERATE
- TERMINATE
- USE BEFORE REPORTING

INITIATE Statement

The INITIATE statement begins processing of a report.

```
INITIATE report-name-1 [, report-name-2]...
```

Explanation of Format

INITIATE

The INITIATE statement resets all data-name entries that contain SUM clauses associated with the designated report.

The INITIATE statement does not open the file with which the report is associated; however, the associated file must be open when the INITIATE statement is executed.

A second INITIATE statement for a particular report-name cannot be executed unless a TERMINATE statement has been executed for that report-name after the execution of the first INITIATE statement.

report-name

Each report-name must be defined by a report-description (RD) entry in the REPORT SECTION of the DATA DIVISION.

Special Registers

The PAGE-COUNTER register, if specified, is set to 1 during execution of the INITIATE statement. If a starting value other than 1 is desired for the associated PAGE-COUNTER, the counter can be reset after the INITIATE statement is executed.

The LINE-COUNTER register, if specified, is set to 0 (zero) before or during execution of the INITIATE statement.

GENERATE Statement

This statement links the PROCEDURE DIVISION to the Report Writer at processing time. The Report Writer is described in the REPORT SECTION of the DATA DIVISION.

Data is moved to the data item in the report-group-description entry of the REPORT SECTION. This data is edited under the control of the Report Writer according to the same rules for movement and editing described for the MOVE statement.

GENERATE statements for a report are executed only after an INITIATE statement for the report is executed and before a TERMINATE statement for the report has been executed.

```
GENERATE identifier
```

Explanation of Format

The identifier represents a DETAIL report group or an RD entry.

If the identifier is the name of a DETAIL report group, the GENERATE statement performs all automatic operations of the Report Writer and produces an output DETAIL report on the output medium during processing. This type of reporting is called detail reporting.

If the identifier represents the name of a report-description (RD) entry, the GENERATE statement performs all automatic operations of the Report Writer and updates the FOOTING report groups in a particular report description (RD) without producing a DETAIL report group associated with the report. In this case, all sum counters associated with the report description (RD) are algebraically incremented. This type of reporting is called summary reporting. For summary reporting, no more than one TYPE DETAIL report group and at least one body group must be present, and the CONTROL clause must be specified for the report.

GENERATE Statement Actions

A GENERATE statement implicitly produces the following automatic operations (if defined) in both detail and summary reporting:

- Steps and tests the LINE-COUNTER register, the PAGE-COUNTER register, or both special registers to produce appropriate PAGE FOOTING, PAGE HEADING, or both report groups
- Recognizes any specified control breaks to produce appropriate CONTROL FOOTING report groups, CONTROL HEADING report groups, or both report groups
- Accumulates all specified identifiers into the sum counters, resets the sum counters on an associated control break, and performs an updating procedure between control-break levels for each set of sum counters
- Executes any specified routines defined by a USE statement before the generation of the associated report groups

Producing Report Groups

During execution of the first GENERATE statement, the following report groups associated with the report, if specified, are produced in the specified order:

- REPORT HEADING report group
- PAGE HEADING report group
- All CONTROL HEADING report groups in the following order: final, major, minor
- DETAIL report group statement

The following steps occur if the Report Writer recognizes a control break when the GENERATE statement is executed. The GENERATE statement cannot be the first statement executed for a report.

1. All CONTROL FOOTING report groups specified for the report are produced, from the minor report group up to and including the report group specified for the identifier that caused the control break.
2. The CONTROL HEADING report groups specified for the report are produced descending order, from the report group specified for the identifier that caused the control break down to the minor report group.
3. The DETAIL report group specified in the GENERATE statement is produced.

TERMINATE Statement

The TERMINATE statement ends the processing of a report.

```
TERMINATE report-name-1 [, report-name-2]...
```

Explanation of Format

Each report-name given in a TERMINATE statement must be defined by a report-description (RD) entry in the REPORT SECTION of the DATA DIVISION.

The TERMINATE statement produces all CONTROL FOOTING report groups associated with the report as if a control break had just occurred at the highest level, and the statement completes the Report Writer functions for the named reports. The TERMINATE statement also produces the last PAGE FOOTING and REPORT FOOTING report groups associated with the report.

If no GENERATE statements have been executed for a report during the interval between the execution of an INITIATE statement and a TERMINATE statement for the same report, then associated FOOTING report groups are not produced.

Appropriate PAGE HEADING report groups, PAGE FOOTING report groups, or both report groups are prepared in their respective order for the report group description.

A second TERMINATE statement for a particular report cannot be executed unless a second INITIATE statement has been executed for that report. If a TERMINATE statement has been executed for a report, a GENERATE statement for that report must not be executed unless an intervening INITIATE statement for that report is executed.

The TERMINATE statement does not close the file associated with the report. You must use a CLOSE statement for the file. However, the associated file must be open at the time the TERMINATE statement is executed. The TERMINATE statement performs Report Writer functions for individually described reports analogous to the I/O functions that the CLOSE statement performs for individually described files.

The SOURCE clauses used in the CONTROL FOOTING FINAL or REPORT FOOTING report groups refer to the values of the items during execution of the TERMINATE statement.

USE BEFORE REPORTING Statement

This statement specifies PROCEDURE DIVISION statements to be executed immediately before Report Writer processes a report group named in the REPORT SECTION of the DATA DIVISION.

```
USE BEFORE REPORTING identifier-1
```

Explanation of Format

A USE BEFORE REPORTING statement, when present, must immediately follow section header in the DECLARATIVES SECTION of the PROCEDURE DIVISION and must be followed by a period followed by a space. The remainder of the section must consist of one or more procedural paragraphs defining the procedures to be used.

Identifier-1 represents a report group named in the REPORT SECTION of the DATA DIVISION. An identifier must not appear in more than one USE statement.

No Report Writer statement (GENERATE, INITIATE, or TERMINATE) can be written in a procedural paragraph or paragraphs following the USE statement in the DECLARATIVES SECTION.

The USE statement itself is never executed; rather, it defines the conditions that call for execution of the USE procedures.

Report Writer executes the designated procedures immediately before it produces the named report group, regardless of control-break associations with report groups.

A USE BEFORE REPORTING statement must not alter the value of any control data item.

Report Writer Program Example

The sample program in Example 12–1 uses the Report Writer program to produce the report shown in Figure 12–3. The data file input to the program is shown in Figure 12–2.

Comment-entry lines in the program indicate the portions of code that describe the various report group types. The corresponding output is shown with numbers on the far right that indicate the report group type that caused the line to be printed.

```
000200 IDENTIFICATION DIVISION.
000300 PROGRAM-ID. FED-SCHOOL-SYSTEM.
000400 AUTHOR. BERKOWITZ.
000500 ENVIRONMENT DIVISION.
000600 INPUT-OUTPUT SECTION.
000700 FILE-CONTROL.
000800     SELECT PENNI ASSIGN TO SORT DISK.
000900     SELECT INFILE ASSIGN TO DISK.
001000     SELECT REPORTFILE ASSIGN TO PRINTER.
001100 DATA DIVISION.
001200 FILE SECTION.
001300 FD  INFILE BLOCK CONTAINS 30 RECORDS.
001350 01  IN-REC          PICTURE X(84).
001400 SD  PENNI.           01  FROMM.
001500    02  FILLER       PICTURE XX.
001600    02  STUDENT.
001700       03  NAME-L    PICTURE X(30).
001800       03  NAME-F    PICTURE X(10).
001900    02  FILLER       PICTURE XX.
002000    02  GRADE        PICTURE 99.
002100    02  FILLER       PICTURE XX.
002200    02  ROOM         PICTURE 999.
002300    02  FILLER       PICTURE 99.
002400    02 MONTHH        PICTURE 99.
002500    02  DAYY         PICTURE 99.
002600    02  YR           PICTURE 99.
002700    02  FILLER       PICTURE X(2).
002800    02  TAL          PICTURE 9.
002850    02  FILLER       PICTURE X(22).
002900 FD REPORTFILE    REPORT IS ABS-REPORT.
003000 WORKING-STORAGE SECTION.
003100 77  SAVED-MONTH     PICTURE 99 VALUE IS 0.
003200 77  CONTINUED       PICTURE X(11) VALUE IS SPACE.
003300 77  ABSS PIC X(8)  VALUE "ABSENCES".
003400 77  CA PIC X(19)   VALUE "CUMULATIVE ABSENCES".
003500 77  TAL-CTR  BINARY  PIC 9999.
003600 77  MTHIX           PICTURE 99.
003700 01  HEAD-1.
```

Example 12–1. Sample Report Writer Program

```
003800     02  FILLER  PIC X(22) VALUE SPACES.
003900     02 HEAD-LINE PIC X(74) VALUE "MONTH        DAY
004000-       "GRADE        ROOM                  NAME  ".
004100     02  FILLER  PIC X(36) VALUE SPACES.
004200 01  MONTH-TABLE.
004300     02  MONTH-1.
004400         03  FILLER  PICTURE A(9) VALUE IS "JANUARY  ".
004500         03  FILLER  PICTURE A(9) VALUE IS "FEBRUARY ".
004600         03  FILLER  PICTURE A(9) VALUE IS "MARCH    ".
004700         03  FILLER  PICTURE A(9) VALUE IS "APRIL    ".
004800         03  FILLER  PICTURE A(9) VALUE IS "MAY      ".
004900         03  FILLER  PICTURE A(9) VALUE IS "JUNE     ".
005000         03  FILLER  PICTURE A(9) VALUE IS "JULY     ".
005100         03  FILLER  PICTURE A(9) VALUE IS "AUGUST   ".
005200         03  FILLER  PICTURE A(9) VALUE IS "SEPTEMBER".
005300         03  FILLER  PICTURE A(9) VALUE IS "OCTOBER  ".
005400         03  FILLER  PICTURE A(9) VALUE IS "NOVEMBER ".
005500         03  FILLER  PICTURE A(9) VALUE IS "DECEMBER ".
005600         03  FILLER  PICTURE A(9) VALUE SPACES.
005700      02 MONTH-2 REDEFINES MONTH-1.
005800         03  MONTHNAME  PICTURE A(9) OCCURS 13 TIMES.
005900 REPORT SECTION.
006000 RD  ABS-REPORT   CONTROLS ARE FINAL, MONTHH, DAYY, GRADE
006100     PAGE LIMIT IS 56 LINES      HEADING 2
006200     FIRST DETAIL 10   LAST DETAIL 45     FOOTING 55.
006210*
006220* The following lines produce the report heading.
006230* See <--1 in sample Report Writer report.
006240*
006300 01 TYPE IS REPORT HEADING.
006400    02 LINE NUMBER IS 2   COLUMN 57       PIC X(17)
006500      VALUE "FED SCHOOL SYSTEM".
006510*
006520* The following lines produce the page heading.
006530* See <--2 in sample Report Writer report.
006540*
006600 01  PAGE-HEAD   TYPE IS PAGE HEADING.
006700     02  LINE NUMBER IS 3  COLUMN 52       PIC X(26)
006800        VALUE "STUDENT  ABSENTEEISM REPORT".
006900     02  LINE NUMBER IS 6.
007000        03  COLUMN IS 56    PIC X(9)
007100            SOURCE IS MONTHNAME OF MONTH-2(MONTHH)
007200        03  COLUMN IS 66    PIC X(8)       SOURCE IS ABSS.
007300        03  COLUMN IS 76    PIC X(11)      SOURCE IS CONTINUED.
007400     02  LINE IS 8.
007500        03  COLUMN IS 1     PIC X(132)     SOURCE HEAD-1.
007510*
```

Example 12–1. Sample Report Writer Program

```
007520* The following lines produce the detail lines.
007530* See <--3 in sample Report Writer report.
007540*
007600 01  DETAIL-LINE TYPE IS DETAIL LINE NUMBER IS PLUS 1.
007700     02  COLUMN IS 24 GROUP INDICATE PIC X(9)
007800         SOURCE IS MONTHNAME OF MONTH-2(MONTHH).
007900     02  COLUMN IS 41 GROUP INDICATE PICTURE IS 99
008000         SOURCE IS DAYY.
008100     02  COLUMN IS 54 GROUP INDICATE PIC  99 SOURCE IS GRADE.
008200     02  COLUMN IS 67 PIC 999 SOURCE IS ROOM.
008300     02  COLUMN IS 80 PIC X(20) SOURCE IS  NAME-L.
008400     02  COLUMN IS 101 PIC X(10) SOURCE IS NAME-F.
008410*
008420* The following lines produce the grade control footing.
008430* See <--4 in sample Report Writer report.
008440*
008500 01 TYPE IS CONTROL FOOTING GRADE.
008600     02  LINE NUMBER IS PLUS 2.
008700        03  COLUMN 1 PIC X(132) VALUE SPACE.
008710*
008720* The following lines produce the day control footing.
008730* See <--5 in sample Report Writer report.
008740*
008800 01  TESTER   TYPE IS CONTROL FOOTING DAYY.
008900     02  LINE NUMBER IS PLUS 2.
009000        03  COLUMN 2 PIC X(12)          VA "ABSENCES FOR".
009100        03  COLUMN 24  PICTURE Z9 SOURCE SAVED-MONTH.
009200        03  COLUMN 26  PICTURE X VALUE --.
009300        03  COLUMN 27  PICTURE 99 SOURCE DAYY.
009400        03  NO-ABS          COLUMN 49      PIC 999   SUM TAL.
009500        03  COLUMN 65 PIC X(19) SOURCE CA.
009600        03  COLUMN 85  PIC 999 SUM TAL RESET ON FINAL.
009700      02 LINE PLUS 1 COLUMN 1    PIC X(132)     VA ALL *.
009720* The following lines produce the month control footing.
009730* See <--6 in sample Report Writer report.
009740*
009800 01   TYPE CONTROL FOOTING MONTHH
009900      LINE PLUS 2 NEXT GROUP NEXT PAGE.
010000      02 COLUMN 16 PIC X(28) VALUE "TOTAL NUMBER OF ABSENCES FOR".
010100      02 COLUMN IS 46 PIC X(9)
010200         SOURCE MONTHNAME OF MONTH-2(SAVED-MONTH).
010300      02  COLUMN 57 PIC XXX VALUE "WAS".
010400      02  TOT      COLUMN 61 PIC 999        SUM NO-ABS.
010410*
010420* The following lines produce the page heading.
```

```
010430* See <--7 in sample Report Writer report.
010440*
```

Example 12–1. Sample Report Writer Program

```
010500 01   TYPE PAGE FOOTING LINE PLUS 1.
010600      02  COLUMN 59   PICTURE X(12) VALUE "REPORT-PAGE-".
010700      02  COLUMN 71   PICTURE 99 SOURCE PAGE-COUNTER.
010710*
010720* The following lines produce the report footing.
010730* See <--8 in sample Report Writer report.
010740*
010800 01   TYPE REPORT FOOTING.
010900      02  LINE PLUS 1 COLUMN 32 PICTURE A(13)
011000          VALUE "END OF REPORT".
011100 PROCEDURE DIVISION.
011200 DECLARATIVES.
011300 PAGE-HEAD-RTN SECTION.
011400     USE BEFORE REPORTING PAGE-HEAD.
011500 TEST-CONT.
011600     IF MONTHH = SAVED-MONTH MOVE "(CONTINUED)" TO CONTINUED
011700         ELSE MOVE SPACES TO CONTINUED
011800             MOVE MONTHH TO SAVED-MONTH.
011900 END DECLARATIVES.
012000 SORTING SECTION.
012100 SORTER.
012200     SORT PENNI ON ASCENDING KEY
012300     MONTHH, DAYY, GRADE, ROOM, STUDENT
012320     USING INFILE OUTPUT PROCEDURE IS REPORTER.
012330     DISPLAY MONTHH.
012340     MOVE MONTHH TO MTHIX.
012500 END-OF-THE-SORT.      STOP RUN.
012600 REPORTER SECTION.
012700 INITIATE-REPORT.
012900     OPEN OUTPUT REPORTFILE.
013000     INITIATE ABS-REPORT.
013100 UNWIND-THE-SORT.
013200     RETURN PENNI RECORD AT END
013300     TERMINATE ABS-REPORT STOP RUN.
013400     GENERATE DETAIL-LINE GO TO UNWIND-THE-SORT.
013460     STOP RUN.
```

Example 12–1. Sample Report Writer Program

Figure 12–2 shows the input data file (INFILE) that produces the output shown in Figure 12–3.

```
CODDINGTON                KIMBERLY    03  125  091288  1
MILLSTEIN                 SANDRA      03  121  091288  1
BURKLAND                  JOSEPH      03  121  091288  1
MCCOY                     JUDY        01  142  091088  1
LUBASCH                   DANIEL      01  142  091088  1
JOFFEE                    JOHN        01  142  091088  1
EAGLE                     MIKE        05  153  090788  1
DANIELSON                 FRED        05  153  090788  1
HUBERT                    THOMAS      03  115  090788  1
WONG                      SUSIE       03  111  090788  1
CODDINGTON                DARIN       02  103  090788  1
CARROLL                   JENNIFER    02  102  090788  1
HANSON                    KAREN       02  102  090788  1
AUSTIN                    EUGENE      02  101  090788  1
```

Figure 12–2. Input Data File to Sample Report Writer Program

Figure 12–3 shows output from the sample Report Writer program. The numbers on the far right side (for example, <–1) indicate the report group type that caused the line to be printed. These numbers shown on the output correspond to the numbers shown in the comment-entry lines of the program.

```
                    FED SCHOOL SYSTEM                          <-- 1
                  STUDENT ABSENTEEISM REPORT                   <-- 2


                     SEPTEMBER ABSENCES                        <-- 2

MONTH        DAY   GRADE    ROOM            NAME               <-- 2

SEPTEMBER    07    02       101    AUSTIN           EUGENE     <-- 3
                            102    CARROLL          JENNIFER   <-- 3
                            102    HANSON           KAREN      <-- 3
                            103    CODDINGTON       DARIN      <-- 3

                                                               <-- 4
SEPTEMBER    07    03       111    WONG             SUSIE      <-- 3
                            115    HUBERT           THOMAS     <-- 3
                                                               <-- 4
SEPTEMBER    07    05       153    DANIELSON        FRED       <-- 3
                            153    EAGLE            MIKE       <-- 3
                                                               <-- 4
                                                               <-- 5
ABSENCES FOR   9-07    008     CUMULATIVE ABSENCES 008         <-- 5
************************************************************ <-- 5
```

Figure 12–3. Sample Report Writer Report

```
SEPTEMBER    10    01       142    JOFFEE            JOHN      <-- 3
                             142    LUBASCH           DANIEL    <-- 3
                             142    MCCOY             JUDY      <-- 3
                                                              <-- 4
                                                              <-- 5
 ABSENCES FOR   9-10    003     CUMULATIVE ABSENCES 011         <-- 5
 ************************************************************ <-- 5
 SEPTEMBER    12    03       121    BURKLAND          JOSEPH    <-- 3
                             121    MILLSTEIN         SANDRA    <-- 3
                             125    CODDINGTON        KIMBERLY  <-- 3
                                                              <-- 4
                                                              <-- 5
 ABSENCES FOR   9-12    003     CUMULATIVE ABSENCES 014         <-- 5
 ************************************************************ <-- 5
                                                              <-- 6
          TOTAL NUMBER OF ABSENCES FOR SEPTEMBER WAS 014       <-- 6


                       REPORT-PAGE-01                         <-- 7
                       END OF REPORT                          <-- 8
```

Figure 12–3. Sample Report Writer Report

Section 13
ANSI Inter-Program Communication (IPC)

The Inter-Program Communication (IPC) module provides a way for a program to communicate with one or more other programs. This communication is defined as

- The ability to transfer control from one program to another within a run unit
- The ability of both programs to have access to the same data items

ANSI-74 IPC is a subset of libraries, which are described in Section 15, "Libraries." The IPC implementation requires that a called program adhere to the following rules:

- The SHARING compiler control option has the value SHAREDBYRUNUNIT.
- The program does not return a value.
- The program has a single entry point, named PROCEDUREDIVISION.

For More Information

Deviations from these rules are extensions to ANSI X3.23-1974 COBOL, which are described in Section 15, "Libraries."

LINKAGE SECTION in the IPC Module

You must code a LINKAGE SECTION in the DATA DIVISION when using IPC. The LINKAGE SECTION appears in a called program. It describes data that is available through the calling program but that is accessed in both the calling and the called programs. No space is allocated in the called program for data items referred to by data-names in the LINKAGE SECTION of that program. PROCEDURE DIVISION references to these data items are resolved at run 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 in the called and calling programs always refer to separate indexes.

The LINKAGE SECTION in a program is meaningful only if the object program functions under the control of a CALL statement and if the CALL statement in the calling program contains a USING phrase.

Data items defined in the LINKAGE SECTION of the called program can be accessed within the PROCEDURE DIVISION of the called program only if they are specified as operands of the USING phrase of the PROCEDURE DIVISION header or if they are subordinate to such operands and the object program is under the control of a CALL statement that specifies a USING phrase.

The structure of the LINKAGE SECTION is the same as that previously described for the WORKING-STORAGE SECTION, beginning with a section header and followed by data-description entries for noncontiguous data items, record-description entries, or both.

Each LINKAGE SECTION record-name and noncontiguous item name must be unique within the called program because it cannot be qualified. Data items defined in the LINKAGE SECTION of the called program must not be associated with data items defined in the REPORT SECTION of the calling program.

Of those items defined in the LINKAGE SECTION, the only items that can be referred to in the PROCEDURE DIVISION are the following:

- Data-name-1, data-name-2, and so on, in the USING phrase of the PROCEDURE DIVISION header
- Data items subordinate to these data-names
- Condition-names, index-names, or both associated with data-names, subordinate data items, or both

In both calling and called programs, the following rules apply to data items used by the operating system as parameters for parameter-matching:

- 01-level items of DISPLAY, COMP, or INDEX usages, and 77-level items with USAGE IS DISPLAY are treated as EBCDIC arrays.
- 77-level COMP or INDEX items are treated as hex arrays.

For More Information

See Section 15, “Libraries,” for details on using data items of other usages as parameters.

Noncontiguous Linkage Storage

Items in the LINKAGE SECTION that have no hierarchic relationship to one another need not be grouped into records. Instead, these items are classified and defined as noncontiguous elementary items. Each item is defined in a separate data-description entry that begins with the special level-number 77.

The following data clauses are required in each data-description entry:

- Level-number 77
- DATA-NAME
- PICTURE clause or USAGE IS INDEX clause

Other data-description clauses are optional and can complete the description of the item, if necessary.

Linkage Records

Data elements in the LINKAGE SECTION that have a definite hierarchic relationship to one another must be grouped into records according to the rules for formation of record descriptions. Any clause used in an input or an output record description can be used in a LINKAGE SECTION.

Usages other than DISPLAY, COMP, or INDEX are extensions to ANSI X3.23-1974 COBOL and their use as parameters is described in Section 15, "Libraries." The WITH LOWER-BOUNDS clause, also an extension to ANSI X3.23-1974 COBOL, is ignored for parameter-linking purposes in all IPC and library interfaces.

The VALUE clause must not be specified in the LINKAGE SECTION except in condition-name entries (88-level).

PROCEDURE DIVISION in the IPC Module

The following constructs apply to the PROCEDURE DIVISION when IPC is used.

- PROCEDURE DIVISION Header
- CALL statement
- CANCEL statement
- EXIT PROGRAM statement
- STOP RUN statement

PROCEDURE DIVISION Header

The PROCEDURE DIVISION is identified by, and must begin with, the following header:

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

The USING phrase is present if the object program is to function under the control of a CALL statement and the CALL statement in the calling program contains a USING phrase. Each operand in the USING phrase of the PROCEDURE DIVISION header must be defined as a data item in the LINKAGE SECTION of the program in which this header occurs, and each operand must have a level-number of 01 or 77.

Within a called program, LINKAGE SECTION data items are processed according to the data descriptions given in the called program.

Inter-Program Communication (IPC) implementation is not allowed if a data item is declared RECEIVED BY CONTENT. The RECEIVED BY CONTENT declaration on a parameter is supported only for tasking and bound procedures.

CALL Statement

The execution of a CALL statement causes control to pass from a calling program to a called program. The calling program is the program in which the CALL statement appears. The called program is the program specified by the value of the identifier or the literal in the CALL statement.

```
      ┌               ┐
      │identifier-1   │
CALL  │               │  [USING data-name-1 [, data-name-2] ...]
      │literal-1      │
      └               ┘

[;ON OVERFLOW imperative-statement]
```

Note: *The following format instructions and general rules assume that the program being called is a COBOL74 program with the value of the SHARING compiler control option equal to SHAREDBYRUNUNIT (the default). This restriction is required for adherence to ANSI-74 COBOL Inter-Program Communication (IPC) conventions. For the effects of a CALL verb on programs that do not meet the preceding criteria, see Section 15, "Libraries."*

Explanation of Format

The literal must be a nonnumeric literal.

The identifier must be defined as an alphanumeric data item whose value can be a program-name.

The USING option is included in the CALL statement only if a USING phrase is present in the PROCEDURE DIVISION header of the called program. The number of operands in each USING phrase must be identical.

The data-names indicate the data items available to a calling program that can be accessed in the called program. A data-name must be defined as a data item in the FILE, WORKING-STORAGE, COMMUNICATION, or LINKAGE SECTION; must have a level-number of 01 or 77; and must not redefine another data item. The data-name can be qualified when it refers to a data item defined in the FILE SECTION or COMMUNICATION SECTION.

The order of appearance of the data-names in the USING phrase of the CALL statement and the USING phrase in the PROCEDURE DIVISION header is critical. Corresponding data-names refer to a single set of data available to the called and calling programs. The correspondence is by position, not by name. In index-names, no such correspondence is established; index-names in the called and calling programs always refer to separate indexes.

A parameter in a USING clause cannot redefine another item either implicitly in the FILE SECTION or explicitly with a REDEFINES clause. Also, 77-level items that have been redefined can yield unexpected results. For example, referring to a 77-level item in a REDEFINES clause can cause it to be treated as if it had been declared as a 01-level item.

The ON OVERFLOW option performs the imperative-statement if a syntax error occurs.

General Rules

A called program is in the initial state the first time it is called within a run unit and the first time it is called after a CANCEL statement is sent to the called program.

Called programs can contain CALL statements. However, a called program must not contain a CALL statement that directly or indirectly calls the calling program.

If you are using the Segmentation module, the CALL statement can appear anywhere within a segmented program. The implementor must provide all controls necessary to ensure that the proper logic flow is maintained. Therefore, when a CALL statement appears in a section with a segment-number greater than or equal to 50, that segment is in the last-used state when the EXIT PROGRAM statement returns control to the calling program.

CANCEL Statement

The CANCEL statement releases the memory areas occupied by the called program.

```
        ⎧identifier-1⎫   [   ⎧identifier-2⎫   ]
CANCEL  ⎨            ⎬   [ , ⎨            ⎬   ] ...
        ⎩literal-1   ⎭   [   ⎩literal-2   ⎭   ]
```

Explanation of Format

The literal must be a nonnumeric literal.

The identifier must be an alphanumeric data item.

General Rules

Note: *The following rules assume that the program being called is a COBOL74 program with the value of the SHARING compiler control option equal to SHAREDBYRUNUNIT (the default). This restriction is required for adherence to ANSI-74 COBOL Inter-Program Communication (IPC) conventions. For the effects of a CANCEL verb on programs that do not meet the preceding criteria, see Section 15, "Libraries."*

After execution of a CANCEL statement, the called program ceases to have any logical relationship to the run unit in which the CANCEL statement appears. A CALL statement naming the same program that is subsequently executed results in initiation of that program in its initial state. The memory areas associated with the named program are released so as to be made available for disposition by the operating system.

A program named in the CANCEL statement must not refer to any program that has been called and has not yet executed an EXIT PROGRAM statement.

A logical relationship to a canceled program is established only by execution of a subsequent CALL statement.

A called program is canceled if it is used as the operand of a CANCEL statement or if the run unit of which the program is a member is terminated.

No action is taken when an executed CANCEL statement names a program that has not been called in this run unit or that has been called and is canceled. Control passes to the next statement.

EXIT PROGRAM Statement

The EXIT PROGRAM statement marks the logical end of a called program.

```
EXIT PROGRAM .
 ___ _______
```

Explanation of Format

The EXIT PROGRAM statement must appear in a sentence by itself. The EXIT PROGRAM sentence must appear in a paragraph by itself.

An execution of an EXIT PROGRAM statement in a called program causes control to be passed to the calling program. Execution of an EXIT PROGRAM statement in a program that is not called causes the program to act as if the statement were an EXIT statement.

For more information on the EXIT statement, refer to “EXIT” in Section 9, “PROCEDURE DIVISION Statements.”

STOP RUN Statement

The STOP RUN statement permanently ends the called program and all other programs in the run unit.

The STOP literal statement in a called program has no effect and should not be used.

For more information on the STOP statement, refer to “STOP” in Section 9, “PROCEDURE DIVISION Statements.”

Section 14
COMMUNICATION SECTION

The communication module provides the ability to access, process, and create messages or portions thereof. The module also gives you the capability to communicate through a message control system (MCS) with local and remote communication devices.

Note: *The ANSI-74 MCS concept differs from the Unisys MCS concept.*

Messages are communicated in a device-independent symbolic manner. Specific devices and system structures are known to the COBOL74 user only symbolically. This generality between the compiled program and the particular system is achieved by an interface called data communications interface (DCI). The general abilities of the compiler are adapted to the specific device needs of an application through a DCI library.

For information on using Transaction Server headers, refer to Volume 2 of this manual.

DCILIBRARY Library

A DCI library is a library to which the compiler builds references whenever a program uses the ACCEPT, DISABLE, ENABLE, RECEIVE, or SEND statements; in fact, a DCI library must be present for a COBOL program to use these verbs.

A DCI library allows programs to deal with symbolic sources and destinations instead of the actual peripherals and thus avoid recompilation when the actual peripherals are changed or rearranged.

DCIENTRYPOINT

The library reference is built with the title DCILIBRARY and the entry-point name DCIENTRYPOINT. This entry point is an untyped procedure with the eight parameters described on the following pages.

Parameter 1 of DCIENTRYPOINT

Parameter 1 is an integer with a value indicating which one of the following 11 functions to perform: Values 6 through 11 are extensions to ANSI X3.23-1974 COBOL. Refer to Volume 2 for additional information about values 6 through 11.

Value	Function to Perform
1	ACCEPT MESSAGE COUNT
2	DISABLE
3	ENABLE
4	RECEIVE
5	SEND
6	BEGIN TRANSACTION WITH TEXT
7	BEGIN TRANSACTION ABORT
8	MID TRANSACTION
9	END TRANSACTION ABORT
10	END TRANSACTION WITH NO TEXT
11	END TRANSACTION WITH TEXT

Parameter 2 of DCIENTRYPOINT

Parameter 2 is an EBCDIC array (unindexed descriptor) of the COBOL program communication description (CD). This parameter must have one of the following two formats. Format 1 is used with all input functions, and Format 2 is used with all output functions.

Format 1: (ACCEPT, DISABLE input, ENABLE input, or RECEIVE)

```
01 CD-ARRAY.
   02 QUEUE-NAME                PIC X(12).
   02 SUB-QUEUE-1-NAME          PIC X(12).
   02 SUB-QUEUE-2-NAME          PIC X(12).
   02 SUB-QUEUE-3-NAME          PIC X(12).
   02 MESSAGE-DATE              PIC 9(6).
   02 MESSAGE-TIME              PIC 9(8).
   02 SOURCE-NAME               PIC X(12).
   02 TEXT-LENGTH               PIC 9(4).
   02 END-KEY                   PIC X.
   02 STATUS-KEY                PIC XX.
   02 MESSAGE-COUNT             PIC 9(6).
```

Format 2: (DISABLE output, ENABLE output, or SEND)

```
01 CD-ARRAY.
   02 DESTINATION-COUNT           PIC 9(4).
   02 TEXT-LENGTH                 PIC 9(4).
   02 STATUS-KEY                  PIC XX.
   02 DESTINATION-TABLE      OCCURS <nnnn> TIMES.
      03 ERROR-KEY                PIC X.
      03 DESTINATION-NAME         PIC X(12).
```

Parameter 3 of DCIENTRYPOINT

Parameter 3 is an integer specifying the size of the destination array within the ANSI CD. This integer is meaningful only for the SEND verb and should be ignored by the custom-written DCILIBRARY for all other verbs.

Note: *Refer to Volume 2 for information about the procedure that links Transaction Server to its own DCI library.*

Parameter 4 of DCIENTRYPOINT

Parameter 4 is an EBCDIC ARRAY (unindexed descriptor) containing either the messages (if the function is a SEND statement or a RECEIVE statement) or the password (if the function is an ENABLE statement or a DISABLE statement.

Parameter 5 of DCIENTRYPOINT

Parameter 5 is an integer with a value indicating the length, in characters, of the array (in parameter 5) that contains the messages or the password. This value is distinct from the length of the information contained in this array, which you maintain in the CD TEXT-LENGTH field of the array in parameter 2.

Parameter 6 of DCIENTRYPOINT

Parameter 6 is an integer with a value indicating either the type of end indicator to be sent or received, or the type of enable or disable operation to be performed. For a SEND statement or a RECEIVE statement, the value specifies the end indicator as follows:

Value	End Indicator
1	ESI (end of segment/receive segment)
2	EMI (end of message/receive message)
3	EGI (end of group)

For a DISABLE statement or an ENABLE statement, the value specifies the device type as follows:

Value	Device Type
11	Input terminal
12	Input
13	Output

Parameter 7 of DCIENTRYPOINT

Parameter 7 is an integer with a value that indicates advancing control when the function being performed is a SEND statement, and indicates whether or not to wait for messages when the function is a RECEIVE statement.

For a SEND statement, the value specifies advancing as follows:

Value	Type of Advancing
0	No advancing
1	After lines
2	Before lines
3	After page
4	Before page

For a RECEIVE statement, the value of the integer can be set in the COBOL program or by the DCI library.

When set in the program, the value indicates to the DCI library whether or not to wait if no message or text is available. A value of 0 (zero) means do not wait, and a value greater than 0 means wait that number of seconds. If this parameter is set to 0 and a NO DATA clause is supplied, no waiting occurs and the action specified in the NO DATA clause is taken if no message is available.

The DCI library, before returning control to the program, sets the value of the integer to 1 if no text is available; otherwise, a value of 0 is returned with the text to the program.

Parameter 8 of DCIENTRYPOINT

Parameter 8 is an integer that equals the amount to advance in a SEND function. If a SEND function is not being performed, this parameter's value is 0.

Program Sample: CD Array

The second parameter, the CD array, allows communication between the COBOL program and the DCI library. The COBOL program maintains the CD array passed by the compiler. The DCI library updates information in the CD array it receives.

The information in the CD should be updated in coordination with the COBOL74 program according to the rules for updating items in the CC array. These updating operations include setting the values of the status key, collecting messages, handling queues, and checking password validity.

The DCI library can be written in COBOL74, ALGOL, or DCALGOL and allows access to disk files, remote files, or port files. The symbolic queues, selection algorithms, and sources and destinations as established in the ANSI-74 COBOL standard can be tailored to the particular application by using the DCI library.

Example 14–1 shows a typical DCI library entry point written in COBOL that declares eight parameters.

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID. DCIENTRYPOINT.
        ENVIRONMENT DIVISION.
        DATA DIVISION.
        LINKAGE SECTION.
(1)     77 DCI-FUNCTION                COMP PIC 9.
(2)     01 THE-CD                           PIC X(244).
        01 INPUT-CD                     REDEFINES THE-CD.
           02 SYMBOLIC-QUEUE                 PIC X(12).
           02 SYMBOLIC-SUB-QUEUE-1           PIC X(12).
           02 SYMBOLIC-SUB-QUEUE-2           PIC X(12).
           02 SYMBOLIC-SUB-QUEUE-3           PIC X(12).
           02 MESSAGE-DATE                   PIC 9(6).
           02 MESSAGE-TIME                   PIC 9(8).
           02 SYMBOLIC-SOURCE                PIC X(12).
           02 TEXT-LENGTH                    PIC 9(4).
           02 END-KEY                        PIC X.
           02 STATUS-KEY                     PIC XX.
           02 MESSAGE-COUNT                  PIC 9(6).
        01 OUTPUT-CD                    REDEFINES THE-CD.
           02 DESTINATION-COUNT              PIC 9(4).
           02 TEXT-LENGTH                    PIC 9(4).
           02 STATUS-KEY                     PIC XX.
           02 DESTINATION-TABLE         OCCURS 1 TO 18 DEPENDING
                                         ON THE-CD-OCCURRENCES.
              03 ERROR-KEY                   PIC X.
              03 SYMBOLIC-DESTINATION        PIC X(12).
```

Example 14–1. Example Program for DCI Library Entry Point

```
    (3)    77 THE-CD-OCCURRENCES          COMP PIC 999.
    (4)    01 THE-MESSAGE
                 03 THE-MESSAGE-SUB            PIC X
                                              OCCURS 1 TO 9999 DEPENDING
                                                ON THE-MESSAGE-LENGTH.
    (5)    77 THE-MESSAGE-LENGTH          COMP PIC 999.
    (6)    77 IO-OR-END-INDICATOR         COMP PIC 99.
    (7)    77 NO-DATA-OR-ADVANCING-TYPE   COMP PIC 99.
    (8)    77 ADVANCING-VALUE             COMP PIC 99.
*
           PROCEDURE DIVISION
                     USING DCI-FUNCTION,
                     THE-CD,
                     THE-CD-OCCURRENCES,
                     THE-MESSAGE,
                     THE-MESSAGE-LENGTH,
                     IO-OR-END-INDICATOR,
                     NO-DATA-OR-ADVANCING-TYPE,
                     ADVANCING-VALUE.
           MAIN SECTION.
           P1.
                PERFORM FUNCTION-SPECIFIED-BY-DCI-FUNCTION.
                EXIT PROGRAM.
```

Example 14–1. Example Program for DCI Library Entry Point

DATA DIVISION in the Communication Module

You must code a COMMUNICATION SECTION in the DATA DIVISION when you use the communication module.

In a COBOL program, the communication-description (CD) entries represent the highest level of organization in the COMMUNICATION SECTION. The COMMUNICATION SECTION header is followed by a communication-description (CD) entry consisting of a level indicator, a data-name, and a series of independent clauses. These clauses indicate the queues and subqueues, the message date and time, the source, the text length, the status and end keys, and message count for input. These clauses also specify the destination count, the text length, the status and error keys, and destinations for output. The entry itself is terminated by a period. These record areas can be implicitly redefined by user-specified, record-description entries following the various communication- description clauses.

The general formats of this entry are described on the following pages.

CD Format 1

```
CD cd-name FOR [INITIAL] INPUT

[;SYMBOLIC QUEUE IS data-name-1]

[;SYMBOLIC SUB-QUEUE-1 IS data-name-2]

[;SYMBOLIC SUB-QUEUE-2 IS data-name-3]

[;SYMBOLIC SUB-QUEUE-3 IS data-name-4]

[;MESSAGE DATE IS data-name-5]

[;MESSAGE TIME IS data-name-6]

[;SYMBOLIC SOURCE IS data-name-7]

[;TEXT LENGTH IS data-name-8]

[;END KEY IS data-name-9]

[;STATUS KEY IS data-name-10]

[;MESSAGE COUNT IS data-name-11]

[;CONVERSATION AREA IS data-name-12 SIZE IS literal-1]

[data-name-1, data-name-2,.., data-name-11]
```

Explanation of CD Format 1

A CD entry must appear only in the COMMUNICATION SECTION.

If none of the optional clauses are specified, a 01-level data-description entry must follow the CD entry.

For each standard input CD, a record area of 87 characters is allocated.

Use of the 01-level data-description entry or the optional clauses results in a record with an implicit description that is equivalent to the following description:

```
Implicit Description                          Comment

01 data-name-0.
   02 data-name-1  PICTURE X(12).             SYMBOLIC QUEUE
   02 data-name-2  PICTURE X(12).             SYMBOLIC SUB-QUEUE-1
   02 data-name-3  PICTURE X(12).             SYMBOLIC SUB-QUEUE-2
   02 data-name-4  PICTURE X(12).             SYMBOLIC SUB-QUEUE-3
   02 data-name-5  PICTURE 9(06).             MESSAGE DATE
   02 data-name-6  PICTURE 9(08).             MESSAGE TIME
   02 data-name-7  PICTURE X(12).             SYMBOLIC SOURCE
   02 data-name-8  PICTURE 9(04).             TEXT LENGTH
   02 data-name-9  PICTURE X.                 END KEY
   02 data-name-10 PICTURE XX.                STATUS KEY
   02 data-name-11 PICTURE 9(06).             MESSAGE COUNT
```

Note: *In the preceding listing, the information under "Comment" is for clarification and is not part of the description.*

Record-description entries following an input CD implicitly redefine this record. The entries must describe a record of exactly 87 characters for the standard CD. (Literal-1 is a numeric literal representing the size of the conversation area in characters.)

Data-names 1 through 11 must be unique. Within this series, any data-name can be replaced by the reserved word FILLER.

The input CD information constitutes the communication between the MCS and the program about the message being handled. This information is not displayed on the terminal as part of the message.

INITIAL

In a single program, the INITIAL clause can be specified in only one CD. The INITIAL clause must not be used in a program that specifies the USING phrase in the PROCEDURE DIVISION header.

If the MCS attempts to schedule a program lacking an INITIAL clause, the results are undefined.

Except for the INITIAL clause, the optional clauses can be written in any order.

data-name-1 through data-name-4

The data items referenced by data-name-1, data-name-2, data-name-3, and data-name-4 contain symbolic names designating queues, subqueues, and so on. All symbolic names must follow the rules for the formation of system-names and must have been previously defined to the MCS.

The contents of the data items referenced by data-name-2, data-name-3, and data-name-4, when not being used, must contain spaces.

A RECEIVE statement causes the serial return of the next message or a portion of a message from the queue as specified by the entries in the CD.

If, during execution of a RECEIVE statement, a message from a more specific source is needed, the contents of the data item referenced by data-name-1 can be made more specific by the use of the contents of the data items referenced by data-name-2, data-name-3, and data-name-4. When a given level of the queue structure is specified, all higher levels must also be specified.

If any levels of the queue hierarchy are not specified, the MCS determines the next message or the portion of a message to be accessed.

After execution of a RECEIVE statement, the contents of the data items referenced by data-name-1 through data-name-4 contain the symbolic names of all levels of the queue structure.

Whenever a program is scheduled by the MCS to process a message, the symbolic names of the queue structure that demanded this activity are placed in the data items referenced by data-name-1 through data-name-4 of the CD associated with the INITIAL clause, as applicable. In all other cases, the contents of the data items referenced by data-name-1 through data-name-4 of the CD associated with the INITIAL clause contain spaces.

The symbolic names are inserted, or the initialization to spaces is completed, prior to the execution of the first PROCEDURE DIVISION statement.

The execution of a subsequent RECEIVE statement naming the same contents of the data items referenced by data-name-1 through data-name-4 returns the message that caused the program to be scheduled. Only at that time is the remainder of the CD updated.

data-name-5

Data-name-5 has the format YYMMDD (year, month, day). The contents represent the date on which the MCS recognizes that the message is complete.

The contents of the data item referenced by data-name-5 are only updated by the MCS as part of the execution of a RECEIVE statement.

data-name-6

The contents of data-name-6 have the format HHMMSSTT (hours, minutes, seconds, hundredths of a second). The contents represent the time at which the MCS recognizes that the message is complete.

The contents of the data item referenced by data-name-6 are only updated by the MCS as part of the execution of a RECEIVE statement.

data-name-7

During execution of a RECEIVE statement, the MCS provides, in the data item referenced by data-name-7, the symbolic name of the communication terminal that is the source of the message being transferred. However, if the symbolic name of the communication terminal is not known to the MCS, the contents of the data item referenced by data-name-7 contain spaces.

data-name-8

The contents of the data item referenced by data-name-8 indicate to the MCS that the number of character positions filled as a result of the execution of the RECEIVE statement.

data-name-9

The contents of the data item referenced by data-name-9 are set only by the MCS as part of the execution of a RECEIVE statement, according to the following rules:

- When the RECEIVE MESSAGE phrase is specified, the following action is taken:
  - If an end-of-group indicator has been detected, the contents of the data item referenced by data-name-9 are set to 3.
  - If an end-of-message indicator has been detected, the contents of the data item referenced by data-name-9 are set to 2.
  - If less than a whole message is transferred, the contents of the data item referenced by data-name-9 are set to 0.
- When the RECEIVE SEGMENT phrase is specified, the following action is taken:
  - If an end-of-group indicator has been detected, the contents of the data item referenced by data-name-9 are set to 3.
  - If an end-of-message indicator has been detected, the contents of the data item referenced by data-name-9 are set to 2.
  - If an end-of-segment indicator has been detected, the contents of the data item referenced by data-name-9 are set to 1.
  - If less than a message segment is transferred, the contents of the data item referenced by data-name-9 are set to 0.
- When more than one of the preceding conditions is satisfied simultaneously, the rule first satisfied in the order listed determines the contents of the data item referenced by data-name-9.

data-name-10

The contents of the data item referenced by data-name-10 indicate the status condition of the previously executed RECEIVE, ACCEPT MESSAGE COUNT, ENABLE INPUT, or DISABLE INPUT statements.

The actual association between the contents of the data item referenced by data-name-10 and the status condition itself is defined in Figure 14–1.

For the 01-level, Figure 14–1 indicates the possible contents of the data items referenced by data-name-10 for Format 1 and by data-name-3 for Format 2 at the completion of each statement shown. An X on a line in a statement column indicates that the associated code shown for that line is possible for that statement.

RECEIVE	SEND	ACCEPT MESSAGE COUNT	ENABLE INPUT (without terminal)	ENABLE OUTPUT	DISABLE INPUT (without terminal)	DISABLE OUTPUT	Status Key Code	
X	X	X	X	X	X	X	00	No error detected. Action completed.
	X						10	Destination is disabled. Action completed.
	X			X		X	20	Destination unknown or access thereto denied by system. No action taken for unknown destination. Data-name-4 (ERROR KEY) indicates unknown.
X		X	X		X		20	One or more queues or subqueues unknown or access to queue denied by system. No action taken.
	X			X		X	30	Content of DESTINATION COUNT clause invalid. No action taken.
			X	X	X	X	40	Password invalid. No enabling/ disabling action taken.
	X						50	Character count greater than length of sending field. No action taken.
X	X	X	X	X	X	X	91	No MCS present. No action taken.

Figure 14–1. Communication Status Condition in the 01-level

data-name-11

The contents of the data item referenced by data-name-11 indicate the number of messages that exist in a queue, sub-queue-1, and so forth. The MCS updates the contents of the data item referenced by data-name-11 only as part of the execution of an ACCEPT statement with the COUNT phrase.

For More Information

Refer to "PROCEDURE DIVISION Header" in Section 8, "PROCEDURE DIVISION Concepts," for information about the restrictions for a program with a USING clause in the header.

CD Format 2

```
CD cd-name FOR OUTPUT

[;DESTINATION COUNT IS data-name-1]

[;TEXT LENGTH IS data-name-2]

[;STATUS KEY IS data-name-3]
┌                                                          ┐
│;DESTINATION TABLE OCCURS integer-2 TIMES                  │
│                                                          │
│        [;INDEXED BY index-name-1 [index-name-2]...]      │
└                                                          ┘
[;ERROR KEY IS data-name-4]

[;SYMBOLIC DESTINATION IS data-name-5]

[;CONVERSATION AREA IS data-name-6 SIZE IS literal-1]
```

Explanation of CD Format 2

A CD entry must appear only in the COMMUNICATION SECTION.

If none of the optional clauses of the CD are specified, a 01-level data-description entry must follow the CD entry.

For each standard-defined output CD, a record area of contiguous standard data-format characters is allocated according to the following formula:

```
10 + 13 * integer-2
```

The first three words of the record area define the DESTINATION COUNT clause, the TEXT LENGTH clause, and the STATUS KEY clause. Note that when the 01-level data description is used instead of the optional clauses, the destination table is assumed to occur only once and any remaining data greater than 5 words (30 characters) is assumed to define the conversation area.

For the standard-defined CD, the record area is defined as follows:

- The DESTINATION COUNT clause defines data-name-1 as the name of a data item whose implicit description is an integer without an operational sign. The data item occupies positions 1 through 4 in the record.
- The TEXT LENGTH clause defines data-name-2 as the name of an elementary data item whose implicit description is an integer of 4 digits without an operational sign. The data item occupies character positions 5 through 8 in the record.
- The STATUS KEY clause defines data-name-3 as the name of an elementary, alphanumeric data item of 2 characters that occupies positions 9 and 10 in the record.
- Character positions 11 through 23 and every set of 13 characters thereafter form table items of the following description:
  - The ERROR KEY clause defines data-name-4 as the name of an elementary, alphanumeric data item of 1 character.
  - The SYMBOLIC DESTINATION clause defines data-name-5 as the name of an elementary, alphanumeric data item of 12 characters.
- The CONVERSATION AREA clause contains user-defined text and is valid only when the USAGE BINARY clause is used. Refer to Volume 2 for information.

Use of the preceding clauses results in a record with an implicit description that is equivalent to the following standard CD definition:

```
Implicit Description                                Comment

01 data-name-0.
   02 data-name-1  PICTURE 9(04).                   DESTINATION COUNT
   02 data-name-2  PICTURE 9(04).                   TEXT LENGTH
   02 data-name-3  PICTURE XX.                      STATUS KEY
   02 data-name OCCURS integer-2 TIMES.             DESTINATION TABLE
   02 data-name-4  PICTURE X.                       ERROR KEY
   02 data-name-5  PICTURE X(12).                   SYMBOLIC DESTINATION
```

Note: *In the preceding listing, the information under "Comment" is for clarification and is not part of the description.*

Record descriptions following an output CD implicitly redefine this record. Multiple redefinitions of this record are permitted; however, only the first redefinition can contain VALUE clauses.

Data-name-1 through data-name-6 must be unique.

If the DESTINATION TABLE OCCURS clause is not specified, one ERROR KEY and one SYMBOLIC DESTINATION clause are assumed. In this case, neither subscripting nor indexing is permitted when these data items are referenced.

If the DESTINATION TABLE OCCURS clause is specified, data-name-4 and data-name-5 can be referenced only by subscripting or indexing.

In the 01-level, the value of the data item referenced by data-name-1 and integer-2 must be 1. In the 02-level, no restriction exists on the value of the data item referenced by data-name-1 and integer-2.

The nature of the output CD information is such that it is not sent to the terminal but constitutes the communication between the program and the MCS about the message being handled.

During execution of a SEND, an ENABLE OUTPUT, or a DISABLE OUTPUT statement, the contents of the data item referenced by data-name-1 indicate to the MCS the number of symbolic destinations to be used from the area referenced by data-name-5.

The MCS finds the first symbolic destination in the first occurrence of the area referenced by data-name-5, the second symbolic destination in the second occurrence of the area referenced by data-name-5, and so on, up to and including the occurrence of the area referenced by data-name-5 indicated by the contents of data-name-1.

If, during execution of a SEND, an ENABLE OUTPUT, or a DISABLE OUTPUT statement, the value of the data item referenced by data-name-1 is outside the range 1 through integer-2, an error condition is indicated and the execution of the SEND, the ENABLE OUTPUT, or the DISABLE OUTPUT statement is terminated.

You must ensure that the value of the data item referenced by data-name-1 is valid at the time of the execution of the SEND, the ENABLE OUTPUT, or the DISABLE OUTPUT statement.

As part of the execution of a SEND statement, the MCS interprets the contents of the data item referenced by data-name-2 to be your indication of the number of leftmost character positions of the data item referenced by the associated SEND identifier from which data is to be transferred.

Each occurrence of the data item referenced by data-name-5 contains a symbolic destination previously known to the MCS. These symbolic destination names must follow the rules for the formation of system-names.

The contents of the data item referenced by data-name-3 indicate the status condition of the previously executed SEND, ENABLE OUTPUT, or DISABLE OUTPUT statements.

The association between the contents of the data item referenced by data-name-3 and the status condition itself is defined in Figure 14–1.

If, during execution of a SEND, an ENABLE OUTPUT, or a DISABLE OUTPUT statement, the MCS determines that any specified destination is unknown or chooses to deny the program access to any destination, then the contents of the data item referenced by data-name-3 and all occurrences of the data items referenced by data-name-4 are updated.

When the contents of the data item referenced by data-name-4 are equal to 1, this value indicates that either the associated value in the area referenced by data-name-5 has not been defined previously to the MCS, or that the MCS has been denied access to this destination. Otherwise, the contents of the data item referenced by data-name-4 are set
to 0.

For the 02-level, Figure 14–2 indicates the possible contents of the data items referenced by data-name-10 for Format 1 and by data-name-3 for Format 2 at the completion of each statement shown. An X on a line in a statement column indicates that the associated code shown for that line is possible for that statement.

RECEIVE	SEND	ACCEPT MESSAGE COUNT	ENABLE INPUT (without terminal)	ENABLE INPUT (with terminal)	ENABLE OUTPUT	DISABLE INPUT (without terminal)	DISABLE INPUT (with terminal)	DISABLE OUTPUT	Status Key Code	
X	X	X	X	X	X	X	X	X	00	No error detected. Action completed.
	X								10	One or more destinations are disabled. Action completed.
	X				X			X	20	One or more destinations unknown, or accsess denied by system. Action completed for known destinations. No action taken for known destinations. Data-name-4 (ERROR KEY) indicates known or unknown (includes system denied access).
X		X	X			X			20	One or more queues or subqueues unknown or access to queue denied by system. No action taken.
				X			X		20	The source is unknown, or access thereto denied by the system. No action taken.
	X				X			X	30	Content of DESTINATION COUNT clause invalid. No action taken.
			X	X	X	X	X	X	40	Password invalid. No enabling/ disabling action taken.
	X								50	Character count greater than length of sending field. No action taken.
	X								60	Partial segment with either zero character count or no sending area specified. No action taken.
X	X	X	X	X	X	X	X	X	91	No MCS present. No action taken.

Figure 14–2. Communication Status Key Condition in the 02-level

Example

Example 14–2 shows a sample program that uses some of the communication module constructs. The associations of physical terminals with specific, named input and output queues and the appropriate passwords (keys) are maintained by the DCILIBRARY.

The program enables output to the two terminals associated with OUT-QUEUE-1 and OUT-QUEUE-2 by using the password SIMON1. It then sends the message *TERMINALS ENABLED OUTPUT* to both. The program enables input from the terminal associated with IN-QUEUE-1 and TERMINAL #1 by using the password SIMON1. It receives one message from that terminal. The program then enables input from the terminal associated with IN-QUEUE-2 and TERMINAL #2 by using the password SIMON1. It receives one message from that terminal. It sends the following messages to both terminals, using keywords ESI and EMI for end indicators:

```
SEGMENT INITIATED-CONTINUED
-MSG COMPLETE.
```

The program sends the following messages to both terminals, using the identifier END-FLAG to contain the end indicator.

```
SEGMENT INITIATED-CONTINUED
-MSG COMPLETE.
-GROUP  COMPLETE.
```

Then the program sends the message *ONLY TERMINAL OUT-QUEUE-1 SHOULD RECEIVE THIS MESSAGE* only to the terminal associated with OUT-QUEUE-1. It finally disables output and input to both terminals by using the password SIMON1.

```
 00100 IDENTIFICATION DIVISION.
 02200 ENVIRONMENT DIVISION.
 03200 DATA DIVISION.
 04000 WORKING-STORAGE SECTION.
 04100 77  END-FLAG         PIC 9.
 06200 01  ENABLE-MSG       PIC X(24) VALUE "TERMINALS ENABLED OUTPUT".
 07900 01  ONE-TERMINAL-MSG PIC X(53) VALUE
 08000     "ONLY TERMINAL OUT-QUEUE-1 SHOULD RECEIVE THIS MESSAGE".
 08500 01  SEG-INIT         PIC X(17) VALUE "SEGMENT INITIATED".
 09000 01  SEG-CONT         PIC X(10) VALUE "-CONTINUED".
 09100 01  MSG-COMP         PIC X(14) VALUE "-MSG COMPLETE.".
 09200 01  GROUP-COMP       PIC X(16) VALUE "-GROUP COMPLETE.".
 10000 01  MSG-1            PIC X(180).
 20700 COMMUNICATION SECTION.
 20800 CD  CM-INQUE-1 INPUT.
 20900 01  INQUE-1-RECORD.
 21000     02  QUEUE-SET PIC X(12).
 21200     02  FILLER PIC X(36) VALUE SPACES.
```

Example 14–2. Maintaining Associations of Physical Terminals with Queues

```
21300     02  FILLER PIC X(14).
21400     02  SYM-SOURCE PIC X(12).
21500     02  IN-LENGTH PIC 9999.
21600     02  END-KEY PIC X.
21700     02  IN-STATUS PIC XX.
21800     02  MSG-COUNT PIC 9(6).
21900 CD  CM-OUTQUE-1 OUTPUT
22000     DESTINATION COUNT DEST-COUNT
22100     TEXT LENGTH OUT-LENGTH
22200     STATUS KEY OUT-STATUS
22300     DESTINATION TABLE OCCURS 2 TIMES INDEXED BY I1
22400     ERROR KEY ERR-KEY
22500     DESTINATION SYM-DEST.
22600 PROCEDURE DIVISION. 22800 ENABLE-MSGS.
23000     MOVE 2 TO DEST-COUNT.
23500     MOVE "OUT-QUEUE-1" TO SYM-DEST (1).
24000     MOVE "OUT-QUEUE-2" TO SYM-DEST (2).
24100     ENABLE OUTPUT CM-OUTQUE-1 WITH KEY "SIMON1".
24460     WAIT (10).
24500 SEND-MSG.
24600     MOVE 24 TO OUT-LENGTH.
24700     SEND CM-OUTQUE-1 FROM ENABLE-MSG WITH EMI.
28000 RECEIVE-MSGS.
28200     MOVE "IN-QUEUE-1" TO QUEUE-SET.
28400     MOVE "TERMINAL #1" TO SYM-SOURCE.
28450     ENABLE INPUT TERMINAL CM-INQUE-1 WITH KEY "SIMON1".
28460     WAIT (15).
28500     RECEIVE CM-INQUE-1 MESSAGE INTO MSG-1 NO DATA DISPLAY "ERR1".
29000     DISPLAY MSG-1.
30000     MOVE "IN-QUEUE-2" TO QUEUE-SET.
31000     MOVE "TERMINAL #2" TO SYM-SOURCE.
31500     ENABLE INPUT TERMINAL CM-INQUE-1 WITH KEY "SIMON1".
31600     WAIT (15).
31700     RECEIVE CM-INQUE-1 SEGMENT INTO MSG-1 NO DATA DISPLAY "ERR2".
31800     DISPLAY MSG-1.
50700 SEGMENTED-MSGS-01.
50900     MOVE 17 TO OUT-LENGTH.
51000     SEND CM-OUTQUE-1 FROM SEG-INIT.
51100     MOVE 10 TO OUT-LENGTH.
51200     SEND CM-OUTQUE-1 FROM SEG-CONT WITH ESI.
51300     MOVE 14 TO OUT-LENGTH.
51400     SEND CM-OUTQUE-1 FROM MSG-COMP WITH EMI.
51500 SEGMENTED-MSGS-02.
51700     MOVE 0 TO END-FLAG.
51800     MOVE 17 TO OUT-LENGTH.
51900     SEND CM-OUTQUE-1 FROM SEG-INIT WITH END-FLAG.
52000     MOVE 1 TO END-FLAG.
```

Example 14–2. Maintaining Associations of Physical Terminals with Queues

```
52100     MOVE 10 TO OUT-LENGTH.
52200     SEND CM-OUTQUE-1 FROM SEG-CONT WITH END-FLAG.
52300     MOVE 2 TO END-FLAG.
52400     MOVE 14 TO OUT-LENGTH.
52500     SEND CM-OUTQUE-1 FROM MSG-COMP WITH END-FLAG.
52600     MOVE 3 TO END-FLAG.
52700     MOVE 16 TO OUT-LENGTH.
52800     SEND CM-OUTQUE-1 FROM GROUP-COMP WITH END-FLAG.
53700 SINGLE-TERMINAL-MSG.
53800     MOVE 1 TO DEST-COUNT.
53900     MOVE 53 TO OUT-LENGTH.
54000     SEND CM-OUTQUE-1 FROM ONE-TERMINAL-MSG WITH EMI.
54300 DISABLE-MSGS.
54350     WAIT (10).
54400     MOVE 2 TO DEST-COUNT.
54500     MOVE "OUT-QUEUE-1" TO SYM-DEST (1).
54600     MOVE "OUT-QUEUE-2" TO SYM-DEST (2).
54700     DISABLE OUTPUT CM-OUTQUE-1 WITH KEY "SIMON1".
55000     MOVE "TERMINAL #1" TO SYM-SOURCE.
55100     DISABLE INPUT TERMINAL CM-INQUE-1 WITH KEY "SIMON1".
55200     MOVE "TERMINAL #2" TO SYM-SOURCE.
55300     DISABLE INPUT TERMINAL CM-INQUE-1 WITH KEY "SIMON1".
56300     STOP RUN.
```

Example 14–2. Maintaining Associations of Physical Terminals with Queues

PROCEDURE DIVISION in the Communication Module

The following constructs apply to the PROCEDURE DIVISION when the communication module is used.

Note: *Refer to* COBOL ANSI-74 Reference Manual, Vol. 2 *for information about using PROCEDURE DIVISION statements with Transaction Server headers.*

ACCEPT MESSAGE COUNT Statement

The ACCEPT MESSAGE COUNT statement makes available the number of messages in a queue.

```
ACCEPT cd-name MESSAGE COUNT
```

Explanation of Format

Cd-name must reference an input CD.

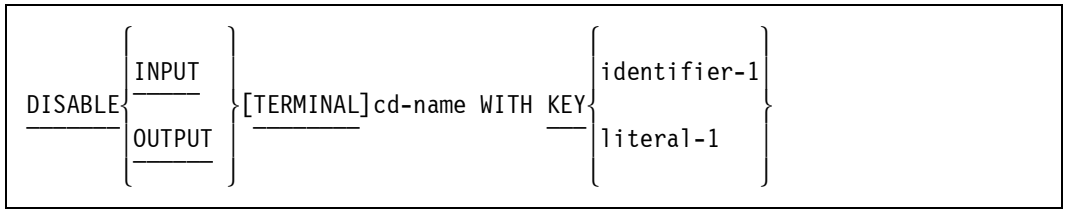
The ACCEPT MESSAGE COUNT statement causes the MESSAGE COUNT field specified for cd-name to be updated to indicate the number of messages present in a queue, sub-queue-1, and so on.

When the ACCEPT MESSAGE COUNT statement is executed, the contents of the area specified by a communication-description (CD) entry must contain at least the name of the symbolic queue to be tested. When the condition is tested, the contents of the data items referenced by data-name-10 (STATUS KEY phrase) and data-name-11 (MESSAGE COUNT phrase) of the area associated with the communication description (CD) are updated as necessary.

See "DATA DIVISION in the Communication Module" earlier in this section for details on the STATUS KEY and MESSAGE COUNT phrases in Format 1 of the CD entries.

DISABLE Statement

The DISABLE statement notifies the MCS to inhibit data transfer between specified output queues and destinations for output or to inhibit data transfer between specified sources and input queues for input.



Explanation of Format

The DISABLE statement provides a logical disconnection between the MCS and the specified sources or destinations. When this logical disconnection has already occurred or is handled by some means external to this program, the DISABLE statement is not required in this program. The logical path for the transfer of data between the COBOL programs and the MCS is not affected by the DISABLE statement.

When the logical disconnection specified by the DISABLE statement has already occurred, is handled by some means external to this program, or is denied by the MCS, the status-key data item in the area referenced by cd-name is updated.

INPUT

When the INPUT phrase with the optional word TERMINAL is specified, the logical path between the source and all associated queues and subqueues is deactivated. Only the contents of the data item referenced by data-name-7 (SYMBOLIC SOURCE phrase) of the area referenced by cd-name are meaningful.

When the INPUT phrase without the optional word TERMINAL is specified, the logical paths for all sources associated with the queues and subqueues specified by the contents of data-name-1 (SYMBOLIC QUEUE phrase) through data-name-4 (SYMBOLIC SUB-QUEUE-3 phrase) of the area referenced by cd-name are deactivated.

OUTPUT

When the OUTPUT phrase is specified, the logical path for the destination or the logical paths for all destinations specified by the contents of the data item referenced by data-name-5 (SYMBOLIC DESTINATION phrase) of the data referenced by cd-name are deactivated.

cd-name

The cd-name must refer to an input CD when the INPUT phrase is specified and must refer to an output CD when the OUTPUT phrase is specified.

literal-1 or identifier-1

Literal-1 or the contents of the data item referenced by identifier-1 must be defined as alphanumeric.

The MCS handles a password of between 1 and 10 characters, inclusive. Literal-1 or the content of the data item referenced by identifier-1 is transferred to the MCS according to the rules for the MOVE statement. The MCS receives the password in an area considered to be an elementary, alphanumeric data item that is 10 characters long.

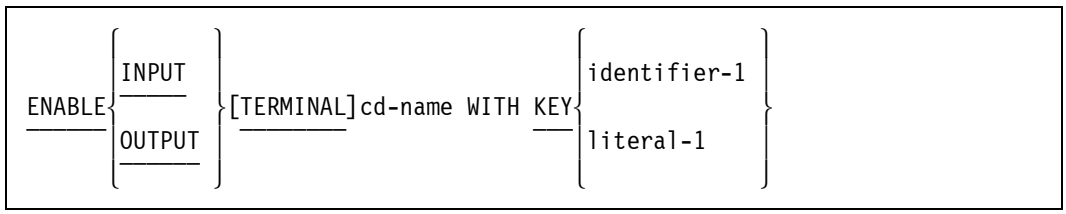
For More Information

For information about the location of the status-key data item, refer to “CD Format 1” and “CD Format 2” earlier in this section.

ENABLE Statement

The ENABLE statement notifies the MCS to allow data transfer between specified output queues and destinations for output or to allow data transfer between specified sources and input queues for input.

The general format of this statement is as follows:



Explanation of Format

The ENABLE statement provides a logical connection between the MCS and the specified sources or destinations. When this logical connection is already present or is handled by some means external to this program, the ENABLE statement is not required in this program. The logical path for the transfer of data between the COBOL programs and the MCS is not affected by the ENABLE statement.

When the logical connection specified by the ENABLE statement already exists, is to be handled by some means external to this program, or is denied by the MCS, the status-key data item in the area referenced by cd-name is updated.

INPUT

When the INPUT phrase with the optional word TERMINAL is specified, the logical path between the source and all associated queues and subqueues that are already enabled is activated. Only the contents of the data item referenced by data-name-7 (SYMBOLIC SOURCE phrase) of the area referenced by cd-name are meaningful to the MCS.

When the INPUT phrase without the optional word TERMINAL is specified, the logical paths for all sources associated with the queue and subqueues specified by the contents of data-name-1 (SYMBOLIC QUEUE phrase) through data-name-4 (SYMBOLIC SUB-QUEUE phrase) of the area referenced by cd-name are activated.

OUTPUT

When the OUTPUT phrase is specified, the logical paths for all destinations specified by the contents of the data item referenced by data-name-5 (SYMBOLIC DESTINATION phrase) of the area referenced by cd-name are activated.

cd-name

Cd-name must refer to an input CD when the INPUT phrase is specified and must refer to an output CD when the OUTPUT phrase is specified.

literal-1 or identifier-1

Literal-1 or the contents of the data item referenced by identifier-1 must be defined as alphanumeric.

The MCS handles a password of between 1 and 10 characters, inclusive. Literal-1 or the contents of the data item referenced by identifier-1 are transferred to the MCS according to the rules for the MOVE statement. The MCS receives the password in an area considered to be an elementary, alphanumeric data item that is 10 characters long.

For More Information

For information about the location of the status-key data item, refer to "CD Format 1" and "CD Format 2."

RECEIVE Statement

The RECEIVE statement makes a message, a message segment (or a portion of a message or a segment), and pertinent information about the data available to the COBOL74 program from a queue maintained by the MCS. The RECEIVE statement enables a specific imperative-statement when no data is available.

When execution of a RECEIVE statement returns a portion of a message, only the subsequent execution of RECEIVE statements in that run unit can cause the remaining portion of the message to be returned.

After execution of a STOP RUN statement, the disposition of a remaining portion of a message partially obtained in that run unit is defined by the data communications interface (DCI) library.

```
                    ⎛         ⎞
RECEIVE cd-name     ⎜ MESSAGE ⎟ INTO identifier-1
                    ⎜         ⎟
                    ⎝ SEGMENT ⎠

[;NO DATA imperative-statement]
```

Explanation of Format

The data items identified by the input CD are updated appropriately by the MCS at each execution of a RECEIVE statement.

cd-name

Cd-name must reference an input CD.

The contents of the data items specified by data-name-1 (SYMBOLIC QUEUE phrase) through data-name-4 (SYMBOLIC SUB-QUEUE-3 phrase) of the area referenced by cd-name designate the queue structure containing the message.

MESSAGE

When the MESSAGE phrase is used, end-of-segment indicators are ignored and the following rules apply to the data transfer:

- If a message is the same size as the area referenced by identifier-1, the message is stored in the area referenced by identifier-1.
- If a message size is smaller than the area referenced by identifier-1, the message is aligned to the leftmost character position of the area referenced by identifier-1 with no zero fill.
- If a message size is larger than the area referenced by identifier-1, the message fills the area referenced by identifier-1 from left to right, starting with the leftmost character of the message. In the 01-level, the disposition of the remainder of the message is undefined.

  In the 02-level, the remainder of the message can be transferred to the area referenced by identifier-1. Subsequent RECEIVE statements refer to the same queue, subqueue, and so on. The remainder of the message, in applying the preceding bulleted rules, is treated as a new message.

A single execution of a RECEIVE statement never returns more than a single message to the data item referenced by identifier-1. However, the MCS does not pass any portion of a message to the object program until the entire message is available in the input queue, even if the SEGMENT phrase of the RECEIVE statement is specified.

SEGMENT

When the SEGMENT phrase is used, the following rules apply:

- If a segment is the same size as the area referenced by identifier-1, the segment is stored in the area referenced by identifier-1.
- If the segment size is smaller than the area referenced by identifier-1, the segment is aligned to the leftmost character position of the area referenced by identifier-1 with no space fill.
- If a segment size is greater than the area referenced by identifier-1, the segment fills the area referenced by identifier-1 from left to right, starting with the leftmost character of the segment. The remainder of the segment can be transferred to the area referenced by identifier-1. Subsequent RECEIVE statements call out the same queue, subqueue, and so on.
- If the text to be accessed by the RECEIVE statement has an end-of-message indicator (EMI) or end-of-group (EGI) indicator associated with it, the existence of an end-of-segment indicator (ESI) associated with the text is implied and the text is treated as a message segment.

A single execution of a RECEIVE statement never returns more than a single segment to the data item referenced by identifier-1. However, the MCS does not pass any portion of a message to the object program until the entire message is available in the input queue, even if the SEGMENT phrase of the RECEIVE statement is specified.

identifier-1

The message, the message segment, or the portion of a message or a segment is transferred to the receiving character positions of the area referenced by identifier-1 and is aligned to the left without space-fill.

During execution of a RECEIVE statement, the MCS makes data available in the data item referenced by identifier-1, and control is transferred to the next executable statement, whether or not the NO DATA phrase is specified.

NO DATA

During execution of a RECEIVE statement, the MCS does not make data available in the data item referenced by identifier-1, and the following action takes place:

- If the NO DATA phrase is specified, the receive operation is terminated with the indication that action is complete and the imperative-statement in the NO DATA phrase is executed.
- If the NO DATA phrase is not specified, execution of the object program is suspended until data is made available in the data item referenced by identifier-1.
- If one or more queues or subqueues is unknown to the MCS or if the MCS denies this program access to a queue or subqueue, control passes to the next executable statement whether or not the NO DATA phrase is specified. (See Figures 14–1 and 14-2 earlier in this section.)

The imperative-statement can be the NEXT SENTENCE phrase.

For More Information

Refer to "DATA DIVISION in the Communication Module" in this section for information about the data items identified by the input CD.

SEND Statement

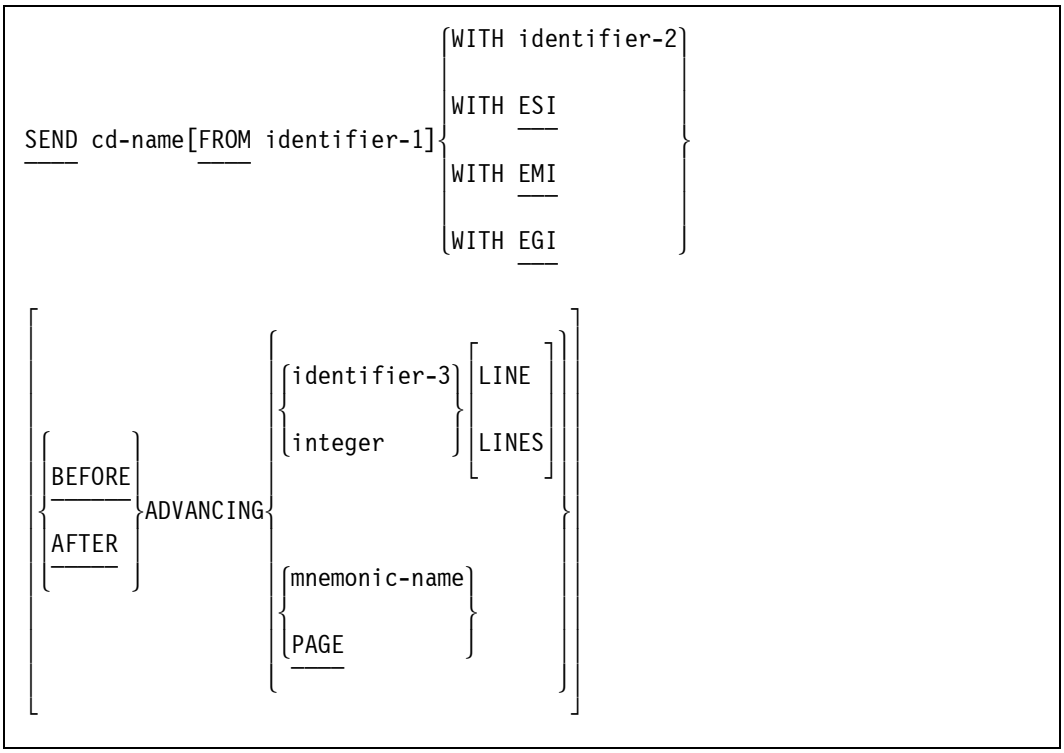
The SEND statement releases a message, a message segment, or a portion of a message to one or more output queues maintained by the MCS.

A single execution of a Format 1 SEND statement releases only a single portion of a message or a message segment to the MCS. A single execution of a Format 2 SEND statement never releases to the MCS more than a single message or a single message segment as indicated by the contents of the data item referenced by identifier-2 or by the specified end-of-segment indicator (ESI), end-of-message indicator (EMI), or end-of-group indicator (EGI). However, the MCS does not transmit any portion of a message to a communications device until the entire message is placed in the output queue.

Format 1

```
SEND cd-name FROM identifier-1
```

Format 2



Explanation of Formats

Cd-name must reference an output CD.

Identifier-1 must be the data-name of the area where the data is made available to the MCS so that the data can be sent.

Identifier-2 must reference a 1-character integer without an operational sign.

Identifier-3, if used, must be the name of an elementary integer item.

If a mnemonic-name phrase is used, the name is identified with a particular feature specified in the SPECIAL-NAMES paragraph in the ENVIRONMENT DIVISION.

Integer or the value of identifier-3 can be 0.

General Rules

The contents of the data items specified by data-name-1 (SYMBOLIC QUEUE phrase) IDATE-WRITTEN of the area referenced by cd-name designate the queue structure containing the message. (Refer to “DATA DIVISION in the Communication Module” earlier in this section.

The message, the message segment, or the portion of a message or a message segment is to be moved to the SEND character positions of the area referenced by identifier-1 and aligned to the left with zero fill.

When a receiving communication device (such as a printer or a display screen) is oriented to a fixed line size, the following rules apply:

- Each message or message segment begins at the leftmost character position of the physical line.
- A message or a message segment smaller than the physical line size is released to appear to be zero-filled to the right.
- Excess characters of a message or a message segment are not truncated. Characters are packed to a size equal to that of the physical line and then are sent to the device. This process continues on the next line with the excess characters.

When a receiving communication device (such as a printer or another computer) is oriented to handling variable-length messages, each message or message segment begins with the next available character position of the communications device.

As part of the execution of a SEND statement, the MCS interprets the contents of the data item referenced by data-name-8 (TEXT LENGTH phrase) of the area referenced by cd-name to be your indication of the number of leftmost character positions of the data item referenced by identifier-1 from which data is to be transferred.

If the value of data-name-8 (TEXT LENGTH phrase) is 0, no characters of the data item referenced by identifier-1 are transferred.

The value of data-name-8 (TEXT LENGTH phrase) cannot be outside the range 0 through the size of the data item referenced by identifier-1, inclusive. If the value of data-name-8 is outside the range, an error is indicated by the value of the data item referenced by data-name-10 (STATUS KEY phrase) in the area referenced by cd-name, and no data is transferred.

As part of the execution of a SEND statement, the contents of the data item referenced by data-name-10 (STATUS KEY phrase) of the area referenced by cd-name are updated by the MCS.

The effect of special control characters in the contents of the data item referenced by identifier-1 is undefined.

During the execution of the run unit, the disposition of a portion of a message not ended by an EMI or an EGI is undefined. Thus, the message does not logically exist for the MCS and cannot be sent to a destination.

After the execution of a STOP RUN statement, any portion of a message transferred from the run unit as a result of a SEND statement, but not terminated by an EMI or an EGI, is purged from the system. Thus, no portion of the message is sent.

Once the execution of a SEND statement has released a portion of a message to the MCS, only the subsequent execution of SEND statements in the same run unit can cause the remaining portions of the message to be released.

Format 2

The contents of the data item referenced by identifier-2 indicate that the contents of the data item referenced by identifier-1 are to have an associated end-of-segment indicator (ESI), end-of-message indicator (EMI), or end-of-group indicator (EGI) according to the schedule in Table 14–1.

Table 14–1. Transmission Indicator Schedule

Identifier-2	Identifier-1	Meaning
0	No indicator	No indicator
1	ESI	An end-of-segment indicator
2	EMI	An end-of-message indicator
3	EGI	An end-of-group indicator

Any character other than 1, 2, or 3 is interpreted as 0. If the content of the data item referenced by identifier-2 is other than 1, 2, or 3 and if identifier-1 is not specified, then an error is indicated by the value in the data item referenced by data-name-10(STATUS KEY phrase) in the area referenced by cd-name, and no data is transferred.

The ESI indicates to the MCS that the message segment is complete, the EMI indicates to the MCS that the message is complete, and the EGI indicates to the MCS that the group of messages is complete. The MCS recognizes these indicators and establishes the appropriate linkage to maintain group, message, and segment control.

The hierarchy of ending indicators (major to minor) is EGI, EMI, and ESI. An EGI need not be preceded by an ESI or an EMI, and an EMI need not be preceded by an ESI.

The ADVANCING phrase enables you to control the vertical positioning of each message or message segment on a communications device where vertical positioning applies. If vertical positioning is not applicable on the device, the MCS ignores the specified or implied vertical positioning.

If identifier-2 is specified and the contents of the data item referenced by identifier-2 are equal to 0, the ADVANCING phrase is ignored by the MCS.

On a device where vertical positioning applies and the ADVANCING phrase is not specified, automatic advancing occurs as if you had specified the AFTER ADVANCING 1 LINE phrase.

If the ADVANCING phrase is implicitly or explicitly specified and vertical positioning applies, the following rules apply:

- If identifier-3 or integer is specified, characters transmitted to the communications device are repositioned vertically downward the number of lines equal to the value associated with the data item referenced by identifier-3 or integer.
- If mnemonic-name is specified, characters transmitted to the communications device are positioned according to the rules specified for that communications device.
- If the BEFORE phrase is used, the message or the message segment is represented on the communications device before the characters are repositioned vertically, according to the rules explained in the preceding list items.
- If the AFTER phrase is used, the message or the message segment is represented on the communications device after the characters are repositioned vertically, according to the rules explained in the preceding list items.
- If the PAGE phrase is specified, characters transmitted to the communications device are represented on the device before or after the device is repositioned to the next page (depending on the phrase used). If the PAGE phrase is specified but has no meaning in conjunction with a specific device, then advancing occurs as if you had specified either the BEFORE ADVANCING 1 LINE or AFTER ADVANCING 1 LINE phrase.

Section 15
Libraries

Libraries include the capabilities of the Inter-Program Communication (IPC) module described in Section 13, "ANSI Inter-Program Communication (IPC)." With the exception of the IPC capabilities described in Section 13, all features described in this section are extensions to ANSI X3.23-1974 COBOL.

A library program is a program that provides a procedure or a set of procedures that can be called by other programs. The procedure or set of procedures are in object code. Therefore, a library program can be thought of as a collection of library objects accessible through an entry point. Each library object is accessible to other programs, including other library programs.

The program that calls the library object is called the user program. Libraries cannot be called recursively; that is, a called program cannot call another program that, in turn, calls the original program.

Libraries can be created with, or called from, COBOL programs. Libraries created by COBOL are limited to one entry point.

For More Information

A detailed discussion of the use of libraries is found in the *Task Management Guide*; familiarity with this material is assumed in the following discussion.

Creating a Library

The compiler automatically creates a COBOL74 library program, provided that the following constraints are true of the library program:

- Only parameters that are allowed for libraries appear in the USING phrase of the PROCEDURE DIVISION statement.
- No constructs occur in the programs that are incompatible with its use as a library.

COBOL library programs have a single entry point. The name of that entry point is PROCEDUREDIVISION, unless the library program contains a program name in the PROGRAM-ID clause of the IDENTIFICATION DIVISION and is compiled with the FEDLEVEL compiler control option set to 5. If such is the case, the name described in the PROGRAM-ID clause is used as the entry point name.

The following code makes a program ineligible for use as a library:

- A VALUE statement or a RECEIVED BY CONTENT clause on a data item listed in the USING clause of the PROCEDURE DIVISION header
- A USE AS EXTERNAL PROCEDURE directive in the DECLARATIVES SECTION of the PROCEDURE DIVISION

For information about specifying and evaluating levels of COBOL to measure compliance with U.S. Government COBOL standards, refer to "FEDLEVEL" in Section 17, "Control of the Compilation Process."

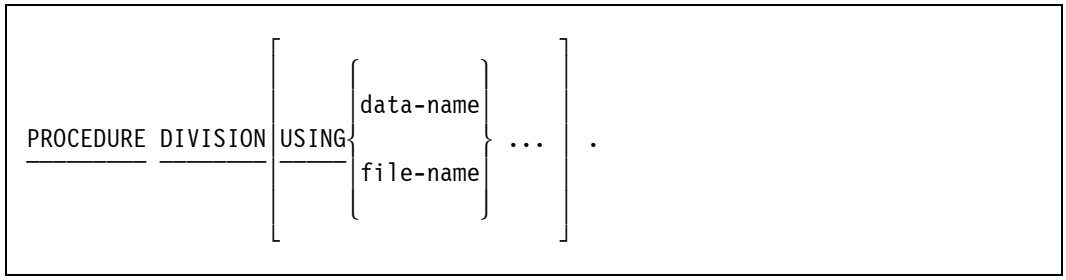
PROCEDURE DIVISION Header in Library Program

Parameters are optional in libraries; however, if parameters are used, the library program must identify them with a USING clause in the PROCEDURE DIVISION.

In a COBOL74 library, the USING clause of the PROCEDURE DIVISION header can specify the following three types of parameters:

- Data items passed by reference
- Data items passed by reference with type conversion
- Files passed by reference

The format of the header is as follows:



Explanation of Format

The data-name must be defined in the LINKAGE SECTION of the program in which the PROCEDURE DIVISION header occurs must have a 01-level or a 77-level number, and must not be a redefined item.

The file-name must be defined in the FILE SECTION of the program.

Rules for Parameters

Parameters to the library entry point can be any of the following types:

- COMP, 01, 77
- BINARY, 01, 77
- DISPLAY, 01
- DOUBLE
- FILE
- INDEX
- INTEGER (COMP)
- REAL
- STRING (DISPLAY)

Parameters of the form INTEGER (COMP) provide a method to receive parameters of type INTEGER. Parameters of the form STRING (DISPLAY) provide a method to receive parameters of type STRING.

Each of the data items in the USING phrase must be defined as 01 or 77 level, and must not redefine another data item.

All data items that are used as parameters either in a COBOL74 library or in a program that calls a COBOL74 library are treated as by-reference parameters.

Any program that declares parameters RECEIVED BY CONTENT are not library capable. The only way for a by-value parameter to be used either in the IPC module or in the library extension is with the type INTEGER conversion in the library mechanism.

When a user program and a library program are both written in COBOL74, the data types of the parameters must be the same. When the user program and the library programs are written using different compilers, the data types must correspond to one another. Because of the constraints imposed by the IPC module, parameter matching has special rules for some data items. These special rules are shown in Table 15–1.

Table 15–1. Parameter Matching for Data Items Requiring Special Rules

COBOL74 Data Type	General Data Type
01-level COMP or INDEX item	EBCDIC array
77-level COMP or INDEX item	HEX array
01-level BINARY item	INTEGER array
77-level BINARY item with less than 12 digits	INTEGER variable
77-level BINARY item with 12 or more digits	DOUBLE variable
77-level REAL item	REAL variable
01-level REAL item	REAL arrays
All other cases	Parameter-matching conventions follow the declared usage of the item.

There is an important distinction for BINARY, REAL, and DOUBLE items between those declared at the 01-level and those declared at the 77-level. The 01-level items are unconditionally treated as arrays, whereas the 77-level items are treated as words. Thus, a 01-level REAL item is described in the library interface as a REAL ARRAY BY REFERENCE, whereas a 77-level REAL item is described as a REAL BY REFERENCE.

A COBOL74 program receiving a lower-bound parameter from another program must declare an extra parameter, a 77-level PIC 9(11) BINARY item. This item name must appear in the USING clause immediately following the lower-bound receiving item. In addition, the LOWER-BOUNDS clause must not be declared for the receiving item.

Exiting a Library

A COBOL library program must have an EXIT PROGRAM statement as a means of exiting and returning to the user program. A STOP RUN statement in a library program ends both the library and the calling program. A STOP literal statement in a library program has no effect on the program and is ignored at execution time.

The EXIT PROGRAM statement causes program control to be returned to the user program at the statement following the statement that called the library.

The general format of this statement is as follows:

```
EXIT PROGRAM .
```

Explanation of Format

An execution of an EXIT PROGRAM statement in a COBOL74 library causes control to be passed from the library to the user program. Execution of an EXIT PROGRAM statement in a program that is not called causes the program to act as if the statement were an EXIT statement.

The EXIT PROGRAM statement must appear in a sentence by itself.

The EXIT PROGRAM sentence must be the only sentence in the paragraph.

For More Information

For general information about the EXIT statement, refer to “EXIT” in Section 9, “PROCEDURE DIVISION Statements.”

Securing a Library

Compiler locking code is provided in the library to ensure data integrity. Because data is global to both the user program and the entry point, and because library parameters must remain global, the compiler restricts the use of a COBOL library to one user at a time. You must wait until the preceding user is finished before you can enter the library.

Therefore, if a COBOL shared library attempts either directly or indirectly to call itself, the attempt waits for the call that is already in process to complete first. In such instances, the program cannot continue execution and the user must terminate it manually.

If a library program is called by user programs written in other languages and the library is declared with the SHARING compiler control option equal to PRIVATE, then the LIBRARYLOCK option should be set to TRUE to ensure data integrity.

For More Information

- For information on providing compiler locking code to maintain data integrity for private libraries, refer to “LIBRARYLOCK” in Section 17, “Control of the Compilation Process.”
- For information on the ways in which a program can be shared when it is called as a library, refer to “SHARING” in Section 17, “Control of the Compilation Process.”

Referring to a Library

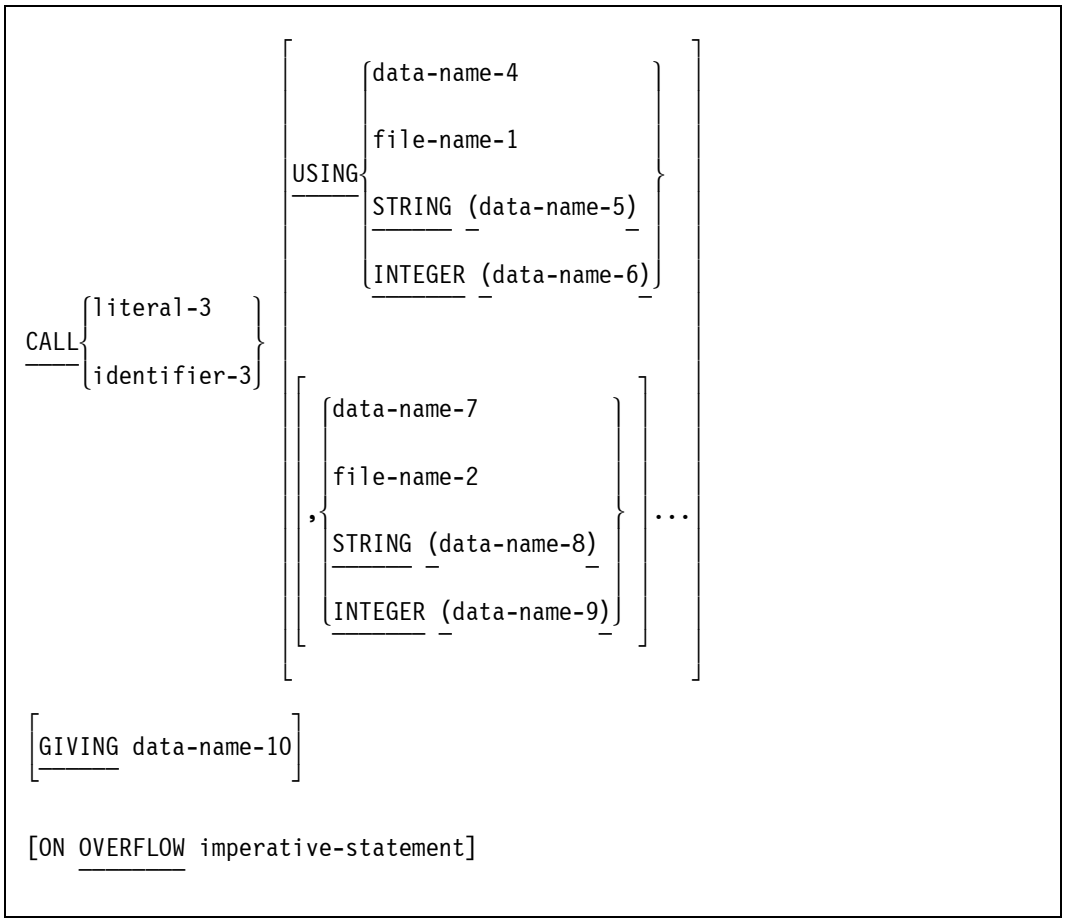
A user program can refer to a library program by using the CALL statement or the CANCEL statement. The CALL statement causes program control to pass from the user program to the specified entry point of the library. The CANCEL statement requests that the operating system terminate the library.

Library programs written in COBOL or COBOL74 have a single entry point, named PROCEDUREDIVISION by default. If a program being called has a PROGRAM-ID clause in the IDENTIFICATION DIVISION and has the FEDLEVEL compiler control option set to 5, the name appearing in the PROGRAM-ID clause is the entry point name. In all other cases, the entry point to COBOL and COBOL74 libraries is PROCEDUREDIVISION.

Other languages have specific syntax for the declaration of entry points. The appropriate entry-point-name must be used for calling such libraries.

CALL Statement for Libraries

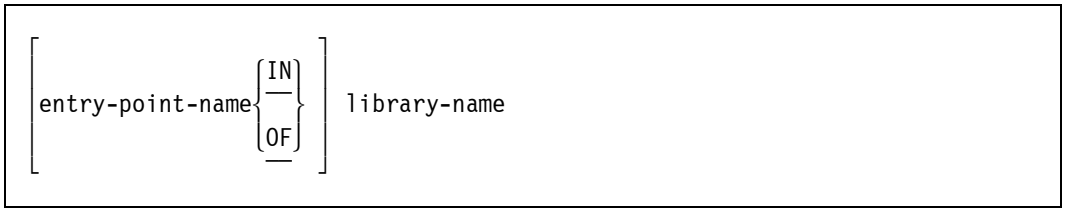
The library being called can be described in different ways depending on whether the library is called with a literal or with an identifier, and depending on whether the LIBACCESS attribute is assigned the value BYTITLE (the default) or the value BYFUNCTION. Refer to “Library Attributes” later in this section for details about the LIBACCESS attribute.



Explanation of Format

literal-3

The contents of literal-3 can be as follows:



The entry-point-name is the name of the entry point in the library or the function being called. In IPC calls, this entry-point-name is PROCEDUREDIVISION by default.

The library-name is a library-title if the LIBACCESS attribute is BYTITLE, and it is a function-name if the LIBACCESS attribute is BYFUNCTION.

A library-title is the object code file-name of the library that contains the entry point. A library-title is meaningful only if the LIBACCESS attribute of the library program is BYTITLE (the default value). A library-title can contain a usercode as its first directory node, but it cannot contain a family name (for example, ON DISK).

A function-name is the name by which an established system library is made available to users. A function-name is meaningful only if the LIBACCESS attribute of the library program is BYFUNCTION. A function-name is limited to a maximum of 17 numeric, uppercase, alphabetic characters and must not contain slashes (/) or other special characters.

identifier-3

Identifier-3 can only contain a library-name. The entry-point-name in a *CALL identifier-3* statement is always PROCEDUREDIVISION for compatibility with the ANSI-74 COBOL Inter-Program Communication (IPC) module.

Note: *In addition to the restrictions for the contents of identifier-3, a* CALL identifier-3 *statement is always significantly less efficient in execution-time performance than the equivalent* CALL literal-3 *statement. The execution of a* CALL identifier-3 *statement requires that a library template be built before the call to the library entry point can be executed, and also that the template be destroyed immediately after control is returned to the library. Before the template can be destroyed, the library must be delinked. No linkage is maintained between the calling program and the library between*
CALL identifier-3 *statements to that library, regardless of the SHARING option setting of the library.*

See the Task Management Programming Guide *for further details on library templates and library linking and delinking. See "Effect of Library State on a CALL Statement" later in this section for details on the impact of the SHARING option as declared in the library.*

USING

The USING clause identifies data items and files passed as parameters to the library procedure and also enables you to manipulate some parameter types with the STRING and INTEGER type conversion mechanisms.

The usage of data items used as parameters to the library entry point can be BINARY, COMP, DISPLAY, DOUBLE, INDEX, or REAL.

The *STRING (data-name-5)* phrase converts the DISPLAY item specified by data-name-5 to a string representation so that the item can be passed to a string parameter of a library object. The string data type cannot be represented directly in COBOL or COBOL74, but appears in other languages such as ALGOL.

The *INTEGER (data-name-6)* phrase converts the COMP item specified by data-name-6 to an integer representation. The compiler generates the object code to pass the numeric value of the packed field in a temporary variable as an integer by reference, and to store the value of that temporary variable (in the event that the library has modified it) back into the original parameter. Data-name-6 must be the name of an elementary 01-level or 77-level data item declared USAGE COMPUTATIONAL.

A parameter in a USING clause cannot redefine another item either implicitly in the FILE SECTION or explicitly with a REDEFINES clause. 77-level items that have been redefined can yield unexpected results. For example, referring to a 77-level item in a

REDEFINES clause can cause the item to be treated as if it had been declared as a 01-level item.

GIVING (Extension to ANSI X3.23-1974 COBOL)

The GIVING clause allows a typed procedure to be called as a library and stores the value of the procedure in data-name-10.

A library written in languages other than COBOL and COBOL74 can be a typed procedure, and thereby return a value.

The GIVING clause specifies the type of procedure and the destination for the value that is returned from the procedure. The procedure type is needed for library linkage matching.

Data-name-10 must specify an elementary numeric item. The rules regarding the relationship between the usage of data-name-10, the type of procedure, and the result returned by the procedure are described in the following table.

Usage of data-name-10	Type of Procedure and Result Returned
77-level USAGE REAL	The user program expects the library procedure to be a REAL PROCEDURE and the result to be of type REAL.
77-level USAGE DOUBLE	The user program expects the library procedure to be a DOUBLE PROCEDURE and the result to be of type DOUBLE.
All other cases	The user program expects the library procedure to be an INTEGER PROCEDURE. The result returned is converted at execution time into the proper form for storing in data-name-10 according to the rules for the COMPUTE statement.

A *CALL identifier-3* statement with a GIVING clause–although a supported feature–is of marginal use for the following reasons:

- A typed entry point cannot be declared in COBOL or COBOL74.
- The only entry point accessible with such a call is PROCEDUREDIVISION, yet the program containing the library cannot be written in COBOL or COBOL74, which are the only languages in which such an entry-point-name is expected.

Effect of Library State on a CALL Statement

User programs can be dependent on a library being in its initial or noninitial state on a given call to a library. A library is in its initial state the first time any user successfully links to it after the library begins executing.

The state of a library depends on the environment in which the library is used and on the value of the SHARING compiler control option in the library itself. Neither of these factors is under the control of the individual user programs linking to the library.

Table 15–2 shows how the SHARING option value affects the initial state of the library.

Table 15–2. Effect of SHARING Option and Library Initial State on CALL Statement

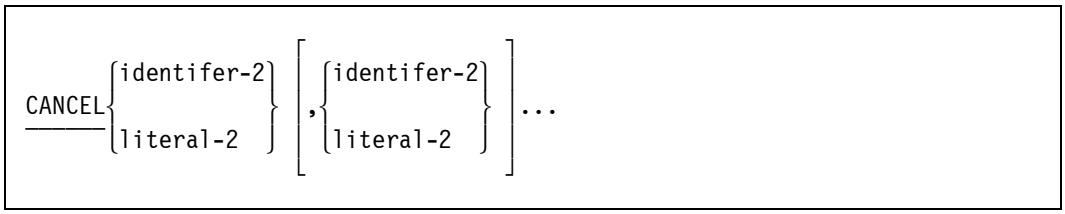
SHARING Option Value	Effect on Library Initial State
SHAREDBYRUNUNIT	The library is in its initial state at the following times: • The first time any program in the run unit calls the library • The first time any program in the run unit calls the library after a CANCEL statement has been successfully performed on the library from within the run unit
PRIVATE	The library is in its initial state at the following times: • Each time the program uses a *CALL identifier-3* statement to call the library. This format causes the library to start and complete execution every time the format is used. • The first time the program uses a *CALL literal-3* statement to call the library. • The first time the program calls the library after it has successfully canceled the library.
SHAREDBYALL	A temporary library is in its initial state at the following times: • The first time any program on the system calls the library • The first time any program on the system calls the library after the operating system has determined that no more users exist and has therefore terminated the library A permanent library is in its initial state the first time any program on the system calls the library.
DONTCARE	Because the state of a DONTCARE library depends on the mechanism chosen by the operating system for that particular execution of the library, you should avoid writing programs that depend on the state of a DONTCARE library.

For More Information

- See "TEMPORARY" in this section for details on temporary or permanent libraries.
- SHAREDBYRUNUNIT libraries ensure that the library state conforms to ANSI-74 IPC requirements. See Section 13, "ANSI Inter-Program Communication (IPC)," for details.

CANCEL Statement for Libraries

The CANCEL statement, implemented to meet the requirements of the ANSI-74 COBOL Inter-Program Communication (IPC) module, causes the operating system to disassociate the specified library from the calling program and attempt to allow the library to end.



Explanation of Format

The CANCEL statement informs the operating system that the linkage between the calling program and the library is to be severed.

The contents of the literal or the identifier used in a CANCEL statement must be a valid library-name (as was discussed earlier in this section).

Identifier-1, identifier-2, and so on must be defined as alphanumeric data items.

Literal-1, literal-2, and so on must be nonnumeric literals.

Effect of Library Initial State on a CANCEL Statement

Because the CANCEL statement is implemented to meet the requirements of the ANSI-74 Inter-Program Communication (IPC) module, its operation and its effect on the initial state of the library depend on the value of the SHARING compiler control option for the library. These dependencies are described in Table 15–3.

Table 15–3. SHARING Option and Library Initial State Effect on CANCEL Statement

SHARING Value	Effect on Library Initial State
SHAREDBYRUNUNIT or PRIVATE	The library is sent to end-of-task (EOT). The repeated execution of CALL and CANCEL statements to the same library is costly because of the overhead involved in task termination and reinitiation. Carelessness in the use of the CANCEL statement when the SHARING option is equal to PRIVATE SHAREDBYRUNUNIT can lead to extremely inefficient application systems.
SHAREDBYALL	The library is not sent to EOT. Instead, the operating system issues an execution-time warning stating that the library was delinked and not terminated. When the next CALL statement for the library is executed, the library is in the state it was in when it last executed an EXIT PROGRAM statement. You should avoid using the CANCEL statement for libraries with the SHARING option equal to SHAREDBYALL because of the overhead associated with the execution-time warning.
DONTCARE	The operating system determines the effect of the CANCEL statement. Because the result is unpredictable, you should avoid using the CANCEL statement for libraries with the SHARING option equal to DONTCARE.

Library Attributes

The following paragraphs describe the types of attributes that are valid for libraries and the way in which library attributes can be changed.

Types of Library Attributes

Libraries, like files, have attributes that can be set programmatically. The following five attributes are associated with libraries:

- FUNCTIONNAME
- INTNAME
- LIBACCESS
- LIBPARAMETER
- TITLE

All the attributes except the LIBACCESS attribute are EBCDIC attributes of type STRING.

The LIBACCESS attribute is a mnemonic-type attribute that has two possible values: BYTITLE (the default) and BYFUNCTION. The effect the values of the LIBACCESS attribute have on the TITLE and the FUNCTIONNAME attributes are shown in Table 15–4.

The default value for the INTNAME attribute is the last node of the title of the file being called. For example, given a statement of the form CALL "X OF A/B/C/D", the INTNAME for the library referenced by this CALL statement is D.

Because of this convention, multiple libraries referenced in a single program can have the same INTNAME. Therefore, always make sure that the attributes of the correct program are being modified, especially when you use file attribute modification mechanisms other than those documented under the CHANGE ATTRIBUTE statement.

Note: *Altering library attributes of COBOL74 programs with file attribute modification mechanisms other than the CHANGE ATTRIBUTE statement is not recommended.*

Table 15–4. Effects of Setting the LIBACCESS Attribute

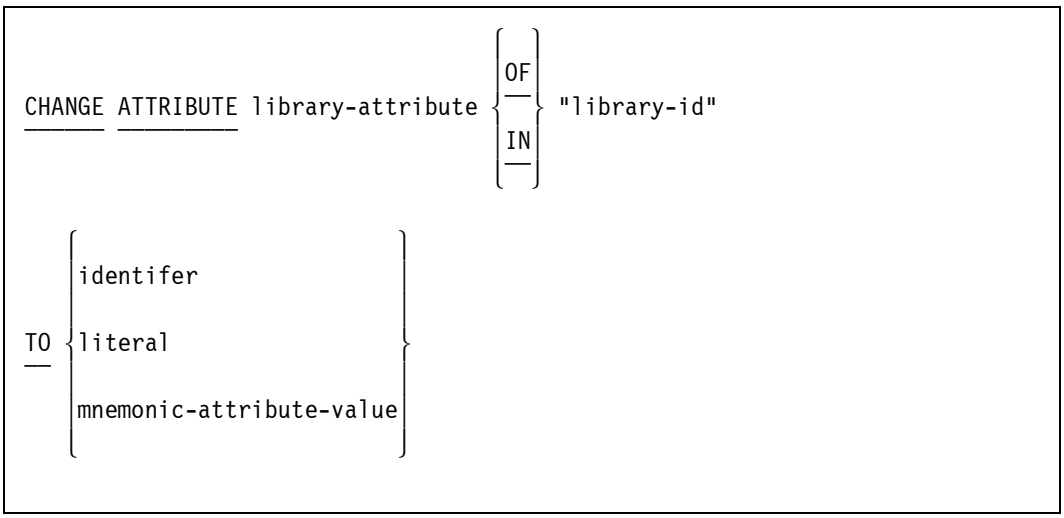
LIBACCESS Attribute Value	Effect
BYFUNCTION	Setting the LIBACCESS attribute to the value BYFUNCTION has the following effects: • The TITLE attribute has no effect. • The FUNCTIONNAME attribute is used to locate the appropriate code file in the operating system library function table. • The code file associated with the FUNCTIONNAME attribute is used.
BYTITLE	The TITLE attribute is used to access the code file.

For more information about library attributes, see the *System Software Utilities Operations Reference Manual.*

CHANGE ATTRIBUTE Statement for Libraries

Library attributes can be changed dynamically by the user program before the first attempt to link to the library; they cannot be changed after linkage is established.

The CHANGE ATTRIBUTE statement is used to change library attributes. The format of this statement is as follows:



Explanation of Format

The library-id specifies the library that has changed attributes and is the same format as the run-time library-id in Format 3 of the CALL statement. The identifier must be an alphanumeric data item with USAGE IS DISPLAY. The literal must be a nonnumeric literal. The mnemonic-attribute-value, used only with the LIBACCESS attribute, must be either BYTITLE or BYFUNCTION.

Library attributes cannot be changed after a CALL statement is issued for the library unless the library has been canceled.

If no CHANGE ATTRIBUTE statement is present, the INTNAME and TITLE attributes are set to the name specified by library-name in the CALL statement, and the LIBACCESS attribute is set to BYTITLE. The FUNCTIONNAME attribute has no effect on a library whose LIBACCESS attribute is set to BYTITLE.

If a user program sets the value of the LIBACCESS attribute to BYFUNCTION, but does not set the value of the FUNCTIONNAME attribute before it calls the library program, then the user program sets the value of the FUNCTIONNAME attribute to the code file-name in the CALL statement. Note that the TITLE attribute has no effect on a library whose LIBACCESS attribute is set to BYFUNCTION.

For information on passing program control to a library and on terminating a library, refer to “Referring to a Library” in this section.

Library Compiler Control Options

The LIBRARYLOCK, SHARING, and TEMPORARY compiler control options control the way the library is used.

LIBRARYLOCK

If a library is called only by COBOL74 programs and the library is declared with the SHARING option equal to PRIVATE, then the LIBRARYLOCK option can remain FALSE. Since the library services only one user at a time, the system preserves data integrity without incurring the additional cost of using compiler locking code.

If a library is called by programs written in other languages and the library is declared with the SHARING option equal to PRIVATE, then the LIBRARYLOCK option should be set to TRUE to ensure data integrity.

If the SHARING option is not PRIVATE, the setting of the LIBRARYLOCK option has no effect on the generation of library locking code.

For More Information

Refer to "LIBRARYLOCK" in Section 17, "Control of the Compilation Process" for more information about this option.

SHARING

The creator of the library specifies the allowed simultaneous uses of a library by using the SHARING option. Table 15–5 shows the various option values and their meanings.

Table 15–5. Meanings of SHARING Option Values

Value	Meaning
PRIVATE	A separate instance of the library is started for each user.
SHAREDBYALL	All simultaneous users share the same instance of the library.
SHAREDBYRUNUNIT	All programs in the process family use the same instance of the library. SHAREDBYRUNUNIT is the default value.
DONTCARE	The operating system optimizes the library sharing.

Users of a library can be restricted not only through the different library-sharing specifications, but also through the normal file-access restrictions of the system.

Only libraries with the SHARING option SHAREDBYRUNUNIT (the default value) or PRIVATE can be canceled. Attempts to cancel other types of libraries are completely ineffective and result only in a warning message.

For More Information

- Refer to "SHARING" in Section 17, "Control of the Compilation Process," in this section for more information about this option.
- For information on dissociating a library from a calling program, refer to "CANCEL Statement for Libraries" in "Referring to a Library" in this section.

TEMPORARY

The TEMPORARY compiler control option, in conjunction with the SHARING option, determines whether a library created by the COBOL compiler functions as a temporary or a permanent library. The effectiveness of the TEMPORARY option depends on the value of the SHARING option. A library can be made permanent only if the SHARING option has been specified as SHAREDBYALL or DONTCARE.

A temporary library is created if the value of the TEMPORARY option is TRUE and the SHARING option has the value SHAREDBYALL or DONTCARE. A temporary library terminates if no users are referencing it. If the library is called again at a later time, a new copy is initiated.

A permanent library is created if the value of the TEMPORARY option is FALSE (the default) and the SHARING option has the value SHAREDBYALL or DONTCARE. A permanent library remains in the mix with no users, and its procedures are available for any later programs. A permanent library is terminated only by direct operator action.

Table 15–6 summarizes the combinations of SHARING and TEMPORARY compiler control options values that create temporary or permanent libraries.

Table 15–6. SHARING and TEMPORARY Compiler Control Option Combinations

SHARING Option Value	TEMPORARY Option TRUE	TEMPORARY Option FALSE
PRIVATE	Temporary library	Temporary library
SHAREDBYRUNUNIT	Temporary library	Temporary library
SHAREDBYALL	Temporary library	Permanent library
DONTCARE	Temporary library	Permanent library

Program Samples of Referring to a Library

The following program samples show various ways of calling a library, canceling a library, and setting of library attributes.

Example 1

The first call in Example 15–1 references OBJECT/A/COBOL/LIB, the second and third calls reference OBJECT/ANOTHER/COBOL/LIB, and the fourth call references a library with the internal name LIB and the title OBJECT/SAMPLE/DYNAMICLIB.

The fourth call enables multiple references to use LIB to reference the library. Also, the fourth call shows that the CHANGE ATTRIBUTE TITLE statement might prove useful in applications in which the library title is not known at compile time and when it is desirable to circumvent the performance issues and implementation constraints of the *CALL identifier-3* statement.

```
000100 IDENTIFICATION DIVISION.
000200 ENVIRONMENT DIVISION.
000300 DATA DIVISION.
000400 WORKING-STORAGE SECTION.
000500 01  PARAM PIC 9(21) COMP.
000600 PROCEDURE DIVISION.
000700 ONLY-HEADER.
000800      CALL "PROCEDUREDIVISION OF OBJECT/A/COBOL/LIB"
000900          USING PARAM.
001000      CANCEL "OBJECT/A/COBOL/LIB".
001100      CHANGE ATTRIBUTE TITLE OF "OBJECT/A/COBOL/LIB" TO
001200          "OBJECT/ANOTHER/COBOL/LIB".
001210*
001220*THE FOLLOWING IS AN EXAMPLE OF AN IPC CALL.
001230*
001300      CALL "OBJECT/A/COBOL/LIB" USING PARAM.
001350*
001360*THE FOLLOWING IS AN EXAMPLE OF A LIBRARY CALL.
001370*
001400      CALL "PROCEDUREDIVISION OF OBJECT/A/COBOL/LIB"
881500          USING PARAM.
001600      CHANGE ATTRIBUTE TITLE OF "LIB" TO
001700          "OBJECT/SAMPLE/DYNAMICLIB".
001800      CALL "ENTRYPOINTNAME IN LIB" USING PARAM.
001900      STOP RUN.
```

Example 15–1. Calling a Library

Example 2

Example 15–2 shows calling a library by function. Only libraries that accept parameters by reference can be used with COBOL74 programs. The example also shows access to the arithmetic function RANDOM. The example assumes that SYSTEM/GENERALSUPPORT is installed as GENERALSUPPORT in the library function table of the operating system.

```
002000IDENTIFICATION DIVISION.
004000ENVIRONMENT DIVISION.
006000DATA DIVISION.
008000WORKING-STORAGE SECTION.
01000077  SEED PIC 9(11) USAGE BINARY.
01200077  RANDOM-RESULT USAGE REAL.
014000PROCEDURE DIVISION.
016000ONLY-PARAGRAPH.
018000    CHANGE ATTRIBUTE LIBACCESS OF "GENERALSUPPORT"
020000        TO BYFUNCTION.
022000    CALL "RANDOM OF GENERALSUPPORT"
024000        USING SEED
026000        GIVING RANDOM-RESULT.
026200    CHANGE ATTRIBUTE FUNCTIONNAME OF "SYNONYM1"
026400        TO "GENERALSUPPORT".
026600    CHANGE ATTRIBUTE LIBACCESS OF "SYNONYM1"
026800        TO BYFUNCTION.
027000    CALL "RANDOM OF SYNONYM1"
027200        USING SEED
027400        GIVING RANDOM-RESULT.
028000    STOP RUN.
```

Example 15–2. Calling a Library by Function

Example 3

Family substitution can occur when the library is linked by title. In this situation, it is desirable to link by function to an initialized library or to control the family substitution as shown in Example 15–3.

```
001000*THE FOLLOWING SEQUENCE LINKS A LIBRARY TO A CODE FILE ON "DISK"
001500*WHEN FAMILY SUBSTITUTION KEEPS THE LIBRARY FROM LINKING.
002000 MOVE ATTRIBUTE FAMILY OF MYSELF TO SAVEFAM.
004000 CHANGE ATTRIBUTE FAMILY OF MYSELF TO ".".
006000 CALL "INITIALIZE IN LIB1".
008000 CHANGE ATTRIBUTE FAMILY OF MYSELF TO SAVEFAM.
```

Example 15–3. Substituting a Family Specification

For More Information

Refer to the FAMILY attribute in the *Task Attributes Reference Manual* for information.

Example 4

The TIMEZONENAME_COB procedure returns the text name or abbreviation for a given time zone number. Example 15–4 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the TIMEZONENAME_COB procedure. The declarations identify the category of data-items required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

```
000100 IDENTIFICATION DIVISION.
000200 ENVIRONMENT DIVISION.
000300 DATA DIVISION.
000400 WORKING-STORAGE SECTION.
000500 77 WHICH   PIC 9(11)  USAGE BINARY  VALUE 1.
000600 77 TZ      PIC 9(11)  USAGE BINARY  VALUE 1.
000700 77 RESULT  PIC 9(11)  USAGE BINARY.
000800 01 LANGA   PIC X(64).
000900 01 TZA     PIC X(64).
001000 PROCEDURE DIVISION.
001100 ONLY-PARAGRAPH.
001200     CHANGE ATTRIBUTE LIBACCESS OF "MCPSUPPORT"
001300         TO BYFUNCTION.
001400     CALL "TIMEZONENAME_COB OF MCPSUPPORT"
001500         USING WHICH, TZ, LANGA, TZA
001600         GIVING RESULT.
001700     DISPLAY TZA.
001800     STOP RUN.
```

Example 15–4. Requesting a Time Zone Name

Following is a brief description of the parameters used in the example to request the time zone name for time zone number 1:

- WHICH is an integer passed to the procedure. If it contains a value of 1, the procedure returns the time zone name. If it contains a value of 2, the procedure returns the time zone abbreviation.
- TZ is an integer passed to the procedure. It contains a time zone number in the range 1 through 88.
- LANGA is an EBCDIC character array passed to the procedure. It contains the MLS language in which the time zone name or abbreviation is to be returned. The language name must be terminated by a blank character.
- If the first character in LANGA is a blank, a period (.) or a null (binary zeros), the time zone name or abbreviation is returned in the language of the task or the SYSTEMLANGUAGE.

- TZA is an EBCDIC character array in which the time zone name or abbreviation is returned from the procedure. The procedure blanks out the entire array, and then it stores the time zone name or abbreviation starting at the first character of the array.
- RESULT is returned as the integer value of the procedure. It indicates whether an error occurred during the execution of the procedure. TIMEZONENAME_COB returns the same error values as the MESSAGESEARCHER statement.

Example 5

The CURRENT_DATE procedure in the GENERALSUPPORT library returns a 21-character alphanumeric value that represents the calendar date, time of day, and local time differential factor provided by the system on which the function is evaluated. This procedure provides functionality similar to the CURRENT-DATE function in COBOL85.

Example 15–5 shows the way to pass a star-bounded EBCDIC array parameter by using the WITH LOWER-BOUNDS phrase:

```
000100  IDENTIFICATION DIVISION.
000200  PROGRAM-ID.
000300      YEAR2000.
000400  ENVIRONMENT DIVISION.
000500  CONFIGURATION SECTION.
000600  SOURCE-COMPUTER.
000700      MICROA.
000800  OBJECT-COMPUTER.
000900      MICROA.
001000  INPUT-OUTPUT SECTION.
001100  FILE-CONTROL.
001200  DATA DIVISION.
001300  WORKING-STORAGE SECTION.
001400  01  DATE-ARRAY PIC X(21) DISPLAY WITH LOWER-BOUNDS.
001500  PROCEDURE DIVISION.
001600  S1 SECTION.
001700  P1.
001800     CHANGE ATTRIBUTE LIBACCESS OF "GENERALSUPPORT" TO BYFUNCTION.
001900     CALL "CURRENT_DATE OF GENERALSUPPORT"
002000          USING DATE-ARRAY.
002100     DISPLAY DATE-ARRAY.
002200     GO TO FINI.
002300 FINI.
002400     STOP RUN.
```

Example 15–5. Calling the CURRENT_DATE Procedure

Returned Value

The character positions returned, numbered from left to right, appear as follows:

Table 15–7. Alphanumeric Value Returned by the CURRENT_DATE Procedure

Character Positions	Contents
1 through 4	Four numeric digits for the year in the Gregorian calendar.
5 through 6	Two numeric digits for the month of the year, in the range 01 through 12.
7 through 8	Two numeric digits for the day of the month, in the range 01 through 31.
9 through 10	Two numeric digits for the hours past midnight, in the range 00 through 23.
11 through 12	Two numeric digits for the minutes past the hour, in the range 00 through 59.
13 through 14	Two numeric digits for the seconds past the minute, in the range 00 through 59.
15 through 16	Two numeric digits for the hundredths of a second past the second, in the range 00 through 99.
17	See Table 15–8.
18 through 21	See Table 15–9.

Character position 17 can contain one of three characters as shown in Table 15–8.

Table 15–8. Meaning of Character in Position 17

IF character 17 is a . . .	THEN . . .
Minus sign (–-)	The local time indicated in the previous character positions is behind Greenwich Mean Time.
Plus sign (+)	The local time indicated in the previous character positions is the same as or ahead of Greenwich Mean Time.
Zero (0)	The system on which this value is evaluated does not have the facility to provide the local time differential factor.

Character positions 18 through 21 contain the time in hours and minutes relative to Greenwich Mean Time. The meaning of the time value contained in these positions depends upon the character in position 17 as shown in Table 15–9.

Table 15–9. Meaning of Characters in Positions 18–21

IF character 17 is a . . .	THEN . . .
Minus sign (–)	Character positions 18 and 19 contain a numeric value in the range 00 through 24. This value indicates the number of hours that the reported time is behind Greenwich Mean Time. Character positions 20 and 21 contain a numeric value in the range 00 through 59. This value indicates the number of minutes that the reported time is behind Greenwich Mean Time.
Plus sign (+)	Character positions 18 and 19 contain a numeric value in the range 00 through 24. This value indicates the number of hours that the reported time is ahead of Greenwich Mean Time. Character positions 20 and 21 contain a numeric value in the range 00 through 59. This value indicates the number of minutes that the reported time is ahead of Greenwich Mean Time.
Zero (0)	Character positions 18 through 21 contain zeros.

Example of Output

The following is an example of a value returned by the CURRENT_DATE procedure. Because character position 17 is a minus sign, the digits 0700 indicate the reported time is 7 hours and 0 (zero) minutes behind Greenwich Mean Time.

```
1995062813195795-0700
```

Section 16
Internationalization

Internationalization refers to the software, firmware, and hardware features that enable you to develop and run application systems that can be customized to meet the needs of a specific language, culture, or business environment. The internationalization features provide support for several character sets, different international business and cultural conventions, extensions to data communications protocols, and the ability to use one or more natural languages concurrently.

This section describes the internationalization features you can use to customize an application for the language and conventions of a particular locality. Using these features to write or modify an application is termed *localization*. The MultiLingual System (MLS) environment enables you to process information to localize your applications. Some of the localization methods included in the MLS environment include translating messages to another language, choosing a particular character set to be used for data processing, and defining date, time, number, and currency formats for a particular business application.

In addition to the information described in this section, refer to the *Multilingual System Administration, Operations, and Programming Guide* for information. The *MultiLingual System Administration, Operations, and Programming Guide* provides definitions for and detailed information about ccsversions, character sets, languages, and conventions. It also describes procedures for setting system values for the internationalization features.

Accessing the Internationalization Features

You can use the following two methods separately or together to access the internationalization features. Both of these methods are described later in this section.

- COBOL74 provides language syntax that enables you to localize your program. For example, when you specify a particular ccsversion in your program, the compiler uses the collating sequence associated with the ccsversion for alphanumeric comparisons.
- The system the CENTRALSUPPORT system library, which contains procedures for localizing a program. You can access the library procedures by using the CALL verb. When a call occurs, input parameters describe the type of information that is needed or the action that is to be performed. Output parameters are returned with the result of the procedure call. For example, you can call the CNV_FORMATTIME_COB procedure to format the time according to a language and convention that you specify.

In your program, you must designate that you want to use the internationalization features; otherwise, your program will not access them. In other words, a program is not affected by the features described in this section unless the program specifically invokes them. Any programs that already exist and do not invoke internationalization features are not affected by the features.

Using the Ccsversion, Language, and Convention Default Settings

The program can choose the specific ccsversion, language, and convention settings that it needs by setting the input parameters to a procedure. The system has default settings for the internationalization features at other levels. The default settings can also be accessed by the program. See "Understanding the Hierarchy for Default Settings" later in this section for information on the available levels and on the features supported at each level.

The system default settings can be determined by using one of the following two methods:

- The program calls the CENTRALSTATUS procedure.
- A system administrator, a privileged user, or a user who is allowed to use the system console can use MARC or the SYSTEMOPTIONS system command. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* or the *MCP/AS Menu-Assisted Resource Control (MARC) Operations Guide* for the instructions to display the default ccsversion, language, or convention with MARC.

The system default settings are as follows:

Feature	Default
Ccsversion	ASeriesNative
Language	English
Convention	ASeriesNative

Before you change the default settings for localization, you must consider the feature and the level at which the feature is defined. As an example, the ccsversion can be changed only at the system operations level. A program can avoid making specific settings by taking advantage of the default settings. For example, if the system-defined ccsversion is France, the language is Francais, and the convention is FranceListing, the program can use those default settings without coding the settings within the program. A program can use the default settings with the use of predefined values as input parameters. These parameters tell the procedure to use the current default setting. See "Input Parameters" later in this section for specific information on those parameters.

To use the system default ccsversion, you must specify the CCSVERSION phrase in the SPECIAL-NAMES paragraph and omit the literal-1 option. The clause *alphabet-name IS CCSVERSION* identifies the collating sequence associated with the alphabet-name as the system default collating sequence.

You can specify five date or time formats by using the TYPE clause in the DATA DIVISION. You can use the system default language and convention by omitting the USING phrase of the TYPE clause.

For More Information

- See "SPECIAL-NAMES Paragraph" in Section 5, "ENVIRONMENT DIVISION," for details on the CCSVERSION phrase.
- Refer to "TYPE" in Section 7, "DATA DIVISION," for more information about using this clause for date and time editing.

Understanding the Hierarchy for Default Settings

The default settings for the internationalization features can be established at the following levels:

Task	Established at task initiation
Program	Established at compile time
Session	Handled by MARC or CANDE commands or by programs which support sessioning
Usercode	Established in the USERDATAFILE file
System	Established with a system or MARC command

There is a priority associated with these levels. A setting at the task level overrides a setting at the program level. A setting at the program level overrides a setting at the session level. A setting at the session level overrides a usercode or system level setting and so on. A language and convention can be established at any level, but the ccsversion can be established only at the system level.

Two task attributes enable you to change the language, the convention, or both. These attributes are the LANGUAGE and CONVENTION task attributes. By using these attributes, you have the option to select from among multiple languages and conventions when running a program. Information on the use of task attributes is provided in the *Task Attributes Reference Manual*.

The LANGUAGE task attribute establishes the language used by a program at run time.

The CONVENTION task attribute establishes the convention used by a program at run time. For example, an international bank might have a program to print bank statements for customers in different countries. This program could have a general routine to format dates, times, currency, and numerics according to the selected conventions. To print a bank statement for a French customer, this program could set the CONVENTION task attribute to FranceBureautique and process the general routine. For a customer in Sweden, the program could set the CONVENTION task attribute to Sweden and process the general routine.

As you code your program you can use the defaults in both the source code and the calls to the CENTRALSUPPORT library, or you can use the settings of your choice. The task level and system level are probably the most useful levels for your program. Because the language and convention features have task attributes defined, you can access or set these task attributes in your program.

Components of the MLS Environment

The following pieces of the MLS environment support different languages and cultures:

- Coded character sets
- Ccsversions
- Languages
- Conventions

Coded Character Sets and Ccsversions

A coded character set is a set of rules that establishes a character set and the one-to-one relationship between the characters of the set and their code values. The same character set can exist with different encodings. For example, the LATIN1-based character set can be encoded in an International Organization for Standardization (ISO) format or an EBCDIC format. Coded character sets are defined in the *MultiLingual System Administration, Operations, and Programming Guide*.

A coded character set name and number is given to each coded character set definition. This name or number may be used to set the INTMODE or EXTMODE file attribute value for a file. For details on these attributes, see the *File Attributes Reference Manual*.

A ccsversion is a collection of information necessary to apply a coded character set in a given country, language, or line of business. This information includes the processing requirements such as data classes, lower-to-uppercase mapping, ordering of characters, and escapement rules necessary for output. A ccsversion name and number is given to each unique group of information. This name and number may also be used to set the CCSVERSION file attribute for a file. For more information on these attributes, see the *File Attributes Reference Manual*.

Each system includes a data file, SYSTEM/CCSFILE, containing all coded character sets and ccsversions that are supported. You cannot choose a coded character set directly, but by choosing a ccsversion, you implicitly designate the default coded character set for your system.

Data can be entered and manipulated in only one coded character set and ccsversion at a time. Although there are many ccsversions that can be accessed, there is only one ccsversion active for the entire system at one time. This is called the system default ccsversion. All coded character set and ccsversion information can be accessed by calling CENTRALSUPPORT library procedures.

You can use any of the following ways to find out which coded character sets and ccsversions are available on the system:

- Look in the *MultiLingual System Administration, Operations, and Programming Guide*. Your system might have a subset of the ones defined in that guide.
- Use the MARC menus and screens or the system command SYSTEMOPTIONS. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* or the *System Commands Operations Reference Manual*.
- Call the CCSVSN_NAMES_NUM procedure.

You might want to refer to the *MultiLingual System Administration, Operations, and Programming Guide* for a complete understanding of ccsversions and the relationship of a coded character set and a ccsversion.

You must use language syntax to establish that a ccsversion is to be used in your program. To do this, specify a *PROGRAM COLLATING SEQUENCE* clause and an *alphabet-name is CCSVERSION literal-1* clause in the ENVIRONMENT DIVISION.

You can enter and manipulate data in any one particular coded character set and ccsversion. Although there are many ccsversions that can be accessed, there is only one ccsversion active for the entire system at one time. This is called the system default ccsversion. You can select a ccsversion to be used during the execution of your program by including the literal-1 option in the CCSVERSION clause. If you do not use the literal-1 option, the program uses the system default ccsversion.

Many of the procedures require the specification of a coded character set or ccsversion as an input parameter. A program can choose a specific coded character set or ccsversion by calling the procedure using the name or number of the coded character set or the ccsversion as an input parameter. A program can also use the system default setting by using predefined values as input parameters. See "Input Parameters" later in this section.

It is possible to use a different ccsversion in your program by changing the value of the literal-1 option. For example, by changing the value of literal-1, your program could process data in the ASeriesNative ccsversion and then process data in the Swiss ccsversion.

For more information about the CCSVERSION clause, see "SPECIAL-NAMES Paragraph," in Section 5, "ENVIRONMENT DIVISION."

Mapping Tables

A mapping table is used to map one group of characters to another group of characters or another representation of the original characters. Many CENTRALSUPPORT library procedures store coded character set and ccsversion information in ALGOL-type translate tables as a way of defining, processing, and mapping data. For example, a translate table can exist to translate lowercase characters to uppercase characters. It is not necessary to understand the layout of an ALGOL-type translate table because the table is usually not visible to your program. A description of translate tables is provided in the *ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

The internationalization procedures provide you with access to mapping tables that apply to data specified in coded character sets or to specified ccsversions. These mapping tables are as follows:

- Mapping data from one coded character set to another coded character set
- Mapping data from lowercase to uppercase characters
- Mapping data from uppercase to lowercase characters
- Mapping data from alternative numeric digits defined in a ccsversion to numeric digits in U.S. EBCDIC
- Mapping data from numeric digits in U.S. EBCDIC to alternative numeric digits defined in a ccsversion
- Mapping characters to their escapement values

You must use procedures from the CENTRALSUPPORT library to access these mapping tables or to process data using these tables. For example, you can use the CCSTOCCS_TRANS_TEXT procedure to translate data from one coded character set to another coded character set. You can use the VSNTRANS_TEXT procedure for the rest of the mapping tables.

See the *MultiLingual System Administration, Operations, and Programming Guide* for definitions of mapping tables for each coded character set and ccsversion.

Data Classes

A data class is a group of characters sharing common attributes such as alphabetic, upon which membership tests can be made. Some characters might not have a data class assigned to them. Many CENTRALSUPPORT library procedures store ccsversion information in ALGOL-type truthset tables as a way to define ccsversion data classes. A truthset is a method of storing the declared set of characters that defines a data class in ALGOL. It is not necessary to understand the layout of an ALGOL-type translate table because the table is usually not visible to your program. A description of translate tables is provided in the *ALGOL Reference Manual, Vol. 1.*

The internationalization features provide you with access to additional truthsets that apply to a ccsversion. These truthsets are as follows:

- Ccsversion alphabetic
- Ccsversion numeric
- Ccsversion graphics
- Ccsversion spaces
- Ccsversion lowercase
- Ccsversion uppercase

The alphabetic truthset contains those characters that are considered to be alphabetic for a specified ccsversion; the numeric truthset contains those characters that are considered to be numeric for a specified ccsversion, and so on.

The compiler automatically accesses the alphabetic truthset if you have specified a *PROGRAM COLLATING SEQUENCE* clause and an *alphabet-name IS CCSVERSION literal-1* clause in the ENVIRONMENT DIVISION. Then if you use the *identifier IS ALPHABETIC* clause, the compiler makes the class condition test sensitive to the ccsversion alphabetic data class.

You can use procedures from the CENTRALSUPPORT library to access these truthsets or to process data using these truthsets. For example, if a program manipulates an employee identification number such as 555962364, it might then need to verify that the text is or is not all numeric. The program can call the VSNINSPECT_TEXT CENTRALSUPPORT library procedure to compare the text to the numeric truthset. This procedure returns the information that the text is or is not all numeric.

For More Information

Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for definitions of ccsversions and data classes.

Text Comparisons

You might need to perform a text comparison to sort and merge text, to compare relationships between two pieces of text, or to index a file.

The traditional method for handling text comparisons is based on a strict binary comparison of the character values. The binary method of comparison is not meaningful when used for sorting text if the binary ordering of the coded characters does not match the ordering sequence of the alphabet. This situation is the case for most coded character sets.

Because the binary method is not sufficient for all usage requirements, the system supports the definitions of two other levels of ordering.

The first level is called ORDERING. For this level, each character has an ordering sequence value (OSV). An OSV is an integer in the range 0 (zero) through 255 that is assigned to each code position in a character set. The OSV indicates a relative ordering value of a character. An OSV of 0 (zero) indicates that the character comes before a character with an OSV equal to 1. More than one character can be assigned the same OSV.

The second level is called COLLATING. For this level, each character has an OSV and a priority sequence value (PSV). A PSV is an integer in the range 1 through 15 that is assigned to each code position in a character set. A PSV indicates a relative priority value within each OSV. Each character with a unique OSV has a PSV equal to 1, but two characters with the same OSV have different PSVs to separate them.

When comparing two strings of data, we call a comparison that uses only 1 level, the Ordering level, an equivalent comparison. A comparison that uses both levels, Ordering and Collating, is called a logical comparison.

You can specify the following three types of comparisons by calling procedures in the CENTRALSUPPORT library:

Ordering Type	Explanation
Binary	Compares two records based on the hexadecimal code values of the characters.
Equivalent	Compare two records based on the OSVs of the characters. This type of comparison uses the ORDERING level.
Logical	Compares two records based on the OSVs plus the PSVs of the characters. This type of comparison uses the COLLATING level.

In addition to the three types of ordering, the system also supports the following two types of character substitution:

Substitution	Explanation
Many to One	A predetermined string of up to three characters can be ordered as if it were one character, assigning it a single OSV and PSV pair. Even if a character is part of a predetermined string of characters that are ordered as a single value, the character still has an OSV and a PSV pair assigned to it to allow for cases in which the character appears in other strings or individually. For example, in Spanish, the letter pair ch is ordered as if it were a single letter, different from either c or h, and ordering between c and d.
One to Many	A single character can generate a string of two or three OSV and PSV pairs. For example, the β (the German sharp S) character is ordered as though it were ss.

You can specify a collating sequence to be used for text comparisons. When you designate an internationalized collating sequence at the program level and you are comparing two alphanumeric records, the compiler uses the logical ordering type when generating the text comparison routines.

Historically, text was sorted by using a standard system-provided method that is based on a strict binary comparison of the character values. Within a program, you might also specify a collating sequence to be used for text comparisons.

Your program can call the CENTRALSUPPORT library procedures for comparing and sorting text to obtain ordering information of the ccsversion, and to sort or compare text based on this information.

Note that comparisons involving string-valued attributes in the syntax of the language do not invoke the internationalization feature and are thus performed by using the EBCDIC collating sequence rather than the collating sequence specified in the CCSVERSION clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

If you must compare the value of a string-valued attribute using the internationalized collating sequence, you can first move (MOVE) the value to a field within the program. Then, perform the compare operation using that field rather than the attribute itself for the comparison.

For example, the following statement generates a comparison using EBCDIC:

```
IF ATTRIBUTE TITLE OF FILE1 IS EQUAL TO "THEFILE"...
```

The following statement generates an internationalized comparison of the same information:

```
01  TEMP-STORAGE PIC X(256).
    ...
    MOVE ATTRIBUTE TITLE OF FILE1 TO TEMP-STORAGE.
    IF TEMP-STORAGE IS EQUAL TO "THEFILE"...
```

Sorting and Merging

Because of the complexity of the SORT and MERGE statements, and because the compiler generates the sort compare procedure to be used in the sort and merge operations, you can specify a localized collating sequence to be used at the program level, or at the SORT or MERGE statement level. To use this language syntax, specify an *alphabet-name IS CCSVERSION* option in the SPECIAL-NAMES paragraph. Then if you specify a localized collating sequence at the program level, the compiler generates the text comparison routines.

Creating Indexed Files

If you are creating a localized indexed file, you must use the KEY-LENGTH and COMPARISON phrases of the SELECT clause to specify the key value and the type of comparison to be done.

The KEY-LENGTH phrase specifies the number of 8-bit characters the system uses to store a translated key value. Translated display data can require a larger storage area when including ordering and collating information. The KEY-LENGTH phrase can be used with the prime record key or the alternate record key.

The COMPARISON phrase specifies the type of comparison to be performed when searching for the key. A binary, logical, or equivalent comparison can be specified.

You must use the same ccsversion when you create the file and when you use it. A run-time error occurs when you open an indexed file for output if the run-time ccsversion does not match the compile-time ccsversion.

For More Information

Refer to “FILE-CONTROL Paragraph” in Section 5, “ENVIRONMENT DIVISION,” for the syntax of the COMPARISON and KEY-LENGTH phrases.

Providing Support for Natural Languages

The natural language feature enables users of your application program to communicate with the computer system in their natural language. A natural language is a human language in contrast to a computer programming language.

You must write your COBOL74 program in the subset of the standard EBCDIC character set defined by the COBOL language. Only the contents of string literals, data items with variable character data, or comments can be in a character set other than that subset.

If your program interacts with a user, has a user interface with screens or forms, displays messages or accepts user input, then those aspects of the program should be in the natural language of the user. For example, French would be the natural language of a person from France.

Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for a list of user interfaces that can be localized. The following text explains how to develop a COBOL application program which supports interaction in the natural language of the user.

Creating Messages for an Application Program

In the MLS environment, the messages handled by your application program are grouped into the following categories:

An . . .	Is a message . . .
Output message	That an application program displays to the user. Some examples of output messages are error messages and prompts for input. An output message can be localized so that it can be displayed in the language of the user.
Input message	Received by an interactive program either from a user or from another program in response to a prompt for input. The input message might be in a language that the program cannot recognize. In this case, the message must be translated so that it can be understood by the program.

If you develop input and output messages within an output message array, you make the localization process easier. When messages are in an output message array, the translator can use the MSGTRANS utility to localize the messages into one or more natural languages. The MSGTRANS utility finds all output message arrays in a program and presents them for translation. If messages are not in output message arrays, a translator must search the source file for each message and then translate the message.

You can create an output message array by creating an ALGOL library that contains OUTPUTMESSAGE ARRAY declarations.

An output message array contains output messages to be used by the MultiLingual System (MLS). ALGOL statements within the output message array declaration contain output messages or translate input messages. You can then call the library from your application program. The *MultiLingual System Administration, Operations, and Programming Guide* describes the procedures for creating and using output message arrays.

There is a program on the release media that demonstrates how to create an ALGOL library containing output message arrays. The program is called EXAMPLE/MLS/ALGOL/LIBRARY.

For information on how to call an ALGOL library from a COBOL74 program, refer to Section 15, "Libraries."

Creating Multilingual Messages for Translation

The following are guidelines for creating messages that can be multilingual:

- Put all output messages in output message arrays.
- Allow more space for translated messages. Because the English language is more compact than many other natural languages, a message in English generally becomes about 33 percent longer after it is translated into another language. For example, if a program can display an 80-character message, an English message should be only 60 characters long so that the translated message can expand by one-third and not exceed the maximum display size.
- Accept or display any messages through a library interface similar to that provided on the release media.
- Use complete sentences for messages because phrases are difficult to translate accurately.
- Do not use abbreviations because they are difficult to translate.

Providing Support for Business and Cultural Conventions

The business and cultural features enable users of an application program to display and receive data according to local conventions. A convention consists of formatting instructions for date, time, numeric, currency, and page size.

Standard convention definitions are provided for many formatting styles. For example, some of the conventions are Denmark, Italy, Turkey, and UnitedKingdom1. These convention definitions contain information to create formats for time, date, numbers, currency, and page size required by a particular locality.

Each system includes a data file named SYSTEM/CONVENTIONS that contains all the convention definitions that are supported. Although you can access many conventions, only one convention is active at a time for the entire system. This convention is called the system default convention. You can access conventions as follows:

- Look in the *MultiLingual System Administration, Operations, and Programming Guide*. Your system might have a subset of the ones defined in that guide.
- Use the MARC menus and screens or the system command SYSTEMOPTIONS. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* or the *System Commands Reference Manual*.
- Call the CNV_NAMES procedure to display the names of conventions available on the host computer.

If none of the conventions meet your needs, you can define a new convention. You must use a template to define a convention. A template is a group of predefined control characters that describe the components for date, time, numeric, or currency. For example the data item 02251990 and the template!0o!/!dd!/!yyyy! produce the formatted date, 02/25/1990. To use some of the CENTRALSUPPORT library procedures, you must understand how templates are defined. The *MultiLingual System Administration, Operations, and Programming Guide* describes how to define a template.

Using the Date and Time Features

COBOL74 provides several date and time features for standard use. You can access the conventions either by using language syntax or by calling a CENTRALSUPPORT library procedure.

Formatting Date and Time with Syntax Elements

COBOL74 provides the following language syntax to handle formatting of date and time data items:

- You can declare a data item to have one of several date or time types in the TYPE clause of the data-description entry. You can also designate a language or convention with the TYPE clause.
- COBOL74 provides special registers, date and time editing, and PROCEDURE DIVISION statements for standard formatting of date and time. These standard features include the following:
  - The special registers TODAYS-DATE, DATE, and DAY provide the system date with MMDDYY, YYMMDD, and YYDDD formats respectively.
  - The special register TIME provides the elapsed time after midnight on a 24-hour clock, in the format HHMMSSTT, where HH equals hours, MM equals minutes, SS equals seconds, and TT equals hundredths of a second. For example, 12:01 p.m. is expressed as 12010000.
  - The special register TIMER provides the number of 2.4-microsecond intervals since midnight.

- The special register TODAYS-NAME provides the name of the current day of the week.
- You can specify the date punctuation (space, slash, or hyphen), the time punctuation (colon or space), and the thousand separator (comma or space) in the PICTURE clause. For example,

```
05  DATE-YYMMDD        PIC 99/99/99
05  DATE-MMDDYY        PIC 99-99-99
05  DATE-YYDDD         PIC 99B999
05  TIME-HHMMSS        PIC 99:99:99
05  AMOUNT             PIC 99B999.99
```

- The COBOL74 compiler provides date and time editing with conventions, with the particular convention and language specified as a property of the receiving data item. The MOVE statement causes the editing to occur.
- The ACCEPT statement transfers the formatted system date or time into the data item specified by the identifier using the TYPE, CONVENTION, and LANGUAGE declared for the item.

For More Information

- Refer to "Special Registers" in Section 2, "Language Elements," for details about the special registers.
- Refer to "Editing" in Section 6, "Data Concepts," for information about editing with the PICTURE clause.

Formatting Date and Time with Library Calls

You can call the CENTRALSUPPORT library procedures to format your date and time items. The following types of procedures are available to format the date and time:

Procedure Type	Description
Convention	You supply the convention name and the value for the date or time. The procedure returns the date or time value in the format used by the convention. All the conventions are described in the *MultiLingual System Administration, Operations, and Programming Guide*.
Template	You supply the following: the format that you want for the date or time in a template parameter; the value for the date or time. You must use predefined control characters to create the template. These control characters are described in the *MultiLingual System Administration, Operations, and Programming Guide*.

Procedure Type	Description
System	The system supplies the date and time. There is a procedure that formats the system date, the system time, or both according to a convention and a procedure that formats the system date, the system time, or both according to a template that you supply.

Example

You could use the CNV_SYSTEMDATETIME_COB procedure to display the system date and time according to the language and convention you choose. If you designate the ASeriesNative convention and the ENGLISH language, the date and time are displayed as follows:

```
9:25 AM Monday,  July 4, 1988
```

If you designate the FranceListing convention and the French language, the same date and time might be displayed as follows:

```
9h25, lundi 4 juillet 1988
```

Using the Numeric and Currency Features

COBOL74 provides several numeric and currency features for standard use. You can access the conventions either by using language syntax or by calling a CENTRALSUPPORT library procedure.

Formatting Numerics and Currencies with Syntax Elements

The standard COBOL74 compiler provides you with the following ways to edit currency and numeric displays:

- You can specify a single character as the currency symbol. For example, in the SPECIAL-NAMES paragraph, the *CURRENCY IS literal-9* option specifies the value of literal-9 as the currency symbol throughout the entire program.
- You can specify a comma as the decimal sign and a period as the thousand separator. In the SPECIAL-NAMES paragraph, the DECIMAL-POINT IS COMMA clause causes a comma to be used as the decimal sign and a period to be used as the thousand separator throughout the entire program.
- By using output message arrays, your program can pass numeric values to messages as records of type USAGE IS DISPLAY. These records can contain numeric formatting, which uses a comma as the thousands separator and a period for the decimal sign.

Formatting Numerics and Currencies with Library Calls

In addition to using the features in COBOL74, you can call the CENTRALSUPPORT library procedures to inquire about numeric symbols or format currency amounts. All numeric or currency symbols can be retrieved with a CENTRALSUPPORT library call. Monetary amounts in real number form can be formatted according to different conventions.

Using the Page Size Formatting Features

Page sizes are specific to a locality. COBOL74 provides several standard features for formatting page size. You can set the number of lines per page and the number of characters per line by using language syntax. Alternately, you can call the CNV_FORMSIZE procedure to obtain predetermined page size values for the convention you specify.

Formatting Page Size with Syntax Elements

COBOL74 provides the following language syntax to handle formatting of the size of pages:

- LINAGE clause in the DATA DIVISION
- Report Writer
- ADVANCING phrase of the WRITE statement

Formatting Page Size with Library Calls

The CNV_FORMSIZE procedure enables you to retrieve default lines-per-page and characters-per-line values for a specified convention.

For example, the Netherlands convention definition specifies 70 lines as the default page length and 82 characters as the default page width, while the Zimbabwe convention definition specifies 66 lines as the default page length and 132 lines as the default page width.

Summary of Language Syntax by Division

The following paragraphs describe the changes you can make in the divisions of a COBOL74 program to use internationalization features. No changes are required in the IDENTIFICATION DIVISION.

ENVIRONMENT DIVISION

The language syntax in the ENVIRONMENT DIVISION includes the following:

- The *PROGRAM COLLATING SEQUENCE IS alphabet-name* clause in the OBJECT-COMPUTER paragraph declares that the collating sequence to be used is the one associated with the alphabet-name. The alphabet-name must be named in the SPECIAL-NAMES paragraph. The collating sequence is used for alphabetic comparisons in conditional statements and for sorting and merging routines.
- The *alphabet-name IS CCSVERSION literal-1* clause in the SPECIAL-NAMES paragraph designates the system collating sequence and the ccsversion. If you do not use the literal-1 option, the system uses the system default ccsversion values for the collating sequence and the ccsversion. If you do specify literal-1, then the literal identifies the ccsversion.
- The following two phrases are used by KEYEDIOII to create a localized indexed file:
  - The *KEY-LENGTH IS literal-2* phrase of the SELECT clause specifies the number of 8-bit characters the system uses to store a translated key value. Translated display data can require extra storage area for ordering and collating information and for the length of the key. A run-time error occurs when the indexed file is created if the compile-time and run-time ccsversions are not the same.
  - The *COMPARISON IS* phrase of the SELECT clause specifies the type of comparison to be performed when searching for the key.
- The *DECIMAL-POINT IS COMMA* option of the SPECIAL-NAMES paragraph causes the functions of the EBCDIC comma and decimal point to be switched.
- The *CURRENCY SIGN IS literal-9* clause of the SPECIAL-NAMES paragraph enables you to specify a single character to be used as the currency symbol in numeric data.

DATA DIVISION

The language syntax that you can use in the DATA DIVISION includes the following:

- The TYPE option of the data-description entry for record structures allows a data item to be declared to be one of the following date or time types:
  - LONG-DATE
  - SHORT-DATE
  - NUMERIC-DATE
  - LONG-TIME

  - NUMERIC-TIME
- The USING phrase of the TYPE option allows the data item to be formatted according to a designated language or convention.
- The KANJI phrase of the USAGE clause enables you to display and write messages in Japanese and other natural languages that require the double-octet format.
- Standard COBOL74 editing also can be used to format date and time.

PROCEDURE DIVISION

The language syntax that you can use in the PROCEDURE DIVISION includes the following:

- The *identifier IS ALPHABETIC* clause tests whether an identifier is in the alphabetic truthset. If you have specified a ccsversion in the ENVIRONMENT DIVISION, the system determines the alphabetic truthset with respect to the alphabetic data class of the ccsversion.
- The *ACCEPT identifier* statement transfers a formatted system date or time to the identifier. The format of the system date or time data item depends on the coding of the TYPE, LANGUAGE, and CONVENTION clauses for the item.
- The MOVE statement causes a receiving item with an associated TYPE clause to be formatted according to the TYPE, LANGUAGE, and CONVENTION clauses declared for the item. If no LANGUAGE and CONVENTION clauses are specified, the compiler uses the hierarchy to determine the language and convention to be used.

  If the receiving item is of type SHORT-DATE, LONG-DATE, or NUMERIC-DATE, the sending item must be of format YYYYMMDD where YYYY represents the year, with a value in the range 0000 through 9999; MM represents the month, with a value in the range 01 through 12; and DD represents the day, with a value in the range 01 through 31.

  If the receiving item is of type LONG-TIME or NUMERIC-TIME, the sending item must be of format HHMMSSPPPP where HH represents the hour, with a value in the range 00 through 23; MM represents the minutes, with a value in the range 00 through 59; SS represents the seconds, with a value in the range 00 through 59; and PPPP represents the partial seconds, with a value in the range 0000 through 9999.
- The CALL statement can be used to call the procedures of the CENTRALSUPPORT library.
- The SORT and MERGE statements use the collating sequence of a specified ccsversion if you establish a program collating sequence in the ENVIRONMENT DIVISION and do not override it in the *COLLATING SEQUENCE IS alphabet-name* option in the SORT or MERGE statement.
- The *CHANGE ATTRIBUTE LIBACCESS statement* can be used to change a library call by function to a library call by title or vice-versa.

Examples

Example 16–1 shows the five different types of date and time items and their expected values upon execution of the ACCEPT statement. The PICTURE size of the data-items depends on the definition of the type defined in the conventions file.

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  LONG-DATE-ITEM              PIC  X(29)  TYPE IS LONG-DATE.
 01  SHORT-DATE-ITEM             PIC  X(17)   TYPE IS SHORT-DATE.
 01  NUM-DATE-ITEM               PIC  X(10)   TYPE IS NUMERIC-DATE.
 01  LONG-TIME-ITEM              PIC  X(13)  TYPE IS LONG-TIME.
 01  NUM-TIME-ITEM               PIC  X(8)  TYPE IS NUMERIC-TIME.
*
 PROCEDURE DIVISION.
*
*  Get the system date and time in the various formats defined by the
*  CONVENTION and LANGUAGE task attributes for the task.   Assume a
*  convention of ASERIESNATIVE and a language of ENGLISH, a current
*  date of 31 August 1990, and a current time of 2:37:20 PM.
*
 ACCEPT LONG-DATE-ITEM FROM DATE.
*
*  The LONG-DATE-ITEM now contains Friday, August 31, 1990.
*
 ACCEPT SHORT-DATE-ITEM FROM DATE.
*
*  The SHORT-DATE-ITEM now contains Fri, Aug 31, 1990.
*
 ACCEPT NUM-DATE-ITEM FROM DATE.
*
*  The NUM-DATE-ITEM now contains 08/31/1990.
*
 ACCEPT LONG-TIME-ITEM FROM TIME.
*
*  The LONG-TIME-ITEM now contains14:37:20.0000.
*
 ACCEPT NUM-TIME-ITEM FROM TIME.
*
*  The NUM-TIME-ITEM now contains 14:37:20.

 STOP RUN.
```

Example 16–1. Coding the Format 3 ACCEPT Statement

Example 16–2 shows the five different types of date or time items and their expected values upon execution of the MOVE statement. The PICTURE size of the data-items depends on the definition of the type defined in the conventions file.

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  LONG-DATE-ITEM              PIC  X(29)  TYPE IS LONG-DATE.
 01  SHORT-DATE-ITEM             PIC  X(17)   TYPE IS SHORT-DATE.
 01  NUM-DATE-ITEM               PIC  X(10)   TYPE IS NUMERIC-DATE.
 01  LONG-TIME-ITEM              PIC  X(13)  TYPE IS LONG-TIME.
 01  NUM-TIME-ITEM               PIC  X(8)  TYPE IS NUMERIC-TIME.
*
 PROCEDURE DIVISION.
*  Convert the literal date and time into the various formats defined
*  by the CONVENTION and LANGUAGE task attributes for the task.   Assume
*  a convention of ASERIESNATIVE and a language of ENGLISH, a current
*  date of 31 August 1990, and a current time of 2:37:20 PM.
 MOVE "19900831" TO LONG-DATE-ITEM.
*  The LONG-DATE-ITEM now contains Friday, August 31, 1990.
*
 MOVE "19900831" TO SHORT-DATE-ITEM.
*
*  The SHORT-DATE-ITEM now contains Fri, Aug 31, 1990.
*
 MOVE "19900831" TO NUM-DATE-ITEM.
*
*  The NUM-DATE-ITEM now contains 08/31/1990.
*
 MOVE "143720000000" TO LONG-TIME-ITEM.
*
*  The LONG-TIME-ITEM now contains14:37:20.0000.
*
 MOVE "143720000000" TO  NUM-TIME-ITEM.
*
*  The NUM-TIME-ITEM now contains 14:37:20.
:
 STOP RUN.
```

Example 16–2. Coding the MOVE Statement for Internationalization

For More Information

- For details on the ACCEPT, CALL, SORT, MERGE, and MOVE statements, see Section 9, “PROCEDURE DIVISION Statements.”
- For information about the alphabetic test, refer to “Class Condition” in Section 8, “PROCEDURE DIVISION Concepts.”

Summary of CENTRALSUPPORT Library Procedures

The CENTRALSUPPORT library procedures are integer-valued procedures. The procedures return values in output parameters and as the procedure result.

You can check the result returned by each procedure by using standard programming practices. The result is useful in deciding if an error has occurred. The possible values for each procedure result are listed in the description of each procedure. The meanings of the result values are described at the end of this section.

The CENTRALSUPPORT library procedures are called by application programs and system software. Following are some tasks programs can perform by calling CENTRALSUPPORT library procedures:

- Identify available coded characters sets and ccsversions.
- Map data from one coded character set to another.
- Process data according to ccsversion.
- Compare and sort text.
- Position characters.
- Determine available natural languages.
- Access CENTRALSUPPORT library messages.
- Identify available conventions definitions.
- Obtain convention information.
- Format dates according to convention.
- Format times according to convention.
- Format monetary data according to convention.
- Determine default page size.

Identifying Available Coded Character Sets and Ccsversions

CCSVSN_NAMES_NUMS

Returns the names and numbers of all coded character sets or all ccsversions available on the host computer. The names and numbers are listed in two arrays. These arrays are ordered so that the names in the names array correspond to the numbers in the numbers array.

CENTRALSTATUS

Obtains the values of the default settings for internationalization features on the host computer. This procedure returns the names of the default ccsversion, language, and convention. It also returns the number of the default ccsversion.

VALIDATE_NAME_RETURN_NUM

Verifies that a designated coded character set or ccsversion name is valid on the host computer. If the coded character set or ccsversion is valid, the procedure returns the corresponding number.

VALIDATE_NUM_RETURN_NAME

Verifies that the designated coded character set or ccsversion number is valid on the host computer. If the coded character set or ccsversion is valid, the procedure returns the corresponding name.

Mapping Data from One Coded Character Set to Another

CCSTOCCS_TRANS_TEXT

Maps data from one coded character set to another by using a translate table. Characters are translated using a one-to-one mapping between the two character sets.

Processing Data According to Ccsversion

VSNINSPECT_TEXT

Compares the input text to a designated ccsversion truthset to determine whether the characters in the text are in the truthset. You can use this procedure to determine if characters are in one of the following truthsets:

- Alphabetic
- Numeric
- Spaces
- Presentation
- Lowercase
- Uppercase

VSNTRANS_TEXT

Translates data using a designated ccsversion. You can use this procedure to perform the following types of translations:

- Lowercase to uppercase characters
- Uppercase to lowercase characters
- The digits 0 through 9 to alternate digits
- Alternate digits to the digits 0 through 9
- Characters to their character escapement directions

Comparing and Sorting Text

VSNCOMPARE_TXT

Compares two strings using one of the following comparison methods for a designated ccsversion:

- Binary comparison, which is based on the binary values of the characters
- Equivalent comparison, which is based on the ordering sequence values of characters
- Logical comparison, which is based on the ordering sequence values and priority sequence values of characters

VSNGETORDERINGFOR_ONE_TEXT

Returns the ordering information for the input text. The ordering information determines how the input text is collated. It includes the ordering and priority sequence values of the characters and any substitution of characters to be made when the input text is sorted. You can choose one of the following types of ordering information:

- Equivalent ordering information, which comprises only the ordering sequence values
- Logical ordering information, which comprises the ordering sequence values followed by the priority sequence values.

Note: *There is a source size restriction of 9,000 bytes.*

Positioning Characters

VSNESCAPEMENT

Takes the input text and rearranges it according to the escapement rules of the ccsversion. Both the character advance direction and the character escapement direction are used. If the character advance direction is positive, then the start position of the text is the leftmost position of the starting character. If the character advance direction is negative, then the starting position for the text is the rightmost position of the last character. From that point on, the character advance direction value and the character escapement direction values, in combination, control where each character should be placed in relation to the previous character.

Determining Available Natural Languages

MCPBOUND_LANGUAGES

Returns the names of the languages that are currently bound to the MCP.

Accessing CENTRALSUPPORT Library Messages

GET_CS_MSG

Returns text of the message associated with the designated CENTRALSUPPORT error number. Your program can specify the maximum message length desired. If the returned message is shorter, it is padded with blanks.

An entire message consists of three parts:

- The header, which always takes up the first 80 characters of the return message
- The general description, which takes the next 80 characters
- The specific description, which has no maximum length

Identifying Available Convention Definitions

CENTRALSTATUS

Obtains the values of the default settings for internationalization features on the host computer. This procedure returns the names of the default ccsversion, language, and convention. It also returns the number of the default ccsversion.

CNV_NAMES

Returns the names of the conventions available on the host system.

CNV_VALIDATENAME

Returns a value that indicates whether the specified convention name is currently defined on the host system.

Obtaining Convention Information

CNV_SYMBOLS

Returns the numeric and monetary symbols defined for a designated convention. The symbols in the convention are

- Numeric positive symbol and numeric negative symbol
- Numeric thousands separator symbol
- Numeric decimal symbol
- Numeric left and right enclosure symbols
- Numeric grouping
- Monetary positive symbol
- Monetary negative symbol
- International currency notation
- National currency symbol
- Monetary grouping
- Monetary thousands separator symbol
- Monetary left and right enclosure symbols
- Monetary decimal symbol

CNV_TEMPLATE_COB

Returns the requested template for a designated convention. You can obtain the template for the following:

- Long date format
- Short date format
- Numeric date format
- Long time format
- Numeric time format
- Monetary format
- Numeric format

Formatting Dates According to Convention

CNV_DISPLAYMODEL_COB

Returns either the date or time display model defined for the designated convention. The components of the model are translated to the designated language.

CNV_FORMATDATE_COB

Formats a numeric date that has the form YYYYMMDD. The numeric date is passed as a parameter to the procedure according to a designated convention and language. The date can be formatted using the long, short, or numeric date format defined in the convention.

CNV_FORMATDATETIMETMP_COB

Returns the system date, the time, or both in the designated language, formatted according to a template passed to this procedure.

CNV_SYSTEMDATETIMETMP_COB

Returns the system date, the time, or both, formatted according to the designated convention template and language. You can choose from the following types of formats:

- Long date and long time
- Long date and numeric time
- Short date and long time
- Short date and numeric time
- Numeric date and long time
- Numeric date and numeric time
- Long date only
- Short date only
- Long time only
- Numeric time only

FORMATDATETMP_COB

Formats a numeric date passed as a parameter to the procedure according to a template and language passed as parameters of the procedure.

Formatting Times According to Convention

CNV_DISPLAYMODEL_COB

Returns either the date or time display model defined for the designated convention. The components of the model are translated to the designated language.

CNV_FORMATTIME_COB

Formats a time with the form HHMMSSPPPP. The time is passed as a parameter to the procedure according to a designated convention and language. The time can be formatted using the long or numeric time format defined in the convention.

CNV_FORMATTIMETMP_COB

Formats a time passed as a parameter to the procedure according to a template and language passed as parameters of the procedure.

CNV_SYSTEMDATETIMETMP_COB

Returns the system date, the time, or both in the designated language, formatted according to a template passed to this procedure.

CNV_SYSTEMDATETIME_COB

Returns the system date, the time, or both, formatted according to the designated convention template and language. You can choose from the following types of formats:

- Long date and long time
- Long date and numeric time
- Short date and long time
- Short date and numeric time
- Numeric date and long time
- Numeric date and numeric time
- Long date only
- Short date only
- Long time only
- Numeric time only

Formatting Monetary Data According to Convention

CNV_CURRENCYEDIT_DOUBLE_COB

Formats a monetary value, passed as a parameter to the procedure, according to the monetary editing format template defined in the designated convention. This

procedure receives a double-precision integer and converts it to a formatted monetary value.

CNV_CURRENCYEDITTMP_DOUBLE_COB

Formats a monetary value, passed as a parameter to the procedure, according to a template also passed as a parameter. The template is retrieved from the CNV_TEMPLATE_COB procedure. This procedure receives a double-precision integer and converts it to represent a currency value according to the template specified.

Determining Default Page Size

CNV_FORMSIZE

Returns the default lines-per-page and characters-per-line values defined in a designated convention for formatting printer output.

Library Calls

You can access the CENTRALSUPPORT library procedures by following these steps:

1. Declare the parameters of the library procedure in the WORKING-STORAGE SECTION of the program.
2. Use the CHANGE statement syntax to specify the value of the LIBACCESS library attribute as BYFUNCTION.
3. Use the CALL statement syntax to call a library procedure where the library identity is *procedure-name of CENTRALSUPPORT*, the parameters are listed in the USING clause, and the procedure result is specified in the GIVING clause.

An example of the declarations and the syntax necessary to call the CENTRALSUPPORT library is provided in the description of each procedure later in this section.

For More Information

For general information about calling library procedures, refer to Section 15, "Libraries."

Parameter Categories

All integer parameters are passed by reference rather than by value. The CENTRALSUPPORT library procedures return output parameters and procedure result values. The parameter types are further described on the following pages.

Input Parameters

In many cases, an input parameter requires that you supply the ccsversion name or number, or the language or convention name. You can get this information as follows:

- The *MultiLingual System Administration, Operations, and Programming Guide* describes all the provided ccsversions, languages, and conventions. However, your system might have only a subset of the ones provided. There may also be customized conventions that are not listed in the *MultiLingual System Administration, Operations, and Programming Guide*. These may be identified by the next two options.
- If you are a system administrator, a privileged user, or are allowed to use the system console, you can use MARC menus and screens, or the SYSOPS command to list the options that exist on your system. The *MultiLingual System Administration, Operations, and Programming Guide* provides the instructions needed to obtain information about ccsversion, language, or convention defaults on your system.
- You can call procedures in the CENTRALSUPPORT library that return the default ccsversion, language, and convention. If you are writing an application to be used on another system, you can use these library procedures to verify that the ccsversion, the language, or the convention specified by the user is valid on that system.

The fields of parameters that you supply as 01-level records have fixed positions. This means that you must use blanks or zeros in any fields that you omit.

For any procedure which accepts a ccsversion number as an input parameter, you can specify a –2 as input to indicate that the system default value should be used. For any procedure that accepts a ccsversion name as an input parameter, you can specify all blanks or all zeros as inputs to indicate that the system default value should be used. For any procedure that accepts a language or convention name as an input parameter, you can specify all blanks or all zeros as inputs to indicate that the task attribute should be used. If the task attribute is not available, the CENTRALSUPPORT library searches down the hierarchy until a usable value is found.

Input Parameters with Type Values

Many CENTRALSUPPORT procedures have an input parameter that indicates the type of information to be applied or returned in the procedure. The values in these parameters are referred to as type values. The values used in the convention (CNV) procedures are common across all CNV procedures. The values used in the coded character set and ccsversion (VSN) procedures are common across all CCS and VSN procedures.

For example, there is a parameter used in some procedures that specifies the formatting of the time. In the examples, the parameter is named CS-NTIMEV. You must choose a type value to indicate a format. For example, a value of 3 indicates the long time format is used.

Because the parameters are passed by reference, you need to declare and assign the type values in the WORKING-STORAGE SECTION. You can then use a MOVE statement to move the value into the parameter as shown in the following example:

```
MOVE LONG-TIME-V TO CS-NTIMEV.
```

Example

Example 16–3 shows WORKING-STORAGE declarations for parameters with type values. The examples shown later in this section uses the type value names declared here. Numeric items must be PIC S9(11) USAGE BINARY for parameter matching to type INTEGER.

```
77  CS-BINARYV            PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-EQUIVALENTV        PIC S9(11)  USAGE BINARY  VALUE  1.
77  CS-LOGICALV           PIC S9(11)  USAGE BINARY  VALUE  2.
77  CS-CHARACTER-SETV     PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-CCSVERSIONV        PIC S9(11)  USAGE BINARY  VALUE  1.
77  CS-NUMTOALTDIGV       PIC S9(11)  USAGE BINARY  VALUE  5.
77  CS-ALTDIGTONUMV       PIC S9(11)  USAGE BINARY  VALUE  6.
77  CS-LOWTOUPCASEV       PIC S9(11)  USAGE BINARY  VALUE  7.
77  CS-UPTOLOWCASEV       PIC S9(11)  USAGE BINARY  VALUE  8.
77  CS-ESCMENTPERCHARV    PIC S9(11)  USAGE BINARY  VALUE  9.
77  CS-ALPHAV             PIC S9(11)  USAGE BINARY  VALUE 12.
77  CS-NUMERICV           PIC S9(11)  USAGE BINARY  VALUE 13.
77  CS-PRESENTATIONV      PIC S9(11)  USAGE BINARY  VALUE 14.
77  CS-SPACESV            PIC S9(11)  USAGE BINARY  VALUE 15.
77  CS-LOWERCASEV         PIC S9(11)  USAGE BINARY  VALUE 16.
77  CS-UPPERCASEV         PIC S9(11)  USAGE BINARY  VALUE 17.
77  CS-NOTINTSETV         PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-INTSETV            PIC S9(11)  USAGE BINARY  VALUE  1.
77  CS-CMPLSSV            PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-CMPLEQV            PIC S9(11)  USAGE BINARY  VALUE  1.
77  CS-CMPEQLV            PIC S9(11)  USAGE BINARY  VALUE  2.
77  CS-CMPGTRV            PIC S9(11)  USAGE BINARY  VALUE  3.
```

Example 16–3. Sample Data Declarations for Type Value Data Items

```
77  CS-CMPGEQV            PIC S9(11)  USAGE BINARY  VALUE  4.
77  CS-CMPNEQV            PIC S9(11)  USAGE BINARY  VALUE  5.
77  CS-LDATE-V            PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-SDATE-V            PIC S9(11)  USAGE BINARY  VALUE  1.
77  CS-NDATEV             PIC S9(11)  USAGE BINARY  VALUE  2.
77  CS-LTIMEV             PIC S9(11)  USAGE BINARY  VALUE  3.
77  CS-NTIMEV             PIC S9(11)  USAGE BINARY  VALUE  4.
77  CS-LDATELTIMEV        PIC S9(11)  USAGE BINARY  VALUE  5.
77  CS-LDATENTIMEV        PIC S9(11)  USAGE BINARY  VALUE  6.
77  CS-SDATELTIMEV        PIC S9(11)  USAGE BINARY  VALUE  7.
77  CS-SDATENTIMEV        PIC S9(11)  USAGE BINARY  VALUE  8.
77  CS-NDATELTIMEV        PIC S9(11)  USAGE BINARY  VALUE  9.
77  CS-NDATENTIMEV        PIC S9(11)  USAGE BINARY  VALUE 10.
77  CS-LONGDATE-TEMPV     PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-SHORTDATE-TEMPV    PIC S9(11)  USAGE BINARY  VALUE  1.
77  CS-NUMDATE-TEMPV      PIC S9(11)  USAGE BINARY  VALUE  2.
77  CS-LONGTIME-TEMPV     PIC S9(11)  USAGE BINARY  VALUE  3.
77  CS-NUMTIME-TEMPV      PIC S9(11)  USAGE BINARY  VALUE  4.
77  CS-MONETARY-TEMPV     PIC S9(11)  USAGE BINARY  VALUE  5.
77  CS-NUMERIC-TEMPV      PIC S9(11)  USAGE BINARY  VALUE  6.
77  CS-DATE-DISPLAYMODEL  PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-TIME-DISPLAYMODEL  PIC S9(11)  USAGE BINARY  VALUE  1.
```

Example 16–3. Sample Data Declarations for Type Value Data Items

Output Parameters

These parameters contain the output values produced by the procedure. For example, the translated text produced by the procedure CCSTOCCS_TRANS_TEXT is returned in an output parameter.

Result Parameter

All the library procedures return an integer value as the procedure result that indicates whether an error occurred during the execution of the procedure. In general, a returned value of 1 means that no error occurred and any other value means that an error occurred. However, the CNV_VALIDATENAME and VSNCOMPARE_TEXT procedures are exceptions to this rule. For these procedures, the returned value can be 0 (zero), 1, or another value. A returned value of 0 (zero) means that no error occurred and the condition is FALSE. A returned value of 1 means that no error occurred and the condition is TRUE. Any other value means that an error occurred.

Each procedure lists the values that can be returned by that procedure. The meanings of these values are explained at the end of this section. You can use these values to call the GET_CS_MSG procedure and display the error that occurred, or you can code error routines to handle the possible errors.

For More Information

Refer to the explanation of the GET_CS_MSG procedure in this section for more information about using that procedure.

Procedure Descriptions

The following pages describe the internationalization procedures accessible from a COBOL74 program. The procedures reside in the CENTRALSUPPORT library.

Each description includes a general overview of the procedure, an example showing how to call the procedure, and an explanation of the parameters used in the example. Not all parameters used to produce the output will be displayed in the output.

You can define the name of a parameter to be whatever name you want. In the following discussions, the parameters are given names in the example, and are identified in the explanation by that name.

The following parameters are used in many of the procedures:

- CS-DATAOKV is a constant with a value of 1 that is compared with the RESULT parameter. This value indicates that the CENTRALSUPPORT library did not find errors and was able to process the information.
- CS-FALSEV is a constant with a value of 0 that is compared with the RESULT parameter. This value indicates that although the CENTRALSUPPORT library did not find errors, it did not process the information.
- SUB represents a subscript.

CCSTOCCS_TRANS_TEXT

This procedure translates data from the coded character set specified in the first parameter to the coded character set specified in the next parameter. Characters are translated using a one-to-one mapping between two coded character sets. For example, you might want to translate data in the LATIN1EBCDIC coded character set to the LATIN1ISO coded character set.

Although there are many coded character set numbers, there is not a mapping table between every combination of coded character sets. The procedure returns an error indicating the data was not found if you pass two valid coded character set numbers for a table that does not exist.

Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for a list of the coded character set to coded character set translate tables that are defined.

Example

Example 16–4 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CCSTOCCS_TRANS_TEXT library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example takes the input string “pañuelo,” which is encoded in the Latin1EBCDIC coded character set and translates it to the Latin1ISO coded character set. The string “pañuelo” is represented by the following hexadecimal codes in Latin1EBCDIC: *978149A4859396*. In Latin1ISO, the hexadecimal codes are *83969586519985958385*. You can use the *MultiLingual System Administration, Operations, and Programming Guide* to determine that the coded character set number for Latin1EBCDIC is 12 and Latin1ISO is 13. You can also retrieve these numbers by calling the procedure VALIDATE_NAME_RETURN_NUM with the coded character set names.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/CCSTOCCSTRANSTEXT."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.

01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(12)   VALUE "DEST-TEXT = ".
    05  OF-DEST-TEXT     PIC X(10).
    05  FILLER           PIC X(58)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

01  DEST-TEXT            PIC X(10).
01  SOURCE-TEXT          PIC X(10).

77  CCS-NUM-FROM         PIC S9(11)  USAGE BINARY.
```

```
 77  CCS-NUM-TO           PIC S9(11)  USAGE BINARY.
 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
```

Example 16–4. Calling the CCSTOCCS_TRANS_TEXT Procedure

```
 77  DEST-START           PIC S9(11)  USAGE BINARY.
  77  RESULT               PIC S9(11)  USAGE BINARY.
  77  SOURCE-START         PIC S9(11)  USAGE BINARY.
  77  TRANS-LEN            PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CCSTOCCS-TRANS-TEXT.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** CCSTOCSS-TRANS-TEXT ***************************************
 CCSTOCCS-TRANS-TEXT.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE 12 TO CCS-NUM-FROM.
   MOVE 13 TO CCS-NUM-TO.
   MOVE "pañuelo" TO SOURCE-TEXT.
   MOVE 10 TO TRANS-LEN.
   CALL "CCSTOCCS_TRANS_TEXT OF CENTRALSUPPORT"
     USING CCS-NUM-FROM,
           CCS-NUM-TO,
           SOURCE-TEXT,
           SOURCE-START,
           DEST-TEXT,
           DEST-START,
           TRANS-LEN
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE DEST-TEXT TO OF-DEST-TEXT
          WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16–4. Calling the CCSTOCCS_TRANS_TEXT Procedure

Explanation

CCS-NUM-FROM is an integer passed to the procedure. It contains the number of the coded character set that you are formatting from.

CCS-NUM-TO is an integer passed to the procedure. It contains the number of the coded character set you are translating the text to. The destination text will be in this coded character set.

SOURCE-TEXT is passed to the procedure. It contains the text to translate. You specify the size of this record.

SOURCE-START is passed to the procedure. It contains the byte offset, relative to 0 (zero), in SOURCE-TEXT where the translation starts.

DEST-TEXT is returned by the procedure. It contains the translated text. The size of this record and the record in the SOURCE-TEXT parameter should be the same.

DEST-START is passed to the procedure. It contains the byte offset (0 relative) in the DEST parameter where the translated text is to be placed.

TRANS-LEN is passed to the procedure. It specifies the number of characters in SOURCE-TEXT to be translated, beginning at SOURCE-START.

RESULT is returned as the integer value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by CCSTOCCS_TRANS_TEXT are as follows:

1	1002	3003
1000	3000	4002
1001	3001	

Sample output from Example 16–4 follows:

```
RESULT =            1
DEST-TEXT = pañuelo
```

For more information on the error result values, see Table 16–1 later in this section.

CCSVSN_NAMES_NUMS

This procedure returns a list of the coded character set names and numbers or a list of the ccsversion names and numbers that are available on your system. You specify which list you want with the first parameter to the procedure. The names and numbers are listed in two arrays. These arrays are coded so that the names in the names array correspond to the numbers in the numbers array.

You might use this procedure to create a menu that lists the ccsversions from which a user can choose. You might also use this procedure to verify that the ccsversion to be used by your program is available on the host computer.

Example

Example 16–5 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CCSVSN_NAMES_NUMS library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example returns a list of available ccsversion names and numbers on a system. This is an arbitrary list of ccsversions and might not be the same on every system.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/CCSVSNNAMESNUMS."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.

01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
```

Example 16–5. Calling the CCSVSN_NAMES_NUMS Procedure

```
    05  OF-RESULT        PIC ZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(15)   VALUE "CCSversion Name".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  FILLER           PIC X(17)   VALUE "CCSversion Number".
    05  FILLER           PIC X(43)   VALUE SPACE.
01  OF-3.
    05  FILLER           PIC X(15)   VALUE ALL "-".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  FILLER           PIC X(17)   VALUE ALL "-".
    05  FILLER           PIC X(43)   VALUE SPACE.
01  OF-4.
    05  OF-NAMES-ELEM    PIC X(17).
    05  FILLER           PIC X(08)   VALUE SPACE.
    05  OF-NUMS-ELEM     PIC ZZZZZZZZZZ9.
    05  FILLER           PIC X(43)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************
    01  NAMES-ARY.
    05 NAMES-ELEM       PIC X(17)   OCCURS 20 TIMES.
    01  NUMS-ARY                        USAGE BINARY.
    05  NUMS-ELEM        PIC S9(11)  OCCURS 20 TIMES.
    01  SUB                 PIC 9(02).

  77  CS-CCSVERSIONV       PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
  77  RESULT               PIC S9(11)  USAGE BINARY.
  77  TOTAL                PIC S9(11)  USAGE BINARY.

  PROCEDURE DIVISION.
  INTLCOBOL74.
    OPEN OUTPUT OUTPUT-FILE.
    PERFORM CCSVSNNAMESNUMS.
    CLOSE OUTPUT-FILE.
    STOP RUN.

 ***** CCSVSNNAMESNUMS ******************************************
  CCSVSNNAMESNUMS.
    CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
    CALL "CCSVSN_NAMES_NUMS OF CENTRALSUPPORT"
      USING CS-CCSVERSIONV,
            TOTAL,
            NAMES-ARY,
```

Example 16–5. Calling the CCSVSN_NAMES_NUMS Procedure

```
NUMS-ARY
       GIVING RESULT.
     MOVE RESULT TO OF-RESULT.
     WRITE OUTPUT-RECORD FROM OF-1.
     IF RESULT IS EQUAL TO CS-DATAOKV
       THEN MOVE SPACE TO OUTPUT-RECORD
            WRITE OUTPUT-RECORD
            WRITE OUTPUT-RECORD FROM OF-2
            WRITE OUTPUT-RECORD FROM OF-3
            MOVE 1 TO SUB
            PERFORM DISPLAYARY UNTIL SUB IS GREATER THAN TOTAL.

  ***** DISPLAYARY ************************************************
   DISPLAYARY.
     MOVE NAMES-ELEM(SUB) TO OF-NAMES-ELEM.
     MOVE NUMS-ELEM(SUB) TO OF-NUMS-ELEM.
     WRITE OUTPUT-RECORD FROM OF-4.
     ADD 1 TO SUB.
```

Example 16-5. Calling the CCSVSN_NAMES_NUMS Procedure

Explanation

CS-CCSVERSIONV is passed to the procedure. It enables you to specify either of the following two types of information to be returned in the output parameters:

Value	Value Name	Return the names and numbers of the . . .
0	CS-CHARACTER-SET-V	Coded character sets
1	CS-CCSVERSION-V	Ccsversions

TOTAL is returned by the procedure. It contains the number of coded character set or ccsversion entries that exist.

NAMES-ARY is returned by the procedure. Each entry contains the name of a coded character set or ccsversion defined in the file SYSTEM/CCSFILE. Each name uses one element of NAMES-ARY and is 17 characters long. In this example, up to 20 names can be stored in the record. The *MultiLingual System Administration, Operations, and Programming Guide* also lists all the coded character sets and ccsversions. The recommended array size is 340 characters.

NUMS-ARY is returned by the procedure. NUMS-ARY contains all the coded character set or ccsversion numbers defined on the host. Each number uses one element of NUMS-ARY. Each element in NUMS-ARY corresponds to a parallel entry in NAMES-ARY and corresponds to a 17-character name. The record can hold up to 20 numbers.

The *MultiLingual System Administration, Operations, and Programming Guide* also provides all the numbers for the coded character sets and ccsversions.

RESULT is returned as the value of the procedure. It indicates whether an error occurred in the CCSVSN_NAMES_NUMS procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CCSVSN_NAMES_NUMS are as follows:

1	1002	3001
1001	3000	3006

Sample output from Example 16–5 follows:

```
RESULT =            1

CCSversion Name     CCSversion Number
---------------     -----------------
ASERIESNATIVE                     0
SWISS                            64
SWEDISH1                         99
SPANISH                          98
CANADAEBCDIC                     74
CANADAGP                         75
FRANCE                           35
NORWAY                           71
```

For more information on the error result values, see Table 16–1 in this section.

CENTRALSTATUS

This procedure returns the name and number of the system default ccsversion, the name of the system default language, and the name of the system default convention.

You might use this procedure to provide a means for your application users to inquire about the default settings on the host computer.

Example

Example 16–6 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CENTRALSTATUS library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example returns the current values for the system default ccsversion, language, and convention. These are arbitrary system values and might not be the same on every system.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/CENTRALSTATUS."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.
```

Example 16–6. Calling the CENTRALSTATUS Procedure

```
01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(15)   VALUE "System Defaults".
    05  FILLER           PIC X(65)   VALUE SPACE.

01  OF-3.
    05  FILLER           PIC X(15)   VALUE ALL "-".
    05  FILLER           PIC X(65)   VALUE SPACE.
01  OF-4.
    05  FILLER           PIC X(13)   VALUE "Field Meaning".
    05  FILLER           PIC X(29)   VALUE SPACE.
    05  FILLER           PIC X(08)   VALUE "Location".
    05  FILLER           PIC X(08)   VALUE SPACE.
    05  FILLER           PIC X(05)   VALUE "Value".
    05  FILLER           PIC X(14)   VALUE SPACE.
01  OF-5.
    05  FILLER           PIC X(13)   VALUE ALL "-".
    05  FILLER           PIC X(29)   VALUE SPACE.
    05  FILLER           PIC X(08)   VALUE ALL "-".
    05  FILLER           PIC X(08)   VALUE SPACE.
    05  FILLER           PIC X(05)   VALUE ALL "-".
    05  FILLER           PIC X(14)   VALUE SPACE.
01  DF-1.
    05  FILLER           PIC X(11)   VALUE "CCSVersion:".
    05  FILLER           PIC X(07)   VALUE SPACE.
    05  D1-SYS-ELEM      PIC X(17).
    05  FILLER           PIC X(45)   VALUE SPACE.
01  DF-2.
    05  FILLER           PIC X(11)   VALUE "Language:  ".
    05  FILLER           PIC X(07)   VALUE SPACE.
    05  D2-SYS-ELEM      PIC X(17).
    05  FILLER           PIC X(45)   VALUE SPACE.
01  DF-3.
    05  FILLER           PIC X(11)   VALUE "Convention:".
    05  FILLER           PIC X(07)   VALUE SPACE.
    05  D3-SYS-ELEM      PIC X(17).
    05  FILLER           PIC X(45)   VALUE SPACE.
01  DF-4.
    05  FILLER           PIC X(39)
        VALUE "System Default CCSVersion Number:      ".
```

Example 16-6. Calling the CENTRALSTATUS Procedure

```
    05  FILLER           PIC X(06)   VALUE SPACE.
    05  FILLER           PIC X(03)   VALUE "(1)".
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D4-CONTROL-ELEM  PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(17)   VALUE SPACE.
****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

 01  CONTROL-INFO                    USAGE BINARY.
     05 CONTROL-ELEM      PIC S9(11)  OCCURS 8 TIMES.
 01  SUB                 PIC 9(02).
 01  SYS-INFO.
     05  SYS-ELEM        PIC X(17)   OCCURS 3 TIMES.

 77  CS-DATAOKV          PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV           PIC S9(11)  USAGE BINARY    VALUE 0.
 77  RESULT              PIC S9(11)  USAGE BINARY.
 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CENTRALSTATUS.
   CLOSE OUTPUT-FILE.
   STOP RUN.
***** CENTRALSTATUS ********************************************
 CENTRALSTATUS.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   CALL "CENTRALSTATUS OF CENTRALSUPPORT"
     USING SYS-INFO,
           CONTROL-INFO
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE SPACE TO OUTPUT-RECORD
          WRITE OUTPUT-RECORD
          WRITE OUTPUT-RECORD FROM OF-2
          WRITE OUTPUT-RECORD FROM OF-3
```

Example 16-6. Calling the CENTRALSTATUS Procedure

```
           MOVE 1 TO SUB
           PERFORM DISPLAYSYSTEMINFO UNTIL SUB IS GREATER THAN 3
           MOVE SPACE TO OUTPUT-RECORD
           WRITE OUTPUT-RECORD
           WRITE OUTPUT-RECORD FROM OF-4
           WRITE OUTPUT-RECORD FROM OF-5
           MOVE 1 TO SUB
           PERFORM DISPLAYCONTROLINFO UNTIL SUB IS GREATER THAN 8.

***** DISPLAYSYSTEMINFO ****************************************
 DISPLAYSYSTEMINFO.
   IF SUB IS EQUAL TO 1
     THEN MOVE SYS-ELEM(SUB) TO D1-SYS-ELEM
          WRITE OUTPUT-RECORD FROM DF-1.
   IF SUB IS EQUAL TO 2
     THEN MOVE SYS-ELEM(SUB) TO D2-SYS-ELEM
          WRITE OUTPUT-RECORD FROM DF-2.
   IF SUB IS EQUAL TO 3
     THEN MOVE SYS-ELEM(SUB) TO D3-SYS-ELEM
          WRITE OUTPUT-RECORD FROM DF-3.
   ADD 1 TO SUB.

***** DISPLAYCONTROLINFO ****************************************
 DISPLAYCONTROLINFO.
   IF SUB IS EQUAL TO 1
     THEN MOVE CONTROL-ELEM(SUB) TO D4-CONTROL-ELEM
          WRITE OUTPUT-RECORD FROM DF-4.
   ADD 1 TO SUB.
```

Example 16-6. Calling the CENTRALSTATUS Procedure

Explanation

SYS-INFO is returned by the procedure. It is recommended that the size of the record be 51 characters. It contains three items, each 17 characters long, in the following order:

1. System default ccsversion name
2. System default language name
3. System default convention name

Each name is 17 characters long. Names shorter than 17 characters are padded on the right with blanks.

CONTROL-INFO is returned by the procedure. It is eight words long and contains the following information.

Location	Information
Word [0]	System default ccsversion number
Word [1] through [7]	Reserved

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CENTRALSTATUS are as follows:

1	1002	3001
1001	3000	

Sample output from Example 16–6 follows:

```
RESULT =            1

System Defaults
---------------
CCSVersion:       ASERIESNATIVE
Language:         ENGLISH
Convention:       ASERIESNATIVE



Field Meaning                              Location        Value
-------------                              --------        -----
System Default CCSVersion Number:             (1)              0
```

For more information on the error result values, see Table 16–1 in this section.

CNV_CURRENCYEDITTMP_DOUBLE_COB

This procedure receives a double-precision integer and formats it to represent a currency value according to the template specified. The template can be retrieved for any convention from the CNV_TEMPLATE_COB procedure or can be created by the user.

For More Information

The *MultiLingual System Administration, Operations, and Programming Guide* describes all the conventions and the type of currency formatting associated with each convention.

You might want to print a report with the numeric and currency formats for the Costa Rica conventions (for example, CRC 89.99) or for the Norway conventions (for example, NKR 89.99).

Example

Example 16–7 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_CURRENCYEDITTMP_DOUBLE_COB library procedure. The declarations identify the category of data-item required for parameter matching.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

```
  IDENTIFICATION DIVISION
            ENVIRONMENT DIVISION.
            INPUT-OUTPUT SECTION.
            FILE-CONTROL.
              SELECT OUTPUT-FILE ASSIGN TO DISK.
            DATA DIVISION.
            FILE SECTION.
            FD OUTPUT-FILE
                LABEL RECORD IS STANDARD
                VALUE OF TITLE IS "OUT/COBOL74/CUREDITTMP."
                PROTECTION SAVE
                RECORD CONTAINS 80 CHARACTERS
                DATA RECORD IS OUTPUT-RECORD.
            01 OUTPUT-RECORD PIC X(80).
            WORKING-STORAGE SECTION.
            01 OF-1.
               05 FILLER PIC X(09) VALUE "RESULT = ".
               05 OF-RESULT PIC ZZZZZZZZZZZ9.
               05 FILLER PIC X(59) VALUE SPACE.
            01 OF-2.
               05 FILLER PIC X(09) VALUE "CE-ARY = ".
               05 OF-CE-ARY PIC X(30).
               05 FILLER PIC X(41) VALUE SPACE.
            ****************************************************************
            *** The following global declarations are used as parameters ***
            *** to the CENTRALSUPPORT procedures. ***
            ****************************************************************
            01 CE-ARY PIC X(30).
            01 CNV-NAME PIC X(17).
            01 TMP-ARY PIC X(48).
            77 AMT PIC S9(23) USAGE BINARY.
            77 PRECISION PIC S9(11) USAGE BINARY.
            77 CS-DATAOKV PIC S9(11) USAGE BINARY VALUE 1.
            77 CS-FALSEV PIC S9(11) USAGE BINARY VALUE 0.
            77 RESULT PIC S9(11) USAGE BINARY.
            PROCEDURE DIVISION.
            INTLCOBOL74.
```

Example 16–7. Calling the CNV_CURRENCYEDITTMP_DOUBLE_COB Procedure

```
  OPEN OUTPUT OUTPUT-FILE.
  PERFORM CNV-CURRENCYEDITTMP-DOUBLE-COB.
  CLOSE OUTPUT-FILE.
  STOP RUN.
***** CNV-CURRENCYEDITTMP-DOUBLE-COB ***************************
CNV-CURRENCYEDITTMP-DOUBLE-COB.
  CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
  MOVE 1234567 TO AMT.
  MOVE 2 TO PRECISION.
  MOVE "ASERIESNATIVE" TO CNV-NAME.
  MOVE "!N[-]CT[,:0,3]D[.]#!" TO TMP-ARY.
  CALL "CNV_CURRENCYEDITTMP_DOUBLE_COB OF CENTRALSUPPORT"
    USING AMT,
          PRECISION,
          TMP-ARY,
          CNV-NAME,
          CE-ARY
    GIVING RESULT.
  MOVE RESULT TO OF-RESULT.
  WRITE OUTPUT-RECORD FROM OF-1.
  IF RESULT IS EQUAL TO CS-DATAOKV
    THEN MOVE CE-ARY TO OF-CE-ARY
         WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-7. Calling the CNV_CURRENCYEDITTMP_DOUBLE_COB Procedure

Explanation

AMT is a double–precision integer passed to the procedure. It contains the monetary value to be formatted.

PRECISION is an integer passed to the procedure. It specifies the name of the number of digits in AMT to be placed after the decimal symbol.

CE-ARY is returned by the procedure. It contains the formatted monetary value. The recommended size of the formatted amount is 20 characters.

TMP-ARY is passed to the procedure. It contains the formatting template used to format the monetary value. The recommended size of a template is 48 characters.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_CURRENCYEDITTMP_DOUBLE_COB are as follows:

1	1002	3000	3009
1001	2002	3002	3038

Sample output from Example 16–7 follows:

```
RESULT =            1
CE-ARY = Kr.12 345,67
```

For more information on the error result values, see Table 16–1 in this section.

CNV_DISPLAYMODEL_COB

This procedure returns either a numeric date or numeric time display model. A display model is a format that you can display to the user to show the form of the requested input. For example, YYDDMM is a display model that shows a user that the date must be entered in that form. The procedure creates the display model according to the convention and language that you specify.

Example

Example 16–8 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_DISPLAYMODEL_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example obtains a date display model from the ASeriesNative convention. The display model is translated to English and returned in DM-ARY.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/CNVDSPMODELCOB."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.
```

Example 16–8. Calling the CNV_DISPLAYMODEL_COB Procedure

```
01  OUTPUT-RECORD        PIC X(80).

 WORKING-STORAGE SECTION.

 01  OF-1.
     05  FILLER           PIC X(09)   VALUE "RESULT = ".
     05  OF-RESULT        PIC ZZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.
 01  OF-2.

     05  FILLER           PIC X(09)   VALUE "DM-ARY = ".
     05  OF-DM-ARY        PIC X(10).
     05  FILLER           PIC X(61)   VALUE SPACE.
****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

 01  CNV-NAME             PIC X(17).
 01  DM-ARY               PIC X(10).
 01  LANG-NAME            PIC X(17).

 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-DATE-DISPLAYMODEL PIC S9(11)  USAGE BINARY    VALUE 0.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  RESULT               PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-DISPLAYMODEL-COB.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** CNV-DISPLAYMODEL-COB ***************************************
 CNV-DISPLAYMODEL-COB.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "ASERIESNATIVE" TO CNV-NAME.
   MOVE "ENGLISH" TO LANG-NAME.
   CALL "CNV_DISPLAYMODEL_COB OF CENTRALSUPPORT"
     USING CS-DATE-DISPLAYMODEL,
           CNV-NAME,
           LANG-NAME,
           DM-ARY
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
```

```
        THEN MOVE DM-ARY TO OF-DM-ARY
             WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-8. Calling the CNV_DISPLAYMODEL_COB Procedure

Explanation

CS-DATE-DISPLAYMODEL is an integer passed to the procedure. It indicates whether you want the display model to be a numeric date or a numeric time.

Value	If the sample data item is . . .	The display model is a . . .
0	CS-DATE-DISPLAYMODEL	Numeric date
1	CS-TIME-DISPLAYMODEL	Numeric time

CNV-NAME is passed to the procedure. It contains the name of the convention from which the date or time model is retrieved if this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for the list of convention names and the explanation of the hierarchy.

LANG-NAME is passed to the procedure. It contains the name of the language in which the date or time components are to be displayed. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for information about determining the valid language names on your system and the explanation of the hierarchy.

DM-ARY is returned by the procedure. It contains the display model. The recommended size of the display model is 10 characters.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_DISPLAYMODEL_COB are as follows:

1	2001	3001
1001	2002	3002
1002	3000	3006

Sample output from Example 16–8 follows:

```
RESULT =            1
DM-ARY = mm/dd/yyyy
```

For more information on the error result values, see Table 16–1 in this section.

CNV_FORMATDATETMP_COB

This procedure formats a date according to a template. You specify the template, date value, and language in which the date is to be displayed. The procedure then returns the formatted date. The template may be retrieved for any convention from the CNV_TEMPLATE_COB procedure or may be created by the user.

Example

Example 16–9 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_FORMATDATETMP_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example formats a date using a template provided by the calling program. The formatted date is translated to English and returned in FD-ARY. The date consists of an unabridged day of week name, abbreviated month name, numeric day of month, day of month suffix “rd,” and numeric year.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/CNVFMTDATETMPCOB."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.
01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.
```

Example 16–9. Calling the CNV_FORMATDATETMP_COB Procedure

```
 01  OF-1.
     05  FILLER           PIC X(09)   VALUE "RESULT = ".
     05  OF-RESULT        PIC ZZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.
 01  OF-2.
     05  FILLER           PIC X(09)   VALUE "FD-ARY = ".
     05  OF-FD-ARY        PIC X(45).
     05  FILLER           PIC X(26)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

 01  DATE-ARY             PIC X(08).
 01  FD-ARY               PIC X(45).
 01  LANG-NAME            PIC X(17).
 01  TMP-ARY              PIC X(48).

 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  RESULT               PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-FORMATDATETMP-COB.
   CLOSE OUTPUT-FILE.
   STOP RUN.
***** CNV-FORMATDATETMP-COB ************************************
 CNV-FORMATDATETMP-COB.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "19901003" TO DATE-ARY.
   MOVE "!W!,!1N!.!DE!,!Y!" TO TMP-ARY.
   MOVE "ENGLISH" TO LANG-NAME.
   CALL "CNV_FORMATDATETMP_COB OF CENTRALSUPPORT"
     USING DATE-ARY,
           TMP-ARY,
           LANG-NAME,
           FD-ARY
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE FD-ARY TO OF-FD-ARY
          WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-9. Calling the CNV_FORMATDATETMP_COB Procedure

Explanation

DATE-ARY is passed into the procedure. It contains the date to be formatted. The date must be in the form YYYYMMDD. The fields of the record have fixed positions. You must use blanks or zeros in any fields that you omit.

TMP-ARY is passed into the procedure. It contains the template used to format the date. The recommended size of a template is 48 characters. The template must use the control characters described in the *MultiLingual System Administration, Operations, and Programming Guide*.

LANG-NAME is passed into the procedure. It contains the name of the language to be used in formatting the date. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for information about determining the valid language names on your system.

FD-ARY is returned by the procedure. It contains the formatted date. The recommended size of the formatted date is 45 characters.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by CNV_FORMATDATETMP_COB are as follows:

1	3001	3030	3048
1001	3002	3045	3057
1002	3011	3046	3058
2001	3012	3047	3059
3000			

Sample output from Example 16–9 follows:

```
RESULT =            1
FD-ARY = Wednesday, Oct. 3rd, 1990
```

For more information on the error result values, see Table 16–1 in this section.

CNV_FORMATDATE_COB

This procedure receives a date and formats it in the form you select according to the convention and language you specify.

You might use this procedure to output a date according to the Greek long-date format and Greek language, for example.

Example

Example 16–10 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_FORMATDATE_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example formats the date in numeric form using the Netherlands convention. The English language is specified.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/CNVFMTDATECOB."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.

01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
```

Example 16–10. Calling the CNV_FORMATDATE_COB Procedure

```
     05  FILLER           PIC X(09)   VALUE "FD-ARY = ".
     05  OF-FD-ARY        PIC X(10).
     05  FILLER           PIC X(61)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

 01  CNV-NAME             PIC X(17).
 01  DATE-ARY             PIC X(08).
 01  FD-ARY               PIC X(10).
 01  LANG-NAME            PIC X(17).

 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  CS-NDATEV            PIC S9(11)  USAGE BINARY    VALUE 2.
 77  RESULT               PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-FORMATDATE-COB.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** CNV-FORMATDATE-COB ****************************************
 CNV-FORMATDATE-COB.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "17760704" TO DATE-ARY.
   MOVE "Netherlands" TO CNV-NAME.
   MOVE "ENGLISH" TO LANG-NAME.
   CALL "CNV_FORMATDATE_COB OF CENTRALSUPPORT"
     USING CS-NDATEV,
           DATE-ARY,
           CNV-NAME,
           LANG-NAME,
           FD-ARY
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
 IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE FD-ARY TO OF-FD-ARY
          WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-10. Calling the CNV_FORMATDATE_COB Procedure

Explanation

CS-NDATEV is an integer passed by reference to the procedure. It indicates which of the following three formats will be used to format the date:

Value	Sample Data Item	Meaning
0	LONG-DATE-V	Use the long date format.
1	SHORT-DATE-V	Use the short date format.
2	NUMERIC-DATE-V	Use the numeric date format.

DATE-ARY is passed to the procedure. It contains the date to be formatted. The date must be in the form YYYYMMDD, left justified. The fields of the array have fixed positions. You must use blanks or zeros in any fields that you omit.

CNV-NAME is passed to the procedure. It contains the name of the convention to be used to format the date value. If this parameter contains all blanks or 17 character zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for the list of convention names and the explanation of the hierarchy.

LANG-NAME is passed to the procedure. It contains the language to be used in formatting the date. If this parameter contains all blanks or 17 character zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for information about determining the valid language names on your system and the explanation of the hierarchy.

FD-ARY is returned by the procedure. It contains the formatted date. The recommended length of a formatted date is 45 characters.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by CNV_FORMATDATE_COB are as follows:

1	3000	3012	3048
2	3001	3045	3057
1001	3002	3046	3058
2001	3006	3047	3059
2002			

Sample output from Example 16–10 follows:

```
RESULT =            1
FD-ARY = 4.7.76
```

For more information on the error result values, see Table 16–1 in this section.

CNV_CURRENCYEDIT_DOUBLE_COB

This procedure converts a double-precision integer containing a monetary value and converts it to a formatted monetary value. The procedure uses the monetary template of the convention you specify to accomplish the formatting.

The *MultiLingual System Administration, Operations, and Programming Guide* describes all the conventions and the type of currency formatting associated with each convention.

You might want to print a report with the numeric and currency formats for the Costa Rica conventions (for example, CRC 89.99), or for the Norway conventions (for example, NKR 89.99).

Example

Example 16–11 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_CURRENCYEDIT_DOUBLE_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example converts a double-precision integer and edits monetary symbols from the Denmark convention into an EBCDIC string.

```
  IDENTIFICATION DIVISION.
            ENVIRONMENT DIVISION.
            INPUT-OUTPUT SECTION.
            FILE-CONTROL.
              SELECT OUTPUT-FILE ASSIGN TO DISK.
            DATA DIVISION.
            FILE SECTION.
            FD OUTPUT-FILE
               LABEL RECORD IS STANDARD
               VALUE OF TITLE IS "OUT/COBOL74/CUREDITDOUB."
               PROTECTION SAVE
               RECORD CONTAINS 80 CHARACTERS
               DATA RECORD IS OUTPUT-RECORD.
            01 OUTPUT-RECORD PIC X(80).
            WORKING-STORAGE SECTION.
            01 OF-1.
               05 FILLER PIC X(09) VALUE "RESULT = ".
               05 OF-RESULT PIC ZZZZZZZZZZZ9.
               05 FILLER PIC X(59) VALUE SPACE.
```

Example 16–11. Calling the CNV_CURRENCYEDIT_DOUBLE_COB Procedure

```
01 OF-2.
   05 FILLER PIC X(09) VALUE "CE-ARY = ".
   05 OF-CE-ARY PIC X(30).
   05 FILLER PIC X(41) VALUE SPACE.
****************************************************************
*** The following global declarations are used as parameters ***
*** to the CENTRALSUPPORT procedures. ***
****************************************************************
1 CE-ARY PIC X(30).
01 CNV-NAME PIC X(17).
77 AMT PIC S9(23) USAGE BINARY.
77 PRECISION PIC S9(11) USAGE BINARY.
77 CS-DATAOKV PIC S9(11) USAGE BINARY VALUE 1.
77 CS-FALSEV PIC S9(11) USAGE BINARY VALUE 0.
77 RESULT PIC S9(11) USAGE BINARY.
PROCEDURE DIVISION.
INTLCOBOL74.
  OPEN OUTPUT OUTPUT-FILE.
  PERFORM CNV-CURRENCYEDIT-DOUBLE-COB.
  CLOSE OUTPUT-FILE.
  STOP RUN.
***** CNV-CURRENCYEDIT-DOUBLE-COB ********************************
CNV-CURRENCYEDIT-DOUBLE-COB.
  CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
  MOVE 1234567 TO AMT.
  MOVE 2 TO PRECISION.
  MOVE "Denmark" TO CNV-NAME.
  CALL "CNV_CURRENCYEDIT_DOUBLE_COB OF CENTRALSUPPORT"
    USING AMT,
          PRECISION,
          CNV-NAME,
          CE-ARY
    GIVING RESULT.
  MOVE RESULT TO OF-RESULT.
  WRITE OUTPUT-RECORD FROM OF-1.
  IF RESULT IS EQUAL TO CS-DATAOKV
    THEN MOVE CE-ARY TO OF-CE-ARY
         WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-11. Calling the CNV_CURRENCYEDIT_DOUBLE_COB Procedure

Explanation

AMT is a double-precision integer passed to the procedure. It contains the monetary value to be formatted. PRECISION is an integer passed to the procedure. It specifies the number of digits in AMT to be placed after the decimal symbol.

CNV-NAME is passed to the procedure. It contains the name of the convention to be used to format the monetary value. If this parameter contains all blanks or zeros, the procedure use the hierarchy to determine the convention to be used. Refer to the

MultiLingual System Administration, Operations, and Programming Guide for the list of convention names and the explanation of the hierarchy.

CE-ARY is returned by the procedure. It contains the formatted monetary value. The recommended size of the formatted amount is 20 characters.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_CURRENCYEDIT_DOUBLE_COB are as follows:

1	1002	3000	3009
1001	2002	3002	3038

Sample output from Example 16–11 follows:

```
RESULT =            1
CE-ARY = Kr.12 345,67
```

For more information on the error result values, see Table 16–1 in this section.

CNV_FORMATTIMETMP_COB

This procedure formats a time according to a template. You specify the template, time value, and language in which the time is to be displayed. The procedure then returns the formatted time. The template may be retrieved for any convention from the CNV_TEMPLATE_COB procedure or may be created by the user.

With this procedure, if the time template is !0t!:!0m!:0s!, the language is English, and the input time is 1255016256, the numeric time is formatted as 12:55:01.

Example

Example 16–12 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_FORMATTIMETMP_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use. This example formats a caller-supplied time using a template also passed in by the calling program. Alphabetic time components are translated to English and returned in FT-ARY.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.
```

Example 16–12. Calling the CNV_FORMATTIMETMP_COB Procedure

```
DATA DIVISION.
 FILE SECTION.
 FD  OUTPUT-FILE
     LABEL RECORD IS STANDARD
     VALUE OF TITLE IS "OUT/COBOL74/CNVFMTTIMETMPCOB."
     PROTECTION SAVE
     RECORD CONTAINS 80 CHARACTERS
     DATA RECORD IS OUTPUT-RECORD.
 01  OUTPUT-RECORD        PIC X(80).
 WORKING-STORAGE SECTION.
 01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
 01  OF-2.
    05  FILLER           PIC X(13)   VALUE "FT-ARY = ".
    05  OF-FT-ARY        PIC X(30).
05  FILLER           PIC X(37)   VALUE SPACE.
****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************
 01  FT-ARY               PIC X(30).
 01  LANG-NAME            PIC X(17).
 01  TIME-ARY             PIC X(10).
 01  TMP-ARY              PIC X(48).
 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  RESULT               PIC S9(11)  USAGE BINARY.
 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-FORMATTIMETMP-COB.
   CLOSE OUTPUT-FILE.
   STOP RUN.
***** CNV-FORMATTIMETMP-COB ************************************
 CNV-FORMATTIMETMP-COB.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "114958" TO TIME-ARY.
   MOVE "!T! !I! !M! !K! !S! !R!" TO TMP-ARY.
   MOVE "ENGLISH" TO LANG-NAME.
   CALL "CNV_FORMATTIMETMP_COB OF CENTRALSUPPORT"
     USING TIME-ARY,
           TMP-ARY,
           LANG-NAME,
           FT-ARY
```

```
       GIVING RESULT.
```

Example 16-12. Calling the CNV_FORMATTIMETMP_COB Procedure

```
MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE FT-ARY TO OF-FT-ARY
          WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-12. Calling the CNV_FORMATTIMETMP_COB Procedure

Explanation

TIME-ARY is passed to the procedure. You specify the time to be formatted in the form HHMMSSPPPP. The partial seconds field, PPPP, is optional. The fields of the array have fixed positions. You must use blanks or zeros in any fields that you omit.

TMP-ARY is passed to the procedure. You specify the template to be used to format the time in this parameter. The recommended length of a template is 48 characters. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for information about creating a template.

LANG-NAME is passed to the procedure. You specify the language to be used in formatting the time in this parameter. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for information about determining the valid language names on your system and the explanation of the hierarchy.

FT-ARY is returned by the procedure. It contains the time value formatted according to the template and language you designated. The recommended length of a formatted time is 45 characters.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by CNV_FORMATTIMETMP_COB are as follows:

1	3000	3013	3053
1001	3001	3030	3054
1002	3002	3051	3055
2001	3011	3052	

Sample output from Example 16–12 follows:

```
RESULT =             1
FT-ARY =     11 hours 49 minutes 58 seconds
```

For more information on the error result values, see Table 16–1 in this section.

CNV_FORMATTIME_COB

This procedure formats a user-supplied time according to the convention, language, and type of time that you specify.

Example

Example 16–13 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_FORMATTIME_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example formats the time in numeric form using the Belgium convention. The formatted time is returned in FT-ARY.

```
   IDENTIFICATION DIVISION.
   ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT OUTPUT-FILE ASSIGN TO DISK.
   DATA DIVISION.
   FILE SECTION.
   FD  OUTPUT-FILE
       LABEL RECORD IS STANDARD
       VALUE OF TITLE IS "OUT/COBOL74/CNVFMTTIMECOB."
       PROTECTION SAVE
       RECORD CONTAINS 80 CHARACTERS
       DATA RECORD IS OUTPUT-RECORD.
 01  OUTPUT-RECORD        PIC X(80).

   WORKING-STORAGE SECTION.

   01  OF-1.
       05  FILLER           PIC X(09)   VALUE "RESULT = ".
       05  OF-RESULT        PIC ZZZZZZZZZZZ9.
       05  FILLER           PIC X(59)   VALUE SPACE.
   01  OF-2.
       05  FILLER           PIC X(09)   VALUE "FT-ARY = ".
       05  OF-FT-ARY        PIC X(30).
       05  FILLER           PIC X(41)   VALUE SPACE.
```

Example 16–13. Calling the CNV_FORMATTIME_COB procedure

```
****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

 01  CNV-NAME            PIC X(17).
 01  FT-ARY              PIC X(30).
 01  LANG-NAME           PIC X(17).
 01  TIME-ARY            PIC X(10).

 77  CS-DATAOKV          PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV           PIC S9(11)  USAGE BINARY    VALUE 0.
 77  CS-NTIMEV           PIC S9(11)  USAGE BINARY    VALUE 4.
 77  RESULT              PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-FORMATTIME-COB.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** CNV-FORMATTIME-COB ****************************************
 CNV-FORMATTIME-COB.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "114958" TO TIME-ARY.
   MOVE "Belgium" TO CNV-NAME.
   MOVE "ENGLISH" TO LANG-NAME.
   CALL "CNV_FORMATTIME_COB OF CENTRALSUPPORT"
     USING CS-NTIMEV,
           TIME-ARY,
           CNV-NAME,
           LANG-NAME,
           FT-ARY       GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE FT-ARY TO OF-FT-ARY
          WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-13. Calling the CNV_FORMATTIME_COB procedure

Explanation

CS-NTIMEV is passed by reference to the procedure. It indicates which of the following two formats will be used to edit the time:

Value	Sample Data Item	Meaning
3	LONG-TIME-V	Use the long time format.
4	NUMERIC-TIME-V	Use the numeric time format.

TIME-ARY is passed to the procedure. It contains the time to be formatted in the form HHMMSSPPPP, left justified. The partial seconds field, PPPP, is optional. The fields of the record have fixed positions. You must use blanks or zeros in any fields that you omit.

CNV-NAME is passed to the procedure. It contains the name of the convention to be used to edit the time value. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for the list of convention names and the explanation of the hierarchy.

LANG-NAME is passed into the procedure. It contains the language to be used in formatting the time. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for information about determining the valid language names on your system and the explanation of the hierarchy.

FT-ARY is returned by the procedure. It contains the formatted time value. The recommended length of the formatted time is 45 characters.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by CNV_FORMATTIME_COB are as follows:

1	2002	3006	3052
1001	3000	3011	3053
1002	3001	3013	3054
2001	3002	3051	3055

Sample output from Example 16–13 follows:

```
RESULT =            1
FT-ARY = 11:49:58
```

For more information on the error result values, see Table 16–1 in this section.

CNV_FORMSIZE

This procedure returns the default lines-per-page and default characters-per-line values from the specified convention. Each convention provides these values to be used with printed output.

Example

Example 16–14 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_FORMSIZE library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example obtains paper dimensions (lines per page and characters per line) from the Denmark convention.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.
DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/CNVFORMSIZE."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.
01  OUTPUT-RECORD        PIC X(80).
WORKING-STORAGE SECTION.
01  OF-1.
    05  FILLER           PIC X(22)
        VALUE "RESULT              = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(46)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(22)
        VALUE "Lines per Page      = ".
    05  OF-LPP           PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(46)   VALUE SPACE.
01  OF-3.
```

Example 16–14. Calling the CNV_FORMSIZE Procedure

```
     05  FILLER           PIC X(22)
         VALUE "Characters per Line = ".
     05  OF-CPL           PIC ZZZZZZZZZZ9.
     05  FILLER           PIC X(46)   VALUE SPACE.
****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

 01  CNV-NAME             PIC X(17).
 77  CPL                  PIC S9(11)  USAGE BINARY.
 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  LPP                  PIC S9(11)  USAGE BINARY.
 77  RESULT               PIC S9(11)  USAGE BINARY.
 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-FORMSIZE.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** CNV-FORMSIZE ************************************************
 CNV-FORMSIZE.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "Denmark" TO CNV-NAME.
   CALL "CNV_FORMSIZE OF CENTRALSUPPORT"
     USING CNV-NAME,
           LPP,
           CPL
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE SPACE TO OUTPUT-RECORD
          MOVE LPP TO OF-LPP
          WRITE OUTPUT-RECORD FROM OF-2
          MOVE CPL TO OF-CPL
          WRITE OUTPUT-RECORD FROM OF-3.
```

Example 16-14. Calling the CNV_FORMSIZE Procedure

Explanation

CNV-NAME is passed to the procedure. It contains the name of the convention to be used to specify the default printer form sizes. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for the list of convention names and the explanation of the hierarchy.

CPL is returned by the procedure. It contains the default number of characters per line specified by the convention you referenced.

LPP is returned by the procedure. It contains the default number of lines per page specified by the convention you referenced.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_FORMSIZE are as follows:

1	1002	3000
1001	2002	

Sample output from Example 16–14 follows:

```
RESULT              =            1
Lines per Page      =           70
Characters per Line =           82
```

For more information on the error result values, see Table 16–1 in this section.

CNV_NAMES

This procedure returns a list of convention names and the total number of convention that are available on the system. The first name is the system default name.

You might use this procedure to obtain the name of a convention to be used as input to another procedure.

Example

Example 16–15 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_NAMES library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example obtains the names of conventions currently available on the system. Note that this is an arbitrary list that may differ from system to system.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/CNVNAMES."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.
```

Example 16–15. Calling the CNV_NAMES Procedure

```
 01  OF-1.
     05  FILLER           PIC X(09)   VALUE "RESULT = ".
     05  OF-RESULT        PIC ZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.
 01  OF-2.
     05  FILLER           PIC X(16)   VALUE "Convention Names".
     05  FILLER           PIC X(64)   VALUE SPACE.
 01  OF-3.
     05  FILLER           PIC X(16)   VALUE ALL "-".
     05  FILLER           PIC X(64)   VALUE SPACE.
 01  OF-4.
     05  OF-NAMES-ELEM    PIC X(17).
     05  FILLER           PIC X(63)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

 01  NAMES-ARY.
     05  NAMES-ELEM       PIC X(17)   OCCURS 45 TIMES.

 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  RESULT               PIC S9(11)  USAGE BINARY.
 77  SUB                  PIC S9(11)  USAGE BINARY.
 77  TOTAL                PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-NAMES.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** CNV-NAMES ************************************************
 CNV-NAMES.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   CALL "CNV_NAMES OF CENTRALSUPPORT"
     USING TOTAL,
           NAMES-ARY
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE SPACE TO OUTPUT-RECORD
```

Example 16-15. Calling the CNV_NAMES Procedure

```
           WRITE OUTPUT-RECORD
           WRITE OUTPUT-RECORD FROM OF-2
           WRITE OUTPUT-RECORD FROM OF-3
           MOVE 1 TO SUB
           PERFORM DISPLAYNAMESARY UNTIL SUB IS GREATER THAN TOTAL.

***** DISPLAYNAMESARY ******************************************
 DISPLAYNAMESARY.
MOVE NAMES-ELEM(SUB) TO OF-NAMES-ELEM.
   WRITE OUTPUT-RECORD FROM OF-4.
   ADD 1 TO SUB.
```

Example 16-15. Calling the CNV_NAMES Procedure

Explanation

TOTAL is returned by the procedure. It contains the total number of conventions that reside on the system.

NAMES-ARY is returned by the procedure. Each element of the record contains the name of a convention defined in the SYSTEM/CONVENTIONS file. Each name uses one element of NAMES-ARY. The record can hold up to 20 names. Each name can be up to 17 characters long and is left justified in the field. If there are less than 17 characters in the name, the field is filled on the right with blanks.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_NAMES are as follows:

1	1001	3001

Sample output from Example 16–15 follows:

```
RESULT =            1

Convention Names
----------------
ASERIESNATIVE
Netherlands
Denmark
UnitedKingdom1
Turkey
Norway
Sweden
Greece
FranceListing
FranceBureautique
EuropeanStandard
```

```
Belgium
Spain
Switzerland
Zimbabwe
Italy
UnitedKingdom2
KENYA
NIGERIA
SOUTHAFRICA
CYRILLIC
BRAZIL
NEWZEALAND
STNDYUGOSLAVIAN
FRENCHCANADA
ARGENTINA
CHILE
COLOMBIA
COSTARICA
MEXICO
PERU
VENEZUELA
AUSTRALIA
EGYPT
ENGLISHCANADA
Japan1
Japan2
```

For more information on the error result values, see Table 16–1 in this section.

CNV_SYMBOLS

This procedure returns a list of numeric and monetary symbols defined for a specified convention.

Example

Example 16–16 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_SYMBOLS library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example obtains monetary and numeric symbols, monetary and numeric grouping specifications, and international currency notation defined for the Norway convention.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.
DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/CNVSYMBOLS."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.
01  OUTPUT-RECORD        PIC X(80).
WORKING-STORAGE SECTION.
01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(13)   VALUE "Field Meaning".
    05  FILLER           PIC X(22)   VALUE SPACE.
    05  FILLER           PIC X(14)   VALUE "Symbols Length".
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  FILLER           PIC X(18)   VALUE "Convention Symbols".
    05  FILLER           PIC X(10)   VALUE SPACE.
01  OF-3.
    05  FILLER           PIC X(13)   VALUE ALL "-".
    05  FILLER           PIC X(22)   VALUE SPACE.
```

Example 16–16. Calling the CNV_SYMBOLS Procedure

```
    05  FILLER           PIC X(14)   VALUE ALL "-".
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  FILLER           PIC X(18)   VALUE ALL "-".
    05  FILLER           PIC X(10)   VALUE SPACE.
01  DF-1.
    05  FILLER           PIC X(32)
        VALUE "International Currency Notation:".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D1-SYMLEN-ELEM   PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D1-SYM-ELEM      PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-2.
    05  FILLER           PIC X(32)
        VALUE "National Currency Notation:     ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D2-SYMLEN-ELEM   PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D2-SYM-ELEM      PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-3.
    05  FILLER           PIC X(32)
        VALUE "Monetary Thousands Separator:   ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D3-SYMLEN-ELEM   PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D3-SYM-ELEM      PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-4.
    05  FILLER           PIC X(32)
        VALUE "Monetary Decimal Symbol:        ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D4-SYMLEN-ELEM   PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D4-SYM-ELEM      PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-5.
    05  FILLER           PIC X(32)
        VALUE "Monetary Positive Symbol:       ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D5-SYMLEN-ELEM   PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D5-SYM-ELEM      PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-6.
```

Example 16-16. Calling the CNV_SYMBOLS Procedure

```
    05  FILLER           PIC X(32)
        VALUE "Monetary Negative Symbol:       ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D6-SYMLEN-ELEM   PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D6-SYM-ELEM      PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-7.
    05  FILLER           PIC X(32)
        VALUE "Monetary Left Enclosure Symbol: ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D7-SYMLEN-ELEM   PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D7-SYM-ELEM      PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-8.
    05  FILLER           PIC X(32)
        VALUE "Monetary Right Enclosure Symbol:".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D8-SYMLEN-ELEM   PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D8-SYM-ELEM      PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-9.
    05  FILLER           PIC X(32)
        VALUE "Numeric Thousands Separator:    ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D9-SYMLEN-ELEM   PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D9-SYM-ELEM      PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-10.
    05  FILLER           PIC X(32)
        VALUE "Numeric Decimal Symbol:         ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D10-SYMLEN-ELEM  PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D10-SYM-ELEM     PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-11.
    05  FILLER           PIC X(32)
        VALUE "Numeric Positive Symbol:        ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D11-SYMLEN-ELEM  PIC ZZZZZZZZZZZ9.
```

```
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D11-SYM-ELEM     PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
```

Example 16-16. Calling the CNV_SYMBOLS Procedure

```
01  DF-12.
    05  FILLER           PIC X(32)
        VALUE "Numeric Negative Symbol:        ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D12-SYMLEN-ELEM  PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D12-SYM-ELEM     PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-13.
    05  FILLER           PIC X(32)
        VALUE "Numeric Left Enclosure Symbol:  ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D13-SYMLEN-ELEM  PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D13-SYM-ELEM     PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-14.
    05  FILLER           PIC X(32)
        VALUE "Numeric Right Enclosure Symbol: ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D14-SYMLEN-ELEM  PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D14-SYM-ELEM     PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-15.
    05  FILLER           PIC X(32)
        VALUE "Monetary Grouping Specification:".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D15-SYMLEN-ELEM  PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D15-SYM-ELEM.
        10  D15-SYM-1    PIC X(12).
        10  D15-SYM-2    PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
01  DF-16.
    05  FILLER           PIC X(32)
        VALUE "Numeric Grouping Specification: ".
    05  FILLER           PIC X(05)   VALUE SPACE.
    05  D16-SYMLEN-ELEM  PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(03)   VALUE SPACE.
    05  D16-SYM-ELEM.
```

```
        10 D16-SYM-1    PIC X(12).
        10  D16-SYM-2    PIC X(12).
    05  FILLER           PIC X(16)   VALUE SPACE.
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
 01  CNV-NAME            PIC X(17).
```

Example 16-16. Calling the CNV_SYMBOLS Procedure

```
 01  SYM-ARY.
     05  SYM-ELEM         PIC X(12)   OCCURS 18 TIMES.
 01  SYMLEN-ARY                       USAGE BINARY.
     05  SYMLEN-ELEM      PIC S9(11)  OCCURS 17 TIMES.
 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  MAX                  PIC S9(11)  USAGE BINARY    VALUE 18.
 77  RESULT               PIC S9(11)  USAGE BINARY.
 77  SUB                  PIC S9(11)  USAGE BINARY.
 77  TOTAL                PIC S9(11)  USAGE BINARY.
 PROCEDURE DIVISION.
 INTLCOBOL74.
   DISPLAY "***  INTL_COBOL74: CNV_SYMBOLS".
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-SYMBOLS.
   CLOSE OUTPUT-FILE.
   STOP RUN.
***** CNV-SYMBOLS **********************************************
 CNV-SYMBOLS.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "Norway" TO CNV-NAME.
   CALL "CNV_SYMBOLS OF CENTRALSUPPORT"
     USING CNV-NAME,
           TOTAL,
           SYMLEN-ARY,
           SYM-ARY
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE SPACE TO OUTPUT-RECORD
          WRITE OUTPUT-RECORD
          WRITE OUTPUT-RECORD FROM OF-2
          WRITE OUTPUT-RECORD FROM OF-3
          MOVE 1 TO SUB
          PERFORM DISPLAYARY UNTIL SUB IS GREATER THAN MAX.
***** DISPLAYARY **********************************************
 DISPLAYARY.
   IF SUB IS EQUAL TO 1
```

```
  THEN MOVE SYMLEN-ELEM(SUB) TO D1-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D1-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-1.
IF SUB IS EQUAL TO 2
  THEN MOVE SYMLEN-ELEM(SUB) TO D2-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D2-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-2.
IF SUB IS EQUAL TO 3
```

Example 16-16. Calling the CNV_SYMBOLS Procedure

```
  THEN MOVE SYMLEN-ELEM(SUB) TO D3-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D3-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-3.
IF SUB IS EQUAL TO 4
  THEN MOVE SYMLEN-ELEM(SUB) TO D4-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D4-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-4.
IF SUB IS EQUAL TO 5
  THEN MOVE SYMLEN-ELEM(SUB) TO D5-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D5-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-5.
IF SUB IS EQUAL TO 6
  THEN MOVE SYMLEN-ELEM(SUB) TO D6-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D6-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-6.
IF SUB IS EQUAL TO 7
  THEN MOVE SYMLEN-ELEM(SUB) TO D7-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D7-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-7.
IF SUB IS EQUAL TO 8
  THEN MOVE SYMLEN-ELEM(SUB) TO D8-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D8-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-8.
IF SUB IS EQUAL TO 9
  THEN MOVE SYMLEN-ELEM(SUB) TO D9-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D9-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-9.
IF SUB IS EQUAL TO 10
  THEN MOVE SYMLEN-ELEM(SUB) TO D10-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D10-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-10.
IF SUB IS EQUAL TO 11
  THEN MOVE SYMLEN-ELEM(SUB) TO D11-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D11-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-11.
IF SUB IS EQUAL TO 12
  THEN MOVE SYMLEN-ELEM(SUB) TO D12-SYMLEN-ELEM
```

```
       MOVE SYM-ELEM(SUB) TO D12-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-12.
IF SUB IS EQUAL TO 13
  THEN MOVE SYMLEN-ELEM(SUB) TO D13-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D13-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-13.
IF SUB IS EQUAL TO 14
  THEN MOVE SYMLEN-ELEM(SUB) TO D14-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D14-SYM-ELEM
       WRITE OUTPUT-RECORD FROM DF-14.
IF SUB IS EQUAL TO 15
```

Example 16-16. Calling the CNV_SYMBOLS Procedure

```
  THEN MOVE SYMLEN-ELEM(SUB) TO D15-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D15-SYM-1.
IF SUB IS EQUAL TO 16
  THEN MOVE SYMLEN-ELEM(SUB) TO D16-SYMLEN-ELEM
       MOVE SYM-ELEM(SUB) TO D15-SYM-2
       WRITE OUTPUT-RECORD FROM DF-15.
IF SUB IS EQUAL TO 17
  THEN MOVE SYM-ELEM(SUB) TO D16-SYM-1.
IF SUB IS EQUAL TO 18
  THEN MOVE SYM-ELEM(SUB) TO D16-SYM-2
       WRITE OUTPUT-RECORD FROM DF-16.
ADD 1 TO SUB.
```

Example 16-16. Calling the CNV_SYMBOLS Procedure

Explanation

CNV-NAME is passed to the procedure. It contains the name of the convention to be used to retrieve the monetary and numeric symbols. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for the list of convention names and the explanation of the hierarchy.

TOTAL is returned by the procedure. It contains the total number of symbols returned.

SYMLEN-ARY is returned by the procedure. It contains the lengths of all symbols being returned in SYM-ARY. The recommended length of SYM-ARY is 16 words.

SYM-ARY is returned by the procedure. Each element of the record contains a symbol defined in the monetary and numeric template for the specified convention. The corresponding entry in SYMLEN-ARY contains the length of each symbol. The maximum length of SYM-ARY is 216 bytes.

SYMLEN-ARY and SYM-ARY are parallel records. Each entry in SYMLEN-ARY specifies the number of characters the corresponding entry in SYM-ARY has. If an entry in SYMLEN-ARY is 0 (zero), it indicates that the symbol is not defined and the corresponding entry in SYM-ARY is filled with blanks. If an entry in SYMLEN-ARY is not 0 (zero), but the corresponding entry in SYM-ARY is all blanks, then the number of blanks specified by the SYMLEN-ARY entry forms the symbol.

The procedure returns the monetary and numeric templates defined by the convention in fixed-length fields. Each field is 12 bytes long except where noted.

The following table shows the monetary and numeric symbols that are returned in the record SYM-ARY and the offset in bytes of the field in which the symbol is returned:

Monetary Symbol	Offset	Numeric Symbol	Offset
International currency notation	0	Thousands separator	96
National currency symbol	12	Decimal symbol	108
Thousands separator	24	Positive sign	120
Decimal symbol	36	Negative sign	132
Positive sign	48	Left enclosure	144
Negative sign	60	Right enclosure	156
Left enclosure	72	Monetary grouping	168
Right enclosure	84	Numeric grouping	192

The monetary and numeric grouping each occupy two adjacent fields (24 bytes) in SYM_ARY. The monetary and numeric groupings, when present, are character strings consisting of unsigned integers separated by commas. The integers specify the number of digits in each group and appear exactly as declared in the monetary and numeric templates including embedded commas. The following table shows the offset in bytes of the fields in the record SYMLEN-ARY, which contain the symbol lengths for the monetary and numeric symbols:

Offset	Contains Length of	Offset	Contains Length of
0	International currency notation	8	Numeric thousands separator symbol
1	National currency symbol	9	Numeric decimal symbol
2	Monetary thousands separator	10	Numeric positive symbol
3	Monetary decimal symbol	11	Numeric negative symbol
4	Monetary positive symbol	12	Numeric left enclosure symbol
5	Monetary negative symbol	13	Numeric right enclosure symbol
6	Monetary left enclosure symbol	14	Monetary grouping
7	Monetary right enclosure symbol	15	Numeric grouping

MAX is not a parameter but a constant with the value of 18. This constant ensures that SUB, a subscript of the SYM-ELEM and SYMLEN-ELEM arrays, does not exceed 18.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV-SYMBOLS are as follows:

1	2002	3002
2	2004	3011
1001	3000	
1002	3001	

Sample output from Example 16–16 follows:

```
RESULT =            1

Field Meaning                      Symbols Length   Convention Symbols
-------------                      --------------   ------------------
International Currency Notation:                3   NKR
National Currency Notation:                     3   KR.
Monetary Thousands Separator:                   1
Monetary Decimal Symbol:                        1   ,
Monetary Positive Symbol:                       0
Monetary Negative Symbol:                       1   -
Monetary Left Enclosure Symbol:                 0
Monetary Right Enclosure Symbol:                0
Numeric Thousands Separator:                    1
Numeric Decimal Symbol:                         1   ,
Numeric Positive Symbol:                        0
Numeric Negative Symbol:                        1   -
Numeric Left Enclosure Symbol:                  0
Numeric Right Enclosure Symbol:                 0
Monetary Grouping Specification:                1   3
Numeric Grouping Specification:                 1   3
```

For more information on the error result values, see Table 16–1 in this section.

CNV_SYSTEMDATETIMETMP_COB

This procedure formats the system date, the system time, or both according to a template and language that you supply. The system obtains the date and time from a single function call to avoid the possibility of splitting the date and time across a day boundary. The template may be retrieved for any convention from the CNV_TEMPLATE_COB procedure or may be created by the user.

Example

Example 16–17 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_SYSTEMDATETIMETMP_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example formats system date and time according to a template provided by the calling program in TMP-ARY. The formatted date and time are translated to English and returned in SDT-ARY. DTEMP-LEN is set to the length of the date template in TMP-ARY.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/CNVSYSDATETIMETMP."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.
```

Example 16–17. Calling the CNV_SYSTEMDATETIMETMP_COB Procedure

```
 01  OUTPUT-RECORD        PIC X(80).
     WORKING-STORAGE SECTION.01  OF-1.
     05  FILLER           PIC X(09)   VALUE "RESULT = ".
     05  OF-RESULT        PIC ZZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.
 01  OF-2.
     05  FILLER           PIC X(10)   VALUE "SDT-ARY = ".
     05  OF-SDT-ARY       PIC X(40).
     05  FILLER           PIC X(30)   VALUE SPACE.


****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

 01  LANG-NAME            PIC X(17).
 01  SDT-ARY              PIC X(40).
 01  TMP-ARY              PIC X(48).

 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  DTEMP-LEN            PIC S9(11)  USAGE BINARY.
 77  RESULT               PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   DISPLAY "***  INTL_COBOL74: CNV_SYSTEMDATETIMETMP_COB".
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-SYSTEMDATETIMETMP-COB.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** CNV-SYSTEMDATETIMETMP-COB *********************************
 CNV-SYSTEMDATETIMETMP-COB.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "!W!, !N! !D!, !YYYY! !OT!:!OM!:!OS!" TO TMP-ARY.
   MOVE 21 TO DTEMP-LEN.
   MOVE "ENGLISH" TO LANG-NAME.
   CALL "CNV_SYSTEMDATETIMETMP_COB OF CENTRALSUPPORT"
     USING TMP-ARY,
           LANG-NAME,
           DTEMP-LEN,
           SDT-ARY
     GIVING RESULT.
```

Example 16-17. Calling the CNV_SYSTEMDATETIMETMP_COB Procedure

```
MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE SDT-ARY TO OF-SDT-ARY
          WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-17. Calling the CNV_SYSTEMDATETIMETMP_COB Procedure

Explanation

TMP-ARY is passed to the procedure. It contains the template you specify, left-justified in the field. The recommended length of a template is 48 characters. If both date and time templates are present, the date template must appear first. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for information about creating a template.

LANG-NAME is passed to the procedure. It contains the name of the language to be used in formatting the date, the time value or both. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for information about determining the valid language names on your system and the explanation of the hierarchy.

DTEMP-LEN is an integer passed by reference to the procedure. It specifies the length of the date template in TMP-ARY. If DTEMP-LEN is 0 (zero), it indicates there is no date template in TMP-ARY. If you specify both a date and time template, then the date template must appear first in TMP-ARY. The date and time are formatted if both date and time templates are present, the date is formatted if only date template is present, and the time is formatted if only time template is present.

SDT-ARY is returned by the procedure. It contains the formatted date, formatted time, or both. The recommended length of the formatted value is 45 characters.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by CNV_SYSTEMDATETIMETMP_COB are as follows:

1	2001	3011	3048	3054
2	3000	3045	3051	3055
1001	3001	3046	3052	3057
1002	3002	3047	3053	

Sample output from Example 16–17 follows:

```
RESULT =            1
SDT-ARY = Thursday, March 7, 1991 18:31:23
```

For more information on the error result values, see Table 16–1 in this section.

CNV_SYSTEMDATETIME_COB

This procedure formats the system date, the system time, or both according to the specified convention. It translates the date and time components to the natural language you specify. The system computes both the date and time from the result of a single function call. Thus, the possibility that the date and time are split across midnight does not exist.

You might use this procedure to output the system date and time in the Spain convention and the Spanish language, for example.

Example

Example 16–18 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_SYSTEMDATETIME_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example formats the system date and time according to formatting definitions in the ASeriesNative convention. The form of date and time is specified by CS-LDATENTIMEV (long date and numeric time). Formatted date and time are translated to English and returned in SDT-ARY.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/CNVSYSDATETIME."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.
```

Example 16–18. Calling the CNV_SYSTEMDATETIME_COB Procedure

```
01  OF-1.

05  FILLER           PIC X(09)   VALUE "RESULT = ".
     05  OF-RESULT        PIC ZZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.
 01  OF-2.
       05  FILLER           PIC X(10)   VALUE "SDT-ARY = ".
       05  OF-SDT-ARY       PIC X(40).
       05  FILLER           PIC X(30)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************
 01  CNV-NAME             PIC X(17).
 01  LANG-NAME            PIC X(17).
 01  SDT-ARY              PIC X(40).

 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  CS-LDATENTIMEV       PIC S9(11)  USAGE BINARY    VALUE 6.
 77  RESULT               PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-SYSTEMDATETIME-COB.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** CNV-SYSTEMDATETIME-COB ************************************
 CNV-SYSTEMDATETIME-COB.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "ASERIESNATIVE" TO CNV-NAME.
   MOVE "ENGLISH" TO LANG-NAME.
   CALL "CNV_SYSTEMDATETIME_COB OF CENTRALSUPPORT"
     USING CS-LDATENTIMEV,
           CNV-NAME,
           LANG-NAME,
           SDT-ARY
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE SDT-ARY TO OF-SDT-ARY
          WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-18. Calling the CNV_SYSTEMDATETIME_COB Procedure

Explanation

CS-LDATENTIMEV is passed to the procedure. It indicates one of the following formats is used when the date and time are returned:

Value	Sample Data Item	Meaning
0	CS-LDATEV	Long date format
1	CS-SDATEV	Short date format
2	CS-NDATEV	Numeric date format
3	CS-LTIMEV	Long time format
4	CS-NTIMEV	Numeric time format
5	CS-LDATELTIMEV	Long date and long time
6	CS-LDATENTIMEV	Long date and numeric time
7	CS-SDATELTIMEV	Short date and long time
8	CS-SDATENTIMEV	Short date and numeric time
9	CS-NDATELTIMEV	Numeric date and long time
10	CS-NDATENTIMEV	Numeric date and numeric time

CNV-NAME is passed to the procedure. It contains the name of the convention to be used to edit the date and time value. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for the list of convention names and the explanation of the hierarchy.

LANG-NAME is passed to the procedure. It contains the language to be used in formatting the date and time value. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the language to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for information about determining the valid language names on your system and the explanation of the hierarchy.

SDT-ARY is returned by the procedure. It contains the formatted date, formatted time, or both. The recommended length of the formatted value is 40 characters.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_SYSTEMDATETIME_COB are as follows:

1	2001	3001	3046	3052	3057
2	2002	3006	3047	3053	
1001	2004	3011	3048	3054	
1002	3000	3045	3051	3055	

Sample output from Example 16–18 follows:

```
RESULT =            1
SDT-ARY = Wednesday, November 7, 1990 17:14:58
```

For more information on the error result values, see Table 16–1 in this section.

CNV_TEMPLATE_COB

This procedure returns the requested format template for a designated convention.

You might want to use this procedure to improve the performance of your program. By retrieving and storing a template that you want to use in many places, you can improve the performance of your program by eliminating the calls to the CENTRALSUPPORT library.

Example

Example 16–19 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_TEMPLATE_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example retrieves a monetary editing template from the Turkey convention. The template is returned in TMP-ARY.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
```

Example 16–19. Calling the CNV_TEMPLATE_COB Procedure

```
SELECT OUTPUT-FILE ASSIGN TO DISK.
  DATA DIVISION.
 FILE SECTION.
 FD  OUTPUT-FILE
     LABEL RECORD IS STANDARD
     VALUE OF TITLE IS "OUT/COBOL74/CNVTEMPLATECOB."
     PROTECTION SAVE
     RECORD CONTAINS 80 CHARACTERS
     DATA RECORD IS OUTPUT-RECORD.
 01  OUTPUT-RECORD        PIC X(80).
 WORKING-STORAGE SECTION.
 01  OF-1.
     05  FILLER           PIC X(09)   VALUE "RESULT = ".
     05  OF-RESULT        PIC ZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.
 01  OF-2.
     05  FILLER           PIC X(10)   VALUE "TMP-ARY = ".
     05  OF-TMP-ARY       PIC X(48).
     05  FILLER           PIC X(22)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************
 01  CNV-NAME             PIC X(17).
 01  TMP-ARY              PIC X(48).
 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  CS-MONETARY-TEMPV    PIC S9(11)  USAGE BINARY    VALUE 5.
 77  RESULT               PIC S9(11)  USAGE BINARY.
 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-TEMPLATE-COB.
   CLOSE OUTPUT-FILE.
   STOP RUN.
***** CNV-TEMPLATE-COB ******************************************
 CNV-TEMPLATE-COB.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "Turkey" TO CNV-NAME.
   CALL "CNV_TEMPLATE_COB OF CENTRALSUPPORT"
     USING CS-MONETARY-TEMPV,
           CNV-NAME,
           TMP-ARY
     GIVING RESULT.
```

Example 16-19. Calling the CNV_TEMPLATE_COB Procedure

```
MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN STRING TMP-ARY DELIMITED BY @OO@,
                 @OO@ DELIMITED BY SIZE
                 INTO OF-TMP-ARY
          WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-19. Calling the CNV_TEMPLATE_COB Procedure

Explanation

CS-MONETARY-TEMPV is passed to the procedure. It specifies the type of template to be returned. This parameter can have the following values:

Value	Sample Data Item	Template to be Retrieved
0	CS-LONGDATE-TEMPV	Long date
1	CS-SHORTDATE-TEMPV	Short date
2	CS-NUMDATE-TEMPV	Numeric date
3	CS-LONGTIME-TEMPV	Long time
4	CS-NUMTIME-TEMPV	Numeric time
5	CS-MONETARY-TEMPV	Monetary template
6	CS-NUMERIC-TEMPV	Numeric template

CNV-NAME is passed to the procedure. It contains the name of the convention that you specify. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for the list of convention names and the explanation of the hierarchy.

TMP-ARY is returned by the procedure. It contains the requested template. The recommended length of a template is 48 characters.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_TEMPLATE_COB are as follows:

1	2002	3002
1001	3000	3006

1002 3001

Sample output from Example 16–19 follows:

```
RESULT =            1
TMP-ARY =!T[.:0,3]D[,]#N[-]C[T]!
```

For more information on the error result values, see Table 16–1 in this section.

CNV_VALIDATENAME

This procedure returns a value in the procedure result that indicates whether the convention name you specified is currently defined on the host system.

You might use this procedure to ensure that a convention used as an input parameter exists on the system on which your program is running.

Example

Example 16–20 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_VALIDATENAME library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example determines whether or not a convention named Sweden is currently available on the system.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/VALIDATENAME."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.
```

Example 16–20. Calling the CNV_VALIDATENAME Procedure

```
01  OF-1.
     05  FILLER           PIC X(09)   VALUE "RESULT = ".
     05  OF-RESULT        PIC ZZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

 01  CNV-NAME             PIC X(17).

 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  RESULT               PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-VALIDATENAME.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** CNV-VALIDATENAME ******************************************
 CNV-VALIDATENAME.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "Sweden" TO CNV-NAME.
   CALL "CNV_VALIDATENAME OF CENTRALSUPPORT"
     USING CNV-NAME
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
```

Example 16-20. Calling the CNV_VALIDATENAME Procedure

Explanation

CNV-NAME, is passed to the procedure. It contains the name of the convention that is to be checked. If this parameter contains all blanks or nulls, the RESULT parameter returns a value of 0 (zero) or FALSE. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for the list of convention names and the explanation of the hierarchy.

RESULT is an integer that is returned by the procedure. It contains the procedure result. The possible values for RESULT and their meanings are as follows:

Value	Condition-Name	Meaning
0	CS-FALSEV	The convention name is not valid.
1	CS-DATAOKV	The convention name is valid.

Sample output from Example 16–20 follows:

```
RESULT =            0
```

GET_CS_MSG

This procedure returns message text associated with the designated message number. The message number is obtained as the result value returned from a call to any of the CENTRALSUPPORT procedures.

When calling the GET_CS_MSG procedure, you can designate the language to which the message is to be translated and the desired length of the returned message. If the returned text is shorter than the length specified, the procedure pads the remaining portion of the record with blanks.

An entire message consists of the following three parts:

- The header, which comprises the first 80 characters of the message text returned by the MSG parameter. The text in the header provides the message number and a concise text description.
- The short description, which comprises the second 80 character of the message text returned by the MSG parameter. If space is a consideration, you might want to limit the description of the message to the header and short description.
- The long description, which comprises the remaining characters of the message text returned by the MSG parameter. The long description provides a complete explanation of the message that was returned.

Part or all of the message text can be returned. Note that the header part starts at offset 0 (zero), the short description at offset 80, and the long description at offset 160. For example, if you specify the MSG-LEN parameter to be equal to 200 characters, then the MSG parameter would contain the header message padded with blanks to offset 80, if necessary, followed by the short description padded with blanks to offset 160, if necessary, followed by the first 40 characters of the long description.

The message length should be at least 80 characters, equal to one line of text. Anything less results in an incomplete message. Using a value of either 80, 160, or 999 is recommended. The value of 999 causes the entire message to be returned.

You might want to use this procedure to retrieve the text of an error message so that it can be displayed by your program.

Example

Example 16–21 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the GET_CS_MSG library procedure. The declarations identify

the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example illustrates how to get the message text associated with a CENTRALSUPPORT error message. Assume that the sample call to VALIDATE_NAME_RETURN_NUM returns the error 3004. (The requested name was not found). When the error is returned, this example gets the first 160 characters (2 lines) of the message text for the error.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/GETCSMSG."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.

01  OF-1.
    05  FILLER           PIC X(39)
        VALUE "RESULT FROM VALIDATE_NAME_RETURN_NUM = ".
    05  OF-RESULT1       PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(23)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(39)
        VALUE "RESULT FROM GET_CS_MSG               = ".
    05  OF-RESULT2       PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(23)   VALUE SPACE.
01  OF-3.
    05  FILLER           PIC X(06)   VALUE "MSG = ".
    05  FILLER           PIC X(74)   VALUE SPACE.
01  OF-4.
    05  OF-MSG           PIC X(80).

****************************************************************
***  The following global declarations are used as parameters  ***
```

```
***  to the CENTRALSUPPORT procedures.                          ***
*****************************************************************
```

Example 16–21. Calling the GET_CS_MSG Procedure

```
 01  LANG-NAME            PIC X(17).
 01  MSG.
     05  MSG-ELEM         PIC X(80)   OCCURS 2 TIMES.
 01  NAME-ARY             PIC X(17).
 77  CS-CHARACTER-SETV    PIC S9(11)  USAGE BINARY    VALUE 0.

 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  MSG-LEN              PIC S9(11)  USAGE BINARY.
 77  NUM                  PIC S9(11)  USAGE BINARY.
 77  RESULT1              PIC S9(11)  USAGE BINARY.
 77  RESULT2              PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   DISPLAY "***  INTL_COBOL74: GET_CS_MSG".
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM GET-CS-MSG.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** GET-CS-MSG ************************************************
 GET-CS-MSG.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "BADNAME" TO NAME-ARY.
   CALL "VALIDATE_NAME_RETURN_NUM OF CENTRALSUPPORT"
     USING CS-CHARACTER-SETV,
           NAME-ARY,
           NUM
     GIVING RESULT1.
   MOVE RESULT1 TO OF-RESULT1.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT1 IS NOT EQUAL TO CS-DATAOKV
     THEN MOVE 160 TO MSG-LEN
          CALL "GET_CS_MSG OF CENTRALSUPPORT"
            USING RESULT1,
                  LANG-NAME,
                  MSG,
                  MSG-LEN
            GIVING RESULT2
```

```
MOVE RESULT2 TO OF-RESULT2
WRITE OUTPUT-RECORD FROM OF-2
WRITE OUTPUT-RECORD FROM OF-3
MOVE MSG-ELEM(1) TO OF-MSG
WRITE OUTPUT-RECORD FROM OF-4
MOVE MSG-ELEM(2) TO OF-MSG
WRITE OUTPUT-RECORD FROM OF-4.
```

Example 16–21. Calling the GET_CS_MSG Procedure

Explanation

NUM is passed to the procedure. It contains the number of the message for which you want the text. These values are returned by calls on other CENTRALSUPPORT procedures. The message numbers and their meanings are listed at the end of this section.

LANG-NAME is passed to the procedure. It specifies the language in which the message is to be displayed. The maximum length of a language name is 17 characters. If this parameter contains all blanks or zeros, the procedure uses the default language hierarchy to determine the language to be used. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for details about determining the valid language names on the system and for the explanation of the default language hierarchy.

MSG is returned by the procedure. It contains the message text associated with the specified message number. It is recommended that the size of this record be at least 80 characters.

MSG_LEN in passed to the procedure. For an output parameter, MSG_LEN contains an update value. For input, it specifies the maximum length of the message to be returned. If MSG_LEN is equal to 0 (zero), one line of text (80 characters) is returned. If MSG_LEN is between 1 and 79, then only a partial message is returned. MSG_LEN should not be greater than the size of the MSG record. Recommended values for MSG_LEN are 80, 160, or a large number that returns all of the message. For output, MSG_LEN specifies the actual length of the message returned by the procedure.

NAME-ARY is passed to the procedure. It contains the coded character set or ccsversion name for which a message number is being requested. The name can be up to 17 characters long. If this parameter contains zeros or blanks, the procedure uses the hierarchy to determine the ccsversion or character set to be used. If there is no system default, the procedure returns an error in RESULT.

CS-CHARACTER-SETV is passed to the procedure. If this flag represents 0 (zero), the coded character set is being checked. If it represents 1 (one), the ccsversion is being checked.

RESULT1 is passed to the procedure. It contains the number of the message for which you want the text. These values are returned by calls on other CENTRALSUPPORT procedures. The message numbers and their meanings are listed at the end of this section. In Example 16–23, the RESULT1 field is from an executed VALIDATE_NAME_RETURN_NUM procedure that requested a ccsversion number for the name BADNAME.

RESULT2 is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by GET_CS_MSG are as follows:

1	2004	3002
1001	3000	3003
1002	3001	

Sample output from Example 16–21 follows:

```
RESULT FROM VALIDATE_NAME_RETURN_NUM =         3004
RESULT FROM GET_CS_MSG               =            1
MSG =
>>> CENTRALSUPPORT INTERFACE ERROR (#3004) <<<
INVALID CHARACTER SET OR CCSVERSION NAME
```

For more information on the error result values, see Table 16–1 in this section.

MCP_BOUND_LANGUAGES

This procedure returns the names of languages that are currently bound to the operating system. For information about binding a language to the operating system, refer to the *MultiLingual System Administration, Operations, and Programming Guide*.

You might use this procedure to determine which languages are available on the system.

Example

Example 16–22 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the MCP_BOUND_LANGUAGES library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example returns the languages bound by the operating system. Assume for this example that the bound language is English.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/MCPBOUNDLANGUAGES."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.
01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.

01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(09)   VALUE "Languages".
    05  FILLER           PIC X(71)   VALUE SPACE.
01  OF-3.
```

Example 16–22. Calling the MCP_BOUND_LANGUAGES Procedure

```
 05  FILLER           PIC X(09)   VALUE ALL "-".
     05  FILLER           PIC X(71)   VALUE SPACE.
 01  OF-4.
     05  OF-LANG-ELEM     PIC X(17).
     05  FILLER           PIC X(63)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************
 01  LANGUAGES-ARY.
     05  LANGUAGES-ELEM   PIC X(17)   OCCURS 20 TIMES.
 01  SUB                  PIC 9(02).

 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  RESULT               PIC S9(11)  USAGE BINARY.
 77  TOTAL                PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM MCP-BOUND-LANGUAGES.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** MCP-BOUND-LANGUAGES ****************************************
 MCP-BOUND-LANGUAGES.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   CALL "MCP_BOUND_LANGUAGES OF CENTRALSUPPORT"
     USING TOTAL,
           LANGUAGES-ARY
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE SPACE TO OUTPUT-RECORD
          WRITE OUTPUT-RECORD
          WRITE OUTPUT-RECORD FROM OF-2
          WRITE OUTPUT-RECORD FROM OF-3
          MOVE 1 TO SUB
          PERFORM DISPLAYLANGUAGESARY
            UNTIL SUB IS GREATER THAN TOTAL.
```

Example 16-22. Calling the MCP_BOUND_LANGUAGES Procedure

```
***** DISPLAYLANGUAGESARY ***************************************
DISPLAYLANGUAGESARY.

 MOVE LANGUAGES-ELEM(SUB) TO OF-LANG-ELEM.
   WRITE OUTPUT-RECORD FROM OF-4.
   ADD 1 TO SUB.
```

Example 16-22. Calling the MCP_BOUND_LANGUAGES Procedure

Explanation

TOTAL is an integer returned by the procedure. It contains the total number of languages that are bound to the operating system.

LANGUAGES-ARY is returned by the procedure. It contains the names of the languages bound to the operating system. The maximum length of each name is 17 characters, and the names are left justified. For any name that is less than 17 characters, the field is filled on the right with blanks. In the example, the size of the record is 84 characters; a record of that size holds 5 names.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by this procedure are as follows:

1	1002	3001
1001	3000	

Sample output from Example 16–22 follows:

```
RESULT =            1

Languages
---------
ENGLISH
```

For more information on the error result values, see Table 16–1 in this section.

VALIDATE_NAME_RETURN_NUM

This procedure examines a coded character set or ccsversion name to determine if it resides in the file SYSTEM/CCSFILE. The first parameter specifies whether you want to examine a coded character set or ccsversion. The next parameter specifies the name to be validated. The procedure returns the number of the coded character set or ccsversion in the last parameter.

You might use this procedure to obtain the ccsversion number needed as a parameter to other CENTRALSUPPORT library procedures.

Example

Example 16–23 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VALIDATE_NAME_RETURN_NUM library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example checks to see if a ccsversion named CanadaGP is valid. Assume for this example that CanadaGP is valid and its associated number is 75.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/VALIDATENAMERTRN."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.

01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
```

Example 16–23. Calling the VALIDATE_NAME_RETURN_NUM Procedure

```
01  OF-2.
     05  FILLER           PIC X(09)   VALUE "NUM    = ".
     05  OF-NUM           PIC ZZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.

*****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
*****************************************************************

 01  NAME-ARY             PIC X(17).

 77  CS-CCSVERSIONV       PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  NUM                  PIC S9(11)  USAGE BINARY.
 77  RESULT               PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM VALIDATE-NAME-RETURN-NUM.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** VALIDATE-NAME-RETURN-NUM ***********************************
 VALIDATE-NAME-RETURN-NUM.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE "CanadaGP" TO NAME-ARY.
   CALL "VALIDATE_NAME_RETURN_NUM OF CENTRALSUPPORT"
     USING CS-CCSVERSIONV,
           NAME-ARY,
           NUM
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE NUM TO OF-NUM
          WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-23. Calling the VALIDATE_NAME_RETURN_NUM Procedure

Explanation

CS-CCSVERSIONV is passed to the procedure. It indicates whether the entry specified in NAME is a coded character set or ccsversion name. The allowable values are as follows:

Value	Sample Data Item	Meaning
0	CS-CHARACTER-SET-V	Coded character set name
1	CS-CCSVERSION-V	Ccsversion name

NAME-ARY is passed to the procedure. It contains the coded character set or ccsversion name for which a number is being requested. The name can be up to 17 characters long. If this parameter contains zeros or blanks and type is equal to 1, the procedure validates the system default ccsversion.

NUM is returned by the procedure. It contains the coded character set or ccsversion number requested.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by VALIDATE_NAME_RETURN_NUM are as follows:

1	3000	3006
1001	3002	
1002	3004	

Sample output from Example 16–23 follows:

```
RESULT =             1
NUM    =            75
```

For more information on the error result values, see Table 16–1 in this section.

VALIDATE_NUM_RETURN_NAME

This procedure examines the number of a coded character set or ccsversion to determine if it resides in the SYSTEM/CCSFILE. The first parameter designates whether a coded character set or ccsversion is to be examined. The second parameter specifies the number to be validated. The procedure then returns the name of the given character set or ccsversion number. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for the list of numbers for coded character sets and ccsversions.

You might use this procedure to display the name of the coded character set or the ccsversion being used.

Example

Example 16–24 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VALIDATE_NUM_RETURN_NAME library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example checks to see if the ccsversion number 75 is valid. Assume for this example that 75 is valid and its associated name is CanadaGP.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/VALIDATENUMRTRN."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.
```

Example 16–24. Calling the VALIDATE_NUM_RETURN_NAME Procedure

```
01  OF-1.
     05  FILLER           PIC X(09)   VALUE "RESULT = ".
     05  OF-RESULT        PIC ZZZZZZZZZZZ9.


     05  FILLER           PIC X(59)   VALUE SPACE.
 01  OF-2.
     05  FILLER           PIC X(11)   VALUE "NAME-ARY = ".
     05  OF-NAME-ARY      PIC X(17).
     05  FILLER           PIC X(52)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

 01  NAME-ARY             PIC X(17).
 77  CS-CCSVERSIONV       PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  NUM                  PIC S9(11)  USAGE BINARY.
 77  RESULT               PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM VALIDATE-NUM-RETURN-NAME.
   CLOSE OUTPUT-FILE.
   STOP RUN.

***** VALIDATE-NUM-RETURN-NAME **********************************
 VALIDATE-NUM-RETURN-NAME.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE 75 TO NUM.
   CALL "VALIDATE_NUM_RETURN_NAME OF CENTRALSUPPORT"
     USING CS-CCSVERSIONV,
           NUM,
           NAME-ARY
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE NAME-ARY TO OF-NAME-ARY
          WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-24. Calling the VALIDATE_NUM_RETURN_NAME Procedure

Explanation

CS-CCSVERSIONV is passed to the procedure. It indicates whether the value specified in NUM is a coded character set number or a ccsversion number. The following values are allowed:

Value	Sample Data Item	Meaning
0	CS-CHARACTER-SET-V	Coded character set number
1	CS-CCSVERSION-V	Ccsversion number

NUM is passed by reference to the procedure. It contains the number of the coded character set or ccsversion for which the name is being requested. If you supply the value –2 in the NUM parameter when you are checking a ccsversion, the procedure returns the name of the system default ccsversion. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for more information about the hierarchy.

NAME-ARY is returned by the procedure. It contains the coded character set or ccsversion name. The recommended length of the name is 17 characters.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by VALIDATE_NUM_RETURN_NAME are as follows:

1	3000	3006
1001	3001	
1002	3003	

Sample output from Example 16–24 follows:

```
RESULT =            1
NAME-ARY = CANADAGP
```

For more information on the error result values, see Table 16–1 in this section.

VSNCOMPARE_TEXT

This procedure compares two records, using one of three comparison methods. The comparison is specified as one of the following types:

- A binary comparison, which is based on the hexadecimal code values of the characters
- An equivalent comparison, which is based on the ordering sequence values (OSVs) of the characters
- A logical comparison, which is based on the ordering sequence values (OSVs) plus the priority sequence values (PSVs) of the characters

The procedure retrieves the OSVs and PSVs from the file SYSTEM/CCSFILE based on the specified ccsversion.

Example

Example 16–25 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VSNCOMPARE_TEXT library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example compares two strings using the CanadaEBCDIC ccsversion. The first string is “hotel” and the second string is “hôtel.” Assume the following ordering values for the characters:

Character	Ordering Sequence Value (OSV)	Priority Sequence Value (PSV)
e	69	2
h	72	2
l	76	2
o	79	2
t	84	2
ô	79	8

The compare relation is CsCmpEql (=) to determine if “hotel” is equal to “hôtel” using a logical comparison. You can use the *MultiLingual System Administration, Operations, and Programming Guide* to determine that the ccsversion number for CanadaEBCDIC is 74. You can also retrieve this number by calling the procedure VALIDATE_NAME_RETURN_NUM with the name CanadaEBCDIC.

```
      IDENTIFICATION DIVISION.
      ENVIRONMENT DIVISION.
      INPUT-OUTPUT SECTION.
      FILE-CONTROL.
        SELECT OUTPUT-FILE ASSIGN TO DISK.

      DATA DIVISION.
      FILE SECTION.
      FD  OUTPUT-FILE
          LABEL RECORD IS STANDARD
          VALUE OF TITLE IS "OUT/COBOL74/VSNCOMPARETEXT."
          PROTECTION SAVE
          RECORD CONTAINS 80 CHARACTERS
          DATA RECORD IS OUTPUT-RECORD.

      01  OUTPUT-RECORD        PIC X(80).

      WORKING-STORAGE SECTION.

      01  OF-1.
          05  FILLER           PIC X(09)   VALUE "RESULT = ".
          05  OF-RESULT        PIC ZZZZZZZZZZ9.
          05  FILLER           PIC X(59)   VALUE SPACE.
     ****************************************************************
     ***  The following global declarations are used as parameters  ***
     ***  to the CENTRALSUPPORT procedures.                         ***
     ****************************************************************     01
    TEXT1-TEXT           PIC X(05).
      01  TEXT2-TEXT           PIC X(05).

      77  CS-CMPEQLV           PIC S9(11)  USAGE BINARY    VALUE 2.
      77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
      77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
      77  CS-LOGICALV          PIC S9(11)  USAGE BINARY    VALUE 2.
      77  TEXT1-START          PIC S9(11)  USAGE BINARY.
      77  TEXT2-START          PIC S9(11)  USAGE BINARY.
      77  COMPARE-LEN          PIC S9(11)  USAGE BINARY.
      77  RESULT               PIC S9(11)  USAGE BINARY.
      77  VSN-NUM              PIC S9(11)  USAGE BINARY.

      PROCEDURE DIVISION.
      INTLCOBOL74.
```

Example 16–25. Calling the VSNCOMPARE_TEXT Procedure

```
    OPEN OUTPUT OUTPUT-FILE.
    PERFORM VSNCOMPARE-TEXT.
    CLOSE OUTPUT-FILE.
    STOP RUN.

***** VSNCOMPARE-TEXT ******************************************
 VSNCOMPARE-TEXT.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE 74 TO VSN-NUM.
   MOVE 5 TO COMPARE-LEN.
   MOVE "hotel" TO TEXT1-TEXT.
   MOVE "hôtel" TO TEXT2-TEXT.
   CALL "VSNCOMPARE_TEXT OF CENTRALSUPPORT"
     USING VSN-NUM,
           TEXT1-TEXT,
           TEXT1-START,
           TEXT2-TEXT,
           TEXT2-START,
           COMPARE-LEN,
           CS-CMPEQLV,
           CS-LOGICALV
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
```

Example 16-25. Calling the VSNCOMPARE_TEXT Procedure

Explanation

VSN-NUM is passed to the procedure. It contains the number of the ccsversion that is used to compare the text records. The number can be obtained by referring to the *MultiLingual System Administration, Operations, and Programming Guide.*

The following values are allowed:

Value	Meaning
0 (zero) or greater	Designate a ccsversion.
–2	Use the system default ccsversion. If that ccsversion is not available, the procedure returns an error in RESULT.

TEXT1-TEXT is passed to the procedure. It contains the first record of text to be compared. You determine the size of the record.

TEXT1-START is passed by reference to the procedure. It contains the byte offset in TEXT1-TEXT, relative to 0 (zero), at which the comparison begins.

TEXT2-TEXT is passed to the procedure. It contains the second record of text to be compared. You determine the size of the record.

TEXT2-START is passed to the procedure. It contains the byte offset in TEXT2-TEXT, relative to 0 (zero), at which the comparison begins.

COMPARE-LEN is passed by reference to the procedure. It contains the number of characters to compare. If COMPARE-LEN is larger than the number of characters in the strings, then the procedure might be comparing invalid data. The value of COMPARE-LEN should not exceed the bounds of either TEXT1-TEXT or TEXT2-TEXT.

The strings should be of equal size or padded with blanks up to the value of COMPARE-LEN. If all pairs of characters compare equally, the strings are considered equal. Otherwise, the first pair of unequal characters encountered is compared to determine their relative ordering. The string that contains the character with the higher ordering (higher PSV and higher OSV) is considered to be the string with the greater value. If substitution forms strings of unequal length, the comparison proceeds as if the shorter string were padded with blanks on the right. This padding ensures that the strings are of equal length.

CS-CMPEQLV is passed by reference to the procedure. It indicates the relational operator of the comparison. The valid values are

Value	Sample Value Name	Meaning
0	CS-CMPLSSV	TEXT1-TEXT is less than TEXT2-TEXT.
1	CS-CMPLEQV	TEXT1-TEXT is less than or equal to TEXT2-TEXT.
2	CS-CMPEQLV	TEXT1-TEXT is equal to TEXT2-TEXT.
3	CS-CMPGTRV	TEXT1-TEXT is greater than TEXT2-TEXT.
4	CS-CMPGEQV	TEXT1-TEXT is greater than or equal to TEXT2-TEXT.
5	CS-CMPNEQV	TEXT1-TEXT is not equal to TEXT2-TEXT.

CS-LOGICALV is passed by reference to the procedure. It indicates the type of comparison to be performed by the procedure. The following are the valid values:

Value	Sample Value Name	Meaning
0	BINARY-V	Perform a binary comparison
1	EQUIVALENT-V	Perform an equivalent comparison
2	LOGICAL-V	Perform a logical comparison

RESULT is returned as the value of the procedure. It contains the procedure result or indicates that an error occurred during the execution of the procedure. The possible values for RESULT and their meanings are as follows:

Value	Condition-Name	Meaning
0	CS-FALSEV	No error and the condition is FALSE
1	CS-DATAOKV	No error and the condition is TRUE

Other possible values returned by the procedure are as follows:

1000	1002	3006
1001	3003	

The meanings of the error result values are explained at the end of this section.

Sample output from Example 16–25 follows:

```
RESULT =            0
```

For more information on the error result values, see Table 16–1 in this section.

VSNESCAPEMENT

This procedure takes the input text and rearranges it according to the escapement rules of the ccsversion. Both the character advance direction and the character escapement direction are used. If the character advance direction is positive, then the starting position for the escapement process is the leftmost position of the text in the DEST-TEXT parameter. If the character advance direction is negative, then the starting position for the escapement process is the rightmost position of the text in the DEST-TEXT parameter. From that point on, the character advance direction value and the character escapement direction values, in combination, control where each character should be placed in relation to the previous character.

Example

Example 16–26 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VSNESCAPEMENT library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example takes the string *ABCDEFG* and rearranges it according to the escapement rules of the ccsversion. Assume for this example a ccsversion number of 999 with a character advance direction of plus (+, left to right) and with the following character escapements:

Character	Escapement	Meaning
A	+	Left to right.
B	–	Right to left.
C	–	Right to left.
D	–	Right to left.
E	+	Left to right.
F	+	Left to right.
G	blank	Use character advance direction value.

```
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT OUTPUT-FILE ASSIGN TO DISK.
 DATA DIVISION.
 FILE SECTION.
 FD  OUTPUT-FILE
     LABEL RECORD IS STANDARD
     VALUE OF TITLE IS "OUT/COBOL74/VSNESCAPEMENT."
     PROTECTION SAVE
     RECORD CONTAINS 80 CHARACTERS
     DATA RECORD IS OUTPUT-RECORD.
 01  OUTPUT-RECORD        PIC X(80).
 WORKING-STORAGE SECTION.
 01 OF-1.
     05  FILLER           PIC X(09)   VALUE "RESULT = ".
     05  OF-RESULT        PIC ZZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.
 01  OF-2.
     05  FILLER           PIC X(12)   VALUE "DEST-TEXT = ".
     05  OF-DEST-TEXT     PIC X(07).
     05  FILLER           PIC X(61)   VALUE SPACE.
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
 01  DEST-TEXT            PIC X(07).
 01  SOURCE-TEXT          PIC X(07).
 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  SOURCE-START         PIC S9(11)  USAGE BINARY.
 77  TRANS-LEN            PIC S9(11)  USAGE BINARY.
 77  RESULT               PIC S9(11)  USAGE BINARY.
 77  VSN-NUM              PIC S9(11)  USAGE BINARY.
 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM VSNESCAPEMENT.
   CLOSE OUTPUT-FILE.
   STOP RUN.
***** VSNESCAPEMENT ********************************************
 VSNESCAPEMENT.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE 999 TO VSN-NUM.
   MOVE "ABCDEFG" TO SOURCE-TEXT.
   MOVE 7 TO TRANS-LEN.
```

Example 16–26. Calling the VSNESCAPEMENT Procedure

```
CALL "VSNESCAPEMENT OF CENTRALSUPPORT"
  USING VSN-NUM,
        SOURCE-TEXT,
        SOURCE-START,
        DEST-TEXT,
        TRANS-LEN
  GIVING RESULT.
MOVE RESULT TO OF-RESULT.
WRITE OUTPUT-RECORD FROM OF-1.
IF RESULT IS EQUAL TO CS-DATAOKV
  THEN MOVE DEST-TEXT TO OF-DEST-TEXT
       WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-26. Calling the VSNESCAPEMENT Procedure

Explanation

VSN-NUM is passed by reference to the procedure. It specifies the ccsversion to be used. The ccsversion contains the escapement rules. The following are the values allowed for VSN-NUM:

Value	Meaning
Greater than or equal to 0	Specifies a ccsversion. The numbers of the ccsversions are listed in the *MultiLingual System Administration, Operations, and Programming Guide*.
–2	Specifies the system default ccsversion. If the system default ccsversion is not available, an error is returned.

SOURCE-TEXT is passed to the procedure. It contains the text to be arranged according to the escapement rules. You must determine the size of the record.

SOURCE-START is passed by reference to the procedure. It specifies where in SOURCE-TEXT the procedure is to begin rearranging the text.

DEST-TEXT is returned by the procedure. It contains the rearranged text. The length of the SOURCE-TEXT parameter and the DEST-TEXT parameter should be the same.

TRANS-LEN is passed by reference to the procedure. It specifies the number of characters to rearrange, beginning at SOURCE-START.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by the procedure are as follows:

1	1002	3003
1000	3000	
1001	3002	

Sample output from Example 16–26 follows:

```
RESULT =            1
DEST-TEXT = ADCBEFG
```

For more information on the error result values, see Table 16–1 in this section.

VSNGETORDERINGFOR_ONE_TEXT

This procedure returns the ordering information for a specified input text. The ordering information determines how the input text is collated. It includes the ordering sequence values (OSVs) and optionally the priority sequence values (PSVs) of the characters. It always includes any substitution of characters to be made when the input text is sorted.

Note: *There is a source size restriction of 9,000 bytes.*

You can choose one of the following types of ordering information:

Type of Ordering	Explanation
Equivalent	The DEST parameter consists of a sequence of OSVs.
Logical	The DEST parameters consists of a sequences of OSVs followed by PSVs.

Example

Example 16–27 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VSNGETORDERINGFOR_ONE_TEXT library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example obtains the OSVs and PSVs for the input text string "ABC¯®." The ccsversion is CanadaEBCDIC. You can use the *MultiLingual System Administration, Operations, and Programming Guide* to determine that the ccsversion for CanadaEBCDIC is 74. You can also retrieve this number by calling the procedure VALIDATE_NAME_RETURN_NUM with the name CanadaEBCDIC. This example requests logical ordering information, so both the OSVs and PSVs are returned. This example also allows for maximum substitution, so the parameter max_osvs is equal to *itext_len * 3* and the parameter total_storage is equal to *max_osvs + round(max_osvs/2.0).*

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OUTPUT-FILE ASSIGN TO DISK.
```

Example 16–27. Calling the VSNGETORDERINGFOR_ONE_TEXT Procedure

```
DATA DIVISION.
FILE SECTION.
FD OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/VSNGETORDONETEXT."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS

    DATA RECORD IS OUTPUT-RECORD.

01 OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.

01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(12)   VALUE "DEST-TEXT = ".
    05  OF-DEST-TEXT     PIC X(51).
    05  FILLER           PIC X(17)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

01  DEST-TEXT            PIC X(51).
01  SOURCE-TEXT          PIC X(51).

77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
77  CS-LOGICALV          PIC S9(11)  USAGE BINARY    VALUE 2.
77  DEST-START           PIC S9(11)  USAGE BINARY.
77  ITEXT-LEN            PIC S9(11)  USAGE BINARY.
77  MAX-OSVS             PIC S9(11)  USAGE BINARY.
77  RESULT               PIC S9(11)  USAGE BINARY.
77  SOURCE-START         PIC S9(11)  USAGE BINARY.
77  TOTAL-STORAGE        PIC S9(11)  USAGE BINARY.
77  VSN-NUM              PIC S9(11)  USAGE BINARY.

PROCEDURE DIVISION.
INTLCOBOL74.
  OPEN OUTPUT OUTPUT-FILE.
  PERFORM VSNGETORDERINGFOR-ONE-TEXT.
  CLOSE OUTPUT-FILE.
  STOP RUN.
```

Example 16-27. Calling the VSNGETORDERINGFOR_ONE_TEXT Procedure

```
***** VSNGETORDERINGFOR-ONE-TEXT *********************************
 VSNGETORDERINGFOR-ONE-TEXT.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE 74 TO VSN-NUM.
   MOVE 5 TO ITEXT-LEN.
   COMPUTE MAX-OSVS = ITEXT-LEN * 3.
   COMPUTE TOTAL-STORAGE = MAX-OSVS + MAX-OSVS / 2.

   MOVE "ABCæÆ" TO SOURCE-TEXT.
   CALL "VSNGETORDERINGFOR_ONE_TEXT OF CENTRALSUPPORT"
     USING VSN-NUM,
           SOURCE-TEXT,
           SOURCE-START,
           ITEXT-LEN,
           DEST-TEXT,
           DEST-START,
           MAX-OSVS,
           TOTAL-STORAGE,
           CS-LOGICALV
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE DEST-TEXT TO OF-DEST-TEXT
          WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-27. Calling the VSNGETORDERINGFOR_ONE_TEXT Procedure

Explanation

VSN-NUM is passed to the procedure. It contains the number of the ccsversion that is used. The number can be obtained by calling the CENTRALSTATUS procedure or by referring to the *MultiLingual System Administration, Operations, and Programming Guide*. The following shows the allowed values:

Value	Meaning
Greater than or equal to 0	Specifies a ccsversion. The numbers of the ccsversions are listed in the *MultiLingual System Administration, Operations, and Programming Guide*.
–2	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

SOURCE-TEXT is a record passed to the procedure. It contains the text for which the ordering information is requested.

SOURCE-START is passed by reference to the procedure. It contains the offset of the location where the translation is to begin.

ITEXT-LEN is passed by reference to the procedure. It contains the length of the text that is to be translated.

DEST-TEXT is a record returned by the procedure. It contains the ordering information for the input text.

DEST-START is returned by the procedure. It designates the starting offset at which the result values are placed.

MAX-OSVS is an integer passed by reference to the procedure. It designates the maximum number of storage bytes to be used to store the ordering sequence values.

The value of MAX-OSVS should be the length of the input text. In the case when substitution is required, the MAX-OSVS value might need to be more than the length of the input text.

The maximum substitution length defined for any ccsversion is 3; therefore, to allow for substitution for every character, the value of MAX-OSVS is as follows:

```
(length of source text in bytes) * 3
```

If the number of OSVs returned is less than MAX-OSVS, then the alphanumeric record is packed with the ordering sequence value for blank.

TOTAL-STORAGE is passed by reference to the procedure. It defines the maximum number of bytes needed to store the complete ordering information for the text. If you request equivalent ordering information, TOTAL-STORAGE and MAX-OSVS should be set the same. If you request logical ordering information, you must provide space for the four-bit priority values in addition to the space allowed for the OSVs. Each OSV has one PSV, and one byte can hold two PSVs. Therefore, the space allowed for PSVs *MAX-OSVS/2*, and the value of TOTAL-STORAGE should be set as follows:

```
MAX-OSVS + (MAX-OSVS)/2
```

When the ordering information is returned by the procedure, all the OSVs are listed first, followed by all the PSVs.

CS-LOGICALV is an integer passed by reference to the procedure. It indicates the type of ordering information you want, as follows:

Value	Sample Value Name	Meaning
1	CS-EQUIVALENTV	OSVs only
2	CS-LOGICALV	PSVs only

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by this procedure are as follows:

0	1002	3003
1	3000	3006
1000	3001	3008
1001	3002	

Sample output from Example 16–27 follows:

```
RESULT = CS_DATAOKV
DEST-TEXT = 414243414541  454040404040  404040111221  111111111111
```

Based on the values of DEST-TEXT, the OSVs are 65, 66, 67, 65, 69, 65, and 69. The PSVs are 1, 1, 1, 2,2, 1, and 1.

For more information on the error result values, see Table 16–1 in this section.

VSNINSPECT_TEXT

This procedure searches a specified text for characters that are present or not present in a requested data class. The SCANNED-CHARS parameter is an integer that represents the number of characters that were searched when the criteria specified in the CS_NOT_INTSETV parameter were met. If SCANNED-CHARS is equal to INSPECT-LEN, then all the characters were searched but none met the criteria. Otherwise, adding the TEXT-START value to the RESULT value gives the location of the character, from the start of the array that met the search criteria.

Example

Example 16–28 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VSNINSPECT_TEXT library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example examines a record that contains two fields, a name and a phone number. The name is verified to contain only alphabetic characters as defined by the France ccsversion. You can use the *MultiLingual System Administration, Operations, and Programming Guide* to determine that the ccsversion number for France is 35. You can also retrieve this number by calling the procedure CCSVSNNUM with the name France.

```
  IDENTIFICATION DIVISION.
     ENVIRONMENT DIVISION.
     INPUT-OUTPUT SECTION.
     FILE-CONTROL.
       SELECT OUTPUT-FILE ASSIGN TO DISK.

     DATA DIVISION.
     FILE SECTION.
     FD  OUTPUT-FILE
         LABEL RECORD IS STANDARD
         VALUE OF TITLE IS "OUT/COBOL74/VSNINSPECTTEXT."
         PROTECTION SAVE
         RECORD CONTAINS 80 CHARACTERS
         DATA RECORD IS OUTPUT-RECORD.

     01  OUTPUT-RECORD        PIC X(80).

    WORKING-STORAGE SECTION.
```

Example 16–28. Calling the VSNINSPECT_TEXT Procedure

```
 01  OF-1.
     05  FILLER           PIC X(09)   VALUE "RESULT = ".

     05  OF-RESULT        PIC ZZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.
 01  OF-2.
     05  FILLER           PIC X(16)   VALUE "SCANNED-CHARS = ".
     05  OF-SCANNED-CHARS PIC ZZZZZZZZZZZ9.
     05  FILLER           PIC X(52)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************
 01  SOURCE-TEXT          PIC X(41).

 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  CS-NOT-INTSETV       PIC S9(11)  USAGE BINARY    VALUE 0.
 77  CS-NUMERICSV         PIC S9(11)  USAGE BINARY    VALUE 13.
 77  ID-LEN               PIC S9(11)  USAGE BINARY    VALUE 10.
 77  INSPECT-LEN          PIC S9(11)  USAGE BINARY.
 77  NAME-LEN             PIC S9(11)  USAGE BINARY    VALUE 30.
 77  SCANNED-CHARS        PIC S9(11)  USAGE BINARY.
 77  SOURCE-START         PIC S9(11)  USAGE BINARY.
 77  RESULT               PIC S9(11)  USAGE BINARY.
 77  VSN-NUM              PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM VSNINSPECT-TEXT.
   CLOSE OUTPUT-FILE.
   STOP RUN.
 ***** VSNINSPECT-TEXT ******************************************
 VSNINSPECT-TEXT.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE 35 TO VSN-NUM.
   MOVE NAME-LEN TO INSPECT-LEN.
   MOVE "7775961089John Alan Smith               " TO SOURCE-TEXT.
   CALL "VSNINSPECT_TEXT OF CENTRALSUPPORT"
     USING VSN-NUM,
           SOURCE-TEXT,
           SOURCE-START,
           INSPECT-LEN,
           CS-NUMERICSV,
```

Example 16-28. Calling the VSNINSPECT_TEXT Procedure

```
            CS-NOT-INTSETV,
            SCANNED-CHARS
      GIVING RESULT.

    MOVE RESULT TO OF-RESULT.
    WRITE OUTPUT-RECORD FROM OF-1.
    IF RESULT IS EQUAL TO CS-DATAOKV AND
       SCANNED-CHARS IS EQUAL TO ID-LEN
      THEN MOVE SCANNED-CHARS TO OF-SCANNED-CHARS
           WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-28. Calling the VSNINSPECT_TEXT Procedure

Explanation

VSN-NUM is passed by reference to the procedure. It specifies the ccsversion to be used. The ccsversion contains the rules for applying a truthset. The following are the values allowed for VSN-NUM:

Value	Meaning
Greater than or equal to 0	Specifies a ccsversion. The numbers of the ccsversions are listed in the *MultiLingual System Administration, Operations, and Programming Guide*.
–2	Specifies the system default ccsversion. If the system default ccsversion is not available, an error is returned.

SOURCE-TEXT is passed to the procedure. The record is searched for a character using the requested truthset and type of search. You determine the size of the record.

SOURCE-START is passed by reference to the procedure. It contains the byte offset in SOURCE-TEXT, relative to 0 (zero), at which the search begins.

ID-LEN is passed by reference to the procedure. It specifies the length of the inspected test; that is, the number of characters found to be numeric.

INSPECT-LEN is passed by reference to the procedure. It specifies the number of characters to be searched beginning at SOURCE-START. It specifies that maximum length of the search.

NAME-LEN passed by reference to the procedure. It specifies the length of the name to be inspected.

CS-NUMERICSV is passed to the procedure. It indicates the type of truthset to be used for the search. The following are the values allowed for CS-NUMERICSV and their meanings:

Value	Sample Data Name	Meaning
12	CS-ALPHAV	Alphabetic truthset. It identifies the characters defined as alphabetic in the specified ccsversion.
13	CS-NUMERICSV	Numeric truthset. It identifies the characters defined as numeric in the specified ccsversion.
14	CS-PRESENTATIONV	Presentation truthset. It identifies the characters in the ccsversion that can be represented on a presentation device, such as a printer.
15	CS-SPACESV	Spaces truthset. It identifies the characters defined as spaces in the specified ccsversion.
16	CS-LOWERCASEV	Lowercase truthset. It identifies the characters defined as lowercase alphabetic in the specified ccsversion.
17	CS-UPPERCASEV	Uppercase truthset. It identifies the characters defined as uppercase alphabetic in the specified ccsversion.

A ccsversion is not required to have a definition for each of these truthsets. Some of the truthsets, such as 16 and 17, are optional. A result of 4002 might be returned if the truthset was not defined for the ccsversion. The input text remains unchanged.

CS-NOT-INSETV is passed to the procedure. It indicates the type of search to be performed. The values allowed for this parameter and their meanings are as follows:

Value	Sample Data Name	Meaning
0	CS-NOTINTSETV	Search the text until a character is found that is not in the requested truthset.
1	CS-INTSETV	Search the text until a character is found that is in the requested truthset.

SCANNED-CHARS is an integer returned by the procedure. It contains the number of characters, relative to 0 (zero), that were scanned when the search criteria was met.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by VSNINSPECT_TEXT are as follows:

1	3000	3007
1000	3001	4002
1001	3003	
1002	3006	

Sample output from Example 16–28 follows:

```
RESULT =            1
SCANNED-CHARS =            10
```

For more information on the error result values, see Table 16–1 in this section.

VSNTRANS_TEXT

This procedure applies a specified mapping table to the source text and places the result into the destination parameter. You can use the same record for both the source and destination text.

You might use this procedure to translate alternative digits received as data into numeric digits for arithmetic processing.

Example

Example 16–29 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VSNTRANS_TEXT library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example translates a string in lowercase letters to uppercase letters using the CanadaEBCDIC ccsversion. The input string is "pæan." You can use the *MultiLingual System Administration, Operations, and Programming Guide* to determine that the ccsversion number for CanadaEBCDIC is 74. You can also retrieve this number by calling the procedure CCSVSNNUM with the name CanadaEBCDIC.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL74/VSNTRANSTEXT."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.

01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
```

Example 16–29. Calling the VSNTRANS_TEXT Procedure

```
 01  OF-2.
     05  FILLER           PIC X(12)   VALUE "DEST-TEXT = ".
     05  OF-DEST-TEXT     PIC X(07).
     05  FILLER           PIC X(61)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

 01  DEST-TEXT            PIC X(07).
 01  SOURCE-TEXT          PIC X(07).

 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  CS-LOWTOUPCASEV      PIC S9(11)  USAGE BINARY    VALUE 7.
 77  DEST-START           PIC S9(11)  USAGE BINARY.
 77  SOURCE-START         PIC S9(11)  USAGE BINARY.
 77  RESULT               PIC S9(11)  USAGE BINARY.
 77  TRANS-LEN            PIC S9(11)  USAGE BINARY.
 77  VSN-NUM              PIC S9(11)  USAGE BINARY.
 PROCEDURE DIVISION.
 INTLCOBOL74.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM VSNTRANS-TEXT.
   CLOSE OUTPUT-FILE.
   STOP RUN.
***** VSNTRANS-TEXT ******************************************
 VSNTRANS-TEXT.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE 74 TO VSN-NUM.
   MOVE 4 TO TRANS-LEN.
   MOVE "pæan" TO SOURCE-TEXT.
   CALL "VSNTRANS_TEXT OF CENTRALSUPPORT"
     USING VSN-NUM,
           SOURCE-TEXT,
           SOURCE-START,
           DEST-TEXT,
           DEST-START,
           TRANS-LEN,
           CS-LOWTOUPCASEV
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE DEST-TEXT TO OF-DEST-TEXT.
          WRITE OUTPUT-RECORD FROM OF-2.
```

Example 16-29. Calling the VSNTRANS_TEXT Procedure

Explanation

VSN-NUM is an integer passed by reference to the procedure. It contains the number of the ccsversion to be used. The ccsversion contains the rules for translation of text. Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for a list of the ccsversion numbers.

The values allowed for VSNNUM and the meanings of the values are as follows:

Value	Meaning
0 (zero) or greater	Use the specified ccsversion number.
–2	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

SOURCE-TEXT is passed to the procedure. It contains the data to translate. You should determine the size of this record.

SOURCE-START is passed to the procedure. It designates the byte offset, relative to 0 (zero), in SOURCE-TEXT at which translation is to begin.

DEST-TEXT is returned by the procedure. It contains the translated text. This record and the record in the SOURCE-TEXT parameter should be the same size.

DEST-START is passed to the procedure. It indicates the offset in the DEST-TEXT parameter where the translated text is to be placed.

TRANS-LEN is passed to the procedure. It designates the number of characters in the SOURCE-TEXT parameter to translate, beginning at SOURCE-START.

CS-LOWTOUPCASEV is passed to the procedure. It designates the type of translation requested. The allowed values for CS-LOWTOUPCASEV and their meanings are as follows:

Value	Sample Data Name	Meaning
5	CS-NUMTOALTDIGV	Translate numbers 0 through 9 to alternate digits specified in the ccsversion.
6	CS-ALTDIGTONUMV	Translate alternate digits to numbers 0 through 9.
7	CS-LOWTOUPCASEV	Translate all characters from lowercase to uppercase.
8	CS-UPTOLOWCASEV	Translate all character from uppercase to lowercase.
9	CS-ESCMENTPERCHARV	Translate a character to its escapement value.

A ccsversion is not required to have a definition for each of these tables. Some of the tables, such as 5, 6, 7, and 8, are optional. A result of 4002 might be returned if the table was not defined for the ccsversion. The input text remains unchanged.

RESULT is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by VSNTRANS_TEXT are as follows:

1	3000	3006
1000	3001	4002
1001	3002	
1002	3003	

Sample output from Example 16–29 follows:

```
RESULT =            1
DEST-TEXT = PÆAN
```

For more information on the error result values, see Table 16–1 in this section.

Errors

All of the procedures in the CENTRALSUPPORT library return integer results to indicate the success or failure of the procedure.

Declarations

Example 16–30 shows a sample set of declarations for the message values.

```
01  ERROR-VALUES       PIC S9(11)  USAGE BINARY.
    88  CS-FILE-ACCESS-ERRORV        VALUE  1000.
    88  CS-FAULTV                    VALUE  1001.
    88  CS-SOFTERRV                  VALUE  1002.
    88  LANGUAGE-NOT-FOUNDV          VALUE  2001.
    88  CONVENTION-NOT-FOUNDV        VALUE  2002.
    88  INCOMPLETE-DATAV             VALUE  2004.
    88  BAD-ARRAY-DESCRIPTIONV       VALUE  3000.
    88  ARRAY-TOO-SMALLV             VALUE  3001.
    88  BAD-DATA-LENV                VALUE  3002.
    88  NO-NUM-FOUNDV                VALUE  3003.
    88  NO-NAME-FOUNDV               VALUE  3004.
    88  NO-MSGNUM-FOUNDV             VALUE  3005.
    88  BAD-TYPE-CODEV               VALUE  3006.
    88  BAD-FLAGV                    VALUE  3007.
    88  BAD-TEXT-PARAMV              VALUE  3008.
    88  BAD-TEMPCHARV                VALUE  3011.
    88  BAD-DATEINPUTV               VALUE  3012.
    88  BAD-TIMEINPUTV               VALUE  3013.
    88  CNV-EXISTS-ERRV              VALUE  3014.
    88  BAD-MAXDIGITSV               VALUE  3015.
    88  BAD-FRACDIGITSV              VALUE  3016.
    88  BAD-ALTFRACDIGITSV           VALUE  3017.
    88  BAD-LDATETEMPV               VALUE  3018.
    88  BAD-SDATETEMPV               VALUE  3019.
    88  BAD-NDATETEMPV               VALUE  3020.
    88  BAD-LTIMETEMPV               VALUE  3021.
    88  BAD-NTIMETEMPV               VALUE  3022.
    88  BAD-MONTEMPV                 VALUE  3023.
    88  BAD-NUMTEMPV                 VALUE  3024.
    88  BAD-LPPV                     VALUE  3027.
    88  BAD-CPLV                     VALUE  3028.
    88  REQSYMBOLV                   VALUE  3029.
    88  BAD-TEMPLENV                 VALUE  3030.
    88  MUTUAL-EXCLUSIVEV            VALUE  3031.
    88  BAD-MINDIGITSV               VALUE  3032.
    88  MISSING-RBRACKETV            VALUE  3033.
```

Example 16–30. Declaring Message Values

```
88  MISSING-TCCOLONV            VALUE  3034.
88  BAD-INPUTVALV               VALUE  3035.
88  CNV-NOTAVAILV               VALUE  3036.
88  CNVFILE-NOTPRESENTV         VALUE  3037.
88  BAD-PRECISIONV              VALUE  3038.
88  NO-CNVNAMEV                 VALUE  3039.
88  DEL-PERMANENTCNV-ERRV       VALUE  3040.
88  NO-HEXCODE-DELIMV           VALUE  3041.
88  BAD-HEXCODEV                VALUE  3042.
88  NO-ALTCURR-DELIMV           VALUE  3043.
88  DATA-NOT-FOUNDV             VALUE  4002.
```

Example 16-30. Declaring Message Values

Explanation of Error Values

The range 1000 through 1999 are error messages that report a system software error.

Values from 2000 through 2999 contain error messages in which the caller passed invalid data to a procedure, but the CENTRALSUPPORT library was able to return some valid data.

Values from 3000 through 3999 contain error messages in which the caller passed invalid data to a CENTRALSUPPORT procedure, and the CENTRALSUPPORT library was unable to return any valid data.

Values from 4000 through 4999 contain error messages in which the caller passed some sort of data for which the CENTRALSUPPORT library could find no return information. CENTRALSUPPORT completed the request, but no data was returned.

Table 16–1 lists the error numbers that can be returned for internationalization and the specific descriptions of the error messages that you can have your program display.

For More Information

- Refer to GET_CS_MSG earlier in this section for information about the message parts.
- Refer to the *MultiLingual System Administration, Operations, and Programming Guide* for a list of the complete error messages, and for information about the corrective actions to be taken if an error occurs.

Table 16–1. Specific Descriptions for Internationalization Error Values

Error Value	Specific Description
1000	An error occurred while accessing the SYSTEM/CCSFILE or the SYSTEM/CONVENTIONS file.
1001	An unexpected fault occurred in CENTRALSUPPORT, and a program dump of the CENTRALSUPPORT library might occur. Your request cannot be processed at this time.
1002	A CENTRALSUPPORT software error was detected, and a program dump of the CENTRALSUPPORT library might occur. Your request cannot be processed at this time.
2001	The data is not in the requested language. It is in MYSELF.LANGUAGE or the SYSTEM LANGUAGE or the first available LANGUAGE.
2002	The data is not in the requested convention; it is in MYSELF.CONVENTION or the SYSTEM CONVENTION.
2004	Only partial data is being returned. There was insufficient space in the output array.
3000	A parameter was incorrectly specified as less than or equal to 0.
3001	The output array size is smaller than the length of the data it is supposed to contain.
3002	At least one array length is invalid or the offset + length is greater than the size of the array.
3003	The requested number was not found.
3004	The requested name was not found.
3005	The requested number was not found.
3006	The type code specified is out of the acceptable range.
3007	The flag specified is out of the acceptable range.
3008	The space for OSVs or total storage allocated in OUTPUT is not big enough for OSVs and/or PSVs.
3011	An invalid control character was detected in the template.
3012	The input date contains invalid characters.
3013	The input time contains invalid characters.
3014	An attempt was made to add a new convention with the name of an existing convention.
3015	The maximum digits value is either missing or out of range.
3016	The fractional digits value is either missing or out of range.
3017	The international fractional digits value is either missing or out of range.

Table 16–1. Specific Descriptions for Internationalization Error Values

Error Value	Specific Description
3018	The long date template is either missing or contains invalid information.
3019	The short date template is either missing or it contains invalid information.
3020	The numeric date template is either missing or contains invalid information.
3021	The long time template is either missing or it contains invalid information.
3022	The numeric time template is either missing or contains invalid information.
3023	The monetary template is either missing or it contains invalid information.
3024	The numeric template is either missing or it contains invalid information.
3027	The lines per page value is either missing or it is out of range.
3028	The characters per line value is either missing or it is out of range.
3029	A required symbol in either the monetary or the numeric template is missing.
3030	An invalid template length value was encountered.
3031	A mutually exclusive combination of control characters has been encountered in a monetary or numeric template.
3032	The mindigits field in a "t" control character in a monetary or numeric template is out of range.
3033	A right bracket "]" is required to terminate a "t" control character symbol definition list.
3034	An expected colon ":" is missing from the "t" control character in a monetary or numeric template.
3035	The input value did not contain digits or an expected symbol was missing.
3036	Specified convention does not exist and cannot be retrieved, modified, or deleted.
3037	A convention definition cannot be added, modified, or deleted.
3038	The "PRECISION" parameter value is out of range.
3039	A required convention name was not provided.
3040	The named convention is a standard convention and cannot be modified or deleted.
3041	A hexadecimal value representing a symbol in a monetary or numeric template is missing a required delimiter.
3042	An invalid character was encountered in a hex value representing a symbol in a monetary or numeric template.
3043	The international currency notation is missing a required terminating delimiter.
3044	The date components are separated by an invalid character.
3045	The year component exceeds two digits.
3046	A nonzero value is required for the year component.
3047	The month value is outside the valid range. Acceptable values are in the range of 1 through 12.

Table 16–1. Specific Descriptions for Internationalization Error Values

Error Value	Specific Description
3048	The day value is outside the valid range. Acceptable values for January through December are: 1 through 31 for January, March, May, July, August, October, and December; 1 through 30 for April, June, September, and November; and 1 through 28 for February (1 through 29 in a leap year).
3049	An input date is required but missing.
3050	Time components are separated by an invalid character.
3051	The hour value is outside the valid range for a 24-hour clock. Acceptable values are in the range of 1 through 24.
3052	The hour value is outside the valid range for a 12-hour clock. Acceptable values are in the range of 1 through 12.
3053	The minute value is outside the valid range. Acceptable values are in the range of 0 through 59.
3054	The second value is outside the valid range. Acceptable values are in the range of 0 through 59.
3055	The partial second value contains invalid characters.
3056	An input time is required but missing.
3057	The month value is required but missing.
3058	The day of year value is outside the valid range. Acceptable values are in the range of 1 through 365 (1 through 366 in a leap year).
3059	The day of year value cannot be calculated because a date component (year, month, or day) is missing.
4002	The requested data was not found.

Section 17
Control of the Compilation Process

The COBOL74 compiler enables you to control portions of the compilation process. You can control the way you start the compilation, the files you use as input to and the files produced as output from the compiler, and the various options that direct the compilation.

Starting a Compilation

Programs can be compiled through CANDE or WFL. You can choose the method that is the most familiar or the most convenient.

You can make a COBOL74 file in CANDE by using the following CANDE syntax:

```
MAKE <MYFILE> C74
```

If a program is created of type C74, then you can enter the COMPILE command to begin the compilation.

The default value of a compiler option can depend on whether the compilation starts from CANDE or WFL. If this is the case for a particular option, the default for both CANDE and WFL compiles is documented under the description of the option.

For More Information

- Refer to the *CANDE Operations Reference Manual* for information about the MAKE and COMPILE commands. These commands can be used to make and compile a file through CANDE.
- WFL is invoked by using the WFL *START* command in CANDE. Refer to the *WFL Reference Manual* for information about compiling a program using a WFL deck.

Using Cross-Reference Files

A cross-reference file is a file that contains an alphabetized list of user-defined words that appear in a program.

For each user-defined word, the cross-reference information contains the following data as well as other information:

- The type of the item the word identifies
- The sequence number of the source input record at which the identifier is declared
- The sequence number of the source input record at which the word is declared
- The sequence numbers of the input records at which the word is accessed

Depending on the values of the related compiler options–NOXREFLIST, XDECS, XREF, XREFFILES, and XREFS—the compiler optionally generates cross-reference information.

The following factors can be controlled through the use of compiler control options:

- Whether the compiler saves cross-reference information as it processes the source input
- Which user-defined words and which references to these words are cross-referenced
- Whether the compiler starts the SYSTEM/XREFANALYZER utility automatically
- If the compiler starts the SYSTEM/XREFANALYZER utility automatically, whether the utility produces printed output, disk files suitable for input to the SYSTEM/INTERACTIVEXREF utility and the Editor, or both the printed output and the disk files

The compiler saves cross-reference information if the XREF option, the XREFFILES option, or both options are TRUE. If used, these options should be assigned the value TRUE before the end of the IDENTIFICATION DIVISION.

Table 17–1 shows the effects of setting the XREFS option, the XDECS option, or both compiler control options. The XDECS option must be used with the XREF option, the XREFFILES option, or both options. The XREFS option must be used with the XDECS option, and the XREF and XREFFILES options.

Table 17–1. Effects of the XDECS and XREFS Compiler Control Options

Options That Are Set	Effects
Any of the following combinations: • XREF and XDECS • XREFFILES and XDECS • XREF, XREFFILES, and XDECS	Selects the user-defined words to be cross-referenced.
XDECS	No effect when set by itself.
Any of the following combinations: • XREF and XREFS • XREFFILES and XREFS • XREF, XREFFILES, and XREFS	Selects the user-defined words to be cross-referenced. If XDECS is FALSE, the XREFS option has no effect.
XREFS	No effect when set by itself.

When the compiler is saving cross-reference information, this information is written to a disk file in raw form. Before this information can be printed or read by the SYSTEM/INTERACTIVEXREF utility or the Editor, it must be analyzed by the SYSTEM/XREFANALYZER utility.

You have the option of instructing the compiler not to start the SYSTEM/XREFANALYZER utility, keeping the raw file of the compiler on disk. Table 17–2 shows the effects of setting the NOXREFLIST option.

Table 17–2. Effects of the NOXREFLIST Option

If the value is . . .	The compiler . . .
FALSE	Automatically starts the SYSTEM/XREFANALYZER utility to process the raw cross-reference file.
TRUE	Does not start the SYSTEM/XREFANALYZER utility, and the raw file of the compiler is left on disk with the title XREF/code-file-name, where code-file-name is the name of the object code file produced by the compiler. You can then run the SYSTEM/XREFANALYZER utility directly at a later time to analyze the raw file.

If the compiler starts the SYSTEM/XREFANALYZER utility (that is, if the NOXREFLIST option is FALSE), the program produces either a printed listing or a pair of disk files suitable for the Editor and the SYSTEM/INTERACTIVEXREF utility.

Table 17–3 shows the effects of the XREF and XREFFILES options when the NOXREFLIST option is FALSE.

References and declarations in the interactive cross-reference files and variables within a copy library refer to the line number of the COPY statement in the source program. These references and declaratives do not refer to the actual line numbers within the copy library. However, when both the XREF and XREFFILES options are set, the actual line numbers within the COPY file are used for the references and the declarations.

Table 17–3. Effects of the XREF and XREFFILES Options

If the option is. . .	The compiler produces . . .
XREF	A listing.
XREFFILES	Two disk files. These file are titled *XREFFILES/code-file-name/XREFS* and *XREFFILES/code-file-name/XDECS*.
XREF and XREFFILES	Both a listing and the disk files.

Note: *If syntax errors occur, the cross-reference information produced might be unreliable. In extreme cases, the SYSTEM/XREFANALYZER utility fails.*

For More Information

- Refer to "NOXREFLIST," "XDEC," "XREF," "XREFFILES," and "XREFS" in this section for more information about these options.
- Refer to the discussion of the SYSTEM/XREFANALYZER and SYSTEM/INTERACTIVEXREF utilities in the *System Software Utilities Manual* for more information about cross-reference files.
- Refer to the *Editor Operations Guide* for information about getting cross-reference information while in the Editor.

Performing a Separate Compilation

You can separately recompile selected areas of a previously compiled COBOL program by using the SEPCOMP compiler control option. To perform a separate compilation, you must first compile the program with the compiler control option MAKEHOST assigned the value TRUE. This step creates the host file. The host file is an executable code file that contains additional information necessary for a separate compilation. Given only the name of the host file, the name of the original source, and the patches to change the source, the compiler is able to separately compile the patched areas, resulting in an abbreviated compilation. Only the affected areas of the program are actually compiled.

You can use the separate compilation facility as a supplement to, not a replacement for, the standard method of compilation. This separate compilation is meant to be used in development work on large COBOL programs where a reduction in the time required for compilation of large programs can be beneficial.

The following considerations for using the SEPCOMP option apply when you are performing a separate compilation procedure:

- The SEPCOMP option cannot be explicitly referenced after the beginning of the compilation.
- The SEPCOMP option cannot be set more than once because when the option is first set, the SEPCOMP option starts the preprocessing of the CARD file input.
- The title of the host program can be specified either as a string within parentheses immediately following the word SEPCOMP on the CCR, or by a file equation to the host file in the COBOL compiler. The optional string specification has precedence over the file equation. The host file contains the name of the symbolic file from which it was compiled.

Examples

Example 17–1 shows a separate compilation using a WFL deck in which the title of the host program is given as a string.

```
?BEGIN JOB COMPILE/A/B;
       COMPILE A/B WITH COBOL74  LIBRARY;
       COBOL DATA
000010$ SEPCOMP ("A/HOST") LIST MAP
     :
<patch records>
     :
?END JOB
```

Example 17–1. Separate Compilation with the Host Title Given as a String

Example 17–2 shows a separate compilation using a WFL deck in which the title of the host program is file-equated to the COBOL file host.

```
?BEGIN JOB COMPILE/A/B;
       COMPILE A/B WITH COBOL74 LIBRARY;
       COBOL FILE HOST (FILENAME=A/HOST);
       COBOL DATA
000010$ SEPCOMP LIST MAP
    :
<patch records>
    :
?END JOB
```

Example 17–2. Separate Compilation with the Host Title File-Equated

Providing the Changed Records

A patch record is a record with a nonblank sequence number; that is, at least one digit of the sequence number is required. Once the host file name is known, the patch records must be provided. The compiler examines the patch records of a SEPCOMP compilation to determine the sections or code segments that need to be recompiled. The output from a SEPCOMP compilation is an executable code file that, unlike ALGOL and NEWP files, cannot be used as the host file for the next SEPCOMP compilation.

The compiler accepts compiler control records (CCRs) with blank sequence numbers following the compiler control record that sets the SEPCOMP option and before the first patch record.

Sequence errors are not allowed among patch records having nonblank sequence numbers.

Observing Compilation Restrictions

You are responsible for observing the following restrictions when performing a separate compilation:

- The host symbolic file and all of the COPY library files must be unchanged since the MAKEHOST compilation. A SEPCOMP compilation verifies this requirement by checking the date and timestamp.
- Both the host symbolic file and the patch file for a SEPCOMP compilation must be in sequential order.
- You must use a COBOL compiler of the same release level for both the MAKEHOST and SEPCOMP compilations.
- The SOURCE (host symbolic) file must be assigned to a mass-storage device, because random access is necessary.
- All new label declarations must be unique, regardless of qualification.
- Only PROCEDURE DIVISION statements are allowed to be patched with the SEPCOMP compilation facility.

- New PROCEDURE DIVISION statements can be added, and existing statements can be modified or deleted. PROCEDURE DIVISION labels can be deleted as long as the label is referenced only within the code segment in which the label resides.
- The SORT, MERGE, USE, and ALTER verbs cannot be used in either the MAKEHOST or the SEPCOMP compilation.
- A DECLARATIVES SECTION cannot be added, deleted, or modified.
- The DEBUG facility of COBOL cannot be used with a SEPCOMP compilation.

If any of the preceding restrictions are violated, the COBOL compiler aborts the SEPCOMP compilation.

The compiler control options MERGE, NEW, OPTIMIZE, and STATISTICS are ignored during a MAKEHOST or a SEPCOMP compilation.

The compiler control options MERGE and XREF are ignored during a SEPCOMP compilation.

For More Information

- For information about creating a host file, refer to “MAKEHOST” in this section.
- For information about separately recompiling selected areas of a previously compiled program, refer to “SEPCOMP” in this section.

Compiler Control Option Concepts

Compiler control options provide control of the following compiler functions:

- Source language input
- Source language output
- Optional compilation mechanisms
- Printed output
- Compiler diagnostic messages
- Compiler debugging
- User options
- Code generation

You can specify these options on compiler control records (CCRs). A CCR contains compiler control statements made up of options or groups of options and the associated parameters, if any. By using CCRs, you can control the options provided by the compiler.

Types of Compiler Control Records (CCRs)

A CCR can be either temporary or permanent. The differences between the two types of CCRs are described in the following list:

- Temporary CCRs apply only to a given compilation. These CCRs have a dollar currency sign ($) in column 7.
- Permanent CCRs remain associated with the source language. These CCRs have a dollar currency sign ($) in columns 7 and 8. The compiler includes permanent CCRs in any NEWSOURCE file created.

Types of Compiler Control Options

A compiler control option can be of type Boolean, value, or immediate. The following lists show the options belonging to each category.

Boolean

A Boolean option is either enabled (set to TRUE) or disabled (set to FALSE). When enabled, the option directs the compiler to apply an associated function to all subsequent processing until the option is disabled. Boolean options can also have associated parameters related to the function affected by the Boolean option.

The Boolean options are as follows:

BINARYCOMP	INFO	NEWID	STERNFATAL
BINDINFO	LIB$	NOXREFLIST	STERNMULTDEST
CODE	LIBDOLLAR	OMIT	STRICTPICTURE
COMPILERDEBUG	LIBRARYLOCK	OPT	SUMMARY
DATADICTINFO	LINEINFO	OPTIMIZE	TADS
DELETE	LIST	OWN	TEMPORARY
DICTIONARY	LISTDELETED	OWNTEMP	User Option
DELETE	LIST$	SEPCOMP	WARNFATAL
ERRORLIST	LISTDOLLAR	SEQ	WARNSUPR
FREE	LISTOMITTED	SEQCHECK	XDECS
GLOBAL	LISTP	SEQUENCE	XREF
GLOBALTEMP	LIST1	SPEC	XREFFILES
GROUPMOVEWARN	MERGE	STATISTICS	XREFS
	NEW	SPEC	

Value

A value option directs the compiler to store a value associated with a given function. The value options are as follows:

DEBUG	Sequence Increment
ERRORLIMIT	SHARING
FEDLEVEL	Symbolic ID
LEVEL	TARGET
Sequence Base	

Immediate

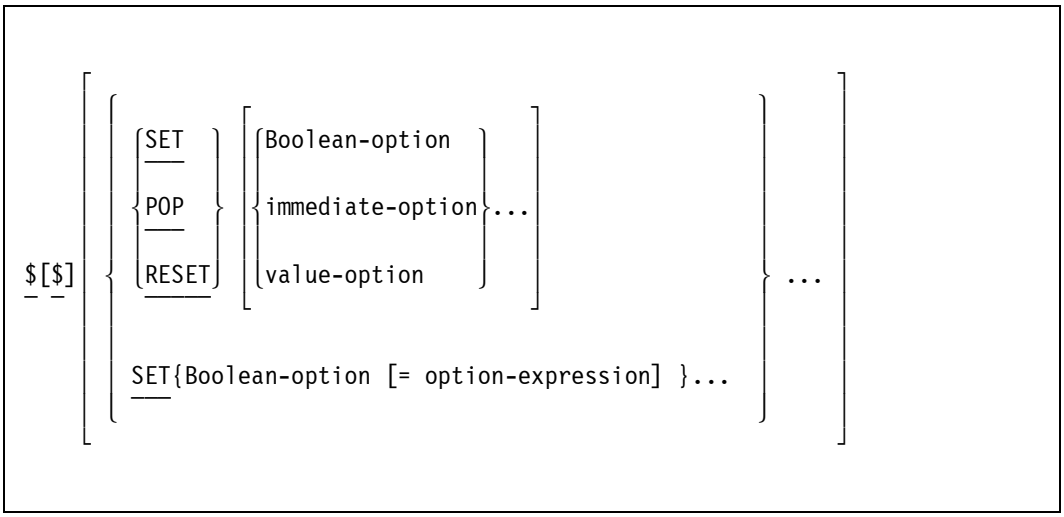
An immediate option directs the compiler to apply a function that is independent of subsequent processing. Immediate options can also have associated parameters. The immediate options are CLEAR, PAGE, and VOID.

Compiler Control Option Formats

The following text describes how compiler control options are formatted.

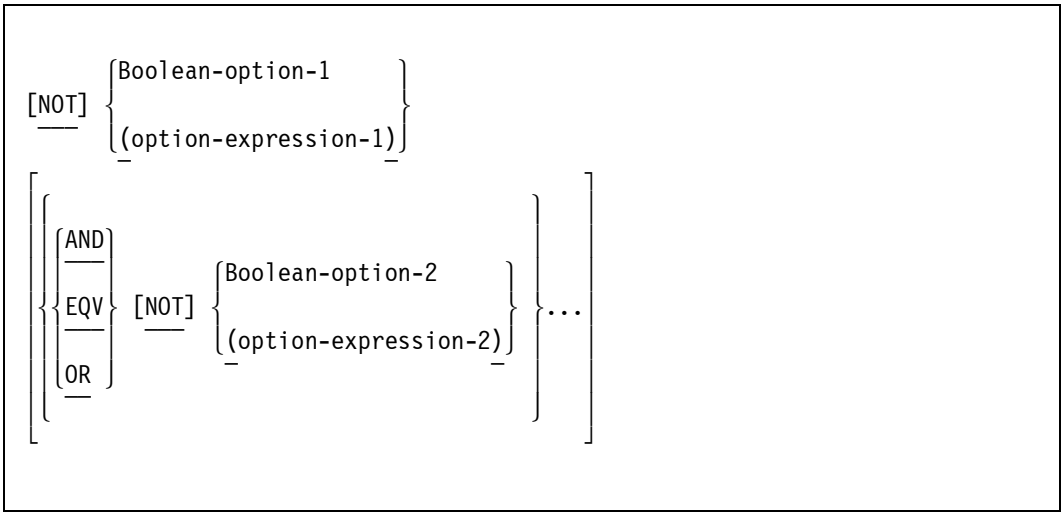
compiler control record

The following is the general format for a compiler control record (CCR):



option-expression

The following is the general format for an option-expression:



Explanation of Formats

CCRs must have a dollar currency sign ($) in column 7 and can have an optional dollar sign in column 8. The options follow the dollar currency sign, and one or more spaces follow each option. No option can continue past column 72 of a CCR.

The option expressions follow the standard rules of Boolean algebra for statement evaluation.

CCRs can be interspersed at any point in the source language input. However, a CCR might affect the syntactical element that precedes it for the following reason. The CCR that follows a COBOL74 syntactical element actually is processed while the compiler is preparing that syntactical element for future analysis and delivering the previous syntactical element for parsing.

Special attention should be paid to the use of CCRs in the immediate vicinity of COPY statements. For example, if a COPY statement immediately follows a CCR, the options specified in the CCR can affect the first record included as a result of the COPY statement, unless these options are explicitly prevented from doing so. Adding a semicolon (;) after the ending period in the COPY statement ensures that the option changes take effect after all source information from the COPY statement has been processed.

Because the CCR can affect the preceding statement or clause and cause unexpected results, it is recommended that you insert dummy statement between the two records.

Some options affect generated object code. These effects are discussed in the description of the particular option.

For More Information

- For information about instructing the compiler to treat all COMPUTATIONAL items as if they were declared USAGE IS BINARY, refer to “BINARYCOMP” in this section.
- For information on causing all data items in the WORKING-STORAGE SECTION to be global except those specifically declared LOCAL or OWN, refer to “GLOBAL” in this section.
- For information on causing all data items in the WORKING-STORAGE SECTION to be OWN except those specifically declared LOCAL or GLOBAL, refer to “OWN” in this section.
- For information about the COPY statement, refer to “COPY” in Section 9, “PROCEDURE DIVISION Statements.”

Option Action Indicators

Option action indicators preceding a list of options set, reset, or recall the last setting of each settable option in the list.

A register that reflects the last 47 settings of the option is associated with each of the settable options. The presence of any preceding option action indicator is ignored when nonsettable options are processed.

The option action indicators are defined as follows:

If . . .	The current setting of each option specified is . . .
SET	Saved, and each of the options is set to TRUE (enabled or on). The option can also be set to the value of an optional Boolean expression.
RESET	Saved, and each of the options is set to FALSE (disabled or off).
POP	Discarded, and each of the options is set to its prior setting. When no prior setting exists, a POP option action indicator leaves the option set to FALSE.

A CCR that consists solely of a dollar currency sign ($) and no option information has no effect unless the MERGE option is TRUE. If the MERGE option is TRUE and a record is present in the secondary input file (SOURCE file) that has the same sequence number as the blank CCR, then the source record is ignored.

A CCR that contains valid information in the primary input file (CARD file) always takes effect before a CCR that already exists in the secondary input file (SOURCE file) if the MERGE option is set and both records have the same sequence number.

COBOL74 Source and Object Files

Source input is submitted to the COBOL compiler as one or more punched card, disk, or magnetic tape files. If more than one source file is used, the input from these files is merged on the basis of sequence numbers or as specified by the COPY statement. In addition to the object code file, the COBOL compiler produces the following optional output files: an updated symbolic file, an error message file, and a printer listing containing the source records and compiler control records (CCRs) used by the compiler, and other types of information.

Figure 17–1 shows the flow of input and output data during compilation.

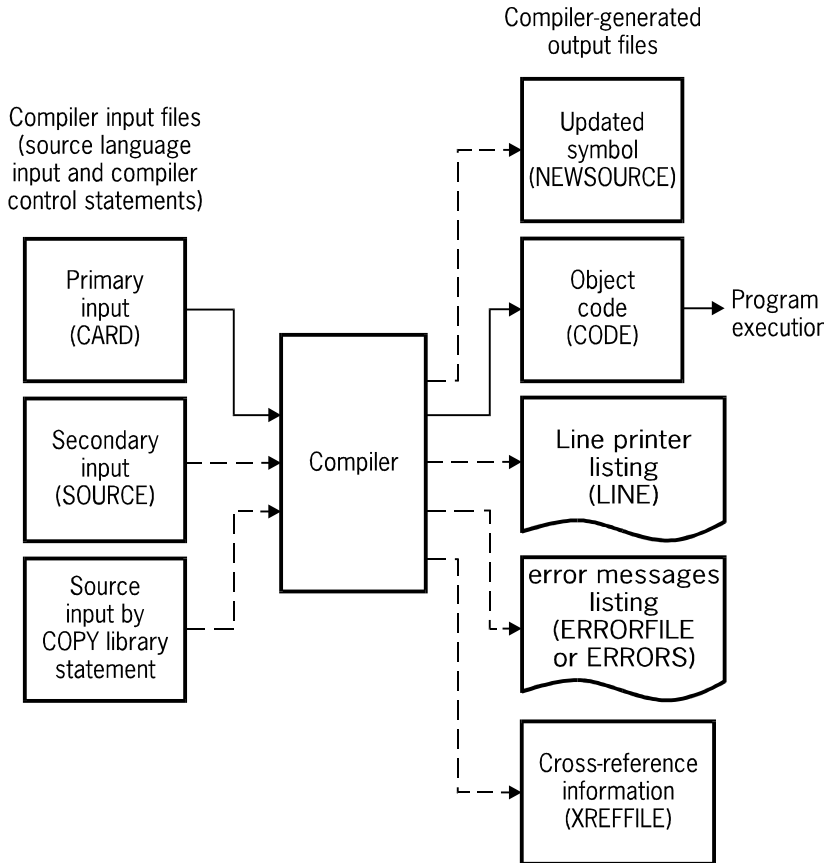


Figure 17–1. Compiler Data Flow

Input Files

The compiler accepts input from the CARD file, the SOURCE file, and the COPY library files. Input can be from the CARD file only, from the CARD file and the SOURCE file, from the CARD file and the COPY library files, or from all three.

Table 17–4 shows the file attribute name, its assigned value, and its purpose for the compiler input files.

Table 17–4. Attribute Values for the Compiler Input File

Name	Value	Purpose
EXTMODE	EBCDIC or ASCII	Specifies the character type of the physical file.
DEPENDENTSPECS	TRUE	Specifies that the format of the records and the structure of the logical file should be determined by the structure of the associated permanent file.
INTMODE	EBCDIC	Specifies the units for the BLOCKSIZE and MAXRECSIZE attributes.
MAXRECSIZE	Taken from the length of the physical file.	Defines the smallest unit of the file that the program can read from or write to at any one time.
BLOCKSIZE	Taken from the length of the physical file.	Defines the smallest unit of the file that the I/O subsystem can physically read from or write to. Because the value of the BLOCKSIZE attribute for this file is assumed from the physical file, this attribute does not require explicit assignment.

The DEPENDENTSPECS file attribute is set to TRUE for all compiler input files. To set file attributes that are different from those of the permanent file (for example, BLOCKSIZE), you must set DEPENDENTSPECS to FALSE.

Each file has an internal name specified by the INTNAME file attribute. Each file is assigned to a device specified by the KIND file attribute. Table 17–5 shows the internal name and the device to which the file is assigned for each compiler input file. In some cases, the default device depends on whether the compilation is started from CANDE or WFL.

Table 17–5. Compiler Input Files

Internal Name	Language Initiation	Device	Explanation
CARD	WFL	READER	The CARD file supplies the primary source input to the compiler and must be present for each compilation. If the compilation is started from CANDE or WFL, and the compiler file equation is not applied to the CARD file, the file is assumed to be a card reader file.
CARD	CANDE	DISK	If you compile from CANDE and compiler file equation is not applied to the CARD file, the file is assumed to be a disk file.
SOURCE	WFL, CANDE	SOURCE	The SOURCE file, which is optional, supplies secondary source input to the compiler. If the compiler file equation is not applied to the SOURCE file, the file is always assumed to be a disk file. When this file is present and the MERGE option is TRUE, records from the SOURCE file are merged with those of the CARD file on the basis of sequence numbers. If a record from the CARD file and a record from the SOURCE file have the same sequence number, the CARD file record is compiled and the SOURCE file record is ignored.
COPY files	WFL, CANDE	DISK	COPY library files provide source input to the compiler in addition to that supplied by the CARD and SOURCE files and are present only if one or more COPY statements appear in the CARD or SOURCE files.

For More Information

- For details on merging source-language records, see "MERGE."
- For details on the COPY statement, refer to "PROCEDURE DIVISION Statements."
- See the *I/O Subsystem Programming Guide* for information on using file attributes.
- See the *WFL Reference Manual* for details pertaining to the compiler file equation.

Output Files

The compiler produces one to five output files. These files are described next.

CODE File

The CODE file is produced unconditionally and contains the executable object code produced by the compiler. Whether this file is discarded after compilation, executed and then discarded, or stored permanently depends on the specifications in the COMPILE statement and whether or not syntax errors occurred during compilation.

NEWSOURCE File

The NEWSOURCE file is produced only if the NEW option is TRUE. The NEWSOURCE file is an updated source file that contains the portion of source input from the CARD and SOURCE files that was actually compiled. If the compiler file equation is not applied to the NEWSOURCE file, the file is written to disk.

LINE File

The LINE file is produced unless the LIST option is FALSE. If the compiler is started from CANDE, the default value of the LIST option is FALSE and must be explicitly set to TRUE for the LINE file to be produced. If the compiler is started from WFL, the LINE file is produced by default. If the compiler file equation is not applied to the LINE file, the file is written to the printer. The contents of the LINE file depends on the values of the CODE option and the MAP option; however, when printed, the line file contains the following minimum information:

- Source used as input to the compiler
- Code segmentation information
- Error messages and error count, if syntax errors occur

ERRORFILE File

Note: *ERRORFILE is for use with compilations initiated by CANDE, in which the compiler automatically sets ERRORLIST to TRUE and assigns ERRORFILE to the originating CANDE station. Use the synonymous ERRORS file, which is a general-purpose error file, to avoid the conditional default device assignment.*

The ERRORFILE file is produced only if the ERRORLIST option is TRUE. If the compiler is started from WFL, the default value of the ERRORLIST option is FALSE and must be explicitly set to TRUE for the ERRORFILE file to be produced. If the compiler file equation is not applied to the ERRORFILE file, the file is written to the printer. If the compiler is started from CANDE, the ERRORFILE file is produced by default and written to the remote station that started the compiler. If no syntax errors occur during compilation, no ERRORFILE file is produced regardless of the value of the ERRORLIST option.

For every record that contains syntax errors, a copy of the source record containing the error is written to the ERRORFILE file, followed by the syntax errors that occurred for that record.

The synonym *ERRORS* can be used instead of *ERRORFILE*. If you do not file equate to either ERRORFILE or ERRORS, the file is created with the title ERRORS. If you file equate to both ERRORFILE and ERRORS, only the equation to ERRORS is used.

XREFFILE File

When either the XREF or XREFFILES compiler control option is TRUE, the compiler saves raw cross-reference information in the XREFFILE file. The XREFFILE file is named XREFFILES/code-file-name, where code-file-name is the name of the object code file produced by the compiler. The SYSTEM/XREFANALYZER utility can read this file to produce a printed cross-reference file, a disk XREFFILE file, or both for use with the SYSTEM/INTERACTIVEXREF utility or the Editor utility. The SYSTEM/XREFANALYZER utility is run automatically unless the NOXREFLIST compiler control option is TRUE.

Attribute Settings

The following table summarizes the attribute settings for each type of compiler output file. For the file ERRORFILE, the default device depend on whether the compilation starts from CANDE or from WFL.

Table 17–6. Attribute Assignments for Compiler Output Files

INTNAME	KIND	INTMODE	MAXRECSIZE	BLOCKSIZE
CODE	Disk	HEX	30 words	150 words
NEWSOURCE	Disk	EBCDIC	15 words	450 words
LINE	Printer	EBCDIC	22 words	22 words
ERRORFILE (WFL)	Printer	EBCDIC	12 words	12 words
ERRORFILE (CANDE)	Remote	EBCDIC	12 words	12 words
XREFFILE	Disk	EBCDIC	510 words	510 words

Legend

ERRORFILE is Synonymous with ERRORS

For More Information

- Refer to the discussions of the COMPILE statement in the *WFL Reference Manual* and the *CANDE Operations Reference Manual*.
- For information on creating a new source-language symbolic, refer to "NEW" in this section.
- For information on requesting a listing of the object code, refer to "CODE" in this section.
- For information on requesting a map of variables in the object program, refer to "MAP" in this section.
- For information about cross-reference files, refer to "Using Cross-Reference Files" in this section.

Compiler Control Options

The following paragraphs describe the general format, the default settings, and the details of the individual compiler control options. The options are listed in alphabetic order.

BINARYCOMP

```
BINARYCOMP
```

(Type: Boolean, Default: FALSE)

The BINARYCOMP option directs the compiler to treat all COMPUTATIONAL items as if they were declared USAGE IS BINARY.

The compiler examines the current setting of the BINARYCOMP option as it parses each data declaration, after it examines the level-number, and before it examines the data-name. A change to the BINARYCOMP option within a data declaration affects the data item if the option change occurs before the first clause following the data-name.

It is recommended that you do not change the BINARYCOMP option within the clauses of a data declaration. Instead, the option should be changed either before the level number or after the ending period of the data declaration.

BINDINFO

```
BINDINFO
```

(Type: Boolean, Default: FALSE)

The BINDINFO option causes information used for binding to be placed in the code file unconditionally. This information is then available to the PROGRAMDUMP and DUMPANALYZER utilities.

The BINDINFO option has no effect on programs compiled with the LEVEL option set to 3 or higher, nor does the BINDINFO option affect programs that call an external procedure. Binding information is always produced for such programs.

The setting of this option cannot be changed after the first source record.

The BINDINFO option is incompatible with the STATISTICS and TADS options. If an attempt is made to set both the STATISTICS and BINDINFO options to TRUE, the first option that became TRUE is unaffected, the second option is set to FALSE, and a warning message is issued.

CLEAR

```
CLEAR
```

(Type: Immediate)

The CLEAR option sets to FALSE the current setting and all previous settings of all Boolean options except MERGE, OPTIMIZE, and, conditionally, NEW, STATISTICS, FREE, and LINEINFO.

If a source-language record has been written to the new symbolic file (NEWSOURCE) as a result of the NEW option, then the NEW option is not set to FALSE.

If the first source record has been processed, STATISTICS, FREE, and LINEINFO are not set to FALSE.

CODE

```
CODE
```

(Type: Boolean, Default: FALSE)

The CODE option provides a listing of the object code.

COMPILERDEBUG

This option is for Unisys internal use only.

DEBUG

This option is for Unisys internal use only.

DELETE

```
DELETE
```

(Type: Boolean, Default: FALSE)

The DELETE option discards source-language records from the secondary input (SOURCE file) until the option becomes FALSE.

This option can appear only on a CCR in the primary source-language input (CARD file).

If the MERGE option is FALSE, the compiler ignores the DELETE option. The DELETE option does not alter the normal merging process. The compiler does not carry forward the discarded source-language records to the output symbolic file (NEWSOURCE file) if the NEW option is TRUE, and these records are not listed unless the LISTDELETED option is TRUE.

During the merging process, the compiler control options from the secondary input (SOURCE file) within the DELETE range take effect. You must ensure that SETs, POPs, and RESETs within the DELETE option do not accidentally destroy other matching SETs, POPs, and RESETs, causing unexpected results. If the NEW option is TRUE, however, the compiler control options within the DELETE range are not carried forward to the output symbolic file (NEWSOURCE file), even if these compiler control options begin with two dollar currency signs ($$).

For information on listing discarded source-language records, refer to "LISTDELETED" in this section.

DOUBLE

```
DOUBLE
```

(Type: Boolean, Default: FALSE)

The DOUBLE option double-spaces all printed output.

ERRORLIMIT

```
ERRORLIMIT = integer
```

(Type: Value, Default: 10 for compilations originated through CANDE; otherwise, 150)

The ERRORLIMIT option specifies the maximum number of errors that the compiler can produce before a compilation is ended.

If the error limit is exceeded, the compiler produces a listing of the errors and informs you that the compilation was ended because the error limit was exceeded.

If the error limit is exceeded and the NEW option is TRUE, then the new symbolic file (NEWSOURCE) is purged.

ERRORLIST

(Type: Boolean, Default: TRUE for compilations originated through CANDE; otherwise, FALSE)

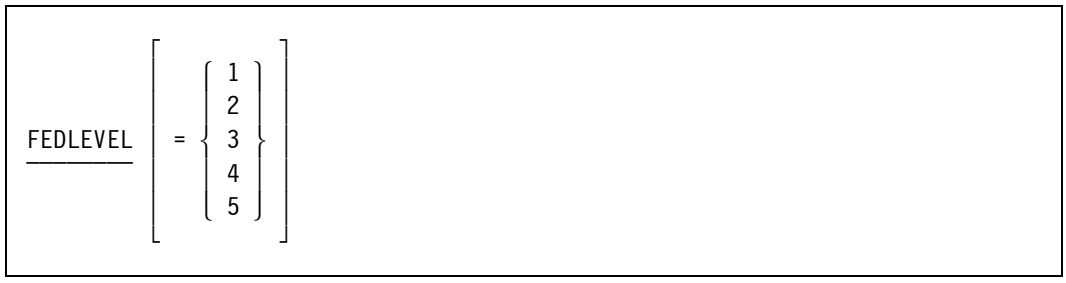
The ERRORLIST option causes syntax errors to be written to the file ERRORFILE.

When the compiler detects a syntax error in the source input, it writes the line of text in error, an error message, and a pointer to the syntactical item in question on two lines in the file ERRORFILE. The ERRORLIST option is provided primarily for use when the compiler is called from a remote terminal through CANDE, but the option can be used regardless of the manner in which the compiler is called.

When the compiler is called from CANDE, the default value of the ERRORLIST option is TRUE and the file ERRORFILE is automatically equated to the remote device from which the compilation originated.

Note: *ERRORFILE is for use with compilations initiated by CANDE, in which the compiler automatically sets ERRORLIST to TRUE and assigns ERRORFILE to the originating CANDE station. Use the synonymous ERRORS file, which is a general purpose error file, to avoid the conditional default device assignment.*

FEDLEVEL



(Type: Value, Default: 0 or 4)

The FEDLEVEL option provides a facility for specifying and evaluating levels of COBOL to measure compliance with U.S. Government COBOL standards.

The default value for the FEDLEVEL option is either 0 or 4, depending on whether the option is stated or not. If FEDLEVEL appears with no value, then the default is level 4. If the FEDLEVEL option is not stated, then the default is level 0.

The compiler produces nonfatal warnings for constructs not available at the level at which the program was compiled. For example, if FEDLEVEL is set to 2, all constructs allowed only for level 3 and higher produce warnings.

The 01-level corresponds to the U.S. Government low-level, the 02-level corresponds to low-intermediate, the 03-level corresponds to high-intermediate, and the 04-level corresponds to high-level. The 05-level includes all the extensions to the ANSI standard.

If the FEDLEVEL option is set to 5 and there is a program-name specified in the PROGRAM-ID clause, that name is used as the entry-point-name. Otherwise, the entry-point-name is PROCEDUREDIVISION.

A FEDLEVEL option of 4 might be helpful to reduce the number of syntax errors produced in an environment where WARNFATAL is set. For example, if both WARNFATAL and GROUPMOVEWARN options are set, a FEDLEVEL option of 4 could eliminate some syntax error messages that are unrelated to examining the program for violations of MOVE statement rules. If FEDLEVEL is left to default or set to a low value in this case, a large number of syntax errors could be produced that would make it more difficult to locate the particular statements that violate the MOVE rules.

FREE

```
FREE
```

(Type: Boolean, Default: TRUE for compilations originated through CANDE; otherwise, FALSE)

The FREE option removes most of the margin restrictions required by COBOL. This option must be FALSE when you are using a debugging line in the program. Because the default setting is TRUE for compilations started from CANDE, you must remember to explicitly set the option to FALSE.

For a description of debugging lines, refer to Section 11, “Debugging.”

GLOBAL

```
GLOBAL
```

(Type: Boolean, Default: FALSE)

The GLOBAL option causes all data items in the WORKING-STORAGE SECTION and the FILE SECTION to be global except those specifically declared LOCAL or OWN. The GLOBAL option only affects the data descriptions in the FILE SECTION, not the files

themselves. The GLOBAL option has no effect on entries in the ENVIRONMENT DIVISION or the COMMUNICATION SECTION.

If the compilation is an 02-level, the GLOBAL option is ignored. The OWN and GLOBAL options cannot both be TRUE.

As the compiler parses each data declaration, it checks the setting of the GLOBAL option when the current syntactical element is the data-name in the declaration and the next syntactical element has been prepared for future analysis. If an ending period follows the data-name and the next syntactical element is a CCR that changes the setting of the GLOBAL option, the new GLOBAL option setting affects the data item.

To prevent the new GLOBAL option setting from affecting the data item, redundant clauses or semicolons (;) can be inserted in the data declaration that precedes the CCR so that at least one syntactical element exists between the data-name and the ending period of the data declaration.

It is recommended that you do not change the GLOBAL option within the clauses of a data declaration. Instead, change the GLOBAL option either before the level number or after the ending period of the data declaration.

The VALUE clause cannot be used with the GLOBAL option.

GLOBALTEMP

```
GLOBALTEMP
```

(Type: Boolean, Default: FALSE)

The GLOBALTEMP option causes the temporary arrays generated by the compiler for the processing of some statements to be global. The GLOBALTEMP and OWNTEMP compiler control options cannot both be TRUE. If the GLOBALTEMP option is TRUE in a program compiled at the 02-level, then the temporary arrays for the host program are also shared by all bound procedures in which the GLOBALTEMP option is TRUE.

This option must appear before the first source record and cannot appear on any compiler control record (CCR) after the first source record.

Note: *This option can produce unpredictable errors or results if used in a bound procedure that is then used as an asynchronous process (that is, by an ALGOLPROCESS statement).*

GROUPMOVEWARN

```
GROUPMOVEWARN
```

(Type: Boolean, Default: FALSE)

The GROUPMOVEWARN option issues a warning whenever the compiler encounters a MOVE statement in which one operand is a group item, the other operand is an elementary item, and the usages of the two operands differ. Some compilers and systems generate incorrect results for MOVE statements of this type. The GROUPMOVEWARN option is useful for programs migrating from such systems because it identifies code that is expecting results contrary to the requirements of the standard.

If the ERRORLIST option is TRUE, the compiler produces an ERRORFILE file. The ERRORFILE file contains the warnings produced by the GROUPMOVEWARN option even if there were no actual syntax errors in the program.

The GROUPMOVEWARN option produces warning messages that are issued as “stern" messages. The WARNSUPR option does not suppress these messages. If either the WARNFATAL or STERNFATAL option is set, then the messages are considered syntax errors.

Example

Example 17–3 illustrates the situation that the GROUPMOVEWARN option is intended to help diagnose.

```
*  The record description
*
 01  THE-RECORD.
     03  A-PACKED-DATE  PIC 9(6)  COMP VALUE 031990.
     03  A-GROUP-ITEM.
         05  DATE-YEAR  PIC 99.
         05  DATE-MONTH PIC 99.
         05  DATE-DAY   PIC 99.
     03  AN-ELEMENTARY-ITEMS REDEFINES A-GROUP-ITEM  PIC 9(6).
     :
 PROCEDURE DIVISION
     :
*
*  After execution of the next statement, the correct value for
*  THE-RECORD is @031990031990404040@. Since a move operation
*  involving group items is an alphanumeric-to-alphanumeric move,
*  the move operation transfers 3 bytes of data from A-PACKED-DATE
*  to the first 3 bytes of A-GROUP-ITEM, and fills the remainder of
*  of A-GROUP-ITEM with spaces.
*
     MOVE A-PACKED-DATE TO A-GROUP-ITEM.
*
*  After execution of the next statement, the correct value for
*  THE-RECORD is @031990F0F3F1F9F9F0@. This is an elementary numeric
*  move operation, and as such, the data is transformed from
*  computational to display form in the process.
*
```

```
MOVE A-PACKED-DATE TO AN-ELEMENTARY-ITEM.
:
```

Example 17–3. Understanding the GROUPMOVEWARN Option

INFO

```
INFO
____
```

(Type: Boolean, Default: FALSE)

This option is for Unisys internal use only.

LABELSINTABLE

```
LABELSINTABLE
_____________
```

(Type: Boolean, Default: FALSE)

The LABELSINTABLE option causes the compiler to allocate an execution-time table for the Program Control Words (PCWs) associated with paragraph-names and section-names instead of placing these PCWs directly on the hardware stack. Correspondingly, when the label is referenced, these PCWs are retrieved from the execution-time table instead of directly from the hardware stack. This option has the effect of significantly reducing the stack requirements of programs that have a large number of paragraph-names and/or section-names.

This option should be used only if a program fails to compile with a syntax error indicating that the maximum hardware stack size for a program has been exceeded. The execution-time performance of GO TO and PERFORM statements, as well as that of transitions from one code segment to another, is necessarily inferior to that obtained when this option is reset.

The compiler places the PCW for a label in the table or retrieves it from the table only under the following circumstances (some of which overlap):

- The label is not the target of an ALTER verb.
- The label is not, and is not contained within, a procedure declared USE AS INTERRUPT.
- The label is not, and is not contained within, a procedure declared USE FOR DEBUGGING.
- The label is not a SECTION within DECLARATIVES.
- Neither $TADS nor $OPTIMIZE is set.

- Neither $MAKEHOST nor $SEPCOMP is set.
- The label is not explicitly or implicitly referenced in a USE FOR DEBUGGING procedure in DECLARATIVES.
- The label is not a SORT INPUT or OUTPUT procedure, nor is it contained within such a procedure.

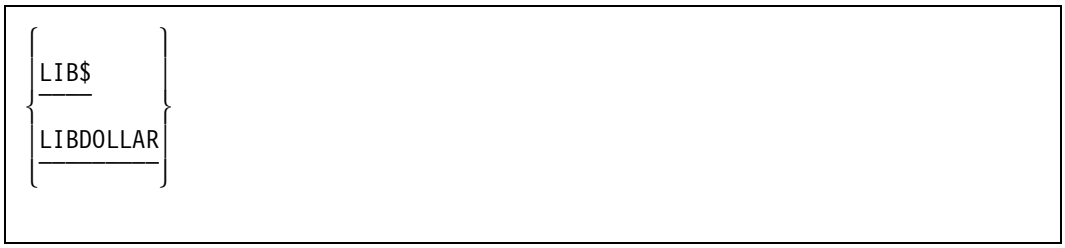
If no labels in the program are found to be eligible to be placed in the table when this option is set, a warning is issued by the compiler.

LEVEL

(Type: Value, Default: 2)

The LEVEL option controls the lexicographical level at which the compilation is to occur. The integer must be in the range from 2 through 14. The LEVEL option must appear before the IDENTIFICATION DIVISION.

LIB$ OR LIBDOLLAR



(Type: Boolean, Default: FALSE)

The LIB$ option causes a CCR that occurs as part of a COBOL library file to be processed. When this option is FALSE, CCRs in a COBOL library file are ignored.

This option is treated as an error if it occurs on a CCR in a COBOL library file.

LIB$ and LIBDOLLAR are synonymous.

LIBRARYLOCK

(Type: Boolean, Default: FALSE)

When TRUE, the LIBRARYLOCK option provides locking code to maintain data integrity for private libraries.

For More Information

- For information on specifying the way a program is shared when it is called as a library, refer to "SHARING" in this section.
- For a description of the compiler control options that determine the way a library is used, refer to Section 15, "Libraries."

LINEINFO

```
LINEINFO
```

(Type: Boolean, Default: TRUE for compilations originated through CANDE; otherwise, FALSE)

The LINEINFO option saves source-language sequence numbers in the code file so that if the object program abnormally ends, you can investigate the problem by program sequence number rather than by code address.

LIST

```
LIST
```

(Type: Boolean, Default: FALSE for compilations originated through CANDE; otherwise, TRUE)

The LIST option produces

- A source-language listing
- A compilation summary
- An error message listing
- A warning message listing

$LIST does not take effect until after the Compiler Control Image containing it has been processed completely. Consequently, Unisys recommends that listing-related Compiler Control Options be contained in one or more separate Compiler Control

Images, rather than included in the same Compiler Control Images as other options against which the compiler might issue syntax warnings.

For example, rather than

```
000100$ LIST MERGE "SOMEFILE" TADS OPTIMIZE
```

Unisys recommends

```
000100$ LIST MERGE "SOMEFILE"
000102$ TADS OPTIMIZE
```

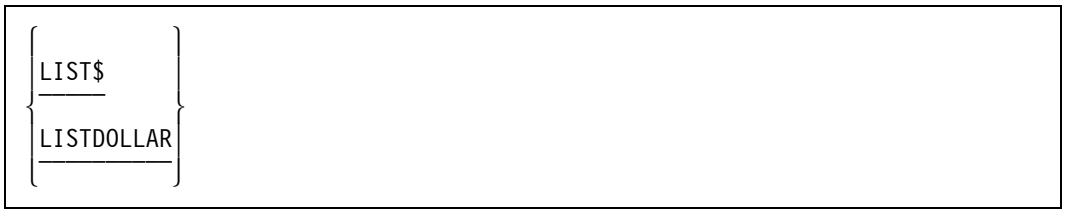
or

```
000100$ LIST
000102$ TADS OPTIMIZE
000104$ MERGE "SOMEFILE"
```

This ensures that the following appropriate syntax warnings are reflected in the compilation listing:

- TADS and OPTIMIZE are mutually exclusive.
- $OPTIMIZE was not set as directed.

LIST$ or LISTDOLLAR



(Type: Boolean, Default: FALSE)

The LIST$ option lists all temporary CCRs encountered during the compilation while the LIST option is set. CCRs encountered while the LIST option is reset are not listed regardless of the setting of the LISTDOLLAR option.

LIST$ and LISTDOLLAR are synonymous.

LISTDELETED

(Type: Boolean, Default: FALSE)

While the LIST option is set, the LISTDELETED option lists that part of the source-language input that was deleted by the DELETE or the VOID option. Deleted records encountered while the LIST option is reset are not listed regardless of the setting of the LISTDELETED option.

LISTOMITTED

```
LISTOMITTED
```

(Type: Boolean, Default: FALSE)

The LISTOMITTED option lists that part of the source-language input that was omitted by the OMIT option while the LIST option is set. Omitted records encountered while the LIST option is reset are not listed regardless of the setting of the LISTOMITTED option.

LISTP

```
LISTP
```

(Type: Boolean, Default: FALSE)

The LISTP option lists source-language records that originate from the primary input file (CARD file).

If the LIST option is TRUE, this option has no effect.

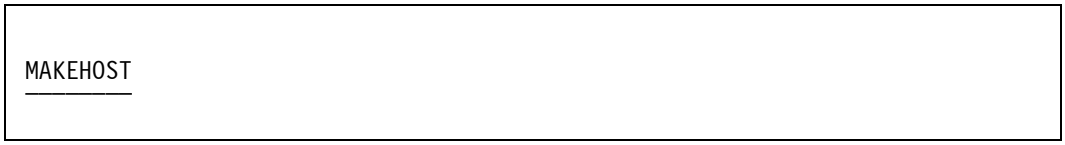
LIST1

```
LIST1
```

(Type: Boolean, Default: FALSE)

The LIST1 option produces a listing during the first pass.

MAKEHOST



(Type: Boolean, Default: FALSE)

The MAKEHOST option creates a host file when compiling a COBOL program. The output host file is an executable code file that is then used as one of the inputs to a SEPCOMP compilation.

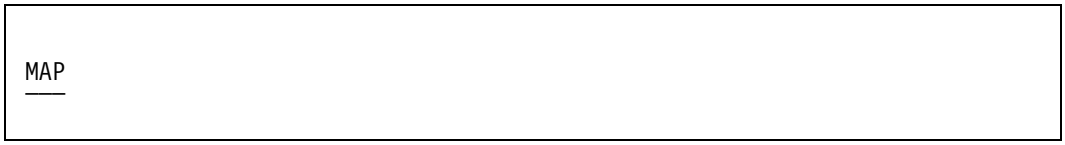
The code segments of a MAKEHOST compilation are different from those of a normal compilation. All MAKEHOST sections are placed into separate code segments, as opposed to putting contiguous sections with the same segment-number in the same code segment, which is done in a normal compilation.

The MAKEHOST option must appear as the first record of the compiler input file.

The MAKEHOST option is incompatible with the MERGE, NEW, OPTIMIZE, and STATISTICS options. If one of these options is used with the MAKEHOST option, then the option is ignored and a warning message is issued.

Note: *The 'MAKEHOST' option will be deimplemented in subsequent releases. Use bound programs instead.*

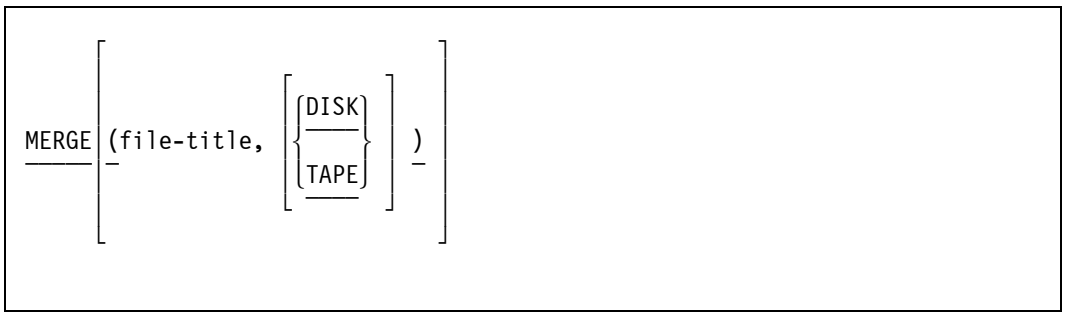
MAP



(Type: Boolean, Default: FALSE)

The MAP option includes information concerning the allocation of variables in the object program as part of the output listing.

MERGE



(Type: Boolean, Default: FALSE)

The MERGE option starts the process that merges the primary source-language records (CARD file) with the secondary source-language input records (SOURCE file) specified by the MERGE parameters.

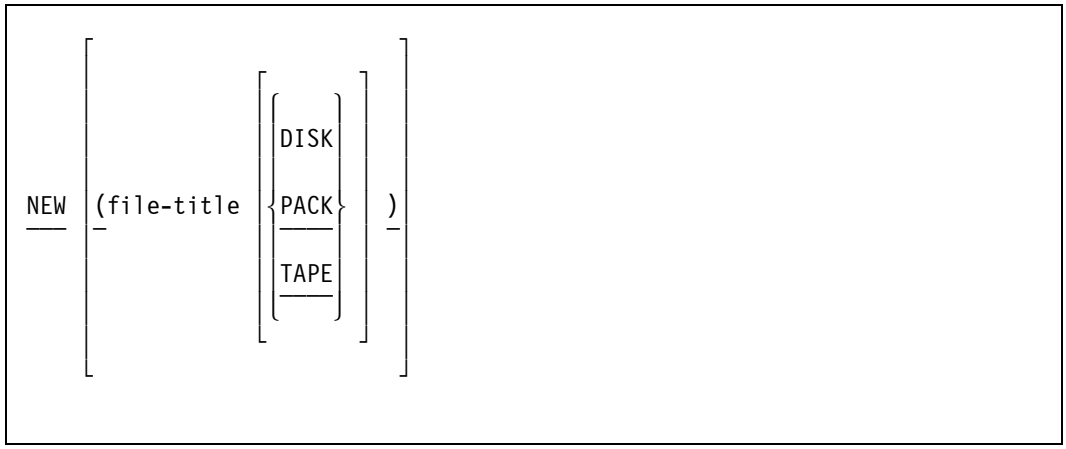
The file title must be a string. If you do not specify a file title, SOURCE is assumed, unless this file title is preempted by a file equation.

If you do not specify a device, DISK is assumed, unless preempted by a file equation.

This option remains TRUE throughout a compilation. If you try to change this option, your attempt is treated as an error and is ignored.

You cannot use the MERGE option with the MAKEHOST and SEPCOMP options. If the MAKEHOST option is TRUE or becomes TRUE, the compiler reads the SOURCE file as the compiler input file, but the compiler issues a warning message and stops reading records from the CARD file. If the SEPCOMP option is TRUE or becomes TRUE, the MERGE option is set to FALSE and the compiler issues a warning message.

NEW



(Type: Boolean, Default: FALSE)

The NEW option creates a new source-language symbolic file (NEWSOURCE file) of all source-language records accepted for compilation.

The file title must be a string. If you do not specify a file title, NEWSOURCE is assumed unless it is preempted by a file equation.

If you do not specify a device, DISK is assumed unless it is preempted by a file equation.

This option starts the writing of all input source-language records accepted for compilation to a new symbolic file (NEWSOURCE file). Source-language records discarded by the DELETE or VOID options are excluded; however, input records omitted by the OMIT option and permanent CCRs are included.

Once set to TRUE, this option remains TRUE throughout a compilation. An attempt to change this option is treated as an error and is ignored.

You cannot use the NEW option with the MAKEHOST and SEPCOMP options. If the MAKEHOST or the SEPCOMP option is TRUE or becomes TRUE, the compiler sets the NEW option to FALSE and issues a warning message.

NEWID

```
NEWID
```

(Type: Boolean, Default: FALSE)

The NEWID option replaces the rightmost eight character positions of the source-language record with the *symbolic-id* literal associated with this option.

If this option is FALSE, the rightmost eight character positions of the source-language record remain unchanged.

For More Information

Refer to "Symbolic ID" in this section for information on this literal.

NORMALMCPWARNING

```
NORMALMCPWARNING = integer
```

(Type: Value; Default: 0)

The NORMALMCPWARNING option allows a programmer to specify that a syntax message retrieved from the MCP can be treated as a "normal" (nonstern) syntax warning instead of a stern warning. A stern warning appears in both the compiler ERRORFILE and LINE files, and cannot be suppressed through the WARNSUPR compiler control option.

The option must be followed by an equal sign and one integer numeric literal. The integer is the number of the message number in the MCP, not in the internal number of the compiler for messages retrieved from the MCP.

Any number of NORMALMCPWARNING options can appear in a given program, each with a single integer argument.

Once a given MCP warning number is selected for the nonstern status, the warning associated with that MCP warning number is treated as a nonstern warning for the remainder of the compilation.

The setting of the NORMALMCPWARNING option has no effect on whether the compiler associates the message number with the generated object code file through the WARNINGS attribute.

An example of the application of this option is the function of the Automatic Insertion Editing mechanism in the PICTURE clause. The use of this mechanism causes a message to be

- Retrieved from the MCP.
- Associated with the object code file through the WARNINGS attribute.
- Included unconditionally in the compiler ERRORFILE listing.
- Included in the LINE file if the LIST Compiler Control Option is set, regardless of whether the WARNSUPR Compiler Control Option is set.

The text of the stern syntax warning generated for this case begins as follows:

```
WARNING 853: THE FOLLOWING MESSAGE WAS RECEIVED FROM THE MCP:
Warning 159: Automatic Insertion Editing ...
```

To make this a normal warning, the appropriate Compiler Control image is set as follows:

```
$NORMALMCPWARNING = 159
```

NOXREFLIST

```
NOXREFLIST
```

(Type: Boolean, Default: FALSE)

When set to TRUE, the NOXREFLIST option prevents the SYSTEM/XREFANALYZER utility from being started by the compiler when cross-reference information is being saved (that is, when either the XREF option or the XREFFILES option is TRUE). Instead, the file XREF/code-file-name, where code-file-name is the name of the object code file generated by the compiler, remains on disk. The SYSTEM/XREFANALYZER utility can be run later by using the file XREF/code-file-name as input. The NOXREFLIST option has no effect if both the XREF option and the XREFFILES option are FALSE.

For more information about cross-reference files, refer to "Using Cross-Reference Files" in this section.

OMIT

(Type: Boolean, Default: FALSE)

The OMIT option causes all source-language records, except other compiler control records (CCRs), to be ignored (but not discarded) for compilation until the option is FALSE.

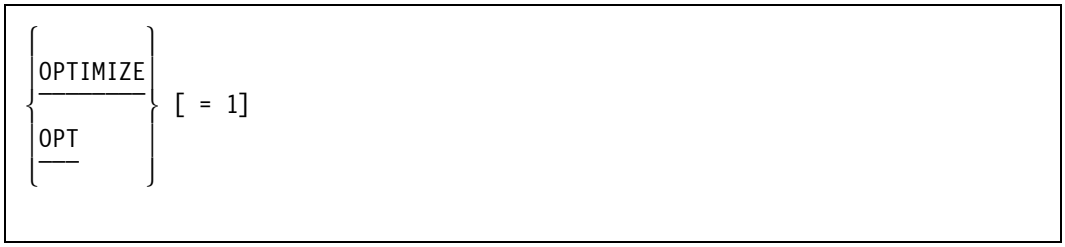
This option can appear on a CCR in either the primary (CARD file) or the secondary (SOURCE file) source-language input.

The omitted source-language records are carried forward to the output symbolic file (NEWSOURCE file) if the NEW option is also TRUE. The records are not listed unless the LISTOMITTED option is TRUE.

CCRs encountered in the source-language input while this option is TRUE are processed in the normal fashion.

For information on listing that part of the source-language input that was omitted by the OMIT option, refer to "LISTOMITTED" in this section.

OPTIMIZE or OPT



(Type: Value, Default: No optimization)

The OPTIMIZE option applies optional optimization functions that are included at appropriate levels during the compilation process.

OPT and OPTIMIZE are synonymous.

The setting of this option cannot be altered after the first source record.

The setting of this option results in the following actions:

- Numerical comparisons involving some unsigned display items are compared as characters, rather than being converted to binary items and then compared. The zone portion of each character is masked with hex letter F so that uninitialized data items compare equal to 0 (zero).
- Simple ADD statements involving some unsigned display items are executed as character additions with all zones masked with hex letter F so that uninitialized data items are considered 0 (zero).
- When variables or expressions are used as subscripts, object code to verify that the subscript is within range at execution time is omitted. The responsibility of ensuring the validity of any given subscript rests with the programmer. System protection against accesses beyond the bounds of the record remains in effect.

  Note that if a numeric literal is used as a subscript, the value of that literal is verified at compilation time, and if it is found to be outside the range of permissible values, a warning is issued and object code is generated unconditionally to force an INVALID INDEX at execution time, whether $OPTIMIZE is set or not.
- The characteristics of the PERFORM statements and ranges of the program are analyzed. If the increase in the amount of object code is not too great, the statements in the PERFORM range are expanded in line.

The use of the USE FOR DEBUGGING statement to monitor items might not always work correctly when the OPTIMIZE option is TRUE. Some items in certain statements might not be monitored correctly (or at all).

You cannot use the OPTIMIZE option with the TADS, MAKEHOST, and SEPCOMP options. If the TADS, the MAKEHOST, or the SEPCOMP option is TRUE or becomes TRUE, the compiler sets the OPTIMIZE option to FALSE and issues a warning message.

OWN

```
OWN
```

(Type: Boolean, Default: FALSE)

The OWN option causes all WORKING-STORAGE SECTION data items to assume the declaration OWN except those for which LOCAL or GLOBAL is explicitly declared. The OWN option is ignored if the compilation is at the 02-level, OWN and GLOBAL options cannot both be TRUE.

As the compiler parses each data declaration, it checks the setting of the OWN option when the current syntactical element is the data-name in the declaration and the next syntactical element has been prepared for future analysis. If an ending period follows the data-name and the next syntactical element is a CCR that changes the setting of the OWN option, the new OWN option setting affects the data item.

To prevent the new OWN option setting from affecting the data item, redundant clauses or semicolons (;) can be inserted in the data declaration that precedes the CCR so that at least one syntactical element exists between the data-name and the ending period of the data declaration.

It is recommended that you do not change the OWN option within the clauses of a data declaration. Instead, change the OWN option either before the level number or after the ending period of the data declaration.

The VALUE clause cannot be used with the OWN option.

OWNTEMP

```
OWNTEMP
```

(Type: Boolean, Default: FALSE)

The OWNTEMP option causes the temporary arrays generated by the compiler for the processing of some statements to be OWN. The OWNTEMP and GLOBALTEMP compiler control options cannot both be TRUE. The OWNTEMP option is ignored in a program compiled at level 2.

This option must appear before the first source record and cannot appear on any compiler control record (CCR) after the first source record.

Note: *This option can produce unpredictable errors or results if used in a bound procedure that is then used as an asynchronous process (that is, by an ALGOL* PROCESS *statement).*

PAGE

```
PAGE
```

(Type: Immediate)

The PAGE option ejects a page on the output listing.

If the LIST option is FALSE, the PAGE option is ignored.

If the OMIT option is TRUE and the LISTOMITTED option is FALSE, the PAGE option is ignored.

SEPCOMP

```
SEPCOMP [(file-title)]
```

(Type: Boolean, Default: FALSE)

The SEPCOMP option separately recompiles selected areas of a previously compiled COBOL program. To do a SEPCOMP compilation, you must first compile the program with the compiler control option MAKEHOST compilation set to TRUE. This step creates the host file. The host file is an executable code file that contains additional information necessary for a SEPCOMP compilation. Given only the name of the host file, the name of the original source, and the patches to change the source, the compiler is able to separately compile the patched areas, resulting in an abbreviated compilation. Only the affected areas of the program are actually compiled.

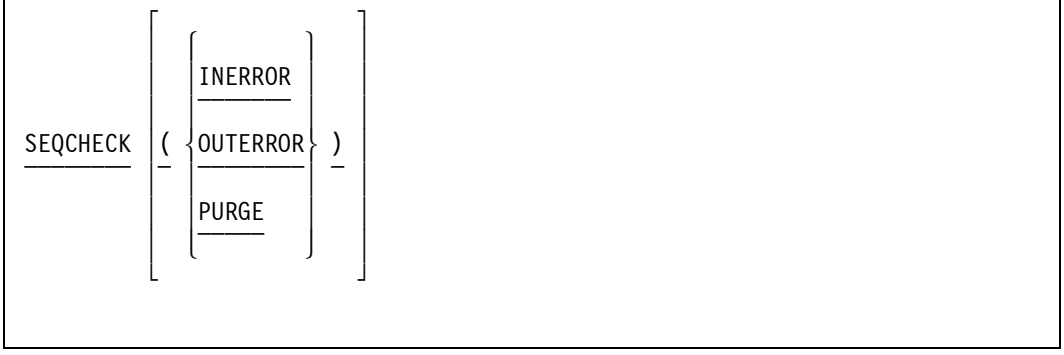
The file-title must be a quoted string containing a file title. If you do not specify a file title, HOST is assumed unless it is preempted by a file equation.

The SEPCOMP option can be specified only in the compiler file CARD.

For specific information on using the SEPCOMP option, refer to "Performing a Separate Compilation" in this section.

Note: *The 'SEPCOMP' option will be deimplemented in subsequent releases. Use bound programs instead.*

SEQCHECK



(Type: Boolean, Default: FALSE)

This option checks the sequence numbers of the primary, secondary, and/or new symbolic records to ensure that the numbers are in ascending order in a manner specified by the parameters.

This option then verifies that the sequence number of the current source-language record acquired from the primary input (CARD file) or the secondary input (SOURCE file) or directed to the output (NEWSOURCE file) is greater than the sequence number of the previous source-language record acquired from or directed to the same file.

If the sequence numbers are not in ascending order, a sequencing warning is produced.

For the primary input only, if the previous sequence number was made up of spaces and if the current sequence number is also made up of spaces, then a sequencing warning is not produced.

The comparison of sequence numbers is done to allow nonnumeric sequence number values.

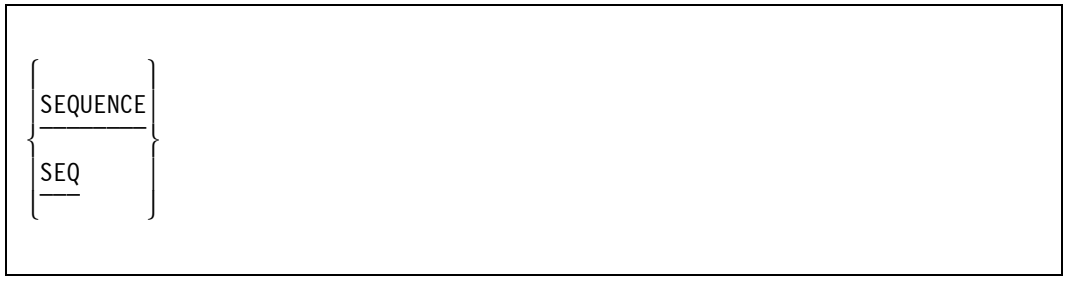
The specification of the parameter INERROR directs the compiler to treat sequencing warnings on the primary or secondary inputs as syntax errors.

The specification of the parameter OUTERROR directs the compiler to treat sequencing warnings on the output symbolic file (NEWSOURCE file) as syntax errors.

The specification of the parameter PURGE directs the compiler to purge the output symbolic file (which is not locked or entered into the directory) if sequencing warnings occur that refer to the output symbolic file.

The warning messages produced when SEQCHECK is set can be treated as stern warning messages under some circumstances. The WARNSUPR option does not suppress stern warnings. These warnings are considered syntax errors when either the STERNFATAL or WARNFATAL option is set.

SEQUENCE or SEQ



(Type: Boolean, Default: FALSE)

The SEQUENCE option assigns new sequence numbers to the source-language records accepted for compilation.

SEQUENCE and SEQ are synonymous.

This option affects only input source-language records encountered by the compiler following the merging process, including records that were omitted because of the OMIT option.

The SEQUENCE option assigns the current Sequence Base option value to the current source-language record and increments the Sequence Base option value by the Sequence Increment option value. Sequencing occurs before the production of a new symbolic file record and before the source-language record is listed.

If the result base exceeds 999999 when the compiler increments the Sequence Base option value by the Sequence Increment option value, then the SEQUENCE option is disabled and a sequencing error is produced.

Sequence Base

```
integer
```

(Type: Value, Default: 10)

The Sequence Base option stores the specified integer value as the Sequence Base associated with the SEQUENCE option.

This option must appear on a CCR so that it is dissociated from any other option that requires an integer as a parameter or a value.

The maximum value of the integer is 999999.

This option can be specified independently of the SEQUENCE option.

Sequence Increment

```
+ integer
```

(Type: Value, Default: 10)

The Sequence Increment option stores the specified integer value as the Sequence Increment associated with the SEQUENCE option.

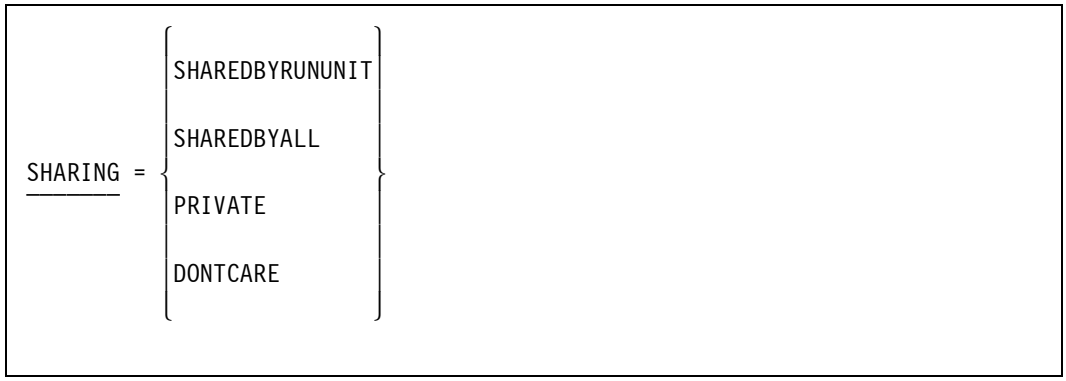
This option must appear on a CCR so that it is dissociated from any other option requiring a positive integer as a parameter or a value.

The maximum value of the integer is 999999.

The plus sign (+) must immediately precede the integer.

This option can be specified independently of the SEQUENCE option.

SHARING



(Type: Value, Default: SHAREDBYRUNUNIT)

The SHARING option specifies the way in which the program is shared when called as a library. Table 17–7 shows the result of settings each SHARING option value.

Table 17–7. Effects of the SHARING Option

Value	Effect
SHAREDBYRUNUNIT	All users of this run unit (the program and any libraries that are called) share the same instance of the library. This definition of run unit should not be confused with the ANSI COBOL74 and COBOL85 definition of run unit.
SHAREDBYALL	All simultaneous users share the same instance of the library.
PRIVATE	A separate instance of the library is started for each user. If the library can be called by programs other than COBOL74 programs, consider setting the LIBRARYLOCK option to TRUE.
DONTCARE	The sharing is determined by the operating system.

If the library is called by a COBOL74 program, the library services only one user at a time, regardless of the setting of the SHARING option. In a complex environment where multiple libraries are linked together, a COBOL74 library with the SHARING option set to PRIVATE should also have the LIBRARYLOCK option set to TRUE to ensure data integrity.

For More Information

- For information about the state of a library, refer to "Effect of Library Initial State on a CANCEL Statement" in Section 15, "Libraries."

- For information on locking and sharing libraries, refer to "LIBRARYLOCK" and "SHARING" earlier in this section.

SPEC

```
SPEC
____
```

(Type: Boolean, Default: FALSE)

The SPEC option suppresses printing of warning messages, sequence error messages, the expansion of the Enterprise Database Server INVOKE statement, and the list of elementary items in a CORRESPONDING option.

STATISTICS

```
STATISTICS
__________
```

(Type: Boolean, Default: FALSE)

The STATISTICS option compiles the source language input so that statistical data about the object program is written when the object program is executed. Use of this option is designed for a development environment, not a production environment, because extra processor overhead is generated.

The STATISTICS option is incompatible with the MAKEHOST, SEPCOMP, and BINDINFO options. In addition, the STATISTICS option is not valid if the LEVEL option has a value greater than 2.

If the MAKEHOST or the SEPCOMP option is TRUE or becomes TRUE, then the STATISTICS option is set to FALSE and a warning message is issued. If an attempt is made to set both the STATISTICS and BINDINFO options to TRUE, the first option that became TRUE is unaffected, the second option is set to FALSE, and a warning message is issued.

The use of the STATISTICS option is incompatible with programs that use the USE EXTERNAL PROCEDURE statement. If STATISTICS is TRUE when the USE EXTERNAL PROCEDURE is encountered during compilation, then a warning message is issued and the STATISTICS option is set to FALSE.

STERNFATAL

```
STERNFATAL
__________
```

(Type: Boolean, Default: FALSE)

The STERNFATAL option causes warning messages marked as "stern" during the compilation process to be treated as syntax errors.

An example of the application of this option can be drawn from the function of the GROUPMOVEWARN option. This option causes the production of stern warnings. Setting STERNFATAL and GROUPMOVEWARN in most cases allows the compiler to detect the fact that the compiler might have generated incorrect code in prior releases, and to treat such violations as syntax errors; it does not producing syntax errors or less serious (non-stern) warnings that might occur in the program as would be the case if WARNFATAL and GROUPMOVEWARN were set.

Note: *When the STERNFATAL option is set, the COBOL74 compiler issues warnings to comply with the requirements of a formal deimplementation process. These warnings are considered stern warnings and are treated as syntax errors.*

STERNMULTDEST

(Type Boolean, Default: FALSE)

The STERNMULTDEST option causes the compiler to treat warnings as "stern" warnings if they alert the user about possible precision loss on multiple-destination ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements.

STRICTPICTURE

(Type: Boolean, Default: FALSE)

The STRICTPICTURE option directs the compiler to issue a syntax error for any PICTURE string that does not conform to ANSI X3.23-1974 COBOL rules.

When the STRICTPICTURE option is reset, Manual Insertion Editing is allowed.

SUMMARY

```
SUMMARY
```

(Type: Boolean, Default: FALSE)

The SUMMARY option produces a listing of summary information about the compilation. The summary produced by this option is the same as that produced when the LIST option is set to TRUE, with two additions. These additions include

- A listing of the sequence number of each COPY statement in the program and the name of the file used by that COPY statement.
- A summary of the usage levels of several critical tables internal to the compiler, relative to the capacity of those tables.

Symbolic ID

```
" alphanumeric literal "
```

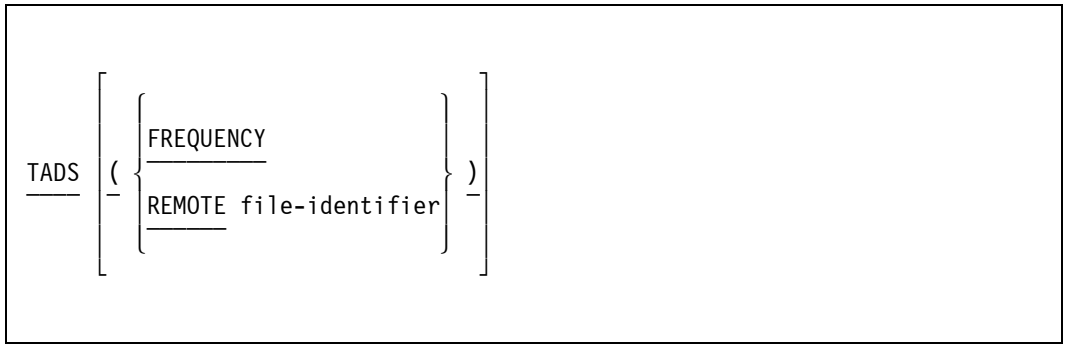
(Type: Value, Default: None)

The Symbolic ID option stores the specified alphanumeric literal as the Symbolic ID option value associated with the NEWID option.

The alphanumeric literal cannot be longer than 8 characters.

This option must appear on a CCR so that it is dissociated from any other option that might require a quoted string as a parameter.

TADS



(Type: Boolean, Default: FALSE)

When the TADS option is TRUE, the Test and Debug System (TADS) generates special debugging code and tables as part of the object program. The tables support the symbolic debugging environment of TADS.

You must place the TADS option before the first record that is not a CCR in the source program.

The FREQUENCY option generates additional code and tables for test coverage and frequency analysis. You must specify the FREQUENCY option if you want to use the TADS commands FREQUENCY or COVERAGE to determine the number of times individual statements have been executed or to determine which statements have not been executed.

The REMOTE option allows TADS to share a REMOTE file with the program being tested. Sharing a file might be necessary because only one REMOTE input file can be open for each station. The file must have been assigned to the REMOTE file identifier, and must be opened I-O. The record size must not be less than 72.

After the compiler reaches the PROCEDURE DIVISION header, if the source file contains sequence numbers, the TADS option performs some of the sequence number checking associated with the SEQCHECK INERROR compiler control option.

The sequence numbers of source records must be in ascending order. If this is not the case, the compiler issues a syntax error. This rule does not apply to CCRs.

A CCR can have the same sequence number as the source record that immediately precedes or follows it. The compiler issues a syntax error if the sequence number of a CCR is less than that of the preceding source record that is not a CCR.

You cannot use the TADS option with the OPTIMIZE, MAKEHOST, SEPCOMP and BINDINFO options.

Programs compiled with the TADS option set to TRUE cannot be used as input to the Binder.

For more information on debugging COBOL74 programs, refer to the *COBOL ANSI-74 Test and Debug System (TADS) Programming Guide.*

TARGET

```
TARGET = target-1 [ ( target-2 [ ,target-3 . . .] ) ]
```

(Type: Enumerated, Default: Installation defined)

This option designates a specific computer system or group of systems as the target for which the generated object code is to be optimized. This option can be used to specify all machines on which the code file needs to run.

TARGET must appear in the source before the IDENTIFICATION DIVISION of the program.

Specification of a secondary target is optional. If specified, a secondary target must be enclosed in parentheses. If more than one secondary target is specified, then the additional targets must be separated from each other by a comma and the entire list must be enclosed in parentheses.

If a secondary target is specified, the compiler does not generate any operators that are valid for the system or systems identified by the primary target but invalid for the system or systems identified by the secondary target.

See the COMPILERTARGET system command in the *System Commands Operations Reference Manual* for a complete list of the target values that are allowed.

Examples

```
TARGET=THIS
```

The compiler optimizes the object code file for the system on which it is compiled.

```
TARGET=THIS (ALL)
```

The compiler optimizes the object code file for the system on which it is compiled, but it does not generate any operators that are invalid for other machines.

TEMPORARY

```
TEMPORARY
_________
```

(Type: Boolean, Default: FALSE)

The TEMPORARY option causes the object program, when called as a library, to function as a temporary library. This option is FALSE by default, causing the program to function as a permanent library.

This option is dictated by the value of the SHARING option. A library is made permanent only if the SHARING option has been specified as SHAREDBYALL or DONTCARE.

For More Information

- For information on sharing a library, refer to "SHARING" in this section.
- For information about temporary and permanent libraries, refer to "TEMPORARY" in Section 15, "Libraries."

USER

```
identifier
```

(Type: Boolean, Default: FALSE)

If an identifier on a compiler control record (CCR) is not recognized as one of the standard options, it is considered a USER option. This option must be an alphanumeric identifier.

A user option can be manipulated exactly like any other Boolean option; that is, you can set, reset, or pop it. In addition, the USER option can be used as a variable in option expressions to assign values to standard Boolean options or to other user options.

VOID

```
VOID sequence-number
```

(Type: Immediate)

The VOID option discards all input records, except other compiler control records (CCRs) in the secondary source-language input file (SOURCE file), until the secondary input sequence-number exceeds the specified sequence-number.

The sequence-number must be an unsigned integer. The sequence-number cannot be less than the sequence-number of the CCR on which the VOID option is specified. The maximum value of sequence-number is 999999.

This option can appear only on a CCR in the primary source-language input (CARD file).

This option is ignored if the MERGE option is FALSE.

The source-language records that are discarded are not carried forward to the output symbolic file (NEWSOURCE file).

The source-language records that are discarded are not listed unless the LISTDELETED option is set.

For information on listing that part of the source language that was deleted by the DELETE or VOID options, refer to “LISTDELETED” earlier in this section.

WARNFATAL

```
WARNFATAL
```

(Type: Boolean, Default: FALSE)

The WARNFATAL option causes any warning messages issued by the compiler (whether stern warnings or not) to be treated as syntax errors.

WARNSUPR

```
WARNSUPR
```

(Type: Boolean, Default: FALSE)

The WARNSUPR option suppresses the printing of warning messages only.

XDECS

```
XDECS
```

(Type: Boolean, Default: TRUE)

When the compiler is saving cross-reference information because the XREF option or the XREFFILES option is TRUE, only user-defined words declared while the XDECS option is TRUE are included in the cross-reference information. If both the XREF option and the XREFFILES option are FALSE, the XDECS option has no effect.

For more information about cross-reference files, refer to "Using Cross-Reference Files" earlier in this section.

XREF

```
XREF
```

(Type: Boolean, Default: FALSE)

The XREF option causes cross-reference information to be saved by the compiler.

If the XREF option is TRUE, a listing is produced. If the XREFFILES option is TRUE, a pair of disk files is produced. These files are titled *XREFFILES/code-file-name/XREFS* and *XREFFILES/code-file-name/XDECS*. If both the XREF option and the XREFFILES option are TRUE, both a listing and the disk files are produced.

If, in addition, USERBACKUPNAME is set for the compiler's LINE file, and the compiler is calling XREFANALYZER to generate a cross-reference listing, the FILENAME of the LINE file used by XREFANALYZER is the FILENAME of the compiler's LINE file with the node XREFLIST appended.

The XREF option can be used anywhere in the source, but the setting of the option at the end of the IDENTIFICATION DIVISION is effective for the entire compilation. Any set, reset, or pop of this option after the end of the IDENTIFICATION DIVISION is ignored.

For more information about cross-reference files, refer to "Using Cross-Reference Files" earlier in this section.

XREFFILES

```
XREFFILES
```

(Type: Boolean, Default: FALSE)

When TRUE, the XREFFILES option causes cross-reference information to be saved by the compiler and causes the SYSTEM/XREFANALYZER utility, if it is started by the compiler, to produce files that can be used by the Editor and the SYSTEM/INTERACTIVEXREF utility. These files have the titles *XREFFILES/code-file-name/DECS* and *XREFFILES/code-file-name/REFS*, where the code-file-name is the name of the object code file that the compiler is generating.

This option can be used anywhere in the source, but the setting of the option at the end of the IDENTIFICATION DIVISION is effective for the entire compilation. Any set, reset, or pop of this option after the end of the IDENTIFICATION DIVISION is ignored.

For more information about cross-reference files, refer to "Using Cross-Reference Files" earlier in this section.

XREFS

```
XREFS
```

(Type: Boolean, Default: FALSE)

When the compiler is saving cross-reference information because the XREF option or the XREFFILES option is TRUE, only references that are found while the XREFS option is TRUE are included in the cross-reference information. References to user-defined words that are declared when the XDECS option is FALSE are not cross-referenced. If both the XREF option and the XREFFILES option are FALSE, then the XREFS option has no effect.

The files produced can be used by the SYSTEM/INTERACTIVEXREF utility or by the Editor. The titles of the files produced are *XREFFILES/code-file-name/DECS* and *XREFFILES/code-file-name/REFS*, where the name of the code file is produced by the compiler.

Generation of the interactive XREFFILES file is inhibited if the NOXREFLIST option is TRUE.

For more information on cross-reference files, refer to “Using Cross-Reference Files” earlier in this section.

Appendix A
General Format Notation

General format notations are syntax diagrams that show the rules for creating COBOL74 code from words and symbols. These format notations visually represent the valid combinations of words and symbols that make COBOL74 statements or clauses.

To familiarize you with format notations, this appendix describes the components of the general format notation and provides examples of using format notations to create COBOL74 entries. You can refer to the examples at the back of this appendix as you read the explanation of each component of the general format notation.

All the format notations, whether simple or complex, follow the same basic rules. Once you understand them, the format notations serve as quick references to the syntax for statements and clauses.

The general format notation shows you the following:

- Required words and optional keywords
- Words you must provide
- The order in which the items must appear
- Entries that can be repeated
- Required and optional punctuation

When more than one format notation appears for a statement or clause, each format describes the valid combinations of COBOL74 code for a particular application. For example, there are three separate formats for three separate types of addition. The formats are numbered and labeled for ease of reference.

You can construct COBOL74 statements and clauses from the components shown in the following list. Some components represent words that you use in your code. Other components symbolize the rules for constructing your code. For example, you include uppercase underlined words in your code. You do not include the braces and brackets in your code. Instead, these symbols indicate alternatives and options for the words they enclose.

- Uppercase words
- Lowercase words
- Level numbers
- Braces ({})
- Brackets ([])
- Ellipses (...)

- Separators
- Special characters

You must code the words in the order in which they appear in the general format. The words can appear in a different order only if it is explicitly stated in the explanation of the format that you can do so.

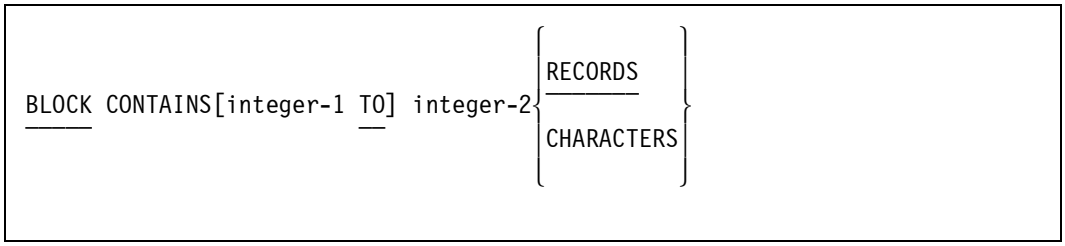
Table A–1 shows each component of the format notation and explains its meaning.

Table A–1. General Format Notation Components

Components	Explanation
Underlined uppercase words	Identify required words that you must include in the statement or clause.
Not underlined uppercase keywords	Show optional keywords that you can include or omit without affecting the meaning of the statement or clause. An optional word enclosed in braces indicates a default option.
Lowercase words	Indicate words that you must supply. If you see a number appended to a lowercase word, that lowercase word represents a particular occurrence of that word.
Level numbers	Describe numbers that you must provide to show the organization of your data.
{ } (Braces)	Enclose a group of phrases stacked vertically from which you must pick only one. If braces enclose a single phrase, you can repeat the phrase. If braces enclose multiple phrases, you must pick one phrase from the list of alternatives.
[ ] (Brackets)	Enclose a group of phrases stacked vertically that are optional. If brackets enclose a single option, you can omit that option. If brackets enclose multiple options, you can omit all the options or you can pick one option from the list of options. Phrases with phrases are allowed. Nested phrases appear enclosed in braces or brackets within another phrase enclosed in braces or brackets.
... (Ellipses)	Represent the point at which you can repeat a part of the format. Ellipses can appear only immediately to the right of a right bracket or a right brace. The part of the format that you can repeat is the part enclosed within the preceding brackets or braces.
Separators	Indicate punctuation in a sentence. Table 2-2 lists all the separators. Commas and semicolons are optional. You can use separators between statements if you want. (You can interchange the comma, semicolon, and space characters). Periods are required. You must end with a period the paragraphs within the IDENTIFICATION and PROCEDURE DIVISIONS and entries within the ENVIRONMENT and DATA DIVISIONS.
Special characters	Indicate arithmetic operations or relation conditions. Special characters are not underlined in the general format notation. However, if a special character appears, it is required. The special characters include the symbols for addition, subtraction, multiplication, division, and exponentiation (+, –, *, /, **) and the symbols for less than, equal to, and greater than (<, =, >).

Example 1

The following example is the general format notation for the BLOCK clause:



Explanation

This example illustrates the following:

- BLOCK is a word that you must enter.
- CONTAINS is an optional keyword for improving readability.
- The *integer-1 TO* phrase is optional. If it is used, TO is required.
- Integer-2 is a required entry. You must substitute a number for integer-2.
- CHARACTERS is the default phrase. If you do not enter either RECORDS or CHARACTERS, the compiler assumes CHARACTERS. However, you can enter CHARACTERS to improve the readability of your code. If you do not want characters and do want records, then you must enter RECORDS.

Using the preceding format notation, you can code any of the following lines. Notice that all three lines of code are equivalent to the compiler, but the full notation is clearer to an outside reader.

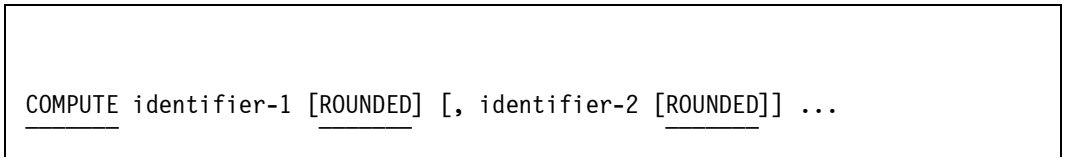
```
BLOCK CONTAINS 400 TO 500 CHARACTERS
BLOCK CONTAINS 400 TO 500
BLOCK 400 TO 500
```

You can also code the following line. In this instance, you can choose not to use the *integer-1 TO* option and you can choose RECORDS.

```
BLOCK CONTAINS 15 RECORDS
```

Example 2

The following example is the general format notation for the COMPUTE statement:



```
= arithmetic-expression [; ON SIZE ERROR imperative-statement].
```

Explanation

This example illustrates the following:

- COMPUTE is a word that you must enter.
- Identifier-1 and arithmetic-expression are entries that you must provide.
- ROUNDED is an option. If you choose to use it, your results are arithmetically rounded.
- , *identifier-2* is an option. It can be repeated with or without the ROUNDED option.
- The equals sign and period are required.
- The comma and semicolon are optional.
- *; ON SIZE ERROR imperative-statement* is an option. If you choose to use this option, the phrase, ON is optional, but SIZE, ERROR, and imperative-statement are required.

Two valid statements that you could form are the following:

```
COMPUTE fruit-inventory = apples + oranges + bananas.
COMPUTE ws-fruit, print-fruit ROUNDED = fruit-inventory + order-size.
```

Appendix B
Reserved Words and Keywords

This section describes and lists the reserved words and the keywords.

Reserved Words

A reserved word can never be declared as an identifier. Its predefined meaning cannot be changed.

Following is the list of reserved words. Words that are reserved but are described in Volume 2 of this manual are also included.

Some of the reserved words in this list represent features that are extensions to ANSI X3.23-1974 COBOL. The reserved words that are not documented in Volume 1 are covered in Volume 2 (for example, BEGIN-TRANSACTION, END-TRANSACTION, FIND and STORE).

A

ABORT-TRANSACTION	ACCEPT	ACCESS
ADD	ADVANCING	AFTER
ALL	ALLOW	ALPHABETIC
ALSO	ALTER	ALTERNATE
AND	ARE	AREA
AREAS	ASCENDING	ASSIGN
AT	ATTACH	AUDIT
AUTHOR	AWAIT-OPEN	

B

BEFORE	BEGIN-TRANSACTION	BEGINNING
BINARY	BLANK	BLOCK
BOTTOM	BY	

C

CALL
CANCEL
CAUSE
CD
CF
CH
CHANGE
CHARACTER
CHARACTERS
CLOCK-UNITS
CLOSE
CMP
COBOL
CODE
CODE-SET
COLLATING
COLUMN
COMMA
COMMUNICATION
COMP
COMPUTATIONAL
COMPUTE
CONFIGURATION
CONTAINS
CONTENT
CONTINUE
CONTROL
CONTROL-POINT
CONTROLS
COPY
COPY-NUMBER
CORR
CORRESPONDING
COUNT
CP
CREATE
CRUNCH
CURRENCY
CURRENT
CREATE
CRUNCH
CURRENCY

D

DATA
DATA-BASE
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DB
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
DETACH
DETAIL
DICTIONARY
DISABLE
DISALLOW
DISK
DISPLAY
DIVIDE
DIVISION
DMERROR
DMSTATUS
DMSTRUCTURE
DMUPDATECOUNT
DOUBLE
DOWN
DUPLICATES
DYNAMIC

E

EGI
ELSE
EMI
ENABLE
END
END-OF-PAGE
END-TRANSACTION
ENDING
ENVIRONMENT
EOP
EQUAL
ERROR
ESI
EVENT
EVERY
EXCEPTION
EXCLUSIVE
EXECUTE
EXIT
EXTEND
EXTERNAL

F

FALSE
FD
FIELD
FILE
FILE-CONTROL
FILLER
FINAL
FIND
FIRST
FOOTING
FOR
FORM
FORM-KEY
FREE
FROM
FUNCTION

G

GENERATE
GIVING
GO
GREATER
GROUP

H

HEADING
HERE
HIGH-VALUE
HIGH-VALUES

I

I-O
I-O-CONTROL
ID
IDENTIFICATION
IF
IN
INDEX
INDEXED
INDICATE
INITIAL
INITIALIZE
INITIATE
INPUT
INPUT-OUTPUT
INQUIRY
INSERT
INSPECT
INSTALLATION
INTERRUPT
INTO
INVALID
INVOKE
IS

J

JUST
JUSTIFIED

K

KANJI
KEY

L

LABEL
LAST
LD
LEADING
LEFT
LENGTH
LESS
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCAL
LOCAL-STORAGE
LOCK
LOCKED
LOW-VALUE
LOW-VALUES
LOWER-BOUND
LOWER-BOUNDS

M

MEMORY
MERGE
MESSAGE
MID-TRANSACTION
MODE
MODULES
MOVE
MULTIPLE
MULTIPLY

N

NATIVE
NEGATIVE
NEXT
NO
NO-AUDIT
NONE
NOT
NULL
NUMBER
NUMERIC

O

OBJECT-COMPUTER
OC
OCCURS
ODT-INPUT-PRESENT
OF
OFF
OFFSET
OMITTED
ON
OPEN
OPTIONAL
OR
ORGANIZATION
OUTPUT
OVERFLOW
OWN

P

PAGE	PAGE-COUNTER	PC
PERFORM	PF	PH
PIC	PICTURE	PLUS
POINT	POINTER	PORT
POSITION	POSITIVE	
PRIOR	PROCEDURE	PROCEDURES
PROCEED	PROCESS	PROGRAM
PROGRAM-ID	PURGE	

Q

QUEUE	QUOTE	QUOTES

R

RANDOM	RD	READ
READ-OK	READ-WRITE	REAL
RECEIVE	RECEIVED	RECORD
RECORDS	RECREATE	REDEFINES
REEL	REF	REFERENCE
REFERENCES	RELATIVE	RELEASE
REMAINDER	REMOVAL	REMOVE
RENAMES	REPLACING	REPORT
REPORTING	REPORTS	RERUN
RESERVE	RESET	RESPOND
RETURN	REVERSED	REWIND
REWRITE	RF	RH
RIGHT	ROUNDED	RUN

S

SAME
SAVE
SD
SEARCH
SECTION
SECURE
SECURITY
SEEK
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN
SINGLE
SIZE
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STACK
STANDARD
STANDARD-1
START
STATUS
STOP
STORE
STRING
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SUM
SYMBOLIC
SYNC
SYNCHRONIZED
SYSTEM
SYSTEMERROR

T

TABLE
TAG-KEY
TAG-SEARCH
TALLYING
TAPE
TAPES
TASK
TB
TERMINAL
TERMINATE
TEXT
THAN
THEN
THROUGH
THRU
TIME
TIMER
TIMES
TO
TODAYS-DATE
TODAYS-NAME
TOP
TRACTORS
TRAILING
TRANSACTION
TRANSPORT
TRUE
TYPE

U

UNIT
UNLOCK
UNSTRING
UNTIL
UP
UPDATE
UPON
USAGE
USE
USING

V

VA
VALUE
VALUES
VARYING
VIA

W

WAIT
WHEN
WHERE
WITH
WORDS
WORKING-STORAGE
WRITE
WRITE-OK

Z

ZERO
ZEROES
ZEROS

Context-Sensitive Keywords

A context-sensitive keyword can be redeclared as an identifier, and if it is used where the syntax calls for that reserved word, it carries the predefined meaning; otherwise, it has the user-declared meaning.

The following are the context-sensitive keywords:

AREAOVERFLOW
ASCII
BACKUP
CAPABLE
CATEGORY
CHANNEL
COLON
CONVERSATION
DBKIND
DEFAULT
DIV
DMEXCEPTIONMSG
DMNEXTEXCEPTION
DMTERMINATE
DUMP
EBCDIC
ERRORTYPE
FILEOVERFLOW
HEADER
HERE
INTEGER
KEYSPERENTRY
LIST
MOD
ODT
OPEN
PORT
PRINTER
PRODUCTION
READER
REM
REMOTE
SEMANTIC
SPO
STRUCTURE
SW1
SW2
SW3
SW4
SW5
SW6
SW7
SW8
SW9
TAPE
TAPES
TEMPORARY
TEST
TITLE
TRUNCATED
URGENT
USER
VERSION
WFL
WHERE

Application-Specific Keywords

An application-specific keyword is reserved by the compiler for the extent of the program when you are using some network applications.

The compiler reserves the following words in programs only when you specify the RESERVE NETWORK clause in the SPECIAL-NAMES paragraph.

AWAIT-OPEN

RESPOND

For More Information

Refer to Volume 2 of this manual for the list of words the compiler reserves when you specify the RESERVE SEMANTIC clause in the SPECIAL-NAMES paragraph.

Appendix C
EBCDIC and ASCII Character Sets

Tables C–1 and C–2 list the hexadecimal representation and ordinal number for each ASCII and EBCDIC character. Table C–1 is sorted by EBCDIC ordinal number and represents the EBCDIC-to-ASCII translation performed when necessary. Table C–2 is sorted by ASCII ordinal number and represents the ASCII-to-EBCDIC translation performed when necessary.

Note: *In the following tables, ASCII characters greater than 127 (4"7F") are invalid.*

Table C–1. EBCDIC-to-ASCII Translation Chart

EBCDIC		ASCII			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
00	0	00	0	NUL	Null
01	1	01	1	SOH	Start of Heading
02	2	02	2	STX	Start of Text
03	3	03	3	ETX	End of Text
04	4	9C	15		
05	5	09	9	HT	Horizontal Tabulation
06	6	86	13		
07	7	7F	127	DEL	Delete
08	8	97	15		
09	9	8D	14		
0A	10	8E	14		
0B	11	0B	11	VT	Vertical Tabulation
0C	12	0C	12	FF	Form Feed
0D	13	0D	13	CR	Carriage Return
0E	14	0E	14	SO	Shift Out
0F	15	0F	15	SI	Shift In
10	16	10	16	DLE	Data Link Escape

Table C–1. EBCDIC-to-ASCII Translation Chart

EBCDIC		ASCII			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
11	17	11	17	DC1	Device Control 1
12	18	12	18	DC2	Device Control 2
13	19	13	19	DC3	Device Control 3
14	20	9D	157		
15	21	85	13		
16	22	08	8	BS	Backspace
17	23	87	135		
18	24	18	24	CAN	Cancel
19	25	19	25	EM	End of Medium
1A	26	92	146		
1B	27	8F	143		
1C	28	1C	28	FS	File Separator
1D	29	1D	29	GS	Group Separator
1E	30	1E	30	RS	Record Separator
1F	31	1F	31	US	Unit Separator
20	32	80	128		
21	33	81	129		
22	34	82	130		
23	35	83	131		
24	36	84	132		
25	37	0A	10	LF	Line Feed
26	38	17	23	ETB	End of Transmission Block
27	39	1B	27	ESC	Escape
28	40	88	136		
29	41	89	137		
2A	42	8A	138		
2B	43	8B	139		
2C	44	8C	140		

Table C–1. EBCDIC-to-ASCII Translation Chart

EBCDIC		ASCII			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
2D	45	05	5	ENQ	Enquiry
2E	46	06	6	ACK	Acknowledge
2F	47	07	7	BEL	Bell
30	48	90	144		
31	49	91	145		
32	50	16	22	SYN	Synchronous Idle
33	51	93	147		
34	52	94	148		
35	53	95	149		
36	54	96	150		
37	55	04	4	EOT	End of Transmission
38	56	98	152		
39	57	99	153		
3A	58	9A	154		
3B	59	9B	155		
3C	60	14	20	DC4	Device Control 4
3D	61	15	21	NAK	Negative Acknowledge
3E	62	9E	158		
3F	63	1A	26	SUB	Substitute
40	64	20	32	SP	Space
41	65	A0	160		
42	66	A1	161		
43	67	A2	162		
44	68	A3	163		
45	69	A4	164		
46	70	A5	165		
47	71	A6	166		
48	72	A7	167		
49	73	A8	168		

Table C–1. EBCDIC-to-ASCII Translation Chart

EBCDIC		ASCII			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
4A	74	5B	91	[	Opening Bracket
4B	75	2E	46	.	Period
4C	76	3C	60	<	Less Than
4D	77	28	40	(	Opening Parenthesis
4E	78	2B	43	+	Plus
4F	79	21	33	!	Exclamation Point
50	80	26	38	&	Ampersand
51	81	A9	169		
52	82	AA	170		
53	83	AB	171		
54	84	AC	172		
55	85	AD	173		
56	86	AE	174		
57	87	AF	175		
58	88	B0	176		
59	89	B1	177		
5A	90	5D	93	]	Closing Bracket
5B	91	24	36	$	Dollar Sign
5C	92	2A	42	*	Asterisk
5D	93	29	41	)	Closing Parenthesis
5E	94	3B	59	;	Semicolon
5F	95	5E	94	^	Circumflex (ASCII); Not Sign (EBCDIC)
60	96	2D	45	-	Hyphen (Minus)
61	97	2F	47	/	Slant (Slash)
62	98	B2	178		
63	99	B3	179		
64	100	B4	180		
65	101	B5	181		
66	102	B6	182		

Table C–1. EBCDIC-to-ASCII Translation Chart

EBCDIC		ASCII			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
67	103	B7	183		
68	104	B8	184		
69	105	B9	185		
6A	106	7C	124	|	Vertical Line
6B	107	2C	44	,	Comma
6C	108	25	37	%	Percent
6D	109	5F	95	_	Underline
6E	110	3E	62	>	Greater Than
6F	111	3F	63	?	Question Mark
70	112	BA	186		
71	113	BB	187		
72	114	BC	188		
73	115	BD	189		
74	116	BE	190		
75	117	BF	191		
76	118	C0	192		
77	119	C1	193		
78	120	C2	194		
79	121	60	96	`	Grave Accent (Opening Single Quotation Mark)
7A	122	3A	58	:	Colon
7B	123	23	35	#	Number Sign
7C	124	40	64	@	Commercial At
7D	125	27	39	'	Apostrophe (Closing Single Quotation Mark)
7E	126	3D	61	=	Equals
7F	127	22	34	"	Quotation Marks
80	128	C3	195		
81	129	61	97	a	Lowercase a
82	130	62	98	b	Lowercase b

Table C–1. EBCDIC-to-ASCII Translation Chart

EBCDIC		ASCII			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
83	131	63	99	c	Lowercase c
84	132	64	100	d	Lowercase d
85	133	65	101	e	Lowercase e
86	134	66	102	f	Lowercase f
87	135	67	103	g	Lowercase g
88	136	68	104	h	Lowercase h
89	137	69	105	i	Lowercase i
8A	138	C4	196		
8B	139	C5	197		
8C	140	C6	198		
8D	141	C7	199		
8E	142	C8	200		
8F	143	C9	201		
90	144	CA	202		
91	145	6A	106	j	Lowercase j
92	146	6B	107	k	Lowercase k
93	147	6C	108	l	Lowercase l
94	148	6D	109	m	Lowercase m
95	149	6E	110	n	Lowercase n
96	150	6F	111	o	Lowercase o
97	151	70	112	p	Lowercase p
98	152	71	113	q	Lowercase q
99	153	72	114	r	Lowercase r
9A	154	CB	203		
9B	155	CC	204		
9C	156	CD	205		
9D	157	CE	206		
9E	158	CF	207		
9F	159	D0	208		

Table C–1. EBCDIC-to-ASCII Translation Chart

EBCDIC		ASCII			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
A0	160	D1	209		
A1	161	7E	126	~	Overline (Tilde)
A2	162	73	115	s	Lowercase s
A3	163	74	116	t	Lowercase t
A4	164	75	117	u	Lowercase u
A5	165	76	118	v	Lowercase v
A6	166	77	119	w	Lowercase w
A7	167	78	120	x	Lowercase x
A8	168	79	121	y	Lowercase y
A9	169	7A	122	z	Lowercase z
AA	170	D2	210		
AB	171	D3	211		
AC	172	D4	212		
AD	173	D5	213		
AE	174	D6	214		
AF	175	D7	215		
B0	176	D8	216		
B1	177	D9	217		
B2	178	DA	218		
B3	179	DB	219		
B4	180	DC	220		
B5	181	DD	221		
B6	182	DE	222		
B7	183	DF	223		
B8	184	E0	224		
B9	185	E1	225		
BA	186	E2	226		
BB	187	E3	227		
BC	188	E4	228		

Table C–1. EBCDIC-to-ASCII Translation Chart

EBCDIC		ASCII			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
BD	189	E5	229		
BE	190	E6	230		
BF	191	E7	231		
C0	192	7B	123	{	Opening Brace
C1	193	41	65	A	Uppercase A
C2	194	42	66	B	Uppercase B
C3	195	43	67	C	Uppercase C
C4	196	44	68	D	Uppercase D
C5	197	45	69	E	Uppercase E
C6	198	46	70	F	Uppercase F
C7	199	47	71	G	Uppercase G
C8	200	48	72	H	Uppercase H
C9	201	49	73	I	Uppercase I
CA	202	E8	232		
CB	203	E9	233		
CC	204	EA	234		
CD	205	EB	235		
CE	206	EC	236		
CF	207	ED	237		
D0	208	7D	125	}	Closing Brace
D1	209	4A	74	J	Uppercase J
D2	210	4B	75	K	Uppercase K
D3	211	4C	76	L	Uppercase L
D4	212	4D	77	M	Uppercase M
D5	213	4E	78	N	Uppercase N
D6	214	4F	79	O	Uppercase O
D7	215	50	80	P	Uppercase P
D8	216	51	81	Q	Uppercase Q
D9	217	52	82	R	Uppercase R

Table C–1. EBCDIC-to-ASCII Translation Chart

EBCDIC		ASCII			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
DA	218	EE	238		
DB	219	EF	239		
DC	220	F0	240		
DD	221	F1	241		
DE	222	F2	242		
DF	223	F3	243		
E0	224	5C	92	\	Reverse Slant
E1	225	9F	159		
E2	226	53	83	S	Uppercase S
E3	227	54	84	T	Uppercase T
E4	228	55	85	U	Uppercase U
E5	229	56	86	V	Uppercase V
E6	230	57	87	W	Uppercase W
E7	231	58	88	X	Uppercase X
E8	232	59	89	Y	Uppercase Y
E9	233	5A	90	Z	Uppercase Z
EA	234	F4	244		
EB	235	F5	245		
EC	236	F6	246		
ED	237	F7	247		
EE	238	F8	248		
EF	239	F9	249		
F0	240	30	48	0	Zero
F1	241	31	49	1	One
F2	242	32	50	2	Two
F3	243	33	51	3	Three
F4	244	34	52	4	Four
F5	245	35	53	5	Five
F6	246	36	54	6	Six

Table C–1. EBCDIC-to-ASCII Translation Chart

EBCDIC		ASCII			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
F7	247	37	55	7	Seven
F8	248	38	56	8	Eight
F9	249	39	57	9	Nine
FA	250	FA	250		
FB	251	FB	251		
FC	252	FC	252		
FD	253	FD	253		
FE	254	FE	254		
FF	255	FF	255		

Table C–2. ASCII-to-EBCDIC Translation Chart

ASCII		EBCDIC			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
00	0	00	0	NUL	Null
01	1	01	1	SOH	Start of Heading
02	2	02	3	STX	Start of Text
03	3	03	4	ETX	End of Text
04	4	37	55	EOT	End of Transmission
05	5	2D	45	ENQ	Enquiry
06	6	2E	46	ACK	Acknowledge
07	7	2F	47	BEL	Bell
08	8	16	22	BS	Backspace
09	9	05	5	HT	Horizontal Tabulation
0A	10	25	37	LF	Line Feed
0B	11	0B	11	VT	Vertical Tabulation
0C	12	0C	12	FF	Form Feed
0D	13	0D	13	CR	Carriage Return
0E	14	0E	14	SO	Shift Out
0F	15	0F	15	SI	Shift In
10	16	10	16	DLE	Data Link Escape
11	17	11	17	DC1	Device Control 1
12	18	12	18	DC2	Device Control 2
13	19	13	19	DC3	Device Control 3
14	20	3C	60	DC4	Device Control 4
15	21	3D	61	NAK	Negative Acknowledge
16	22	32	50	SYN	Synchronous Idle
17	23	26	38	ETB	End of Transmission Block
18	24	18	24	CAN	Cancel
19	25	19	25	EM	End of Medium
1A	26	3F	63	SUB	Substitute
1B	27	27	39	ESC	Escape
1C	28	1C	28	FS	File Separator

Table C–2. ASCII-to-EBCDIC Translation Chart

ASCII		EBCDIC			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
1D	29	1D	29	GS	Group Separator
1E	30	1E	30	RS	Record Separator
1F	31	1F	31	US	Unit Separator
20	32	40	64	SP	Space
21	33	4F	79	!	Exclamation Point
22	34	7F	127	"	Quotation Marks
23	35	7B	123	#	Number Sign
24	36	5B	91	$	Dollar Sign
25	37	6C	108	%	Percent
26	38	50	80	&	Ampersand
27	39	7D	125	'	Apostrophe (Closing Single Quotation Mark)
28	40	4D	77	(	Opening Parenthesis
29	41	5D	93	)	Closing Parenthesis
2A	42	5C	92	*	Asterisk
2B	43	4E	78	+	Plus
2C	44	6B	107	,	Comma
2D	45	60	96	-	Hyphen (Minus)
2E	46	4B	75	.	Period
2F	47	61	97	/	Slant (Slash)
30	48	F0	240	0	Zero
31	49	F1	241	1	One
32	50	F2	242	2	Two
33	51	F3	243	3	Three
34	52	F4	244	4	Four
35	53	F5	245	5	Five
36	54	F6	246	6	Six
37	55	F7	247	7	Seven
38	56	F8	248	8	Eight
39	57	F9	249	9	Nine

Table C–2. ASCII-to-EBCDIC Translation Chart

ASCII		EBCDIC			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
3A	58	7A	122	:	Colon
3B	59	5E	94	;	Semicolon
3C	60	4C	76	<	Less Than
3D	61	7E	126	=	Equals
3E	62	6E	110	>	Greater Than
3F	63	6F	111	?	Question Mark
40	64	7C	124	@	Commercial At
41	65	C1	193	A	Uppercase A
42	66	C2	194	B	Uppercase B
43	67	C3	195	C	Uppercase C
44	68	C4	196	D	Uppercase D
45	69	C5	197	E	Uppercase E
46	70	C6	198	F	Uppercase F
47	71	C7	199	G	Uppercase G
48	72	C8	200	H	Uppercase H
49	73	C9	201	I	Uppercase I
4A	74	D1	209	J	Uppercase J
4B	75	D2	210	K	Uppercase K
4C	76	D3	211	L	Uppercase L
4D	77	D4	212	M	Uppercase M
4E	78	D5	213	N	Uppercase N
4F	79	D6	214	O	Uppercase O
50	80	D7	215	P	Uppercase P
51	81	D8	216	Q	Uppercase Q
52	82	D9	217	R	Uppercase R
53	83	E2	226	S	Uppercase S
54	84	E3	227	T	Uppercase T
55	85	E4	228	U	Uppercase U
56	86	E5	229	V	Uppercase V

Table C–2. ASCII-to-EBCDIC Translation Chart

ASCII		EBCDIC			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
57	87	E6	230	W	Uppercase W
58	88	E7	231	X	Uppercase X
59	89	E8	232	Y	Uppercase Y
5A	90	E9	233	Z	Uppercase Z
5B	91	4A	74	[	Opening Bracket
5C	92	E0	224	\	Reverse Slant
5D	93	5A	90	]	Closing Bracket
5E	94	5F	95	^	Circumflex (ASCII); Not Sign (EBCDIC)
5F	95	6D	109	_	Underline
60	96	79	121	`	Grave Accent (Opening Single Quotation Mark)
61	97	81	129	a	Lowercase a
62	98	82	130	b	Lowercase b
63	99	83	131	c	Lowercase c
64	100	84	132	d	Lowercase d
65	101	85	133	e	Lowercase e
66	102	86	134	f	Lowercase f
67	103	87	135	g	Lowercase g
68	104	88	136	h	Lowercase h
69	105	89	137	i	Lowercase i
6A	106	91	145	j	Lowercase j
6B	107	92	146	k	Lowercase k
6C	108	93	147	l	Lowercase l
6D	109	94	148	m	Lowercase m
6E	110	95	149	n	Lowercase n
6F	111	96	150	o	Lowercase o
70	112	97	151	p	Lowercase p
71	113	98	152	q	Lowercase q
72	114	99	153	r	Lowercase r

Table C–2. ASCII-to-EBCDIC Translation Chart

ASCII		EBCDIC			
Hex Code	**Ordinal Number**	**Hex Code**	**Ordinal Number**	**Char**	**Description**
73	115	A2	162	s	Lowercase s
74	116	A3	163	t	Lowercase t
75	117	A4	164	u	Lowercase u
76	118	A5	165	v	Lowercase v
77	119	A6	166	w	Lowercase w
78	120	A7	167	x	Lowercase x
79	121	A8	168	y	Lowercase y
7A	122	A9	169	z	Lowercase z
7B	123	C0	192	{	Opening Brace
7C	124	6A	106	|	Vertical Line
7D	125	D0	208	}	Closing Brace
7E	126	A1	161	~	Overline (Tilde)
7F	127	07	7	DEL	Delete

Appendix D
Examples

Following are a group of examples that show various constructs.

Example 1

Example D–1 shows a way to read records from a data file and print them out on a printer.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  EXECTEST.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. A-9.
OBJECT-COMPUTER. A-9.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OPTIONAL CURRFILE
      ASSIGN TO DISK.
    SELECT PFILE ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD CURRFILE
    BLOCK CONTAINS  60 CHARACTERS
    RECORD CONTAINS 60 CHARACTERS.
01 SOURCE-IN-REC.
    05 SOURCE-IN      PIC X(52).
    05 SOURCE-SEQ     PIC 9(8).
FD PFILE RECORD CONTAINS 64 CHARACTERS.
01 PRINT-REC.
    05 PRINT-LINENUM  PIC 9(2).
    05 BLANK-SPACES   PIC X(2).
    05 PRINT-DATA     PIC X(52).
    05 PSEQ           PIC 9(8).
WORKING-STORAGE SECTION.
01 LINE-NUMBER        PIC 9(8).
01 WSREC.
    05 WSNUM          PIC 9(2).
```

Example D–1. Coding READ and WRITE Statements

```
    05 WSPAC          PIC X(2).
    05 WSIN           PIC X(52).
    05 WSEQ           PIC 9(8).
PROCEDURE DIVISION.
PARA-1.
     OPEN INPUT CURRFILE.
     OPEN OUTPUT PFILE.
     MOVE SPACES TO WSPAC.
     MOVE 1 TO LINE-NUMBER.
PARA-2.
     READ CURRFILE AT END GO TO EOJ.
     MOVE LINE-NUMBER TO WSNUM.
     MOVE SOURCE-IN TO WSIN.
     MOVE SOURCE-SEQ TO WSEQ.
     WRITE PRINT-REC FROM WSREC.
     ADD 1 TO LINE-NUMBER.
     GO TO PARA-2.
EOJ.
     CLOSE CURRFILE. CLOSE PFILE.
     STOP RUN.
```

Example D–1. Coding READ and WRITE Statements

Example 2

Example D–2 shows the use of indexed files with alternate keys.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.                    IND-EXMPL.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.                A9.
 OBJECT-COMPUTER.                A9.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
      SELECT KENNEL-FILE ASSIGN TO DISK
          ORGANIZATION IS INDEXED
          ACCESS MODE IS RANDOM
          RECORD KEY IS NAME
          ALTERNATE RECORD KEY IS COLOR WITH DUPLICATES
          ALTERNATE RECORD KEY IS BREED WITH DUPLICATES.
      SELECT PRINT-FILE ASSIGN TO PRINTER.

DATA DIVISION.
FILE SECTION.
FD KENNEL-FILE BLOCK CONTAINS 6 RECORDS
         VALUE OF TITLE IS "KENNEL/RECORDS".
```

Example D–2. Coding Indexed Files with Alternate Keys

```
01 KENNEL-RECORD.
     05 NAME             PIC X(12).
     05 COLOR            PIC X(10).
     05 BREED            PIC X(10).
     05 PRICE            PIC 9999.99.
     05 KENNEL           PIC X(10).

05 KENNEL-NUMBER    PIC 9999.
FD PRINT-FILE.
01 PRINT-RECORD.
     05 PRINT-ITEM       PIC X(12) OCCURS 6 TIMES INDEXED BY J.
     05 PRINT-TRAIL      PIC X(8).

WORKING-STORAGE SECTION.
01 RED                  PIC X(10)       VALUE IS "RED".
01 TAN                  PIC X(10)       VALUE IS "SPOTTED".
01 BLACK                PIC X(10)       VALUE IS "BLACK".
01 EXPENSIVE            PIC 999V99      VALUE IS 800.00.
01 CHEAP                PIC X(10)       VALUE IS "CHEAP".
01 NO-SALE              PIC X(10)       VALUE IS "NO SALE".
01 INDEX-TYPE           PIC X(10) DISPLAY.

PROCEDURE DIVISION.
SECTION-1 SECTION.

PROCEDURE-1.
     OPEN OUTPUT KENNEL-FILE.  OPEN OUTPUT PRINT-FILE.
PROCEDURE-2.
*    This procedure opens an indexed file with
*    alternate keys and then writes data into it.

     MOVE "OTTO" TO NAME.        MOVE "GOLDEN" TO COLOR.
     MOVE EXPENSIVE TO PRICE.    MOVE BLACK TO KENNEL.
     MOVE 13 TO KENNEL-NUMBER.   MOVE "COLLIE" TO BREED.
     WRITE KENNEL-RECORD;
           INVALID KEY DISPLAY "ERROR - PRIMARY KEY NOT UNIQUE".

     MOVE "GERONIMO" TO NAME.    MOVE RED TO COLOR.
     MOVE 350.00 TO PRICE.       MOVE "ANOTHER" TO KENNEL.
     MOVE 97 TO KENNEL-NUMBER.   MOVE "RETRIEVER" TO BREED.
     WRITE KENNEL-RECORD;
           INVALID KEY DISPLAY "ERROR - PRIMARY KEY NOT UNIQUE".

     MOVE "CHARLIE" TO NAME.     MOVE "WHITE" TO COLOR.
     MOVE CHEAP TO PRICE.        MOVE "NONE" TO KENNEL.
     MOVE 01 TO KENNEL-NUMBER.   MOVE "MIXED" TO BREED.
     WRITE KENNEL-RECORD;
           INVALID KEY DISPLAY "ERROR - PRIMARY KEY NOT UNIQUE".
```

Example D–2. Coding Indexed Files with Alternate Keys

```
     CLOSE KENNEL-FILE SAVE.
PROCEDURE-3.
*    This procedure opens an indexed file, reads
*    data with alternate keys and writes to a printer file.

     OPEN I-O KENNEL-FILE.
     MOVE "RED" TO COLOR.
     MOVE "COLOR" TO INDEX-TYPE.
     READ KENNEL-FILE KEY IS COLOR
          INVALID KEY PERFORM INVALID-READ-MARKER GO TO G3.
          PERFORM WRITE-OUT-RECORD.

G3.  MOVE "MIXED" TO BREED.
     MOVE "BREED" TO INDEX-TYPE.
     READ KENNEL-FILE KEY IS BREED
          INVALID KEY PERFORM INVALID-READ-MARKER GO TO G1.
          PERFORM WRITE-OUT-RECORD.

G1.  MOVE "WHITE" TO COLOR.
     MOVE "COLOR" TO INDEX-TYPE.
     READ KENNEL-FILE KEY IS COLOR
          INVALID KEY PERFORM INVALID-READ-MARKER GO TO G2.
          PERFORM WRITE-OUT-RECORD.
G2.  STOP RUN.

WRITE-OUT-RECORD.
     MOVE SPACES TO PRINT-RECORD.
     MOVE NAME TO PRINT-ITEM(1).
     MOVE COLOR TO PRINT-ITEM(2).
     MOVE BREED TO PRINT-ITEM(3).
     MOVE PRICE TO PRINT-ITEM(4).
     MOVE KENNEL TO PRINT-ITEM(5).
     MOVE KENNEL-NUMBER TO PRINT-ITEM(6).
     WRITE PRINT-RECORD.

INVALID-READ-MARKER.
     DISPLAY "ERROR - NO SUCH " INDEX-TYPE " IN FILE"
     MOVE "INVALID RECORD ACCESS" TO PRINT-RECORD.
```

Example D–2. Coding Indexed Files with Alternate Keys

Example 3

Example D–3 shows the use of the OCCURS DEPENDING ON phrase in a WRITE FROM statement. This phrase enables the program to vary the length of the record to be stored. For example, the length of the record PRINT-ODO-ITEM varies from 1 through 13, depending on the length of the record PRINT-SUB.

After execution of this program, the external file PRINT-FILE contains two records. The first records contains the characters *13* followed by 130 occurrences of the letter A. The second record contains the characters *03* followed by three occurrences of the string 1234567890. Note that the value contained in the record PRINT-SUB prior to the execution of the statement WRITE PRINT-RECORD FROM ODO does not determine the number of occurrences of PRINT-ODO-ITEM written to FILEA.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  ODO-EXAMPLE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
    SOURCE-COMPUTER.   B7900.
    OBJECT-COMPUTER.   B7900.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FILEA ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD FILEA
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS PRINT-RECORD.
01 PRINT-RECORD.
    02 PRINT-SUB PIC 99.
    02 PRINT-ODO-GROUP.
        03 PRINT-ODO-ITEM OCCURS 1 TO 13
            TIMES DEPENDING ON PRINT-SUB.
            04 PRINT4  PIC X(4).
            04 PRINT6  PIC X(6).
WORKING-STORAGE SECTION.
01 ODO.
    02 ODO-SUB PIC 99.
    02 ODO-GROUP.
        03 ODO-ITEM OCCURS 1 TO 5 TIMES
            DEPENDING ON ODO-SUB.
            04 ODO6 PIC X(6).
            04 ODO4  PIC X(4).
PROCEDURE DIVISION.
P1.
    OPEN OUTPUT FILEA.
    MOVE 13 TO PRINT-SUB.
    MOVE ALL "A"  TO PRINT-ODO-GROUP.
    WRITE PRINT-RECORD.
P2.
    MOVE 3 TO ODO-SUB.
    MOVE 2 TO PRINT-SUB.
    MOVE ALL "1234567890" TO ODO-GROUP.
    WRITE PRINT-RECORD FROM ODO.
    CLOSE FILEA.

P3.
    STOP RUN.
```

Example D–3. Coding OCCURS DEPENDING ON Phrase in WRITE FROM Statement

Example 4

Example D–4 shows a Sort program with the USING and GIVING options. In this example, the files that are directly involved in the sort operation must not be open when the SORT statement is encountered, because the USING and GIVING options perform the functions of the INPUT PROCEDURE and the OUTPUT PROCEDURE clauses automatically. Although the use of the GIVING and USING options requires a minimum amount of coding, it is not possible to select, add, modify, or delete individual records of the sort file in any way. This means that no special processing records can be incorporated into a sort function when these options are used.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SORTB.
*       THIS SORT PROGRAM EXAMPLE ILLUSTRATES
*       THE FOLLOWING SORT OPTIONS
*       USING
*       GIVING.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. B7900.
OBJECT-COMPUTER. B7900.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT INDATA           ASSIGN TO DISK.
     SELECT SORTED           ASSIGN TO DISK.
     SELECT WORKFL           ASSIGN TO SORT DISK.
     SELECT PRFILE           ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD  INDATA
       BLOCK CONTAINS 60 CHARACTERS
       LABEL RECORDS ARE OMITTED
       DATA RECORD IS A-FILE.
01 A-FILE.
     02 1-ACCT PICTURE IS X(4).
     02 1-DEPT PICTURE IS X(4).
     02 1-BODY PICTURE IS X(52).
FD SORTED
      BLOCK CONTAINS 60 CHARACTERS
      LABEL RECORDS ARE OMITTED
      DATA RECORD IS B-FILE.
01 B-FILE.
     02 2-ACCT PICTURE IS X(4).
     02 2-DEPT PICTURE IS X(4).
     02 2-BODY PICTURE IS X(52).
SD WORKFL
   ; DATA RECORD IS C-FILE.
01 C-FILE.
     02  3-ACCT PICTURE IS X(4).
```

Example D–4. Coding the SORT Program with the USING and GIVING Options

```
02  3-DEPT PICTURE IS X(4).
     02  3-BODY PICTURE IS X(52).
FD PRFILE
     LABEL RECORDS ARE OMITTED
     DATA RECORD IS D-FILE.
01 D-FILE.
     02 4-ACCT PICTURE IS X(4).
     02 4-DEPT PICTURE IS X(4).
     02 4-BODY PICTURE IS X(52).
PROCEDURE DIVISION.
OPENING SECTION.
OPENER.
     OPEN OUTPUT PRFILE.
A-SORT SECTION.
P1.
     SORT WORKFL ON ASCENDING KEY 3-ACCT 3-DEPT
       USING INDATA GIVING SORTED.
OK-PRINT SECTION.
P2. OPEN INPUT SORTED.
A-LOOP SECTION.
P3. READ SORTED AT END GO TO A-1-MOVE.
       PERFORM A-MOVE.
       WRITE D-FILE.
       GO TO A-LOOP.
A-MOVE SECTION.
P4. MOVE B-FILE TO D-FILE.
A-1-MOVE SECTION.
P5.  EXIT.
ALL-DONE SECTION.
P6. CLOSE SORTED PRFILE.
         STOP RUN.
```

Example D–4. Coding the SORT Program with the USING and GIVING Options

Example 5

Example D–5 shows coding of the MERGE program using the USING and GIVING options. In this example, the files that are directly involved in the merge process must not be open when the MERGE statement is encountered. This restriction is necessary because the USING and GIVING options perform the functions of the OPEN, READ, RETURN, WRITE, and CLOSE statements automatically. Although the use of the USING and GIVING options require a minimum amount of coding, it is not possible to select, add, modify, or delete records of the sort-merge file in any way. This means that no special processing procedures can be incorporated into a merge function when these options are used.

The files TO-BE-MERGED1 and TO-BE-MERGED2 are merged into the file AFTER-THE-MERGE. This merged file is then listed after the merge function has been accomplished. The files that were merged were already in sorted order according to the keys defined in the MERGE statement.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MERGEA.
*       THIS MERGE PROGRAM EXAMPLE ILLUSTRATES
*       THE FOLLOWING MERGE OPTIONS
*       USING
*       GIVING.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  B7900.
OBJECT-COMPUTER.  B7900.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TO-BE-MERGED1  ASSIGN TO DISK.
    SELECT TO-BE-MERGED2  ASSIGN TO DISK.
    SELECT AFTER-THE-MERGE ASSIGN TO DISK.
    SELECT WORK-ING       ASSIGN TO MERGE DISK.
    SELECT FOR-THE-PRINTER ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
SD  WORK-ING
   ; RECORD CONTAINS 60 CHARACTERS
   ; DATA RECORD IS C-FILE
   ; BLOCK CONTAINS 60 CHARACTERS.
01   C-FILE.
     02  3-ACCT PICTURE IS X(4).
     02  3-DEPT PICTURE IS X(4).
     02  3-BODY PICTURE IS X(52).
FD  TO-BE-MERGED1
      BLOCK CONTAINS 60 CHARACTERS
      RECORD CONTAINS 60 CHARACTERS
      LABEL RECORDS ARE OMITTED
      DATA RECORD IS A-FILE1 VALUE OF TITLE IS "MG1".
01  A-FILE1.
     02 1-ACCT PICTURE IS X(4).
     02 1-DEPT PICTURE IS X(4).
     02 1-BODY PICTURE IS X(52).
FD  TO-BE-MERGED2
     BLOCK CONTAINS 60 CHARACTERS
     RECORD CONTAINS 60 CHARACTERS
     LABEL RECORDS ARE OMITTED
     DATA RECORD IS A-FILE2 VALUE OF TITLE IS "MG2".
01  A-FILE2.
     02 1-ACCT PICTURE IS X(4).
     02 1-DEPT PICTURE IS X(4).
     02 1-BODY PICTURE IS X(52).
```

Example D–5. Coding MERGE Program with the USING and GIVING Options

```
FD  AFTER-THE-MERGE
     BLOCK CONTAINS 60 CHARACTERS
     RECORD CONTAINS 60 CHARACTERS
     LABEL RECORDS ARE OMITTED
DATA RECORD IS B-FILE VALUE OF TITLE IS "AFTER/MERGE".
01  B-FILE.
     02 2-ACCT PICTURE IS X(4).
     02 2-DEPT PICTURE IS X(4).
     02 2-BODY PICTURE IS X(52).
FD  FOR-THE-PRINTER
     LABEL RECORDS ARE OMITTED
     DATA RECORD IS D-FILE.
01  D-FILE.
     02 4-ACCT PICTURE IS X(4).
     02 4-DEPT PICTURE IS X(4).
     02 4-BODY PICTURE IS X(52).
PROCEDURE DIVISION.
OPENING SECTION.
OPEN-PARA.
     OPEN OUTPUT FOR-THE-PRINTER.
A-MERGE SECTION.
P1. MERGE WORK-ING ON ASCENDING KEY 3-ACCT 3-DEPT
     USING TO-BE-MERGED1 TO-BE-MERGED2
     GIVING AFTER-THE-MERGE.
OK-PRINT SECTION.
P2. OPEN INPUT AFTER-THE-MERGE.
A-LOOP SECTION.
P3. READ AFTER-THE-MERGE AT END GO TO A-1-MOVE.
     PERFORM A-MOVE.
     WRITE D-FILE.
     GO TO A-LOOP.
A-MOVE SECTION.
P4. MOVE B-FILE TO D-FILE.
A-1-MOVE SECTION.
P5. EXIT.
ALL-DONE SECTION.
P6. CLOSE AFTER-THE-MERGE WITH LOCK. CLOSE FOR-THE-PRINTER.
     STOP RUN.
```

Example D-5. Coding MERGE Program with the USING and GIVING Options

Example 6

Example D–6 shows the use of remote files with variable record lengths. Variable length records for a remote file are declared by the use of the RECORD CONTAINS and the RECORD CONTAINS...DEPENDING ON phrases in the FD statement for the remote file in the source code. Run-time code is generated by the compiler to handle

the variable length records. Other than specifying one of the RECORD CONTAINS phrases, you do not need to be concerned with the processing of variable-length records except as noted in the following examples.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.                         VAR1-METHOD1.
ENVIRONMENT DIVISION.

CONFIGURATION SECTION.
SOURCE-COMPUTER.                    B7900.
OBJECT-COMPUTER.                    B7900.

INPUT-OUTPUT SECTION.

FILE-CONTROL.
    SELECT   RMTE                   ASSIGN TO REMOTE.

DATA DIVISION.

FILE SECTION.
FD  RMTE
       RECORD CONTAINS 125 TO 200 CHARACTERS.
01  SHORT-RECORD                   PIC X(125).
01  LONG-RECORD                    PIC X(200).

WORKING-STORAGE SECTION.

01  REMOTE-MESSAGE.
    05  SPACING-AREA               PIC X(50)  VALUE SPACES.
    05  MESSAGE-AREA               PIC X(200) VALUE SPACES.

PROCEDURE DIVISION.

OPEN-REMOTE.
    OPEN I-O RMTE.
 WRITE-SHORT-RECORD.
* ONLY THE FIRST 125 CHARACTERS ARE WRITTEN TO FILE RMTE.
    MOVE "HELLO USER, PLEASE WAIT FOR MY MESSAGE."
       TO MESSAGE-AREA.
    WRITE SHORT-RECORD FROM REMOTE-MESSAGE.
    MOVE SPACES TO MESSAGE-AREA.
    WAIT 2.
 WRITE-LONG-RECORD.
* UP TO 200 CHARACTERS ARE WRITTEN TO REMOTE FILE RMTE.
    MOVE "HELLO USER.            WELCOME TO THE WORLD OF VARIABLE
-    " LENGTH REMOTE FILE RECORDS. WE ARE HAPPY YOU ARE HERE."
       TO MESSAGE-AREA.
```

```
    WRITE LONG-RECORD FROM REMOTE-MESSAGE.
    WAIT 2.
CLOSE-FILE.
    CLOSE RMTE.
    STOP RUN.
```

Example D–6. Coding Remote Files with Variable-Record Lengths

Example 7

Example D–7 shows coding of the PERFORM program with the VARYING UNTIL option and the use of remote files. The program creates a lunch menu and displays it on the screen.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.                    VARI-METHOD1.
ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER.               B7900.
OBJECT-COMPUTER.               B7900.

INPUT-OUTPUT SECTION.

FILE-CONTROL.
    SELECT   RMTE            ASSIGN TO REMOTE.

DATA DIVISION.

FILE SECTION.

FD RMTE
   RECORD CONTAINS 280 TO 680 CHARACTERS.

01  OCCURS-RECORD.
    05  FILLER                   PIC X(120).
    05  VARIABLE-PART.
        10  FIRST-PART           PIC X(80).
        10  SECOND-PART          PIC X(80)
           OCCURS 1 TO 6 TIMES DEPENDING ON Z.

WORKING-STORAGE SECTION.
77  Z                             PIC 9 COMP VALUE 1.
01  ENTREES.
    05  FILLER                PIC X(15)  VALUE "ONION SOUP    ".
    05  FILLER                PIC X(65)  VALUE SPACES.
    05  FILLER                PIC X(15)  VALUE "FRUIT SALAD   ".
    05  FILLER                PIC X(65)  VALUE SPACES.
    05  FILLER                PIC X(15)  VALUE "HAMBURGER     ".
```

```
    05  FILLER                PIC X(65)  VALUE SPACES.
    05  FILLER                PIC X(15)  VALUE "FRENCH FRIES  ".
    05  FILLER                PIC X(65)  VALUE SPACES.
    05  FILLER                PIC X(15)  VALUE "ICED TEA      ".
    05  FILLER                PIC X(65)  VALUE SPACES.
    05  FILLER                PIC X(15)  VALUE "ICE CREAM     ".
    05  FILLER                PIC X(65)  VALUE SPACES.
```

Example D–7. Coding PERFORM Program with the VARYING UNTIL Option

```
01  ENTREE-TABLE REDEFINES ENTREES.
    05  ENTREE PIC X(80) OCCURS 6 TIMES.

PROCEDURE DIVISION.
OPEN-REMOTE.
    OPEN I-O RMTE.
    MOVE SPACES TO OCCURS-RECORD.
    MOVE "  LUNCH MENU: "TO FIRST-PART.
    PERFORM WRITE-REMOTE VARYING Z FROM 1 BY 1 UNTIL Z > 6.
    CLOSE RMTE.
    STOP RUN.
WRITE-REMOTE.
    MOVE ENTREE(Z) TO SECOND-PART(Z).
    WRITE OCCURS-RECORD.
END-OF-JOB.
```

Example D–7. Coding PERFORM Program with the VARYING UNTIL Option

Example 8

Example D–8 shows coding of a remote file.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.                 VARI-METHOD2.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.            B7900.
OBJECT-COMPUTER.            B7900.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT RMTE             ASSIGN TO REMOTE.

DATA DIVISION.
FILE SECTION.
FD RMTE
     RECORD CONTAINS 240 CHARACTERS.
```

```
01 RMTE-RECORD.
     05  ACTUAL-RECORD-AREA    PIC X(240).
```

Example D–8. Coding Remote Files

```
WORKING-STORAGE SECTION.
01  MESSAGE-IN.
    05  MESSAGE-ENTERED        PIC X(14)  VALUE "YOU ENTERED:  ".
    05  MESSAGE-AREA           PIC X(226) VALUE SPACES.
01  MESSAGE-OUT.
    05  GOOD-MORNING           PIC X(14)
         VALUE "GOOD MORNING  ".
    05  USER-NAME              PIC X(17).

05  INSTRUCT               PIC X(209)
          VALUE " CLEAR HOME AND ENTER ANY MESSAGE.  THEN XMIT.".

 PROCEDURE DIVISION.
 OPEN-REMOTE.
     OPEN I-O RMTE.
 READ-AND-WRITE.
*    THE USER IS PROMPTED TO ENTER A MESSAGE.  THE MESSAGE
*    ENTERED WILL BE DISPLAYED BACK ON THE TERMINAL (THE MESSAGE
*    SHOULD NOT BE MORE THAN 209 CHARACTERS LONG), AND THE
*    PROGRAM WILL END.
     MOVE ATTRIBUTE USERCODE OF MYSELF TO USER-NAME.
     MOVE MESSAGE-OUT TO ACTUAL-RECORD-AREA.
     WRITE RMTE-RECORD.
     READ RMTE AT END GO TO CLOSE-FILE.
     MOVE ACTUAL-RECORD-AREA TO MESSAGE-AREA.
     MOVE MESSAGE-IN TO ACTUAL-RECORD-AREA.
     WRITE RMTE-RECORD.
 CLOSE-FILE.
     CLOSE RMTE.
     STOP RUN.
```

Example D–8. Coding Remote Files

Example 9

Example D–9 shows the access of tape labels by means of USE routines.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID.                    TAPELABEL-EXAMPLE.
*    Demonstrate access of tape labels by means of USE routines.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  A-15.
 OBJECT-COMPUTER.  A-15.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.

     SELECT DATA-IN             ASSIGN TO TAPE.
     SELECT DATA-OUT            ASSIGN TO TAPE.

 I-O-CONTROL.
     SAME RECORD AREA FOR DATA-IN, DATA-OUT.

 DATA DIVISION.
 FILE SECTION.

 FD  DATA-IN

   LABEL RECORD IS DATA-LABEL
       VALUE OF SAVEFACTOR IS 3
                DENSITY IS BPI1600
                FILENAME IS "DATATAP".
   01  DATA-IN-RCD                PIC X(100).

   FD  DATA-OUT
       LABEL RECORD IS DATA-LABEL
       VALUE OF SAVEFACTOR IS 3
                DENSITY IS BPI1600
                FILENAME IS "DATATAP".
   01  DATA-OUT-RCD               PIC X(100).
```

Example D–9. Coding Tape Label Access with USE Routines

```
WORKING-STORAGE SECTION.

01  DATA-LABEL.
    05  DATA-LBL-ID            PIC X(4).
    05  DATA-LBLA              PIC X(36).
    05  DATA-LBLB              PIC X(40).

PROCEDURE DIVISION.

DECLARATIVES.

TAPE-BGN-INPT SECTION.
    USE AFTER STANDARD BEGINNING LABEL PROCEDURE ON DATA-IN.

TAPE-BGN-INPT-RTN.
    MOVE DATA-LBLA TO DATA-LBLB.

TAPE-END-INPT SECTION.
    USE AFTER STANDARD ENDING LABEL PROCEDURE ON DATA-IN.

TAPE-END-INPT-RTN.
    MOVE DATA-LBLA TO DATA-LBLB.

TAPE-BGN-OTPT SECTION.
    USE BEFORE STANDARD BEGINNING LABEL PROCEDURE ON DATA-OUT.

TAPE-BGN-OTPT-RTN.
    MOVE "USER DEFINED DATA" TO DATA-LBLA.

TAPE-END-OTPT SECTION.
    USE BEFORE STANDARD ENDING LABEL PROCEDURE ON DATA-OUT.

TAPE-END-OTPT-RTN.
    MOVE "MORE USER DEFINED DATA" TO DATA-LBLA.

END DECLARATIVES.
```

Example D–9. Coding Tape Label Access with USE Routines

```
 MAIN SECTION.
DATA-TAPE-RTN.

 OPEN INPUT DATA-IN.
OPEN OUTPUT DATA-OUT.
    READ DATA-IN.
 WRITE DATA-OUT-RCD.
   CLOSE DATA-IN.
  CLOSE DATA-OUT.

THATS-ALL-FOLKS.

     STOP RUN.
```

Example D–9. Coding Tape Label Access with USE Routines

Index

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

Special Characters

8600 0296–208