VisualAge COBOL

Programming Guide

Version 2.2

SC26-9050-02

VisualAge COBOL

Programming Guide

Version 2.2

SC26-9050-02

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xx.

Third Edition (April 1998)

This edition applies to Version 2.2 of IBM VisualAge COBOL (program number 5639-B92) and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for reader's comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, W92/H3
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (1) the exchange of information between independently created programs and other programs (including this one) and (2) the mutual use of the information which has been exchanged, should contact:

IBM Corporation, W92/H3
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Programming Interface Information

This *VisualAge COBOL Programming Guide* is intended to help the user create, compile, link, and run IBM VisualAge COBOL application programs. This book documents General-Use Programming Interface and Associated Guidance Information provided by IBM VisualAge COBOL.

General-Use programming interfaces allow the customer to write programs that obtain the services of IBM VisualAge COBOL.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AD/Cycle
AIX
AIX/6000
AS/400
C/370
CICS
CICS OS/2
COBOL/370
DATABASE 2
DB2
DFSMS/MVS
DFSORT
IBM
IMS
Language Environment
MQSeries Three Tier
MVS/ESA
Operating System/2
OS/2
OS/390
Presentation Manager
RS/6000
System Object Model
System/390
SOMobjects
VisualAge

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Microsoft, Windows, Windows NT, the Windows 95 logo, and Open Database Connectivity are trademarks or registered trademarks of Microsoft Corporation.

INTERSOLV is a registered trademark and DataDirect a trademark of INTERSOLV, Inc.

MQ/Series is the registered trademark of MQSoftware Inc.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

About This Book

Welcome to IBM VisualAge COBOL[1], IBM's new COBOL development environment for OS/2, Windows 95, and Windows NT! VisualAge COBOL gives you a comprehensive development environment designed specifically for mission-critical, client/server applications.

VisualAge COBOL supports local and remote access to DB2, CICS, and VSAM (remote access only to VSAM using Windows), giving you access to data and transactions nearly anywhere in your enterprise. And all the IBM COBOL family of solutions support the high subset of ANSI 85 COBOL functions, so your applications can be ported across supported platforms, whether they are running on a mainframe, an RS/6000, or a personal computer with OS/2, Windows 95, or Windows NT.

VisualAge COBOL supports object-oriented extensions, allowing you to develop software objects using COBOL and share objects created by other languages, like C++.

VisualAge COBOL provides a complete development environment. The environment includes an editor, debugger, GUI designer, and performance analyzer, all integrated with WorkFrame. WorkFrame integrates your tools and files, so selecting a file also selects the application you need to control the execution of a program, examine and modify data, and perform many other useful functions.

How This Book Will Help You

This book will help you write, compile, link-edit, and run your VisualAge COBOL programs. It will also help you define object-oriented classes and methods, invoke methods, and refer to objects in your programs.

This book assumes experience in developing application programs and some knowledge of COBOL. It focuses on using COBOL to meet your programming objectives and not on the definition of the COBOL language. For complete information on COBOL syntax, refer to *IBM COBOL Language Reference*.

There are some differences between host and PC COBOL. For details on language and system differences between VisualAge COBOL and IBM COBOL for OS/390 & VM, see Appendix A, “Summary of Differences with Host COBOL” on page 540.

This book also assumes familiarity with OS/2 or Windows and the VisualAge COBOL development environment. For information on OS/2 or Windows, see your operating system documentation. To learn about the VisualAge COBOL development environment, see the *Getting Started* guide.

[1] IBM VisualAge COBOL is referred to as VisualAge COBOL throughout this publication.

Throughout this book, we use these indicators to identify platform-specific information:

- Prefix the text with platform-specific text (for example, “Under OS/2...”)
- Add parenthetical qualifications (for example, “(Windows only)”)
- Bracket the text with icons. We use the following icons:

  OS/2 ▶ Informs you of information specific to OS/2. ◀ OS/2

  Windows ▶ Informs you of information specific to Windows. ◀ Windows

Abbreviated Terms

Certain terms are used in a shortened form in this book. Abbreviations for the product names used most frequently in this book are listed alphabetically in Figure 1. Abbreviations for other terms, if not commonly understood, are shown in *italics* the first time they appear, and are listed in the glossary in the back of this book.

Figure 1. Common Abbreviations in this Book

Term Used	Long Form
CICS	CICS for OS/2 or CICS for Windows NT or VisualAge CICS Enterprise Application Development
DB2	Database 2
OS/2	Operating System/2
SOM	System Object Model
STL	Standard Language file system

In addition to these abbreviated terms, the term “COBOL 85 Standard” is used in this book to refer to the combination of the following standards:

- ISO 1989:1985, Programming languages - COBOL
- ISO 1989/Amendment 1, Programming Languages - COBOL - Amendment 1: Intrinsic function module
- X3.23-1985, American National Standard for Information Systems - Programming Language - COBOL
- X3.23a-1989, American National Standard for Information Systems - Programming Language - Intrinsic Function Module for COBOL

The two ISO standards are identical to the American National Standards.

Syntax Notation

Syntax Notation

Throughout this book, syntax for the compiler options is described using the structure defined below.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line. The following table shows the meaning of symbols at the beginning and end of syntax diagram lines.

Symbol	Indicates
►►──	The syntax diagram starts here
───►	The syntax diagram is continued on the next line
►──	The syntax diagram is continued from the previous line
───►◄	The syntax diagram ends here

  Diagrams of syntactical units other than complete statements start with the ►── symbol and end with the ───► symbol.

- Required items appear on the horizontal line (the main path).

```
►►──STATEMENT──required item─────────────────────────────────────────►◄
```

- Optional items appear below the main path.

```
►►──STATEMENT──┬─────────────┬───────────────────────────────────────►◄
               └─optional item┘
```

- When you can choose from two or more items, they appear vertically in a stack.

  If you **must** choose one of the items, one item of the stack appears on the main path.

```
►►──STATEMENT──┬─required choice 1─┬─────────────────────────────────►◄
               └─required choice 2─┘
```

  If choosing one of the items is optional, the entire stack appears below the main path.

```
►►──STATEMENT──┬───────────────────┬─────────────────────────────────►◄
               ├─optional choice 1─┤
               └─optional choice 2─┘
```

- An arrow returning to the left above the main line indicates an item that can be repeated.

```
               ┌──────────────┐
►►──STATEMENT──▼─repeatable item┴────────────────────────────────────►◄
```

  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords appear in uppercase letters (for example, PRINT). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, `item`). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, they must be entered as part of the syntax.
- Use at least one blank or comma to separate parameters.

How Examples Are Shown

This book shows numerous examples of sample COBOL statements, program fragments, and small programs to help illustrate the concepts being discussed. The examples of program code are written in lowercase, uppercase, or mixed case to demonstrate that you can write your programs in any of these three cases.

Where it helps to more clearly separate the examples from the explanatory text, they are indented, `presented in a different font style`, or are shown in boxes.

Names of files, COBOL keywords, commands, and options appearing in text are generally shown in SMALL UPPER CASE, unless they are mixed-case, in which case they are `presented in a different font style`.

Summary of Changes

This section lists the key changes that have been made to the IBM VisualAge COBOL product since Version 2.0. Those documented in this publication have an associated page reference for your convenience. The latest technical changes are marked in the text by a change bar in the left margin.

Major Changes in Version 2.1

- New compiler option `-host` to facilitate setting of all host data compiler options ("Options Supported by cob2" on page 142).
- New compiler option ANALYZE to check the syntax of embedded SQL and CICS statements ("ANALYZE" on page 162).
- Host DBCS, removal of restriction "CHAR (EBCDIC) does not apply to DBCS data.," (removed from Appendix B, "System/390 Host Data Type Considerations" on page 543).
- Default EBCDIC code page based on run time locale, ("Locale Sensitivity" on page 486 and "Definitions of COBOL Environment Variables" on page 135).
- Enable Japanese Era and Chinese Era support in the date/time callable services.
- Remote workstation DL/I calls (Appendix G, "Remote DL/I" on page 638) (Windows only).
- Extension of the ACCEPT statement to cover the recommendation in the *Working Draft for Proposed Revision of ISO 1989:1985 Programming Language COBOL* ("How to Get 4-digit Year Dates" on page 518).
- New intrinsic date functions.

Major Changes in Version 2.2

- The millennium language extensions, enabling compiler-assisted date processing for dates containing 2-digit and 4-digit years (Chapter 31, "Using the Millennium Language Extensions" on page 520).
- Remote workstation DL/I calls for OS/2 (Appendix G, "Remote DL/I" on page 638).
- Host data type support for DB2 and the date and time callable services.
- Support for Version 3 of the Open Database Connectivity (ODBC) interface (Chapter 23, "Open Database Connectivity" on page 418).

Part 1. Coding Your Program

This part of the book explains how to do various programming tasks using the COBOL language. It discusses the most common topics, starting with basic ones, then building on those in succeeding chapters. Topics related to object-oriented COBOL are in Part 3, “Object-Oriented Programming Topics” on page 269. More complex programming topics are treated in Part 4, “Advanced Topics” on page 363.

For complete details on the COBOL language, see *IBM COBOL Language Reference*.

COBOL Terms for Data

Chapter 1. Introduction to COBOL Terms

This chapter is intended to help the non-COBOL programmer relate terms used in other programming languages to COBOL terms.

This chapter introduces COBOL fundamentals for:

- Variables, Structures, Literals, and Constants
- Assignment and Terminal Interaction
- Built-In (Intrinsic) Functions
- Tables and Pointers

Variables, Structures, Literals, and Constants

Most high-level programming languages share the concept of data being represented as variables, structures, literals, and constants. This section describes how these data representations are defined in COBOL. You place all data-item definitions in the DATA DIVISION of your program.

Variables

In COBOL you refer to a variable by a *data-name*. For example, if a customer name is a variable in your program, code:

```
Data Division.
.
.
01  Customer-Name              Pic X(20).
01  Original-Customer-Name     Pic X(20).
    .
    .
Procedure Division.
    .
    .
    Move Customer-Name to Original-Customer-Name
.
.
```

The data used in a COBOL program can be divided into three classes: alphabetic, alphanumeric, and numeric. For complete details on the classes and categories of data and the PICTURE clause rules for defining data, see *IBM COBOL Language Reference*.

Data Structure

Related data items are often parts of a larger, hierarchical data structure. A data item that is composed of subordinated data items is called a group item. An elementary data item is a data item that does not have any subordinate items. A record can be either an elementary data item or a group of data items.

Group Items Example

In the following example, `Customer-Record` is a group item composed of two group items (`Customer-Name` and `Part-Order`) both of which contain elementary data items. You can refer to the entire group item or to parts of the group item as shown in the MOVE statements in the Procedure Division.

```
Data Division.
File Section.
FD  Customer-File
    Record Contains 45 Characters.
01  Customer-Record.
    05  Customer-Name.
        10 Last-Name            Pic x(17).
        10 Filler               Pic x.
        10 Initials             Pic xx.
    05  Part-Order.
        10 Part-Name            Pic x(15).
        10 Part-Color           Pic x(10).
Working-Storage Section.
01  Orig-Customer-Name.
    05  Surname                 Pic x(17).
    05  Initials                Pic x(3).
01  Inventory-Part-Name         Pic x(15).
    .
    .
Procedure Division.
    .
    .
    Move Customer-Name to Orig-Customer-Name
    Move Part-Name to Inventory-Part-Name
    .
    .
```

Literals

When you know the value you want to use for a data item, you don't need to define or refer to a data-name; instead use a literal representation of the data value in the Procedure Division.

For example, you might want to prepare an error message for an output file:

```
Move “Invalid Data” To Customer-Name
```

Or, you might want to compare a data item to a certain number:

```
01  Part-number                 Pic 9(5).
    .
    .
    If Part-number = 03519 then display "Part number was found"
```

In these examples, `"Invalid Data"` is a non-numeric literal, and `03519` is a numeric literal.

Assigning Values to Data

Constants

COBOL does not define a construct specifically for constants, but most programmers define a data item with an initial VALUE (as opposed to initializing a variable using the INITIALIZE statement):

```
Data Division.
.
.
01  Report-Header              pic x(50)  value "Company Sales Report".
.
.
01  Interest                   pic 9v9999 value 1.0265.
```

Figurative Constants

Certain commonly used constants and literals are provided as reserved words, called figurative constants. Because they represent fixed values, figurative constants do not require a data definition: ZERO, SPACE, HIGH-VALUE, LOW-VALUE, QUOTE, NULL, ALL.

For example: `Move Spaces To Report-Header`.

Assignment and Terminal Interactions

After you have defined a data item, you can assign a value to it at any time. Assignment takes many forms in COBOL, depending on the purpose behind the assignment:

Figure 2. How to Assign Values to a Data Item

What You Want to Do	**How to Do It**
Assign values to a data item or large data area	One of these ways: • INITIALIZE statement • MOVE statement • STRING or UNSTRING statement • VALUE clause (To set data items to the values you want them to have when the program is in its initial state.)
Replace characters or groups of characters in a data item	INSPECT statement
Receive input values from the terminal or a file	ACCEPT statement
Receive input values from a file	READ (or READ INTO) statement
Assign the results of arithmetic	COMPUTE statement

Initializing a Variable (INITIALIZE Statement)

The following examples illustrate some uses of the INITIALIZE statement. (The symbol ƀ indicates a space.)

Initializing a Variable to Blanks or Zeroes:

```
INITIALIZE identifier-1
```

IDENTIFIER-1 PICTURE	IDENTIFIER-1 Before	IDENTIFIER-1 After
9(5)	12345	00000
X(5)	AB123	ƀƀƀƀƀ
99XX9	12AB3	ƀƀƀƀƀ
XXBX/XX	ABbC/DE	ƀƀƀƀ/ƀƀ
**99.9CR	1234.5CR	**00.0ƀƀ
A(5)	ABCDE	ƀƀƀƀƀ
+99.99E+99	+12.34E+02	+00.00E+00

Initializing a Right-Justified Field:

```
01  ANJUST                PIC X(8)  JUSTIFIED RIGHT.
01  ALPHABETIC-1          PIC A(4)  VALUE "ABCD".
    .
    .
    INITIALIZE ANJUST
        REPLACING ALPHANUMERIC DATA BY ALPHABETIC-1
```

ALPHABETIC-1	ANJUST Before	ANJUST After
ABCD	ƀƀƀƀƀƀƀƀ	ƀƀƀƀABCD

Initializing an Alphanumeric Variable:

```
01  ALPHANUMERIC-1        PIC X.
01  ALPHANUMERIC-3        PIC X(1) VALUE "A".
    .
    .
    INITIALIZE ALPHANUMERIC-1
        REPLACING ALPHANUMERIC DATA BY ALPHANUMERIC-3
```

ALPHANUMERIC-3	ALPHANUMERIC-1 Before	ALPHANUMERIC-1 After
A	y	A

Initializing a Numeric Variable:

```
01  NUMERIC-1             PIC 9(8).
01  NUM-INT-CMPT-3        PIC 9(7) COMP VALUE 1234567.
    .
    .
    .
    INITIALIZE NUMERIC-1
        REPLACING NUMERIC DATA BY NUM-INT-CMPT-3
```

NUM-INT-CMPT-3	NUMERIC-1 Before	NUMERIC-1 After
1234567	98765432	01234567

Initializing an Edited Alphanumeric Variable:

```
01  ALPHANUM-EDIT-1       PIC XXBX/XXX.
01  ALPHANUM-EDIT-3       PIC X/BB VALUE "M/ƀƀ".
    .
    .
    INITIALIZE ALPHANUM-EDIT-1
        REPLACING ALPHANUMERIC-EDITED DATA BY ALPHANUM-EDIT-3
```

ALPHANUM-EDIT-3	ALPHANUM-EDIT-1 Before	ALPHANUM-EDIT-1 After
M/ƀƀ	ABƀC/DEF	M/ƀƀ/ƀƀƀ

Initializing a Structure (INITIALIZE Statement)

You can reset the values of all subordinate items in a group by applying the INITIALIZE statement to the group item. However, it is inefficient to initialize an entire group unless you really need all the items in the group initialized.

The following example shows how you can reset fields in a transaction record produced by a program to spaces and zeros. (Notice that the fields are not identical in each record produced.)

```
01  TRANSACTION-OUT.
    05  TRANSACTION-CODE              PIC X.
    05  PART-NUMBER                   PIC 9(6).
    05  TRANSACTION-QUANTITY          PIC 9(5).
    05  PRICE-FIELDS.
        10  UNIT-PRICE                PIC 9(5)V9(2).
        10  DISCOUNT                  PIC V9(2).
        10  SALES-PRICE               PIC 9(5)V9(2).
    .
    .
    INITIALIZE TRANSACTION-OUT
```

	TRANSACTION-OUT Before	TRANSACTION-OUT After
Record 1	R001383000240000000000000000	ƀ000000000000000000000000000
Record 2	R001390000480000000000000000	ƀ000000000000000000000000000
Record 3	S001410000120000000000000000	ƀ000000000000000000000000000
Record 4	C001383000000000425000000000	ƀ000000000000000000000000000
Record 5	C002010000000000000010000000	ƀ000000000000000000000000000

Note: The symbol ƀ represents a blank space.

Assigning Values to Data

Assigning Values to Variables or Structures (MOVE Statement)

Use the MOVE statement to assign values to variables or structures. For example, the following statement:

```
Move Customer-Name to Orig-Customer-Name
```

assigns the contents of the variable `Customer-Name` to the variable `Orig-Customer-Name`. If `Customer-Name` were longer than `Orig-Customer-Name`, truncation would occur on the right. If it were shorter, the extra character positions on the right would be filled with spaces.

When you move a group item to another group item, be sure the subordinate data descriptions are compatible. The compiler performs all MOVE statements regardless of whether items fit, even if that means a destructive overlap could occur at run time. In such cases, you'll get a warning message when you compile your program.

Assigning Values to Numeric Variables

For variables containing numbers, moves can be more complicated because there are several ways numbers are represented. In general, the algebraic values of numbers are moved if possible (as opposed to the digit-by-digit move performed with character data):

```
01  Item-x            Pic 999v9.
  .
  .
    Move 3.06 to Item-x
```

This move would result in `Item-x` containing the value 3.0, represented by 0030.

Assigning Terminal/File Input to Variables (ACCEPT Statement)

Another way to assign a value to a variable is to read the value from the terminal or a file. To enter data from the terminal, first associate the terminal with a mnemonic-name in the SPECIAL-NAMES Paragraph:

```
Environment Division.
Configuration Section.
Special-Names.
   Console is Names-Input.
```

Then the statement:

```
Accept Customer-Name From Names-Input
```

assigns the line of input entered at the terminal to the variable `Customer-Name`.

To read from a file instead of the terminal, either:

- Change

  ```
  Console is Names-Input
  ```

  to

  device `is Names-Input`

in the above example, where *device* is any valid system device (for example, SYSIN).

-or-

- Set the environment variable CONSOLE to a valid file specification using the `SET` command. For example:

  ```
  SET CONSOLE=\myfiles\myinput.rpt
  ```

  Note that the environment variable must be the same as the system device used. In the above example, the system device is `Console`, but the method of assigning an environment variable to the system device name is supported for all valid system devices. For example, if the system device is SYSIN, the environment variable which must be assigned a file specification is SYSIN also).

For more information on setting environment variables, see “Setting Environment Variables” on page 134.

Displaying Data Values on the Terminal/File (DISPLAY Statement)

In addition to assigning a variable a value read in from the terminal or a file, you can also display the value of a variable on the terminal or write it to a file. For example, if the contents of the variable `Customer-Name` is JOHNSON, then the following statement:

```
Display “No entry for surname '” Customer-Name “' found in the file.”
```

will display this message on the terminal:

```
No entry for surname 'JOHNSON' found in the file.
```

To write data to a destination other than the system logical output unit, the UPON clause must be used on the DISPLAY statement. For example:

```
Display "Hello" UPON SYSOUT
```

writes to the system logical output device, or to the destination specified in the SYSOUT environment variable, if defined.

Assigning Arithmetic Results

When assigning a number to a variable, it is sometimes better to use the COMPUTE statement instead of the MOVE statement. For example, the following two statements accomplish the same thing in most cases:

```
Move w to z
Compute z = w
```

However, when significant left-order digits would be lost in execution, the COMPUTE statement can detect the condition and allow you to handle it. The MOVE statement carries out the assignment with destructive truncation.

When you use the ON SIZE ERROR phrase of the COMPUTE statement, the compiler generates code to detect a size-overflow condition. If the condition occurs, the code in the ON SIZE ERROR phrase is performed, and the content of `z` remains unchanged. If the ON SIZE ERROR phrase is not specified, the assignment is carried out with truncation. There is no ON SIZE ERROR support for the MOVE statement.

Assigning Results of Computations (COMPUTE Statement)

The COMPUTE statement is also used to assign the result of an arithmetic expression (or intrinsic function) to a variable. For example:

```
Compute z = y + (x ** 3)
Compute x = Function Max(x y z)
```

For information on intrinsic functions, see the *IBM COBOL Language Reference*.

Built-in (Intrinsic) Functions

Some high-level programming languages have built-in functions that you can reference in your program as if they were variables having defined attributes and a predetermined value. In COBOL, these are called *intrinsic functions*; they provide various string- and number-manipulation capabilities.

Introduction to Intrinsic Functions

The groups of highlighted words in the following examples are referred to as *function-identifiers*. A function-identifier is the combination of the COBOL reserved word FUNCTION followed by a *function-name* (such as Max), followed by any arguments to be used in the evaluation of the function (such as x, y, z):

```
Unstring Function Upper-case(Name) Delimited By Space Into Fname Lname

Compute A = 1 + Function Log10(x)

Compute M = Function Max(x y z)
```

A function-identifier represents both the function's invocation and the data value returned by the function. Because it actually represents a data item, a function-identifier can be used in most places in the Procedure Division where a data item having the attributes of the returned value can be used.

Because the value of an intrinsic function is derived automatically at the time of reference, you do not need to define functions in the Data Division. Define only the nonliteral data items that you use as arguments. Figurative constants are not allowed as arguments.

Using Function References in Other Contexts

Function-identifiers are loosely referred to in this book as function references. Whereas the COBOL word FUNCTION is a reserved word, the function-names are not reserved; you can use them in other contexts, such as for the name of a variable, and without references to a function.

For example, you could use SQRT to invoke an intrinsic function and/or to name a variable in your program:

Introducing Intrinsic Functions

```
Working-Storage Section.
01  x                 Pic 99  value 2.
01  y                 Pic 99  value 4.
01  z                 Pic 99  value 0.
01  Sqrt              Pic 99  value 0.
    .
    .
    Compute Sqrt = 16 ** .5
    Compute z = x + Function Sqrt(y)
    .
    .
```

Types of Intrinsic Functions

A function-identifier represents a value that is either a character string (alphanumeric data class) or a number (numeric data class) depending on the type of function. The functions MAX, MIN, DATEVAL, and UNDATE can return either type of value depending on the type of arguments you supply.

Three functions, DATEVAL, UNDATE, and YEARWINDOW are provided with the millennium language extensions to assist with manipulationg and converting windowed date fields. For details on the millennium language extensions, see Chapter 31, "Using the Millennium Language Extensions" on page 520. The three functions are described individually in *IBM COBOL Language Reference*.

Numeric intrinsic functions are further classified according to the type of numbers they return. See the *IBM COBOL Language Reference* for more details.

Nesting Functions

Functions can reference other functions as arguments as long as the results of the nested functions meet the requirements for the arguments of the outer function. For example:

```
Compute x = Function Max((Function Sqrt(5)) 2.5 3.5)
```

In this case, `Function Sqrt(5)` returns a numeric value. Thus, the three arguments to the MAX function are all numeric, which are allowable argument types for this function.

Some of the examples in the next three chapters show nesting of functions.

Substrings of Function-Identifiers

You can include a substring specification (reference modifier) in your function-identifier for alphanumeric functions.

Arguments to Intrinsic Functions

The ALL subscript, which enables you to easily reference all of the elements of an array as function arguments, and allowable types of function arguments are discussed in *IBM COBOL Language Reference*.

Arrays and Pointers

In COBOL, arrays are called tables. The language constructs available for representing and handling tables are discussed in Chapter 4, “Handling Tables” on page 47.

Pointers

Pointer data items can contain virtual storage addresses. You define them explicitly with the USAGE IS POINTER clause in the Data Division or implicitly as ADDRESS OF special registers.

Pointer data items can be:

- Passed between programs using the CALL ... BY REFERENCE statement
- Moved to other pointers using the SET statement
- Compared to other pointers for equality using a relation condition
- Initialized to contain an invalid address, using VALUE IS NULL

Use pointer data items to:

- Accomplish limited base addressing, particularly if you want to pass and receive addresses of a variably located record area.
- Handle a chained list.

Procedure Pointers

A procedure pointer is a pointer to an entry point. Define the entry address for a procedure entry point with the USAGE IS PROCEDURE-POINTER clause in the Data Division.

Chapter 2. Program Structure

A COBOL program consists of four divisions, each with a specific logical function. Only the IDENTIFICATION DIVISION is required.

- The IDENTIFICATION DIVISION, discussed on page 12.
- The ENVIRONMENT DIVISION, discussed on page 13.
- The DATA DIVISION, discussed on page 18.
- The PROCEDURE DIVISION, discussed on page 22.

To define a COBOL class or method, you need to define some divisions differently than you would for a program. For detail on the differences, see “Writing a Class Definition” on page 272 or “Writing a Method Definition” on page 276.

IDENTIFICATION DIVISION

Use the IDENTIFICATION DIVISION to name your program and to optionally give other identifying information. For example:

```
Identification Division.
Program-ID. Helloprog.
Author. A. Programmer.
Installation.  Computing Laboratories.
Date-Written.  08/18/1997.
Date-Compiled. 02/27/1998.
```

You can use the optional AUTHOR, INSTALLATION, DATE-WRITTEN, and DATE-COMPILED paragraphs for descriptive information about your program. The data you enter on the DATE-COMPILED paragraph is replaced with the latest compilation date.

PROGRAM-ID Paragraph

Use the PROGRAM-ID paragraph to name your program. The program name you assign is used in these ways:

- Other programs use the name to call your program.
- The name appears in the header on each page, except the first page, of the program listing generated when the program is compiled. (See “Changing Header of Source Listing” on page 13 for details.)

Marking Programs as RECURSIVE

Code the RECURSIVE attribute on the PROGRAM-ID clause so your program can be recursively re-entered while a previous invocation is still active.

RECURSIVE can be coded only on the outermost program of a compilation unit. Neither programs containing nested subprograms nor nested subprograms can be recursive.

Marking Programs as COMMON

Use the COMMON attribute with the PROGRAM-ID clause so your program can be called by the containing program or by any program in the containing program. However, the COMMON program cannot be called by any program contained in itself. Only contained programs can have the COMMON attribute. For more information, see “Structure of Nested Programs” on page 373.

Marking Programs as INITIAL

Use the INITIAL attribute to specify that whenever a program is called, it is placed in its initial state, and any of its contained programs are also placed in their initial states.

Definition of Initial State: Essentially, a program is in its initial state when data items having VALUE clauses are set to the specified value, changed GO TO statements and PERFORM statements are set to their initial states, and non-EXTERNAL files are closed.

Avoiding Mismatches Between Names

To avoid mistakes when the name is case-sensitive, verify that the appropriate setting of the PGMNAME option is used.

Changing Header of Source Listing

The header on the first page of your source statement listing contains the identification of the compiler and the current release level, plus the date and time of compilation and the page number. For example:

```
PP 5639-B92 IBM VisualAge COBOL (OS/2)  2.2            Date 02/27/1998  Time 15:05:19   Page     1
```

The header indicates the compilation platform used as either OS/2 or Windows. (Throughout this book, all sample headers show OS/2 as being the compilation platform.)

You can customize the header on succeeding pages of the listing with the compiler-directing TITLE statement. See the *IBM COBOL Language Reference* for details of the TITLE statement.

ENVIRONMENT DIVISION

In the ENVIRONMENT DIVISION you can describe those aspects of your program that are dependent upon the characteristics of the computing environment in which you are working.

CONFIGURATION SECTION

You can use the CONFIGURATION SECTION to describe the computer for compiling your program (in the SOURCE-COMPUTER paragraph); describe the computer for running your program (in the OBJECT-COMPUTER paragraph); and specify such items as the currency sign, symbolic characters (in the SPECIAL-NAMES paragraph), and user-defined classes (in the REPOSITORY paragraph). Figure 3 on page 15 shows a sample of some of the entries you might include in the CONFIGURATION SECTION.

Specify the Collating Sequence

Using the PROGRAM COLLATING SEQUENCE clause and the ALPHABET clause of the SPECIAL-NAMES paragraph, you can establish the collating sequence used in the following operations:

- Non-numeric comparisons explicitly specified in relation conditions and condition-name conditions
- HIGH-VALUE and LOW-VALUE settings
- SEARCH ALL
- SORT and MERGE unless overridden by a COLLATING SEQUENCE phrase on the SORT or MERGE statement.

The sequence you use can be based on one of these alphabets:

- NATIVE

  NATIVE is the collating sequence specified by the locale setting. The locale setting refers to the national language locale name in effect at compile time. It is usually set at installation. See “Locale Sensitivity” on page 486 for more information about locale sensitivity.

- EBCDIC
- ASCII (use NATIVE if the native character set is ASCII, STANDARD-1[2] if it is not).
- ISO 7-bit code[3], International Reference Version (use STANDARD-2).
- An alteration of the ASCII sequence that you define in the SPECIAL-NAMES paragraph.

You can also specify a collating sequence of your own definition.

Caution: If the code page is DBCS the ALPHABET-NAME clause is not allowed.

Specifying Collating Sequence Example: Figure 3 on page 15 shows the ENVIRONMENT DIVISION coding used to specify a collating sequence where uppercase and lowercase letters are similarly handled for comparisons and for sorting or merging. When you change the ASCII sequence in the SPECIAL-NAMES paragraph, the overall collating sequence is affected, not just the collating sequence of the characters included in the SPECIAL-NAMES paragraph.

[2] STANDARD-1 refers to *American National Standard X3.4, Code for Information Interchange*.

[3] ISO 7-bit code, as defined in *International 646, 7-Bit Coded Character Set for Information Processing Interchange, International Reference*.

```
Identification Division.
.
.
Environment Division.
  Configuration Section.
   Object-Computer.
      Program Collating Sequence Special-Sequence.
   Special-Names.
      Alphabet Special-Sequence Is
          "A" Also "a"
          "B" Also "b"
          "C" Also "c"
          "D" Also "d"
          "E" Also "e"
          "F" Also "f"
          "G" Also "g"
          "H" Also "h"
          "I" Also "i"
          "J" Also "j"
          "K" Also "k"
          "L" Also "l"
          "M" Also "m"
          "N" Also "n"
          "O" Also "o"
          "P" Also "p"
          "Q" Also "q"
          "R" Also "r"
          "S" Also "s"
          "T" Also "t"
          "U" Also "u"
          "V" Also "v"
          "W" Also "w"
          "X" Also "x"
          "Y" Also "y"
          "Z" Also "z".
```

Figure 3. Example of an Alternate Collating Sequence

Define Symbolic Characters

Use the SYMBOLIC CHARACTER clause to give symbolic names to any character of the specified alphabet. For example, to give a name to the plus character (X'2B' in the ASCII alphabet) code:

```
SYMBOLIC CHARACTERS PLUS IS 44
```

Use ordinal position to identify the character; position 1 corresponds to character X'00'.

Define a User-Defined Class

Use the CLASS clause to give a name to a set of characters listed in the clause. For example, name the set of digits by coding:

```
CLASS DIGIT IS "0" THROUGH "9"
```

ENVIRONMENT DIVISION

The class name can only be referenced in a class condition. (This user-defined class is not the same concept as an object-oriented class.)

INPUT-OUTPUT SECTION:

Your IBM VisualAge COBOL programs can process files with line sequential, sequential, indexed, or relative organization.

Use the FILE-CONTROL and I-O-CONTROL paragraphs to:

- Identify and describe the characteristics of your program files.
- Associate your files with the corresponding system file name, directly or indirectly.
- Optionally identify the file system (for example, VSAM or STL file system) associated with the file. You can also do this at program execution time.
- Provide information on how the file is accessed.

FILE-CONTROL Paragraph

The FILE-CONTROL paragraph associates each file in the COBOL program with a physical file known to your file system. Figure 4 shows an example of a FILE-CONTROL paragraph for a VSAM indexed file.

FILE-CONTROL Entries for a VSAM Indexed File

```
SELECT COMMUTER-FILE  1
  ASSIGN TO COMMUTER  2
  ORGANIZATION IS INDEXED  3
  ACCESS IS RANDOM  4
  RECORD KEY IS COMMUTER-KEY  5
  FILE STATUS IS  5
    COMMUTER-FILE-STATUS
    COMMUTER-VSAM-STATUS.
```

Figure 4. Example of a FILE-CONTROL Paragraph

1 The SELECT clause chooses a file in the COBOL program to be associated with a corresponding system file.

2 The ASSIGN clause associates the program's name for the file with the name of the file as known to the system. COMMUTER may be the system file name or the name of the environment variable whose value (at run time) is used as the system file name with optional directory and path names.

3 Use the ORGANIZATION clause to describe the file's organization. If omitted, the default is ORGANIZATION IS SEQUENTIAL.

4 Use the ACCESS MODE clause to define the manner in which the records in the file will be made available for processing—sequential, random, or dynamic. If you omit this clause, ACCESS IS SEQUENTIAL is assumed.

5 You might have additional statements in the FILE-CONTROL paragraph depending on the type of file and file system you use. See the *IBM COBOL Language Reference* for more information about the FILE-CONTROL paragraph.

Chapter 7, “Processing Files” on page 91 provides a general overview on files and file processing.

Identifying Files to the Operating System

As stated in the previous example, the ASSIGN clause associates the program's name for a file with the name of the file as known to the operating system.

You can use either an environment variable, a system file name, a literal, or a data name in the ASSIGN clause. If you specify an environment variable, its value is evaluated at run time and is used as the system file name with optional directory and path names.

If you plan to use a file system other than the default file system, you need to select the file system explicitly, for example, by specifying the file system identifier before the system file name. For example, if the file MYFILE is a Btrieve file and you use F1 as the file's name in your program, the ASSIGN clause would be

```
SELECT F1 ASSIGN TO BTR-MYFILE
```

Note that this assumes that MYFILE is a system file name and not an environment variable. If MYFILE is an environment variable, then its value will be used. For example, if it is set to MYFILE=VSAM-YOURFILE, the system file name in the ASSIGN clause becomes YOURFILE at run time, and the file is treated as a VSAM file, overriding the file system ID used in ASSIGN clause in the program.

Vary the Input/Output File at Run Time

The *file-name* you code in your SELECT sentence is used as a constant throughout your COBOL program, while the name of the file in your SET command can be associated with a different file at run time.

Changing a *file-name* in your COBOL program requires changing input/output statements and recompiling the program. In contrast, you can change the *assignment-name* in your SET command.

Example of Using Different Input Files: As an example, consider a COBOL program that might be used in exactly the same way for several different master files. It contains this SELECT sentence:

```
SELECT MASTER
    ASSIGN TO MASTERA
```

For example, if you are accessing both checking and savings files using the same MASTER file, you can set the MASTERA environment variable prior to the program execution as follows:

```
set MASTERA=d:\accounts\checking
```

to access the file named checking in the d:\accounts drive and directory and

```
set MASTERA=d:\accounts\savings
```

to do the same for the file named `savings`

The same program can be used to access both `checking` and `savings` files by way of the COBOL `MASTER` file without source program changes or recompilation.

Environment variable values in effect at the time of the program invocation are used for associating COBOL file names to the system file names (including any drive and path specifications).

DATA DIVISION

Define the characteristics of your data and group your data definitions into one of the sections in the DATA DIVISION:

- Define data used in input/output operations in the FILE SECTION (discussed in "FILE SECTION (Using Data in Input/Output Operations)").
- Define data developed for internal processing:
  - To be statically allocated and exist for the life of the run-unit: WORKING-STORAGE SECTION (discussed on page 19).
  - To be allocated each time a program is called and deallocated when the program ends: LOCAL-STORAGE SECTION (discussed on page 20).
- Describe data from another program in the LINKAGE SECTION (discussed on page 21).

Limits in the DATA DIVISION

The IBM VisualAge COBOL compiler limits the maximum size of data division elements. For a complete list of these compiler limits, see *IBM COBOL Language Reference*.

FILE SECTION (Using Data in Input/Output Operations)

Define the data you use in input and output operations in the FILE SECTION:

- Name the input and output files your program will use.

  Use the FD entry to give names to your files that the input/output statements in the PROCEDURE DIVISION can refer to.

  Caution: Data items defined in the FILE SECTION are not available to PROCEDURE DIVISION statements until the file has been successfully opened.
- In the record description following the FD entry describe the records and their fields in the file. The *record-name* established is the object of WRITE and REWRITE statements.

Function and Use of FILE SECTION Entries

Entries in the FILE SECTION are summarized in Figure 5.

Figure 5. FILE SECTION Entries

Clause	To Define
FD	The *file-name* to be referred to in PROCEDURE DIVISION input/output statements (OPEN, CLOSE, READ, START, and DELETE). Must match *file-name* in the SELECT clause. *file-name* is associated with the system file through the *assignment-name*.
RECORD CONTAINS *n*	Size of logical records (fixed length)
RECORD IS VARYING	Size of logical records (variable length)
RECORD CONTAINS *n* TO *m*	Size of logical records (variable length)
VALUE OF	An item in the label records associated with file. Comments only.
DATA RECORDS	Names of records associated with file. Comments only.
RECORDING MODE	Record type for sequential files.

Sharing Files Using the EXTERNAL and GLOBAL Clauses

Programs in the same run unit can refer to the same COBOL file names. You can also share physical files without using external or global file definitions in COBOL source programs.

For example, if you specify:

```
SELECT F1 ASSIGN TO MYFILE.
SELECT F2 ASSIGN TO MYFILE.
```

The application has two COBOL file names, but these COBOL files are associated with one system file.

EXTERNAL: Is used for separately compiled programs. A file that is defined as EXTERNAL can be referenced by any program in the run unit that describes the file. See “Sharing Files between Programs (EXTERNAL Files)” on page 399 for an example.

GLOBAL: Is used for programs in a nested, or contained, structure. If a program contains another program (directly or indirectly), both programs can access a common file by referencing a GLOBAL file name. For more information on contained programs and the GLOBAL clause, see “Structure of Nested Programs” on page 373.

WORKING-STORAGE SECTION and LOCAL-STORAGE SECTION

You can write a program that processes data without performing any input/output operations. All the data would be defined in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION.

Most programs, however, have a combination of input and output file processing and internal data manipulation; the data files are defined in the FILE SECTION, and the data developed by the program is defined in the WORKING-STORAGE SECTION or LOCAL-STORAGE section.

DATA DIVISION

What is the WORKING-STORAGE SECTION?

When a program is invoked, the WORKING STORAGE associated with the program is allocated. Any data items with VALUE clauses are initialized to the appropriate value at that time. For the duration of the run-unit, Working-Storage items persist in their last-used state. Exceptions are:

- A program with INITIAL specified on the PROGRAM-ID.

  In this case, WORKING-STORAGE data items are reinitialized each time the program is entered.

- A subprogram that is called and then cancelled.

  In this case, WORKING-STORAGE DATA items are reinitialized on the first reentry into the program following the CANCEL.

Working-Storage is deallocated at the termination of the run-unit.

What is the LOCAL-STORAGE SECTION?

Local-Storage is allocated each time the program is called and is deallocated when the program returns via an EXIT PROGRAM, GOBACK, or STOP RUN. Any data items with VALUE clauses are initialized to the appropriate value each time the program is called. The value in the data items is lost when the program returns.

Storage Sections Example

The following is an example of a recursive program that uses both Working-Storage and Local-Storage.

```
CBL apost,pgmn(lu)
*********************************
* Recursive Program - Factorials
*********************************
 Identification Division.
 Program-Id. factorial recursive.
 Environment Division.
 Data Division.
 Working-Storage Section.
 01  numb  pic 9(4)  value 5.
 01  fact  pic 9(8)  value 0.
 Local-Storage Section.
 01  num  pic 9(4).
 Procedure Division.
     move numb to num.

     if numb = 0
        move 1 to fact
     else
        subtract 1 from numb
        call 'factorial'
        multiply num by fact
     end-if.

     display num '! = ' fact.
     goback.
 End Program factorial.
```

Figure 6. Storage Sections Example

```
Recursive
 CALL's:  Main  1   2   3   4   5
-----------------------------------
L-S   num   5   4   3   2   1   0
-----------------------------------
W-S  numb   5   4   3   2   1   0
     fact   0   0   0   0   0   0
-----------------------------------

Recursive
GOBACK's:   5   4   3   2   1  Main
-----------------------------------
L-S   num   0   1   2   3   4   5
-----------------------------------
W-S  numb   0   0   0   0   0   0
     fact   1   1   2   6   24  120
-----------------------------------
```

LINKAGE SECTION (Using Data from Another Program)

How you share data depends on whether the programs are separately compiled or are nested.

Separately Compiled Programs

Many times an application's solution consists of many separately compiled programs that call and pass data to one another. The LINKAGE SECTION in the called program describes the data passed from another program. The calling program must use a CALL ... USING or INVOKE ... USING statement to pass the data. For details on using data from other programs, see “Passing Data” on page 387.

Nested Programs

An application's solution might consist of nested programs—programs that are contained in other programs. Level-01 LINKAGE SECTION data items can include the GLOBAL attribute. This allows any nested program that includes the declarations to access these LINKAGE SECTION data items.

A nested program can also access data items in a sibling program (one at the same nesting level in the same containing program) that is declared with the COMMON attribute. For more details, see “Structure of Nested Programs” on page 373.

With Recursion or Multithreading

If you compile your program as recursive or with the THREAD option, data defined in the LINKAGE SECTION may not be accessible between entries.

The ability to address a record in the LINKAGE SECTION is established by:

- Passing an argument to the program and specifying the record in an appropriate position in the USING phrase in the program *or*
- Using the Format 5 SET statement.

If you compile your program as recursive or with the THREAD option, the address to that record is valid for the particular instance of the program invocation. The address to the record in another execution instance of the same program must be re-established for that execution instance. Unpredictable results will occur if reference is made to a data item whose address has not been established.

PROCEDURE DIVISION

In the PROCEDURE DIVISION of a program you code the executable statements that process the data you have defined in the other divisions. The PROCEDURE DIVISION contains one or two headers and the logic of your program.

PROCEDURE DIVISION Headers

The PROCEDURE DIVISION begins with the division header and a *procedure-name* header. The division header for a program can simply be:

```
PROCEDURE DIVISION.
```

Or, you can code your division header to receive parameters with the USING phrase or to return a value with the RETURNING Phrase.

USING Phrase

To receive an argument that was passed by reference (the default) or by content, code the division header for a program like this:

```
PROCEDURE DIVISION USING dataname
```

Or this:

```
PROCEDURE DIVISION USING BY REFERENCE dataname
```

Take Note: *dataname* in these examples would need to be defined in the LINKAGE SECTION of the program.

To receive a parameter that was passed by value, code the division header for a program like this:

```
PROCEDURE DIVISION USING BY VALUE dataname
```

See “Passing Data” on page 387 for more information on BY VALUE.

RETURNING Phrase

To return a value as a result, code the division header like this:

```
Procedure Division RETURNING dataname2
```

You can also combine USING AND RETURNING in a PROCEDURE DIVISION header:

```
Procedure Division USING dataname RETURNING dataname2
```

Take Note: *dataname* and *dataname2* in these examples would need to be defined in the LINKAGE SECTION.

How Logic is Divided in the PROCEDURE DIVISION

The PROCEDURE DIVISION of a program is divided into sections, paragraphs, sentences, and statements:

Section Logical subdivision of your processing logic.

A section can contain several paragraphs.

A section can be the subject of the PERFORM statement.

Paragraph Subdivides a section, procedure, or program.

It contains a set of related statements that provide a function and is one of the basic building blocks of a structured program. A paragraph can be the subject of a statement.

Sentence Series of one or more COBOL statements ending with a period.

Many structured programs do not have separate sentences. Each paragraph can contain one sentence. Using scope terminators instead of periods to show the logical end of a statement is preferred. Scope terminators, both explicit and implicit, are discussed beginning on page 25.

PROCEDURE DIVISION

Statement Performs a defined step of COBOL processing, such as adding two numbers.

A statement is a valid combination of words, beginning with a COBOL verb.

Statements Used in the PROCEDURE DIVISION

In the COBOL language, statements are imperative (indicating unconditional action), conditional, or compiler-directing.

Imperative and conditional statements can be ended implicitly or explicitly. If you end a conditional statement explicitly, it becomes a delimited scope statement (which is an imperative statement).

Imperative Statements

An imperative statement indicates that an unconditional action is to be taken. Examples are ADD, MOVE, INVOKE, and CLOSE. A full list of imperative statements can be found in *IBM COBOL Language Reference*.

Conditional Statements

A conditional statement is either a simple conditional statement (IF, EVALUATE, SEARCH) or a conditional statement made up of an imperative statement that includes a conditional phrase or option.

Examples of Conditional Phrases: For example, an arithmetic statement without ON SIZE ERROR is an imperative statement. But an arithmetic statement with the conditional option ON SIZE ERROR and without a scope terminator is a conditional statement.

The following are examples of conditional statements if they are coded without scope terminators:

- Data-manipulation statements with ON OVERFLOW.
- CALL statements with ON OVERFLOW.
- I/O statements with INVALID KEY, AT END, AT END-OF-PAGE.
- RETURN with AT END.

Using the NOT Phrase: For additional program control, the NOT phrase can also be used with conditional statements. For example, you can provide instructions to be performed when a particular exception does not occur, such as NOT ON SIZE ERROR. The NOT phrase cannot be used with the ON OVERFLOW phrase of the CALL statement, but it can be used with the ON EXCEPTION phrase.

Do Not Nest Conditional Statements: An unterminated conditional statement cannot be contained by (nested within) another statement. Except for nesting statements within IF statements, nested statements must be imperative statements and must follow the rules for imperative statements.

Compiler-Directing Statements

A compiler-directing statement is not part of the program logic. A compiler-directing statement causes the compiler to take specific action about the program structure, COPY processing, listing control, control flow, or CALL interface convention.

A description of compiler-directing statements can be found in *IBM COBOL Language Reference*. See “Compiler-Directing Statements” on page 202 for usage notes.

Explicit Scope Terminators

Explicit scope terminators end certain conditional and imperative forms of PROCEDURE DIVISION statements. Use an explicit scope terminator to make a conditional statement imperative (see “Delimited Scope Statements” on page 26). Or use an explicit scope terminator to clearly end an imperative statement. Explicit scope terminators are provided for certain COBOL verbs, such as scope terminator END-IF for the IF verb, and can be found in *IBM COBOL Language Reference*.

Example of Using Explicit Scope Terminators

```
MOVE 0 TO TOTAL
PERFORM UNTIL X = 10
  ADD 1 TO TOTAL
  IF X = 5
    DISPLAY "HALFWAY THROUGH"
    DISPLAY "TOTAL IS " TOTAL
  END-IF
  ADD 1 TO X
END-PERFORM
DISPLAY "FINAL TOTAL IS " TOTAL
```

Implicit Scope Terminators

An implicit scope terminator is a period (.) that ends the scope of all previous statements not yet ended.

Example of Using Implicitly Terminated Statements:

```
IF CAT
   DISPLAY "It is a cat."
ELSE
   IF DOG
      DISPLAY "It is a dog."
   ELSE
      DISPLAY "It is not a dog or cat.".
```

Each of the two periods in the above program fragment end the IF statements, making the code equivalent to the following example that has explicit scope terminators:

```
IF CAT
   DISPLAY "It is a cat."
ELSE
   IF DOG
      DISPLAY "It is a dog."
   ELSE
      DISPLAY "It is not a dog or cat."
   END-IF
END-IF
```

If you use implicit terminators, it can be unclear where statements end. As a result, you might end statements unintentionally, changing your program's logic. Explicit scope terminators make a program easier to understand and prevent unintentional ending of statements. For example, in the program fragment below, changing the location of the first period in the first implicit scope example changes the meaning of the code:

```
IF ITEM = "A"
  DISPLAY "VALUE OF ITEM IS " ITEM
  ADD 1 TO TOTAL.
  MOVE "C" TO ITEM
  DISPLAY " VALUE OF ITEM IS NOW " ITEM
IF ITEM = "B"
  ADD 2 TO TOTAL.
```

The two statements:

```
MOVE "C" TO ITEM
DISPLAY " VALUE OF ITEM IS NOW " ITEM
```

will be performed regardless of the value of ITEM, despite what the indentation indicates, because the first period terminates the IF statement. For improved program clarity and to avoid unintentional ending of statements, you should use explicit scope terminators instead of implicit scope terminators, especially within paragraphs. Use implicit scope terminators only at the end of a paragraph or the end of the program.

Delimited Scope Statements

A delimited scope statement is a conditional statement ended with an explicit scope terminator. A delimited scope statement can be used in these ways:

- To delimit the range of operation for a COBOL conditional statement and to explicitly show the levels of nesting.

  For example, use an END-IF statement instead of a period to end the scope of an IF statement within a nested IF.

- To code a conditional statement where the COBOL syntax calls for an imperative statement.

  For example, code a conditional statement as the object of an inline PERFORM:

```
PERFORM UNTIL TRANSACTION-EOF
  PERFORM 200-EDIT-UPDATE-TRANSACTION
  IF NO-ERRORS
    PERFORM 300-UPDATE-COMMUTER-RECORD
  ELSE
    PERFORM 400-PRINT-TRANSACTION-ERRORS
  END-IF
  READ UPDATE-TRANSACTION-FILE INTO WS-TRANSACTION-RECORD
    AT END
      SET TRANSACTION-EOF TO TRUE
  END-READ
END-PERFORM
```

An explicit scope terminator is required for the inline PERFORM statement, but it is not valid for the out-of-line PERFORM statement.

Rules for Delimited Scope Statements: Because a period implicitly ends the scope of all previous statements, do not use a period in a delimited scope statement.

In general, a delimited scope statement can be coded wherever an imperative statement is allowed by language rules.

Using Nested Delimited Scope Statements: When nested within another delimited scope statement with the same verb, each explicit scope terminator ends the statement begun by the most recently preceding (and as yet unpaired) occurrence of that verb.

Be careful when coding an explicit scope terminator for an imperative statement that is nested within a conditional statement. Ensure that the scope terminator is paired with the statement for which it was intended. In the following example, the scope terminator will be paired with the second READ statement, though the programmer intended it to be paired with the first.

```
READ FILE1
  AT END
     MOVE A TO B
     READ FILE2
END-READ
```

To ensure that the explicit scope terminator is paired with the intended statement, the preceding example can be recoded in this way:

```
READ FILE1
  AT END
     MOVE A TO B
     READ FILE2
     END-READ
END-READ
```

Declaratives

Declaratives provide one or more special-purpose sections that are executed when an exceptional condition occurs.

PROCEDURE DIVISION

Each Declarative Section starts with a USE statement that identifies the function of the section; the series of procedures that follow specify what actions are to be taken when the condition occurs. See the *IBM COBOL Language Reference* for a complete description of declaratives and Chapter 13, “Debugging Techniques” on page 244 and “Input/Output Error Handling Techniques” on page 123 for instances of their use.

Chapter 3. Numbers and Arithmetic

This chapter explains how COBOL views numeric data and how you can best represent numeric data and perform efficient arithmetic operations. The topics are:

General COBOL View of Numbers (PICTURE clause)

In general, you can view COBOL numeric data in a way similar to character-string data—as a series of decimal digit positions. However, numeric items can have special properties, such as an arithmetic sign.

Defining Numeric Items

Define numeric items using the character "9" in the data description to represent the decimal digits of the number instead of using an "x" like with alphanumeric items:

```
05  Count-x           Pic 9(4)   Value 25.
05  Customer-name     Pic x(20)  Value "Johnson".
```

You can code up to 18 digits in the PICTURE clause, as well as various other characters of special significance. The "s" in the following example means that the value is signed:

```
05  Price             Pic s99v99.
```

The field can hold a positive or negative value. The "v" indicates the position of an implied decimal point. Neither "s" nor "v" are counted in the size of the item, nor do they require extra storage positions, unless the item is coded as USAGE DISPLAY with the SIGN IS SEPARATE clause. An exception is internal floating point data (COMP-1 or COMP-2), for which there is no PICTURE clause.

Separate Sign Position (for Portability)

If you plan to port your program or data to a different machine, you might want to code the sign as a separate digit position in storage:

```
05  Price             Pic S99V99   Sign Is Leading, Separate.
```

This ensures that the convention your machine uses for storing a non-separate sign will not cause strange results when you use a machine that uses a different convention.

How COBOL Stores Your Numbers

Extra Positions for Displayable Symbols (Numeric Editing)

You can also define numeric items with certain editing symbols (such as decimal points, commas, and dollar signs) to make the data easier to read and understand when displayed or printed on reports. For example:

```
05  Price           Pic   9(5)v99.
05  Edited-price    Pic  $zz,zz9v99.
  ⋮
    Move Price To Edited-price
    Display Edited-price
```

If the contents of `Price` were 0150099 (representing the value 1,500.99), then `$ 1,500.99` would be displayed after the code is run.

How to Use Numeric-Edited Items as Numbers

Numeric-edited items are classified as alphanumeric data items, not as numbers. Therefore, they cannot be operands in arithmetic expressions or ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE statements.

Numeric-edited items can be moved to numeric and numeric-edited items. In the following example, the numeric-edited item is *de-edited* and its numeric value is moved to the numeric data item.

```
Move Edited-price to Price
Display Price
```

If these two statements were to immediately follow the statements shown in the previous example, then `Price` would be displayed as 0150099, representing the value 1,500.99.

For complete information on the data descriptions for numeric data, refer to *IBM COBOL Language Reference*.

Computational Data Representation (USAGE Clause)

Control how the computer internally stores your numeric data items by coding the USAGE clause in your data description entries. The numeric data you use in your program will be one of the formats available with COBOL:

External decimal (USAGE DISPLAY)
External floating-point (USAGE DISPLAY)
Internal decimal (USAGE PACKED-DECIMAL)
Binary (USAGE BINARY)
Binary (COMP-5).
Internal floating-point (USAGE COMP-1, USAGE COMP-2)

COMP and COMP-4 are synonymous with BINARY, and COMP-3 is synonymous with PACKED-DECIMAL.

How COBOL Stores Your Numbers

Regardless of what USAGE clause you use to control the computer's internal representation of the value, you use the same PICTURE clause conventions and decimal value in the VALUE clause except for floating point data.

External Decimal (USAGE DISPLAY) Items

When you code USAGE DISPLAY or omit the USAGE clause, each position (or byte) of storage contains one decimal digit. This corresponds to the format used for printing or displaying output, meaning the items are stored in displayable form.

What USAGE DISPLAY Items Are For

External decimal items are primarily intended for receiving and sending numbers between your program and files, terminals, and printers. However, it is also acceptable to use external decimal items as operands and receivers in your program's arithmetic processing, and it is often convenient to program this way.

Should You Use Them for Arithmetic

If your program performs a lot of intensive arithmetic and efficiency is a high priority, you might want to use one of COBOL's computational numeric data types for the data items used in the arithmetic.

The compiler has to automatically convert displayable numbers to the *internal* representation of their numeric value before they can be used in arithmetic operations. Therefore, it is often more efficient to define your data items as computational items to begin with, rather than as DISPLAY items. For example:

```
05  Count-x           Pic s9v9(5)  Usage Comp  Value 3.14159.
```

External Floating-Point (USAGE DISPLAY) Items

Displayable numbers coded in a floating-point format are called *external floating-point items*. Like external decimal items, you define external floating-point items explicitly with USAGE DISPLAY or implicitly by omitting the USAGE clause.

In the following example, `Compute-Result` is implicitly defined as an external floating-point item. Each byte of storage contains one character (except for V).

```
05  Compute-Result     Pic -9v9(9)E-99.
```

The VALUE clause is not allowed in the data description for external floating-point items. Also, the minus signs (-) do not mean that the mantissa and exponent will always be negative numbers, but that when displayed the sign will appear as a blank for positive and a minus sign for negative. If a plus sign (+) were used, positive would be displayed as a plus sign and negative as a minus sign.

Just as with external decimal numbers, external floating-point numbers have to be converted (automatically by the compiler) to an internal representation of the numeric value before they can be operated on.

How COBOL Stores Your Numbers

Binary Items

BINARY, COMP, and COMP-4 are synonyms on all platforms. COMP-5 is a new USAGE type based on the X/OPEN COBOL specification.

Binary format occupies 2, 4, or 8 bytes of storage and is handled for arithmetic purposes as a fixed-point number with the leftmost bit being the operational sign. For byte-reversed binary data, the sign bit is the leftmost bit of the rightmost byte.

How Much Storage BINARY Occupies

A PICTURE description with 4 or fewer decimal digits occupies 2 bytes; with 5 to 9 decimal digits, 4 bytes; with 10 to 18 decimal digits, 8 bytes.

Why Use Binary

Binary items can, for example, contain subscripts, switches, and arithmetic operands or results.

However, you might want to use packed decimal format instead of binary because:

- Binary format might not be as well suited for decimal alignment as packed decimal format.
- Binary format is not converted to and from DISPLAY format as easily as packed decimal format.

Truncation of Binary Data (TRUNC Compiler Option)

Use the TRUNC(STD|OPT|BIN) compiler option (described in “TRUNC” on page 195) to indicate how binary data (BINARY, COMP, and COMP-4) is truncated.

COMP-5 Note: COMP-5 data is truncated according to TRUNC(BIN) regardless of which suboption of TRUNC you set.

Byte-Reversal of Binary Data (BINARY Compiler Option)

On the PC you sometimes need to be concerned with byte reversal. How binary data is stored depends on the platform you're running under or the products you're using.

For example, Intel platforms by default store binary data in *little-endian* format (most significant digit is on the highest address). System/390 and AIX by default store binary data in *big-endian* format (least significant digit is on the highest address).

The BINARY(NATIVE|S390) compiler option (described in “BINARY” on page 163) allows you to indicate whether binary data types BINARY, COMP, and COMP-4 are to be stored in big-endian or little-endian format.

COMP-5 is handled as the native binary data format regardless of the BINARY(NATIVE|S390) option setting.

COMP-5 is the recommended data type to use when interfacing with other languages (such as C or C++) or other products (such as CICS or DB2) that assume native binary data formats. However, a SORT or MERGE statement must not contain both big-endian and little-endian binary keys. That is, if the BINARY(S390) option is in effect and a SORT or MERGE key is a COMP-5 data item, no other SORT or MERGE key may be a COMP, BINARY, or COMP-4 data item.

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

Packed decimal format occupies 1 byte of storage for every two decimal digits you code in the PICTURE description, except that the right-most byte contains only 1 digit and the sign. Packed decimal format is handled as a fixed-point number for arithmetic purposes.

Why Use Packed Decimal

- Packed decimal format requires less storage per digit than DISPLAY format requires.
- Packed decimal format might be better suited for decimal alignment than binary format.
- Packed decimal format is converted to and from DISPLAY format more easily than binary format.
- Packed decimal format can, for example, contain arithmetic operands or results.

Floating-Point (COMP-1 and COMP-2) Items

COMP-1 refers to short (single-precision) floating-point format, and COMP-2 refers to long (double-precision) floating-point format, which occupy 4 and 8 bytes of storage, respectively.

On the PC, COMP-1 and COMP-2 data items are represented in IEEE format if the FLOAT(NATIVE) compiler option is in effect. See "FLOAT" on page 180 for additional information.

A PICTURE clause is not allowed in the data description of floating-point data items, but you can provide an initial value using a floating-point literal in the VALUE clause:

```
05  Compute-result    Usage Comp-1  Value 06.23E-24.
```

The characteristics of conversions between floating-point format and other number formats are discussed in the next section, "Data Format Conversions" on page 35.

Floating-point format is well suited for containing arithmetic operands and results and for maintaining the highest level of accuracy in arithmetic.

For complete information on the data descriptions for numeric data, see *IBM COBOL Language Reference.*

How COBOL Stores Your Numbers

Figure 7. Internal Representation of Numeric Items—Native Data Types. This table assumes that the BINARY(NATIVE), CHAR(NATIVE), and FLOAT(NATIVE) compiler options are in effect.

Numeric Type	PICTURE and USAGE and Optional SIGN Clause	Value	Internal Representation
External Decimal	`PIC S9999 DISPLAY`	+ 1234	31 32 33 34
		- 1234	71 32 33 34
		1234	31 32 33 34
	`PIC 9999 DISPLAY`	1234	31 32 33 34
	`PIC S9999 DISPLAY`	+ 1234	31 32 33 34
	`SIGN LEADING`	- 1234	71 32 33 34
	`PIC S9999 DISPLAY`	+ 1234	2B 31 32 33 34
	`SIGN LEADING SEPARATE`	- 1234	2D 31 32 33 34
	`PIC S9999 DISPLAY`	+ 1234	31 32 33 34 2B
	`SIGN TRAILING SEPARATE`	- 1234	31 32 33 34 2D
Binary	`PIC S9999 BINARY`	+ 1234	D2 04
	`COMP`	- 1234	2E FB
	`COMP-4`		
	`COMP-5`	+ 1234	D2 04
		- 1234	2E FB
	`PIC 9999 BINARY`	1234	D2 04
	`COMP`		
	`COMP-4`		
	`COMP-5`	1234	D2 04
Internal Decimal	`PIC S9999 PACKED-DECIMAL`	+ 1234	01 23 4C
	`COMP-3`	- 1234	01 23 4D
	`PIC 9999 PACKED-DECIMAL`	1234	01 23 4F
	`COMP-3`		
Internal Floating Point	`COMP-1`	+ 1234	00 40 9A 44
		- 1234	00 40 9A C4
Internal Floating Point	`COMP-2`	+ 1234	00 00 00 00 00 48 93 40
		- 1234	00 00 00 00 00 48 93 C0
External Floating Point	`PIC +9(2).9(2)E+99 DISPLAY`	+ 1234	2B 31 32 2E 33 34 45 2B 30 32
		- 1234	2D 31 32 2E 33 34 45 2B 30 32

Figure 8. Internal Representation of Numeric Items—System/390 Host Data Types. This table assumes that the BINARY(S390), CHAR(EBCDIC), and FLOAT(HEX) compiler options are in effect.

Numeric Type	PICTURE and USAGE and Optional SIGN Clause	Value	Internal Representation
External Decimal	PIC S9999 DISPLAY	+ 1234 - 1234 1234	F1 F2 F3 C4 F1 F2 F3 D4 F1 F2 F3 C4
	PIC 9999 DISPLAY	1234	F1 F2 F3 F4
	PIC S9999 DISPLAY SIGN LEADING	+ 1234 - 1234	C1 F2 F3 F4 D1 F2 F3 F4
	PIC S9999 DISPLAY SIGN LEADING SEPARATE PIC S9999 DISPLAY SIGN TRAILING SEPARATE	+ 1234 - 1234 + 1234 - 1234	4E F1 F2 F3 F4 60 F1 F2 F3 F4 F1 F2 F3 F4 4E F1 F2 F3 F4 60
Binary	PIC S9999 BINARY COMP COMP-4	+ 1234 - 1234	04 D2 FB 2E
	COMP-5	+ 1234 - 1234	D2 04 2E FB
	PIC 9999 BINARY COMP COMP-4	1234	04 D2
	COMP-5	1234	D2 04
Internal Decimal	PIC S9999 PACKED-DECIMAL COMP-3	+ 1234 - 1234	01 23 4C 01 23 4D
	PIC 9999 PACKED-DECIMAL COMP-3	1234	01 23 4F
Internal Floating Point	COMP-1	+ 1234 - 1234	43 4D 20 00 C3 4D 20 00
Internal Floating Point	COMP-2	+ 1234 - 1234	43 4D 20 00 00 00 00 00 C3 4D 20 00 00 00 00 00
External Floating Point	PIC +9(2).9(2)E+99 DISPLAY	+ 1234	4E F1 F2 4B F3 F4 C5 4E F0 F2
		- 1234	60 F1 F2 4B F3 F4 C5 4E F0 F2

Data Format Conversions

When the code in your program involves the interaction of items with different data formats, the compiler converts these items:

- Temporarily, for comparisons and arithmetic operations.
- Permanently, for assignment to the receiver in a MOVE, COMPUTE, and other arithmetic statement.

Data Format Conversions

When possible, the compiler performs the move to preserve the numeric "value" as opposed to a direct digit-for-digit move. (For more information on truncation and predicting the loss of significant digits, refer to Appendix C, "Intermediate Results and Arithmetic Precision" on page 545.)

Conversion Takes Time

Conversion generally requires additional storage and processing time because data is moved to an internal work area and converted before the operation is performed. The results might also have to be moved back into a work area and converted again.

Conversions and Precision

Conversions between fixed-point data formats (external decimal, packed decimal, and binary) are completed without loss of precision, as long as the target field can contain all the digits of the source operand.

Conversions Where Loss of Precision Is Possible

A loss of precision is possible in conversions between fixed-point data formats and floating-point data formats (short floating-point, long floating-point, and external floating-point). These conversions happen during arithmetic evaluations that have a mixture of both fixed-point and floating-point operands. (Because fixed-point and external floating-point items both have decimal characteristics, reference to fixed-point items in the following examples includes external floating-point items as well, unless stated otherwise.)

When converting from fixed-point to internal floating-point format, fixed-point numbers in base 10 are converted to the numbering system used internally, base 16.

Although the compiler converts short form to long form for comparisons, zeros are used for padding the short number.

When a USAGE COMP-1 data item is moved to a fixed-point data item with more than 6 digits, the fixed-point data item will receive only 6 significant digits, and the remaining digits will be zero.

Conversions that Preserve Precision: If a fixed-point data item with 6 or fewer digits is moved to a USAGE COMP-1 data item and then returned to the fixed-point data item, the original value is recovered.

If a USAGE COMP-1 data item is moved to a fixed-point data item of 6 or more digits and then returned to the USAGE COMP-1 data item, the original value is recovered.

If a fixed-point data item with 15 or fewer digits is moved to a USAGE COMP-2 data item and then returned to the fixed-point data item, the original value is recovered.

If a USAGE COMP-2 data item is moved to a fixed-point (not external floating-point) data item of 18 digits and then returned to the USAGE COMP-2 data item, the original value is recovered.

Conversions that Result In Rounding: If a USAGE COMP-1 data item, a USAGE COMP-2 data item, an external floating-point data item, or a floating-point literal is moved to a fixed-point data item, rounding occurs in the low-order position of the target data item.

If a USAGE COMP-2 data item is moved to a USAGE COMP-1 data item, rounding occurs in the low-order position of the target data item.

If a fixed-point data item is moved to an external floating-point data item where the PICTURE of the fixed-point data item contains more digit positions than the PICTURE of the external floating-point data item, rounding occurs in the low-order position of the target data item.

Sign Representation and Processing

Sign representation affects the processing and interaction of your numeric data.

Given X'*sd*', where *s* is the sign representation and *d* represents the digit, the valid sign representations for external decimal (USAGE DISPLAY without the SIGN IS SEPARATE clause) are :

Positive:	0, 1, 2, 3, 8, 9, A, and B.
Negative:	4, 5, 6, 7, C, D, E, and F.

When the CHAR(NATIVE) compiler option is in effect, signs generated internally are 3 for positive and unsigned, and 7 for negative.

When the CHAR(EBCDIC) compiler option is in effect, signs generated internally are C for positive, F for unsigned, and D for negative.

Given X'*ds*', where *d* represents the digit and *s* is the sign representation, the valid sign representations for internal decimal (USAGE PACKED-DECIMAL) COBOL data are:

Positive:	A, C, E, and F.
Negative:	B and D.

Signs generated internally are C for positive, F for unsigned, and D for negative.

Checking for Incompatible Data (Numeric Class Test)

The compiler assumes that the values you supply for a data item are valid for the item's PICTURE and USAGE clauses and assigns the value you supply without checking for validity. When an item is given a value that is incompatible with its data description, references to that item in the PROCEDURE DIVISION will be undefined and your results will be unpredictable.

Frequently, values are passed into your program and assigned to items that have incompatible data descriptions for those values. For example, non-numeric data might be moved or passed into a field in your program that is defined as a numeric item. Or, perhaps a signed number is passed into a field in your program that is defined as an unsigned number. In either case, these fields contain invalid data. Ensure that the contents of a data item conforms to its PICTURE and USAGE clauses before using the data item in any further processing steps.

How to Do a Numeric Class Test

You can use the numeric class test to perform data validation. For example:

```
Linkage Section.
01  Count-x      Pic 999.
    .
    .
Procedure Division Using Count-x.
    If Count-x is numeric then display "Data is good"
    .
    .
```

The numeric class test checks the contents of a data item against a set of values that are valid for the particular PICTURE and USAGE of the data item.

Performing Arithmetic

COBOL provides various language features to perform arithmetic:

- ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE statements (discussed in “COMPUTE and Other Arithmetic Statements”).
- Arithmetic expressions (discussed in “Arithmetic Expressions” on page 39).
- Intrinsic functions (discussed in “Numeric Intrinsic Functions” on page 40).

For the complete details of syntax and usage for COBOL language constructs, refer to *IBM COBOL Language Reference*.

COMPUTE and Other Arithmetic Statements

The general practice is to use the COMPUTE statement for most arithmetic evaluations rather than ADD, SUBTRACT, MULTIPLY, and DIVIDE statements. This is because one COMPUTE statement can often be coded instead of several individual statements.

The COMPUTE statement assigns the result of an arithmetic expression to a data item:

```
Compute z = a + b / c ** d - e
```

or to many data items:

```
Compute x y z = a + b / c ** d - e
```

When to Use Other Arithmetic Statements

Some arithmetic might be more intuitive using the other arithmetic statements. For example:

```
Add 1 To Increment
```

instead of:

```
Compute Increment = Increment + 1
```

Or,

```
Subtract Overdraft From Balance
```

instead of:

```
Compute Balance = Balance - Overdraft
```

Or,

```
Add 1 To Increment-1, Increment-2, Increment-3
```

instead of:

```
Compute Increment-1 = Increment-1 + 1
Compute Increment-2 = Increment-2 + 1
Compute Increment-3 = Increment-3 + 1
```

You might also prefer to use the DIVIDE statement (with its REMAINDER phrase) for division in which you want to process a remainder. The REM intrinsic function also provides the ability to process a remainder. For an example of the REM function, see “Mathematics” on page 43.

Arithmetic Expressions

In the examples of COMPUTE shown above, everything to the right of the equal sign represents an arithmetic expression. Arithmetic expressions can consist of a single numeric literal, a single numeric data item or a single intrinsic function reference. They can also consist of several of these items connected by arithmetic operators. These operators are evaluated in a hierarchic order:

Figure 9. Operator Evaluation

Operator	**Meaning**	**Order of Evaluation**
Unary + or -	Algebraic Sign	First
**	Exponentiation	Second
/ or *	Division or multiplication	Third
Binary + or -	Addition or subtraction	Last

Operators at the same level are evaluated from left to right; however, you can use parentheses with these operators to change the order in which they are evaluated. Expressions in parentheses are evaluated before any of the individual operators are evaluated. Parentheses, necessary or not, make your program easier to read.

In addition to using arithmetic expressions in COMPUTE statements, you can also use them in other places where numeric data items are allowed. For example, you can use arithmetic expressions as comparands in relation conditions:

```
If (a + b) > (c - d + 5) Then...
```

Doing Math

Numeric Intrinsic Functions

Intrinsic functions can return an alphanumeric or numeric value.

Numeric intrinsic functions:

- Return a signed numeric value.
- Are considered to be temporary numeric data items.
- Can be used only in the places in the language syntax where expressions are allowed.
- Can save you time because you don't have to provide the arithmetic for the many common types of calculations that these functions cover.

  For more information on the practical application of intrinsic functions, including examples of their usage, refer to “Intrinsic Function Examples” on page 41.

Types of Numeric Functions

Numeric functions are classified into these categories:

Integer	Those that return an integer
Floating-Point	Those that return a long floating-point value
Mixed	Those that return an integer, a long floating-point value, or a fixed-point number with decimal places, depending on the arguments

The numeric functions available in COBOL under these categories are described in *IBM COBOL Language Reference*.

Nesting Functions and Arithmetic Expressions

Numeric functions can be nested; you can reference one function as the argument of another. A nested function is evaluated independently of the outer function, except when determining whether a mixed function should be evaluated with fixed-point or floating-point procedures.

Because numeric functions and arithmetic expressions hold similar status syntactically speaking, you can also nest an arithmetic expression as an argument to a numeric function:

```
Compute x = Function Sum(a b (c / d))
```

In this example, there are only three function arguments: a, b and the arithmetic expression (c / d).

ALL Subscripting and Special Registers

Two other useful features of intrinsic functions are the ALL subscript and special registers:

- You can reference all the elements of an array as function arguments by using the ALL subscript. This feature is used with tables, and examples of its use are shown under “Processing Table Items (Intrinsic Functions)” on page 63.

- The integer-type special registers are allowed as arguments wherever integer arguments are allowed.

Intrinsic Function Examples

You can use intrinsic functions to perform several different kinds of arithmetic, as outlined in Figure 10.

Figure 10. Types of Arithmetic that Numeric Intrinsic Functions Handle

Number Handling	Date/Time	Finance	Mathematics	Statistics
LENGTH	CURRENT-DATE	ANNUITY	ACOS	MEAN
MAX	DATE-OF-INTEGER	PRESENT-VALUE	ASIN	MEDIAN
MIN	DATE-TO-YYYYMMDD		ATAN	MIDRANGE
NUMVAL	DAY-OF-INTEGER		COS	RANDOM
NUMVAL-C	DAY-TO-YYYYDDD		FACTORIAL	RANGE
ORD-MAX	INTEGER-OF-DATE		INTEGER	STANDARD-DEVIATION
ORD-MIN	INTEGER-OF-DAY		INTEGER-PART	VARIANCE
	WHEN-COMPILED		LOG	
	YEAR-TO-YYYY		LOG10	
			MOD	
			REM	
			SIN	
			SQRT	
			SUM	
			TAN	

The following examples and accompanying explanations show intrinsic functions in each of the categories listed in the preceding table.

General Number-Handling: Suppose you want to find the maximum value of two prices (represented as alphanumeric items with dollar signs), put this value into a numeric field in an output record, and determine the length of the output record. You could use NUMVAL-C (a function that returns the numeric value of an alphanumeric string) and the MAX function to do this:

```
01  X                      Pic 9(2).
01  Price1                 Pic x(8)   Value "$8000".
01  Price2                 Pic x(8)   Value "$2000.
01  Output-Record.
    05  Product-Name       Pic x(20).
    05  Product-Number     Pic 9(9).
    05  Product-Price      Pic 9(6).
    .
    .
    .
Procedure Division.
    Compute Product-Price =
      Function Max (Function Numval-C(Price1) Function Numval-C(Price2))
    Compute X = Function Length(Output-Record)
```

Additionally, to ensure that the contents in Product-Name are in uppercase letters, you could use the following statement:

```
    Move Function Upper-case(Product-Name) to Product-Name
```

Date/Time: The following example shows how to calculate a due date that is 90 days from today. The first eight characters returned by the CURRENT-DATE function represent the date in a 4-digit year, 2-digit month, and 2-digit day format (YYYYMMDD). In the example, this date is converted to its integer value. Then 90 is added to this value, and the integer is converted back to the YYYYMMDD format.

```
01  YYYYMMDD               Pic 9(8).
01  Integer-Form           Pic S9(9).
    .
    .
    .
    Move Function Current-Date(1:8) to YYYYMMDD
    Compute Integer-Form = Function Integer-of-Date(YYYYMMDD)
    Add 90 to Integer-Form
    Compute YYYYMMDD = Function Date-of-Integer(Integer-Form)
    Display 'Due Date: ' YYYYMMDD
```

Finance: Business investment decisions frequently require computing the present value of expected future cash inflows to evaluate the profitability of a planned investment. The present value of money is its value today. The present value of an amount that you expect to receive at a given time in the future is that amount which if invested today at a given interest rate would accumulate to that future amount.

For example, assume a proposed investment of $1,000 produces a payment stream of $100, $200, and $300 over the next three years, one payment per year respectively. The following COBOL statements show how to calculate the present value of those cash inflows at a 10% interest rate:

```
01  Series-Amt1            Pic 9(9)V99       Value 100.
01  Series-Amt2            Pic 9(9)V99       Value 200.
01  Series-Amt3            Pic 9(9)V99       Value 300.
01  Discount-Rate          Pic S9(2)V9(6)    Value .10.
01  Todays-Value           Pic 9(9)V99.
  ⋮
    Compute Todays-Value =
      Function
        Present-Value(Discount-Rate Series-Amt1 Series-Amt2 Series-Amt3)
```

The ANNUITY function can be used in business problems that require you to determine the amount of an installment payment (annuity) necessary to repay the principal and interest of a loan. The series of payments is characterized by an equal amount each period, periods of equal length, and an equal interest rate each period. The following example shows how you could calculate the monthly payment required to repay a $15,000 loan at 12% annual interest in three years (36 monthly payments, interest per month = .12/12):

```
01  Loan                  Pic 9(9)V99.
01  Payment               Pic 9(9)V99.
01  Interest              Pic 9(9)V99.
01  Number-Periods        Pic 99.
  ⋮
    Compute Loan = 15000
    Compute Interest = .12
    Compute Number-Periods = 36
    Compute Payment =
      Loan * Function Annuity((Interest / 12) Number-Periods)
```

Mathematics: The following COBOL statement demonstrates how intrinsic functions can be nested, how arguments can be arithmetic expressions, and how previously complex mathematical calculations can be simply performed:

```
Compute Z = Function Log(Function Sqrt (2 * X + 1)) + Function Rem(X 2)
```

Here, the remainder of dividing X by 2 is found with an intrinsic function instead of using a DIVIDE statement with a REMAINDER clause.

Statistics: Intrinsic functions also make calculating statistical information on data easier. Assume you are analyzing various city taxes and want to calculate the mean, median, and range (the difference between the maximum and minimum taxes):

```
01  Tax-S                 Pic 99v999 value .045.
01  Tax-T                 Pic 99v999 value .02.
01  Tax-W                 Pic 99v999 value .035.
01  Tax-B                 Pic 99v999 value .03.
01  Ave-Tax               Pic 99v999.
01  Median-Tax            Pic 99v999.
01  Tax-Range             Pic 99v999.
  ⋮
    Compute Ave-Tax = Function Mean(Tax-S Tax-T Tax-W Tax-B)
    Compute Median-Tax = Function Median(Tax-S Tax-T Tax-W Tax-B)
    Compute Tax-Range = Function Range (Tax-S Tax-T Tax-W Tax-B)
```

Fixed-Point versus Floating-Point Arithmetic

Many statements in your program might involve arithmetic. For example, each of the following COBOL statements requires some kind of arithmetic evaluation:

- General arithmetic.

  ```
  compute report-matrix-col = (emp-count ** .5) + 1
  add report-matrix-min to report-matrix-max giving report-matrix-tot
  ```

- Expressions and functions.

  ```
  compute report-matrix-col = function sqrt(emp-count) + 1
  compute whole-hours = function integer-part((average-hours) + 1)
  ```

- Arithmetic comparisons.

  ```
  if report-matrix-col < function sqrt(emp-count) + 1
  if whole-hours not = function integer-part((average-hours) + 1)
  ```

Fixed-Point vs. Floating-Point

For each arithmetic evaluation in your program—whether it is a statement, an intrinsic function, an expression, or some combination of these nested within each other—how you code the arithmetic determines whether it will be floating-point or fixed-point evaluation.

The following discussion explains when arithmetic and arithmetic comparisons are evaluated in fixed-point and floating-point. For details on the precision of arithmetic evaluations, see Appendix C, “Intermediate Results and Arithmetic Precision” on page 545.

Floating-Point Evaluations

In general, if your arithmetic evaluation has either of the characteristics listed below, it will be evaluated by the compiler in floating-point arithmetic:

- An operand or result field is floating-point.

  A data item is floating-point if you code it as a floating-point literal, or if you define it as USAGE COMP-1, USAGE COMP-2, or as external floating-point (USAGE DISPLAY with a floating-point PICTURE).

  An operand that is a nested arithmetic expression or a reference to numeric intrinsic function results in floating-point when:

  – An argument in an arithmetic expression results in floating-point.
  – The function is a floating-point function.
  – The function is a mixed-function with one or more floating-point arguments.

- An exponent contains decimal places.

  This is true if you use a literal that contains decimal places, give the item a PICTURE containing decimal places, or use an arithmetic expression or function whose result has decimal places.

  An arithmetic expression or numeric function yields a result with decimal places if any operand or argument—excluding divisors and exponents—has decimal places.

Fixed-Point Evaluations

In general, if your arithmetic operation contains neither of the characteristics listed above for floating-point, it will be evaluated by the compiler in fixed-point arithmetic. In other words, your arithmetic evaluations will be handled by the compiler as fixed-point only if all your operands are given in fixed-point, your result field is defined to be fixed-point, and none of your exponents represent values with decimal places. Nested arithmetic expression and function references must represent fixed-point values.

Arithmetic Comparisons (Relation Conditions)

If your arithmetic is a comparison (contains a relational operator), then the numeric expressions being compared—whether they are data items, arithmetic expressions, function references, or some combination of these—are really operands (comparands) in the context of the entire evaluation. This is also true of abbreviated comparisons; although one comparand might not explicitly appear, both are operands in the comparison. For example, in the following statement:

```
if (a + d) = (b + e) and c
```

there are two comparisons: (a + d) = (b + e) and (a + d) = c. Although (a + d) does not explicitly appear in the second comparison, it is nevertheless an operand in that comparison (and thus, evaluation of (a + d) is influenced by the attributes of c).

Implicit Note: Implicit comparisons (no relational operator used) are not handled as a unit—the two expressions being compared are treated separately as to whether they will be evaluated in floating-point or fixed-point. In the following example we actually have five arithmetic expressions that are evaluated independent of one another's attributes, and then are compared to each other.

Thus, the rules outlined so far for determining whether your evaluation will be done in fixed-point or floating-point arithmetic apply to your comparison statement as a unit.

```
evaluate (a + d)
  when (b + e) thru c
  when (f / g) thru (h * i)
   .
   .
   .
end-evaluate
```

Your comparison operation (and the evaluation of any arithmetic expressions nested in your comparison) will be handled by the compiler as floating-point arithmetic if either of your comparands is a floating-point value or resolves to a floating-point value.

Your comparison operation (and the evaluation of any arithmetic expressions nested in your comparison) will be handled by the compiler as fixed-point arithmetic if both of your comparands are fixed-point values or resolve to fixed-point values.

Examples of Fixed-Point and Floating-Point Evaluations

For the examples shown on page 43, if you define the data items in the following manner:

```
01  employee-table.
    05  emp-count           pic  9(4).
    05  employee-record occurs 1 to 1000 times
                        depending on emp-count.
        10 hours            pic +9(5)e+99.
  ⋮
01  report-matrix-col       pic  9(3).
01  report-matrix-min       pic  9(3).
01  report-matrix-max       pic  9(3).
01  report-matrix-tot       pic  9(3).
01  average-hours           pic  9(3)v9.
01  whole-hours             pic  9(4).
```

- These evaluations would be done in floating-point arithmetic:

  ```
  compute report-matrix-col = (emp-count ** .5) + 1
  compute report-matrix-col = function sqrt(emp-count) + 1
  if report-matrix-tot < function sqrt(emp-count) + 1
  ```

Fixed-Point vs. Floating-Point

- These evaluations would be done in fixed-point arithmetic:

```
add report-matrix-min to report-matrix-max giving report-matrix-tot
compute report-matrix-max =
  function max(report-matrix-max report-matrix-tot)
if whole-hours not = function integer-part((average-hours) + 1)
```

Chapter 4. Handling Tables

A table is a collection of data items that have the same description. It is the COBOL equivalent to an array of elements. This chapter explains the concepts and coding techniques necessary for defining, referencing, initializing, searching, and processing table items, including both fixed-length and variable-length items.

Defining a Table (OCCURS Clause)

You could define table items as separate, consecutive entries in the DATA DIVISION, but this practice has disadvantages. First, the code does not clearly show the unity of the items. Second, you cannot take advantage of subscripting and indexing for easy reference to the table elements. (See “Referring to an Item in a Table” on page 49 for information on subscripting and indexing.)

Use the COBOL OCCURS clause in the DATA DIVISION entry to define a table, and you do not need separate entries for repeated data items. The OCCURS clause also supplies the information necessary for the use of subscripts or indexes. (For more information on the format of the OCCURS clause, refer to *IBM COBOL Language Reference*).

To code a table, give the table a group name and define a subordinate item (the table *element*) that is to be repeated *n* times:

```
01  table-name.
    05  element-name OCCURS n TIMES.
      .
      .  (subordinate items of the table element might follow)
      .
```

The following figures show how to code tables:

- One-Dimensional Table—Figure 11 on page 48
- Two-Dimensional Table—Figure 12 on page 48
- Three-Dimensional Table—Figure 13 on page 49.

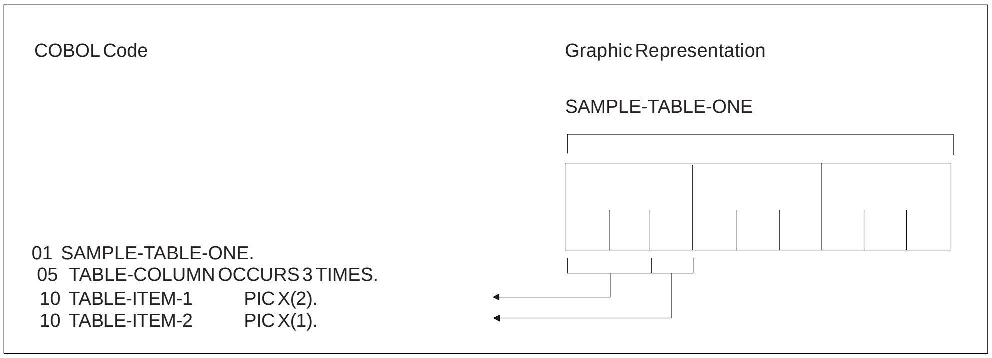
For all the tables, the table element definition (which includes the OCCURS clause) is subordinate to the group item that contains the table; the OCCURS clause cannot appear in a level-01 description.

To create tables of more than one dimension, use nested OCCURS clauses. Tables of up to seven dimensions can be defined using this same method.

One Dimension

To create a one-dimensional table, use one OCCURS clause. For example:

Defining a Table



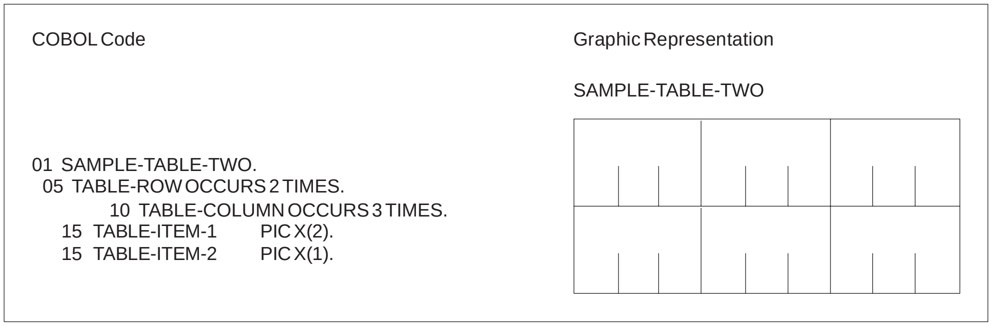
```
01  SAMPLE-TABLE-ONE.
 05  TABLE-COLUMN OCCURS 3 TIMES.
  10  TABLE-ITEM-1        PIC X(2).
  10  TABLE-ITEM-2        PIC X(1).
```

Figure 11. Coding a One-Dimensional Table

`SAMPLE-TABLE-ONE` is the group item that contains the table. `TABLE-COLUMN` names the table element of a one-dimensional table that occurs three times.

Two Dimensions

To create a two-dimensional table, define a one-dimensional table in each occurrence of another one-dimensional table. For example:



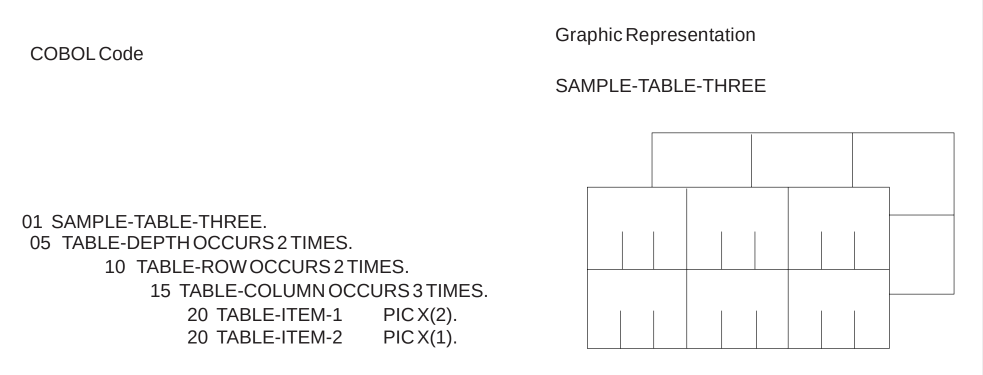
```
01  SAMPLE-TABLE-TWO.
  05  TABLE-ROW OCCURS 2 TIMES.
            10  TABLE-COLUMN OCCURS 3 TIMES.
    15  TABLE-ITEM-1        PIC X(2).
    15  TABLE-ITEM-2        PIC X(1).
```

Figure 12. Coding a Two-Dimensional Table

`SAMPLE-TABLE-TWO` is the name of a two-dimensional table. `TABLE-ROW` is an element of a one-dimensional table that occurs two times. `TABLE-COLUMN` is an element of a two-dimensional table that occurs three times in each occurrence of `TABLE-ROW`.

Three Dimensions

To create a three-dimensional table, define a one-dimensional table in each occurrence of another one-dimensional table, which is itself contained in each occurrence of another one-dimensional table. For example:



COBOL Code

```
01  SAMPLE-TABLE-THREE.
 05  TABLE-DEPTH OCCURS 2 TIMES.
           10  TABLE-ROW OCCURS 2 TIMES.
                15  TABLE-COLUMN OCCURS 3 TIMES.
                    20  TABLE-ITEM-1       PIC X(2).
                    20  TABLE-ITEM-2       PIC X(1).
```

Figure 13. Coding a Three-Dimensional Table

In SAMPLE-TABLE-THREE, TABLE-DEPTH is an element of a one-dimensional table that occurs two times. TABLE-ROW is an element of a two-dimensional table that occurs two times within each occurrence of TABLE-DEPTH. TABLE-COLUMN is an element of a three-dimensional table that occurs three times within each occurrence of TABLE-ROW.

Referring to an Item in a Table

A table element has a collective name, but the individual occurrences within it do not have unique *data-names*. To refer to them, use the *data-name* of the table element, together with the occurrence number, called a *subscript*, of the desired item within the element.

The technique of supplying the occurrence number of individual table elements is called subscripting. See page 49. A related technique, called *subscripting using index-names (indexing)* is also available for table references.

An *index* is a symbol used to locate an item in a table. An index differs from a subscript in that an index is a value to be added to the address of a table to locate an item (the displacement from the beginning of the table). See page 50.

Subscripting

The lowest possible subscript value is 1, which points to the first occurrence of the table-element. In a one-dimensional table, the subscript corresponds to the row number. In a two-dimensional table, the two subscripts correspond to the column and row numbers. In a three-dimensional table, the three subscripts correspond to the depth, column, and row numbers.

You can use a literal subscript or a data-name for a variable subscript.

Referring to a Table Item

Literal Subscripts

The following are valid literal subscript references to SAMPLE-TABLE-THREE:

```
TABLE-COLUMN (2, 2, 1)
TABLE-COLUMN (2 2 1)       (The spaces are required for subscripting.)
```

In the table reference TABLE-COLUMN (2, 2, 1), the first value (2) refers to the second occurrence within TABLE-DEPTH, the second value (2) refers to the second occurrence within TABLE-ROW, and the third value (1) refers to the first occurrence within TABLE-COLUMN.

If a subscript is represented by a literal and the subscripted item is of fixed length, then the compiler resolves the location of the subscripted data item within the table at compile time.

Variable Subscripts

The following is a valid, variable subscript reference to SAMPLE-TABLE-TWO, (assuming that SUB1 and SUB2 are *data-names* containing positive integer values within the range of the table):

```
TABLE-COLUMN (SUB1 SUB2)
```

The *data-name* used as a variable subscript must be described as an elementary numeric integer data item.

If a *data-name* is being used as a subscript or qualifier, it cannot itself be subscripted.

If a subscript is represented by a *data-name*, the code generated for the application resolves the location at run time. The most efficient format for data used as a variable subscript is COMPUTATIONAL (COMP) with a PICTURE size of less than five digits.

Relative Subscripts

In relative subscripting, the subscript can be incremented or decremented by a specified integer amount. Relative subscripting is valid with either literal or variable subscripts. For example:

```
TABLE-COLUMN (SUB1 - 1, SUB2 + 3)
```

Subscripting Using Index-Names (Indexing)

You can refer to table elements by using a subscript, an index, or both. An index is a displacement from the start of the table, based on the length of the table element.

To reference a table by an index:

1. Define the *index-name* for a table in the INDEXED BY clause of the OCCURS clause in the table definition.
2. Choose direct or relative indexing (described below).
3. Initialize the *index-name* with a SET, PERFORM VARYING, or SEARCH ALL statement before using it in a table reference.

4. Use the index in SET, SEARCH, SEARCH ALL, PERFORM VARYING, or relational condition statements.

How the Index Value Is Determined

The compiler determines the index of an entry using the following formula:

$I = L * (S-1)$

where:

I is the index value.
L is the length of a table entry.
S is the subscript (occurrence number) of an entry.

To be valid during execution, an index value must correspond to a table element occurrence of not less than 1 nor greater than the highest permissible occurrence number. This restriction applies to both direct and relative indexing.

Direct Indexing

In direct indexing, the *index-name* is in the form of a displacement. The value contained in the index is then calculated as the occurrence number minus 1, multiplied by the length of the individual table entry.

For example:

```
05  TABLE-ITEM OCCURS 10 INDEXED BY INX-A PIC X(8).
```

For the fifth occurrence of TABLE-ITEM, the binary value contained in INX-A is (5 - 1) * 8 = 32.

Relative Indexing

In relative indexing, the *index-name* is followed by a space, followed by a + or a -, followed by another space, followed by an unsigned numeric literal. The literal is considered to be an occurrence number, and is converted to an index value before being added to or subtracted from the *index-name*.

Relative Indexing Example: If you code indexing for SAMPLE-TABLE-THREE as follows:

```
01  SAMPLE-TABLE-THREE
    05  TABLE-DEPTH OCCURS 3 TIMES INDEXED BY INX-A.
        10  TABLE-COLUMN OCCURS 4 TIMES INDEXED BY INX-B.
            15  TABLE-ROW OCCURS 8 TIMES INDEXED BY INX-C    PIC X(8).
```

a relative indexing reference to:

```
TABLE-ROW (INX-A + 1, INX-B + 2, INX-C - 1)
```

causes the following computation of the displacement:

```
  (contents of INX-A) + (256 * 1)
+ (contents of INX-B) + (64 * 2)
+ (contents of INX-C) - (8 * 1)
```

Referring to a Table Item

That is,

- Each occurrence of `TABLE-DEPTH` is 256 characters in length.
- Each occurrence of `TABLE-COLUMN` is 64 characters in length.
- Each occurrence of `TABLE-ROW` is 8 characters in length.

More Ways to Use Index References

- An index can be modified using a PERFORM, SEARCH, or SET statement.
- To compare two different occurrences of a table element, use a direct indexing reference together with a relative indexing reference, or use subscripting, which is easier to read in your code.
- An index can be shared among different tables. That is, you can use the index defined with one table to index another table if both table descriptions are identical. To be identical, the tables must have the same number of occurrences, as well as occurrences of the same length.
- Store index values in *index data items* you define with the USAGE IS INDEX clause. Use the SET statement to assign to an index the value that you stored in the index data item.

  For example, when you read records to load a variable-length table, you can store the index value of the last record in a data item defined as USAGE IS INDEX. Then, when you use the table index to look through or process the variable-length table, you can test for the end of the table by comparing the current index value with the index value of the last record you stored in the index data item.

  Because you're comparing a physical displacement, you can use index data items only in SEARCH and SET statements or for comparisons with indexes or other index data items. You cannot use index data items as subscripts or indexes.

Referring to a Substring of a Table Item

Both reference modification and subscripting can be coded for a table element in the same statement. For example, if you define a table like this:

```
01  ANY-TABLE.
    05  TABLE-ELEMENT            PIC X(10)
           OCCURS 3 TIMES
           VALUE "ABCDEFGHIJ".
```

the statement

```
MOVE "??" TO TABLE-ELEMENT ( 1 ) ( 3 : 2 )
```

will move the value "??" into table element number 1, beginning at character position 3, for a length of 2.

```
So, if ANY-TABLE looked         ANY-TABLE would look
like this before the change:    like this after the change:

 ABCDEFGHIJ                           AB??EFGHIJ
 ABCDEFGHIJ                           ABCDEFGHIJ
 ABCDEFGHIJ                           ABCDEFGHIJ
```

Putting Values into a Table

Use one of these methods to put values in a table:

- Load the table dynamically.
- Initialize the table (INITIALIZE statement).
- Assign values when you define the table (VALUE clause).

Loading the Table Dynamically

If the initial values of your table are different with each execution of your program, the table can be defined without initial values, and the changed values can be read into the table before your program refers to the table.

To load a table, use:

- The PERFORM . . . VARYING statement.
- Either subscripting or indexing.

When reading data to load your table, test to make sure that the data does not exceed the space allocated for the table. Use a named value giving the item count, rather than using a literal. Then, if you make the table bigger, you need to change only one value, instead of all references to a literal.

Initializing the Table (INITIALIZE Statement)

You can also load your table with a value during execution with the INITIALIZE statement. For example, to fill a table with 3s:

```
INITIALIZE TABLE-ONE REPLACING NUMERIC DATA BY 3.
```

The INITIALIZE statement cannot load a variable-length table (one that was defined using OCCURS DEPENDING ON).

Assigning Values When You Define the Table (VALUE Clause)

If your table contains stable values (for example a table that contains the days and months of the year), set the specific values your table holds when you define it.

Define static values in Working-Storage in one of the these ways:

- Initialize each table item individually.

Putting Values into a Table

- Initialize an entire table at the 01 level.
- Initialize all occurrences of a given table element to the same value.

Initializing Each Table Item Individually

- Describe the table storage area by arranging subordinate data description entries, setting the initial value of each subordinate entry in a VALUE clause.
- Code a REDEFINES entry to describe the table as a record that contains a repeating subordinate entry, defined with an OCCURS clause.

For an example of this method, see "Error Flag Table" and "Error Message Table" in Figure 14 on page 55.

This technique is practical only for small tables. To initialize larger tables, use MOVE, PERFORM, or INITIALIZE statements, as described above.

Initializing a Table at the 01 Level

Code a level-01 record and assign to it, through the VALUE clause, the contents of the whole table. Then, in a subordinate level data item, use an OCCURS clause to define the individual table items.

For example:

```
01  TABLE-ONE                         VALUE "1234".
    05  TABLE-TWO OCCURS 4 TIMES      PIC X.
```

Initializing a Variable-Size Table: A VALUE clause can also be present on a group item that contains an OCCURS clause with the DEPENDING ON option. Each subordinate structure that contains the DEPENDING ON option is initialized using the maximum number of occurrences. If the entire table is defined with the DEPENDING ON option, all the elements are initialized using the maximum defined value of the DEPENDING ON object.

In both cases, if the ODO object has a VALUE clause, it is logically initialized after the *ODO subject* has been initialized. For example:

```
01  TABLE-THREE                       VALUE "3ABCDE".
    05  X                             PIC 9.
    05  Y OCCURS 5 TIMES
          DEPENDING ON X              PIC X.
```

causes Y(1) to be initialized to A, Y(2) to B,... Y(5) to E, and finally the object of the ODO (X) is initialized to 3. Any subsequent reference to TABLE-THREE (such as DISPLAY) would refer to the first 3 elements, Y(1) through Y(3).

Initializing All Occurrences of a Table Element

You can use the VALUE clause on a table element to initialize the element to the indicated value.

As an example, this code:

```
01  T2.
    05  T-OBJ                             PIC 9    VALUE 3.
    05  T OCCURS 5 TIMES DEPENDING ON T-OBJ.
        10  X                             PIC XX   VALUE "AA".
        10  Y                             PIC 99   VALUE 19.
        10  Z                             PIC XX   VALUE "BB".
```

causes all the X elements (1 through 5) to be initialized to AA, all the Y elements (1 through 5) to be initialized to 19, and all the Z elements (1 through 5) to be initialized to BB. T-OBJ is then set to 3.

```
***********************************************************
***           E R R O R   F L A G   T A B L E           ***
***********************************************************
 01  Error-Flag-Table                    Value Spaces.
   88 No-Errors                          Value Spaces.
     05 Type-Error                       Pic X.
     05 Shift-Error                      Pic X.
     05 Home-Code-Error                  Pic X.
     05 Work-Code-Error                  Pic X.
     05 Name-Error                       Pic X.
     05 Initials-Error                   Pic X.
     05 Duplicate-Error                  Pic X.
     05 Not-Found-Error                  Pic X.
     05 Address-Error                    Pic X.
     05 City-Error                       Pic X.
     05 State-Error                      Pic X.
     05 Zipcode-Error                    Pic X.
     05 Home-Phone-Error                 Pic X.
     05 Work-Phone-Error                 Pic X.
     05 Home-Junction-Error              Pic X.
     05 Work-Junction-Error              Pic X.
     05 Driving-Status-Error             Pic X.
 01  Filler Redefines Error-Flag-Table.
     05 Error-Flag Occurs 17 Times
          Indexed By Flag-Index          Pic X.
***********************************************************
***         E R R O R   M E S S A G E   T A B L E       ***
***********************************************************
 01  Error-Message-Table.
     05  Filler                           Pic X(25) Value
          "Transaction Type Invalid".
     05  Filler                           Pic X(25) Value
          "Shift Code Invalid".
     05  Filler                           Pic X(25) Value
          "Home Location Code Inval.".
```

Figure 14 (Part 1 of 2). Table with Static Values Defined for Every Table Element

```
     05  Filler                           Pic X(25) Value
          "Work Location Code Inval.".
     05  Filler                           Pic X(25) Value
          "Last Name - Blanks".
     05  Filler                           Pic X(25) Value
          "Initials - Blanks".
     05  Filler                           Pic X(25) Value
          "Duplicate Record Found".
     05  Filler                           Pic X(25) Value
          "Commuter Record Not Found".
     05  Filler                           Pic X(25) Value
          "Address - Blanks".
     05  Filler                           Pic X(25) Value
          "City - Blanks".
     05  Filler                           Pic X(25) Value
          "State Is Not Alphabetic".
     05  Filler                           Pic X(25) Value
          "ZipCode Is Not Numeric".
     05  Filler                           Pic X(25) Value
          "Home Phone Number Error".
     05  Filler                           Pic X(25) Value
          "Work Phone Number Error".
     05  Filler                           Pic X(25) Value
          "Home Junction Is Blanks".
     05  Filler                           Pic X(25) Value
          "Work Junction Is Blanks".
     05  Filler                           Pic X(25) Value
          "Driving Status Invalid".
 01  Filler Redefines Error-Message-Table.
     05  Error-Message Occurs 17 Times
            Indexed By Message-Index      Pic X(25).
```

Figure 14 (Part 2 of 2). Table with Static Values Defined for Every Table Element

Processing a Table Using Subscripting and PERFORM...VARYING

The procedure shown in Figure 15 processes the entire table shown in Figure 14 on page 55, using subscripting and the PERFORM...VARYING statement.

```
Perform
    Varying Sub From 1 By 1
    Until No-Errors
  If Error-Flag (Sub) = Error-On
    Move Space To Error-Flag (Sub)
    Move Error-Message (Sub) To Print-Message
    Perform 260-Print-Report
  End-If
End-Perform
```

Figure 15. Processing the Sample Table, Using Subscripting

Processing a Table Using Indexing

The procedure shown in Figure 16 processes the entire table, using indexing.

```
Set Flag-Index To 1
Perform Until No-Errors
  Search Error-Flag
    When Error-Flag (Flag-Index) = Error-On
      Move Space To Error-Flag (Flag-Index)
      Set Message-Index To Flag-Index
      Move Error-Message (Message-Index) To
          Print-Message
      Perform 260-Print-Report
  End-Search
End-Perform
```

Figure 16. Processing the Sample Table, Using Indexing

Creating Variable-Length Tables (DEPENDING ON Clause)

If you don't know before execution how many occurrences of a table element there are, you need to set up a variable-length table definition. To do this, use the OCCURS DEPENDING ON (ODO) clause. For example:

```
X OCCURS 1 TO 10 TIMES DEPENDING ON Y
```

`X` is the ODO subject, `Y` is the ODO object.

The cases to consider when using the ODO clause are:

- ODO object and subject are contained within the same group item, and that item is a *sending* field or that item is a *receiving* field.
- ODO object is outside of the group item that contains the subject.

ODO Object and Subject Contained in Group Item

Whether Maximum Length or Actual Length Is Used

If a group item is not complex ODO, contains both the subject and object of the ODO, and it is a receiving item, then the maximum length of the item is used. In this situation it is not necessary to set the value of the ODO object before a reference is made.

If the receiving item is followed by a data-item which is in the same record but is not subordinate to the receiver (is complex ODO), then the actual length is used and a compiler message is issued to inform you that the actual length, not the maximum, will be used. In this situation it is necessary to set the value of the ODO object before any reference to the item.

The following example contrasts how the length is determined for a group item whose subordinate items contain an OCCURS clause with the DEPENDING ON option and the object of that DEPENDING ON option, depending on whether it is the sending group item or the receiving group item.

```
WORKING-STORAGE SECTION.
01  MAIN-AREA.
    03  REC-1.
        05  FIELD-1                       PIC S9.
        05  FIELD-2 OCCURS 1 TO 5 TIMES
             DEPENDING ON FIELD-1         PIC X(05).

01  REC-2.
    05  REC-2-DATA                        PIC X(50).
```

Sending Group Item

If you want to move REC-1 to REC-2, the length of REC-1 is determined immediately prior to the MOVE, using the current value in FIELD-1. If the contents of FIELD-1 do not conform to its PICTURE, that is, if FIELD-1 does not contain an external decimal item, the result is unpredictable. (See Chapter 3, “Numbers and Arithmetic” on page 29 for more information on data and sign representation).

As you can see, you must be sure that you have the correct value placed in the ODO object before the MOVE is initiated.

Receiving Group Item

If you want to do a MOVE to REC-1, the length of REC-1, for the purpose of the MOVE, is determined using the maximum number of occurrences. In this example, that would be 5 occurrences of FIELD-2 plus FIELD-1 for a length of 26.

In this case, the ODO object (FIELD-1) need not be set before referencing REC-1 as a receiving item. However, the sending field's ODO object needs to be set to a valid numeric value between 1 and 5 for the ODO object of the receiving field to be validly set by the move.

Another Record Makes this Complex ODO

However, if REC-2 were followed by a data item which is in the same record but is not subordinate to REC-2, then the actual length of REC-2 is used and the ODO object must be set before the reference.

In the following example, REC-1 is followed by REC-2.

```
01  MAIN-AREA
    03  REC-1.
        05  FIELD-1                       PIC S9.
        05  FIELD-3                       PIC S9.
        05  FIELD-2 OCCURS 1 TO 5 TIMES
             DEPENDING ON FIELD-1         PIC X(05).
    03  REC-2.
        05  FIELD-4 OCCURS 1 TO 5 TIMES
             DEPENDING ON FIELD-3         PIC X(05).
```

If you do a MOVE to REC-1 in this case, the actual length of REC-1 is calculated immediately prior to the move using the current value of the ODO object (FIELD-1), and a compiler message is issued letting you know that the actual length, instead of the maximum length, was used. This case requires that you set the value of the ODO object (FIELD-1) prior to using the item as a receiving field.

ODO Object outside the Group

You must ensure that the object of the OCCURS DEPENDING ON clause contains a value that correctly specifies the current number of occurrences of the table elements. Figure 17 shows how to define a variable-length table.

```
 DATA DIVISION.
 FILE SECTION.

 FD  LOCATION-FILE.
   01 LOCATION-RECORD.
     05  LOC-CODE                      PIC XX.
     05  LOC-DESCRIPTION               PIC X(20).
     05  FILLER                        PIC X(58).

 WORKING-STORAGE SECTION.
   01 FLAGS.
     05 LOCATION-EOF-FLAG              PIC X(5) VALUE SPACE.
         88 LOCATION-EOF               VALUE "FALSE".
   01 MISC-VALUES.
     05 LOCATION-TABLE-LENGTH          PIC 9(3) VALUE ZERO.
     05 LOCATION-TABLE-MAX             PIC 9(3) VALUE 100.
*****************************************************************
***                L O C A T I O N   T A B L E                ***
***                FILE CONTAINS LOCATION CODES.              ***
*****************************************************************
   01 LOCATION-TABLE.
     05 LOCATION-CODE OCCURS 1 TO 100 TIMES
             DEPENDING ON LOCATION-TABLE-LENGTH   PIC X(80).
```

Figure 17. Defining a Variable-Length Table

Figure 18 shows a do-until structure used to control loading of a variable-length table. When initialization is complete, LOCATION-TABLE-LENGTH will contain the subscript of the last item in the table. (This variable-length table is defined in Figure 17.)

Variable-Length Tables

```
Perform Test After
    Varying Location-Table-Length From 1 By 1
    Until Location-EOF
    Or Location-Table-Length = Location-Table-Max
  Move Location-Record To
      Location-Code (Location-Table-Length)
  Read Location-File
      At End Set Location-EOF To True
  End-Read
End-Perform
```

Figure 18. Loading a Variable-Length Table

Two factors that affect the successful manipulation of variable-length records are the correct calculation of records lengths and the conformance of the data in the OCCURS...DEPENDING ON object to its picture. If you are using variable-length group items in either a READ...INTO or WRITE...FROM statement, in conjunction with an OCCURS...DEPENDING ON statement, make sure that the receiver or intermediate field length is correct. The length of the variable portions of a group item is the product of the object of the DEPENDING ON option and the length of the subject of the OCCURS clause.

If the content of the ODO object does not match its PICTURE clause, the program may abnormally terminate. See Chapter 3, “Numbers and Arithmetic” on page 29 for more information on data and sign representation.

Complex OCCURS DEPENDING ON

Complex OCCURS DEPENDING ON is supported as an extension to the COBOL 85 Standard.

The basic forms of complex ODO permitted by the compiler are:

- A data item described by an OCCURS clause with the DEPENDING ON option is followed by a non-subordinate element or group (variably-located item).
- A data item described by an OCCURS clause with the DEPENDING ON option is followed by a non-subordinate data item described by an OCCURS clause with the DEPENDING ON option (variably-located table).
- A data item described by an OCCURS clause with the DEPENDING ON option is nested within another data item described by an OCCURS clause with the DEPENDING ON option (table with variable-length elements).
- Index-name for a table with variable-length elements.

Complex ODO is tricky to use and can make maintaining your code more difficult. If you choose to use it in order to save disk space, follow the guidelines in Appendix D, “Complex OCCURS DEPENDING ON” on page 553.

Searching a Table (SEARCH Statement)

COBOL provides two search techniques for tables: serial and binary.

To perform *serial searches*:

- Use the PERFORM . . . VARYING statement with subscripting or indexing (discussed in "Creating Variable-Length Tables (DEPENDING ON Clause)" on page 57).
- Use SEARCH and indexing.

To perform *binary searches*, use indexing and the SEARCH ALL statement.

The following discussion assumes you are familiar with the format of the SEARCH and SEARCH ALL statements. If you are not, see *IBM COBOL Language Reference*.

Serial Search

Use the SEARCH statement to perform a serial search beginning at the current index setting. To modify the index setting, use the SET statement.

The conditions in the WHEN option are evaluated in the order in which they are written.

- If none of the conditions is satisfied, the index is increased to correspond to the next table element, and the WHEN conditions are evaluated again.
- If one of the WHEN conditions is satisfied, the search ends; the index remains pointing to the table element that satisfied the condition.
- If the entire table has been searched and no conditions were met, the AT END imperative statement is executed, if there is one. If you do not use the AT END option, control passes to the next statement in your program.

Searching More than One Level of a Table

Only one level of a table (a table element) can be referenced with each SEARCH statement. SEARCH statements can be nested to search multiple levels of a table. However, SEARCH statements can be nested only if you delimit each nested SEARCH statement with END-SEARCH. The WHEN condition must be followed by an imperative statement; the SEARCH statement is an imperative statement only when it is delimited by END-SEARCH.

Speeding Up Your Search

It is important to know if the found condition comes after some intermediate point in the table element. You can speed up the SEARCH by using the SET statement to set the index to begin the search after that point.

Arranging the table so that the data used most often is at the beginning also enables more efficient serial searching. If the table is large and is pre-sorted, a binary search is more efficient. See "Binary Search (SEARCH ALL Statement)" on page 62 more information on binary searches.

Searching a Table

Serial Search Example

```
01  TABLE-ONE.
    05  TABLE-ENTRY1 OCCURS 10 TIMES
            INDEXED BY TE1-INDEX.
        10  TABLE-ENTRY2 OCCURS 10 TIMES
                INDEXED BY TE2-INDEX.
            15  TABLE-ENTRY3 OCCURS 5 TIMES
                    ASCENDING KEY IS KEY1
                    INDEXED BY TE3-INDEX.
                20  KEY1                   PIC X(5).
                20  KEY2                   PIC X(10).
    .
    .
PROCEDURE DIVISION.
    .
    .
    SET TE1-INDEX TO 1
    SET TE2-INDEX TO 4
    SET TE3-INDEX TO 1
    MOVE "A1234" TO KEY1 (TE1-INDEX, TE2-INDEX, TE3-INDEX + 2)
    MOVE "AAAAAAAA00" TO KEY2 (TE1-INDEX, TE2-INDEX, TE3-INDEX + 2)
    SEARCH TABLE-ENTRY3
      AT END
        MOVE 4 TO RETURN-CODE
      WHEN TABLE-ENTRY3(TE1-INDEX, TE2-INDEX, TE3-INDEX)
          = "A1234AAAAAAAA00"
        MOVE 0 TO RETURN-CODE
    END-SEARCH
```

Values after execution:

```
TE1-INDEX = 1
TE2-INDEX = 4
TE3-INDEX points to the TABLE-ENTRY3
          that equals "A1234AAAAAAAA00"
RETURN-CODE = 0
```

Binary Search (SEARCH ALL Statement)

When you use SEARCH ALL to perform a binary search, you do not need to set the index before you begin. The index used is always the one associated with the first *index-name* in the OCCURS clause, and it varies during execution to maximize the search efficiency.

To use the SEARCH ALL statement, your table must be ordered on the key or keys coded in the OCCURS clause. You can use any key in the WHEN condition, but all preceding *data-names* in the KEY option must also be tested. The test must be an equal-to condition, and the KEY *data-name* must be either the subject of the condition or the name of a conditional variable with which the tested *condition-name* is associated. The WHEN condition can also be a compound condition, formed from one of the simple conditions listed above, with AND as the only logical connective. The KEY and its object of comparison must be compatible, as stated in the relation test rules.

Binary Search Example

For example, a table defined like this:

```
01  TABLE-A.
    05  TABLE-ENTRY OCCURS 90 TIMES
                   ASCENDING KEY-1, KEY-2
                   DESCENDING KEY-3
                   INDEXED BY INDX-1.
       10  PART-1                   PIC 99.
       10  KEY-1                    PIC 9(5).
       10  PART-2                   PIC 9(6).
       10  KEY-2                    PIC 9(4).
       10  PART-3                   PIC 9(18).
       10  KEY-3                    PIC 9(5).
```

can be searched using the following instructions:

```
SEARCH ALL TABLE-ENTRY
  AT END
    PERFORM NOENTRY
  WHEN KEY-1 (INDX-1) = VALUE-1 AND
       KEY-2 (INDX-1) = VALUE-2 AND
       KEY-3 (INDX-1) = VALUE-3
    MOVE PART-1 (INDX-1) TO OUTPUT-AREA
END-SEARCH
```

These instructions will execute a search on the given table that contains 90 elements of 40 bytes and 3 keys. The primary and secondary keys (KEY-1 and KEY-2) are in ascending order, but the least significant key (KEY-3) is in descending order. If an entry is found in which three keys are equal to the given values (VALUE-1, VALUE-2, and VALUE-3), PART-1 of that entry will be moved to OUTPUT-AREA. If the matching keys are not found in any of the entries in TABLEA, the NOENTRY routine is performed.

Processing Table Items (Intrinsic Functions)

You can process alphanumeric or numeric table items using intrinsic functions as long as the table item's data description is compatible with the function's argument requirements. The *IBM COBOL Language Reference* describes the required data formats for the arguments of the various intrinsic functions.

Use a subscript or index to reference an individual data item as a function argument. Assuming Table-One is a 3x3 array of numeric items, you can find the square root of the middle element with a statement such as:

```
Compute X = Function Sqrt(Table-One(2,2))
```

Processing Multiple Table Items (ALL Subscript)

You might often need to process the data in tables iteratively. For intrinsic functions that accept multiple arguments, you can use the ALL subscript to reference all the items in the table or single dimension of the table. The iteration is handled automatically, making your code shorter and simpler.

Efficient Table Coding

Example 1

This example sums a cross section of `Table-Two`:

```
Compute Table-Sum = FUNCTION SUM (Table-Two(ALL, 3, ALL)))
```

Assuming that `Table2` is a 2x3x2 array, the above statement would cause these elements to be summed:

```
Table-Two(1,3,1)
Table-Two(1,3,2)
Table-Two(2,3,1)
Table-Two(2,3,2)
```

Example 2

This example computes values for all employees.

```
01  Employee-Table.
    05  Emp-Count          Pic s9(4)  usage binary.
    05  Emp-Record         occurs 1 to 500 times
                           depending on Emp-Count.
        10  Emp-Name       Pic x(20).
        10  Emp-Idme       Pic 9(9).
        10  Emp-Salary     Pic 9(7)v99.
    .
    .
Procedure Division.
    Compute Max-Salary = Function Max(Emp-Salary(ALL))
    Compute I = Function Ord-Max(Emp-Salary(ALL))
    Compute Avg-Salary = Function Mean(Emp-Salary(ALL))
    Compute Salary-Range = Function Range(Emp-Salary(ALL))
    Compute Total-Payroll = Function Sum(Emp-Salary(ALL))
```

Example 3

Scalars and array arguments can be mixed for functions that accept multiple arguments:

```
Compute Table-Median = Function Median(Arg1 Table-One(ALL))
```

Efficient Coding for Tables

For efficient table-handling, follow these suggestions:

- If the table is searched sequentially, put the data values most likely to satisfy the search criteria at the beginning of a table.
- Use *index-names* instead of subscripts. This method is more efficient, but subscripting might be easier to understand and maintain. Relative index references are executed as fast as direct index references. For additional details, see “Subscripting” on page 49 and “Subscripting Using Index-Names (Indexing)” on page 50.
- Use binary (COMP) data items with 8 or fewer digits for subscripts and OCCURS DEPENDING ON objects. Use fewer than five digits, if possible.
- Avoid referencing errors by coding subscript and index checks into your program.

Chapter 5. Selection and Iteration

Selection (IF and EVALUATE Statements)

Use control structures to:

- Choose program actions based on the outcome of a decision.
- Control looping in your program.

Selection is providing for different program actions depending on the tested value of some data item or data items.

The IF and EVALUATE statements are COBOL selection constructs. The testing of a data item or data items is done in both of these statements by means of a conditional expression.

IF Statement

Use IF . . . ELSE to code a choice between two processing actions. (The word THEN is optional in a COBOL program.) For example:

```
IF condition-p
  statement-1
ELSE
  statement-2
END-IF
```

IF Statement with a Null Branch

There are two ways you can code an IF statement when one of the processing choices is no action. Because the ELSE clause is optional, you can code the following:

```
IF condition-q
  statement-1
END-IF
```

This coding is suitable for simple programming cases. However, if the logic in your program is complex (for example, you have nested IF constructs with action for only one of the processing choices), you might want to use the ELSE clause and code the null branch of the IF statement with the CONTINUE statement:

```
IF condition-q
  statement-1
ELSE
  CONTINUE
END-IF
```

Nested IF Statements

When an IF statement has another IF statement as one of its possible processing branches, these IF statements are said to be nested IFs. Theoretically, there is no limitation on the depth of nested IF statements. However, when the program has to test a variable for more than two values, EVALUATE is the better choice. (For more information, see "EVALUATE statement" on page 67).

Use nested IF statements sparingly; the logic can be difficult to follow, although proper indentation helps.

Logic of a Nested IF Statement: The following is pseudocode for a nested IF statement:

```
IF condition-p
  IF condition-q
    statement-1
  ELSE
    statement-2
  END-IF
  statement-3
ELSE
  statement-4
END-IF
```

Here an IF is nested, along with a sequential structure, in one branch of another IF. The structure for this logic is shown in Figure 19 on page 66.

When you code a structure like the one in Figure 19, the END-IF closing the inner nested IF becomes very important. Use END-IF instead of a period, because a period would end the outer IF structure as well.

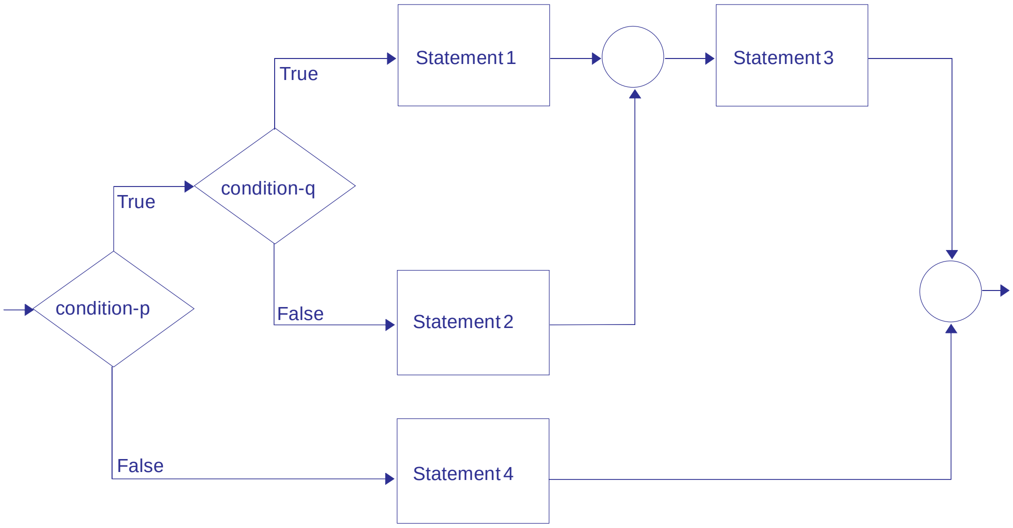


Figure 19. Control Logic Structure for Nested IF Statements

Good Coding Practice for Nested IFs: When you nest IF statements, readability and debugging will be easier if each IF statement has its own END-IF scope-terminator and if you use proper indentation. For example:

```
IF A = 1
  IF B = 2
    PERFORM C
  ELSE PERFORM D.
```

The ELSE PERFORM D phrase is interpreted as the ELSE phrase of the last previous IF which is, `IF B = 2`. If this is the intent, you can make the logic clearer with the following coding:

```
IF A = 1
  IF B = 2
    PERFORM C
  ELSE
    PERFORM D
  END-IF
END-IF
```

If the intent is to have `ELSE PERFORM D` depend on `IF A = 1`, the code would look like this:

```
IF A = 1
  IF B = 2
    PERFORM C
  END-IF
ELSE
  PERFORM D
END-IF
```

EVALUATE statement

The EVALUATE statement is an expanded form of the IF statement. An IF statement allows your program to act on one of two conditions: true or false. If you had three or more possible conditions instead of just two, and you were limited to using IF statements, you would need to nest or cascade the IF statements. Such nested IF statements are a common source of logic errors and debugging problems.

With the EVALUATE statement, you can test any number of conditions in a single statement and have separate actions for each. In structured programming terms, this is a case structure. It can also be thought of as a form of decision table.

Conditional Expressions

The IF and EVALUATE statements let you code different program actions that will be performed depending on the true or false value of a condition expression. COBOL lets you specify any of these simple conditions:

Figure 20. Conditions You Can Test For In COBOL

Condition Type	What It Tests	Where to Look for Information
Class	Whether your data is uppercase alphabetic, lowercase alphabetic, numeric, MBCS Kanji, or consisting entirely of characters listed in the definition of a user-defined class-name. NUMPROC(PFD), which bypasses invalid sign processing, might affect the outcome of a test for numeric data.	"Checking for Incompatible Data (Numeric Class Test)" on page 37.
User-defined	A level-88 condition name, to discover whether a data item contains a particular value or range of values.	See "Condition-Names (Switches and Flags)" on page 68 for details on how to use level-88 items to define *condition-names* that you can test to control the processing of switches and flags.
Relation	Compares two items.	*IBM COBOL Language Reference*.
Sign	Whether a numeric operand is less than, greater than, or equal to zero.	*IBM COBOL Language Reference*.
Switch-status	Whether an UPSI switch is on or off.	*IBM COBOL Language Reference*.

You can create combined conditions by using logical connectives (AND, OR, or NOT), and you can combine conditions. Rules for using conditions are given in *IBM COBOL Language Reference*.

Condition-Names (Switches and Flags)

Some program decisions are based on whether the value of a data item is true or false, on or off, yes or no. To control these two-way decisions in your program, define level-88 items with meaningful names (a condition name) to act as switches.

Some program decisions are based not on an on or off condition of a data item, but instead, depend on the particular value (or range of values) of a data item. When condition-names are used to give more than just on or off values to a field, the field is generally referred to as a flag, not a switch. For details on flags, see the section "Flags" on page 69, below.

Flags and Switches Make Changing Code Easier: Flags and switches make your code easier to change. If you need to change the values for a condition, you have to change only the level-88 condition-name value.

For example, a program that uses a condition-name to test a field for a given numeric range—a salary range—need not be changed. If the program must be changed to check for a different salary range, you would need to change only the condition-name value in the DATA DIVISION. You do not need to make changes in the PROCEDURE DIVISION.

Switches: For example, to test for an end-of-file condition for an input file named Transaction-File, you could use the following data definitions:

```
Working-Storage Section.
01  Switches.
    05  Transaction-EOF-Switch       Pic X  value space.
        88  Transaction-EOF                 value "y".
```

The level-88 description says a condition named Transaction-EOF is turned on when Transaction-EOF-Switch has a value of "y". Referencing Transaction-EOF in your PROCEDURE DIVISION expresses the same condition as testing for `Transaction-EOF-Switch = "y"`. For example, the statement:

```
If Transaction-EOF Then Perform Print-Report-Summary-Lines
```

causes the report to be printed only if your program has read through to the end of the Transaction-File and if the Transaction-EOF-Switch has been set to "y".

Flags: To test for more than two values, assign more than one condition-name to a field by using multiple level-88 items.

Consider a program that updates a master file. The updates are read from a transaction file. The transaction file's records contain a field for the function to be performed: add, change, or delete. In the input file's record description, code a field for the function code using level-88 items:

```
01  Transaction-Input Record
    05  Transaction-Type             Pic X.
        88  Add-Transaction          Value "A".
        88  Change-Transaction       Value "C".
        88  Delete-Transaction       Value "D".
```

The code in the PROCEDURE DIVISION for testing these condition-names might look like this:

```
Evaluate True
  When Add-Transaction
    Perform Add-Master-Record-Paragraph
  When Change-Transaction
    Perform Update-Exisitng-Record-Paragraph
  When Delete-Transaction
    Perform Delete-Master-Record-Paragraph
End-Evaluate
```

Resetting Condition-Names (Switches and Flags)

Throughout your program, you might need to reset your switches or change your flags back to the original values they have in their data descriptions. To do so, you can use either a SET statement or define your own data item to use.

SET condition-name TO TRUE: When you use the SET *condition-name* TO TRUE statement, the switch or flag is set back to the original value it was assigned in its data description.

This method makes it easy for the reader to follow your code if you choose meaningful condition-names and if the value assigned has some association with a logical value of `True`.

The SET statement in the following example does the same thing as `Move "y" to Transaction-EOF-Flag`:

```
01 Switches
   05  Transaction-EOF-Switch        Pic X  Value space.
       88  Transaction-EOF                  Value "y".
         .
         .
         .
Procedure Division.
    000-Do-Main-Logic.
      Perform 100-Initialize-Paragraph
      Read Update-Transaction-File
          At End Set Transaction-EOF to True
      End-Read
```

The following example shows how you can assign a value for a field in an output record based on the transaction code of an input record.

```
01  Input-Record.
    05  Transaction-Type             Pic X(9).
         .
         .
         .
01  Data-Record-Out.
    05  Data-Record-Type             Pic X.
        88 Record-Is-Active          Value "A".
        88 Record-Is-Suspended       Value "S".
        88 Record-Is-Deleted         Value "D".
    05  Key-Field                    Pic X(5).
         .
         .
         .
```

```
   Procedure Division.
            .
            .
            .
       Evaluate Transaction-Type of Input-Record
         When "ACTIVE"
           Set Record-Is-Active to TRUE
         When "SUSPENDED"
           Set Record-Is-Suspended to TRUE
         When "DELETED"
           Set Record-Is-Deleted to TRUE
       End-Evaluate
```

Level-88 Note: For a level-88 item with multiple values (such as `88 Record-is-Active   Value "A" "O" "S"`), SET *condition-name* TO TRUE assigns the first value (here, `A`).

SWITCH-OFF: Establish a data item with this description:

```
     01  SWITCH-OFF      Pic X  Value "n".
```

Then use SWITCH-OFF throughout your program to set on/off switches to off. With this method, whoever reads your code can easily see what you are doing to a switch. From this code:

```
   01  Switches
       05  Transaction-EOF-Switch       Pic X  Value space.
           88  Transaction-EOF                 Value "y".
   01  SWITCH-OFF                       Pic X  Value "n".
          .
          .
          .
   Procedure Division.
          .
          .
          .
       Move SWITCH-OFF to Transaction-EOF-Switch
```

it is easy to see that you are setting the end-of-file switch to off. In other words, you have reset the switch to indicate that the end of the file has not been reached.

Iterative Loops (PERFORM Statement)

For looping (repeating the same code), use one of the forms of the PERFORM statement. You can use the PERFORM statement to loop a set number of times or to loop based on the outcome of a decision.

PERFORM statements can be inline or out-of-line.

Use the PERFORM statement to run a paragraph and then implicitly return control to the next executable statement. In effect, the PERFORM statement is a way of coding a closed subroutine that you can enter from many different parts of the program.

Repeating Program Actions

Coding a Loop to Be Performed a Definite Number of Times

Use the PERFORM . . . TIMES statement to execute a paragraph a certain number of times:

```
PERFORM 010-PROCESS-ONE-MONTH 12 TIMES
INSPECT . . .
```

When control reaches the PERFORM statement, the code for the paragraph 010-PROCESS-ONE-MONTH is executed 12 times before control is transferred to the INSPECT statement.

Conditional Looping

Use the PERFORM . . . UNTIL statement to execute a paragraph until a condition you choose is satisfied. You can use either of the following forms:

```
PERFORM . . . WITH TEST AFTER . . . UNTIL . . .
PERFORM . . . [WITH TEST BEFORE] . . . UNTIL . . .
```

In the following example, the implicit WITH TEST BEFORE phrase provides a do-while structure:

```
PERFORM  010-PROCESS-ONE-MONTH
  UNTIL MONTH EQUAL DECEMBER
INSPECT . . .
```

When control reaches the PERFORM statement, the condition (MONTH EQUAL DECEMBER) is tested. If the condition is satisfied, control is transferred to the INSPECT statement. If the condition is not satisfied, 010-PROCESS-ONE-MONTH is executed, and the condition is tested again. This cycle continues until the condition tests as true. (To make your program easier to read, you might want to code the WITH TEST BEFORE clause.)

Use the PERFORM . . . WITH TEST AFTER . . . UNTIL if you want to execute the paragraph at least once and then test before any subsequent execution. This is equivalent to the do-until structure.

Looping through a Table

Use the PERFORM statement to control a loop through a table. You can use either of the following forms:

```
PERFORM . . . WITH TEST AFTER . . . VARYING . . . UNTIL . . .
PERFORM . . . [WITH TEST BEFORE] . . . VARYING . . . UNTIL . . .
```

PERFORM. . .WITH TEST AFTER Example

You can use PERFORM . . . VARYING to initialize a table. In this form of the PERFORM statement, a variable is increased or decreased and tested until a condition is satisfied. The following code shows an example of looping through a table to check for invalid data:

```
  *** BLANK FIELDS ARE NOT ALLOWED IN THE INPUT DATA  ***

PERFORM TEST AFTER VARYING WS-DATA-IX
 FROM 1 BY 1
 UNTIL WS-DATA-IX = 12
 IF WS-DATA (WS-DATA-IX) EQUALS SPACES
   SET SERIOUS-ERROR TO TRUE
   DISPLAY ELEMENT-NUM-MSG5
 END-IF
END-PERFORM

INSPECT . . .
```

In the code above, when control reaches the PERFORM statement, WS-DATA-IX is set equal to 1 and the PERFORM statement is executed. Then the condition (WS-DATA-IX = 12) is tested. If the condition is true, control drops through to the INSPECT statement. If it is false, WS-DATA-IX is increased by 1, the PERFORM statement is executed, and the condition is tested again. This cycle of execution and testing continues until WS-DATA-IX is equal to 12.

In terms of the application, this loop controls input-checking for the 12 fields of item WS-DATA. Empty fields are not allowed, and this section of code loops through and issues error messages, as appropriate.

Executing a Group of Paragraphs or Sections

In structured programming, the paragraph you execute is usually a single paragraph. However, you can execute a group of paragraphs, a single section, or a group of sections using the PERFORM . . . THRU. statement.

WHEN YOU USE PERFORM . . . THRU use a paragraph-EXIT statement to clearly indicate the end point for the series of paragraphs.

Intrinsic functions can make the task of the iterative processing of tables simpler and easier for you to code. For information on using the ALL subscript with intrinsic functions to reference all the items in a table, see “Processing Table Items (Intrinsic Functions)” on page 63.

Chapter 6. String Handling

COBOL provides language constructs for performing these operations associated with string data items:

Figure 21. COBOL Data Constructs for Manipulating Strings

What You Want to Do	What to Use	Where to Look
Join data items	STRING Statement	On page 74
Split data items	UNSTRING Statement	On page 76
Manipulate null-terminated strings	Usual string handling statements.	On page 79
Reference substrings of data items	Reference modifiers	On page 80
Tally and replace data items	INSPECT statement	On page 83
Convert data items	Intrinsic functions UPPER-CASE, LOWER-CASE, REVERSE, NUMVAL, and NUMVAL-C	On page 85
Evaluate data items	Intrinsic functions CHAR, ORD, MAX, MIN, ORD-MAX, ORD-MIN, LENGTH, and WHEN-COMPILED	On page 87

Joining Data Items (STRING Statement)

Use the STRING statement to join all or parts of several data items into one data item. One STRING statement can save you several MOVE statements.

The STRING statement transfers data into the receiving item in the order you indicate. In the STRING statement you can also specify:

- Delimiters that cause a sending field to be ended and another to be started
- Special actions to be taken when an ON OVERFLOW condition occurs (when the single receiving field is filled before all of the sending characters have been processed).

STRING Statement Example

In the following example, an input record is read, and the STRING statement is used to select and format information as an output line consisting of a line number, customer name and address, invoice number, next billing date, and balance due, truncated to the dollar figure shown. (The symbol ƀ indicates a blank space.)

In the FILE SECTION, the following records are defined:

```
01  RCD-01.
    05  CUST-INFO.
        10  CUST-NAME              PIC X(15).
        10  CUST-ADDR              PIC X(35).
    05  BILL-INFO.
        10  INV-NO                 PIC X(6).
        10  INV-AMT                PIC $$,$$$.99.
        10  AMT-PAID               PIC $$,$$$.99.
        10  DATE-PAID              PIC X(8).
        10  BAL-DUE                PIC $$,$$$.99.
        10  DATE-DUE               PIC X(8).
```

In the WORKING-STORAGE SECTION, the following fields are defined:

```
77  RPT-LINE                       PIC X(120).
77  LINE-POS                       PIC S9(3).
77  LINE-NO                        PIC 9(5) VALUE 1.
77  DEC-POINT                      PIC X VALUE ".".
```

The record, as read, contains the following information:

```
J.B.ƀSMITHƀƀƀƀƀ
444ƀSPRINGƀST.,ƀCHICAGO,ƀILL.ƀƀƀƀƀ
A14275
$4,736.85
$2,400.00
09/22/76
$2,336.85
10/22/76
```

In the PROCEDURE DIVISION, the programmer initializes RPT-LINE to SPACES and sets LINE-POS, the data item to be used as the POINTER field, to 4. (By coding the POINTER phrase of the STRING statement, you can use the explicit pointer field to control placement of data in the receiving field.) Then, the programmer issues this STRING statement:

```
STRING
   LINE-NO SPACE CUST-INFO INV-NO SPACE DATE-DUE SPACE
      DELIMITED BY SIZE
   BAL-DUE
      DELIMITED BY DEC-POINT
   INTO RPT-LINE
   WITH POINTER LINE-POS.
```

STRING Program Results

When the statement is performed, the following steps take place:

1. The field LINE-NO is moved into positions 4 through 8 of RPT-LINE .
2. A space is moved into position 9.
3. The group item CUST-INFO is moved into positions 10 through 59.
4. INV-NO is moved into positions 60 through 65.
5. A space is moved into position 66.
6. DATE-DUE is moved into positions 67 through 74.

7. A space is moved into position 75.
8. The portion of BAL-DUE that precedes the decimal point is moved into positions 76 through 81.
9. The value of LINE-POS is 82 after the STRING statement is performed.

After the STRING statement is performed, RPT-LINE appears as shown in the following:

```
Column

 4    10                                         60     67       76
 │     │                                          │      │        │
 ▼     ▼                                          ▼      ▼        ▼
 00001 J.B. SMITH  444 SPRING ST., CHICAGO, ILL   A14275 10/22/76 $2,336
```

Splitting Data Items (UNSTRING Statement)

Use the UNSTRING statement to split one sending field into several receiving fields. One UNSTRING statement can save you several MOVE statements.

You can indicate delimiters that, when encountered in the sending field, cause the current receiving field to be switched to the next one indicated. You might have the number of characters placed in each receiving field returned to you, and keep a count of the total number of characters transferred. You might also specify special actions for the program to take if all the receiving fields are filled before the end of the sending item is reached.

UNSTRING Statement Example

In the following example, selected information is taken from the input record; some is organized for printing and some for further processing.

In the FILE SECTION, the following records are defined:

```
*  Record to be acted on by the UNSTRING statement:
 01  INV-RCD.
     05  CONTROL-CHARS                PIC XX.
     05  ITEM-INDENT                  PIC X(20).
     05  FILLER                       PIC X.
     05  INV-CODE                     PIC X(10).
     05  FILLER                       PIC X.
     05  NO-UNITS                     PIC 9(6).
     05  FILLER                       PIC X.
     05  PRICE-PER-M                  PIC 99999.
     05  FILLER                       PIC X.
     05  RTL-AMT                      PIC 9(6).99.
*
*  UNSTRING receiving field for printed output:
 01  DISPLAY-REC.
     05  INV-NO                       PIC X(6).
     05  FILLER                       PIC X VALUE SPACE.
     05  ITEM-NAME                    PIC X(20).
     05  FILLER                       PIC X VALUE SPACE.
     05  DISPLAY-DOLS                 PIC 9(6).

*
*  UNSTRING receiving field for further processing:
 01  WORK-REC.
     05  M-UNITS                      PIC 9(6).
     05  FIELD-A                      PIC 9(6).
     05  WK-PRICE REDEFINES FIELD-A   PIC 9999V99.
     05  INV-CLASS                    PIC X(3).
*
*  UNSTRING statement control fields
 77  DBY-1                            PIC X.
 77  CTR-1                            PIC S9(3).
 77  CTR-2                            PIC S9(3).
 77  CTR-3                            PIC S9(3).
 77  CTR-4                            PIC S9(3).
 77  DLTR-1                           PIC X.
 77  DLTR-2                           PIC X.
 77  CHAR-CT                          PIC S9(3).
 77  FLDS-FILLED                      PIC S9(3).
```

In the PROCEDURE DIVISION, the programmer writes the following UNSTRING statement:

Splitting Data Items

```
  * Move subfields of INV-RCD to the subfields of DISPLAY-REC
  *   and WORK-REC:
   UNSTRING INV-RCD
            DELIMITED BY ALL SPACES OR  "/" OR DBY-1
            INTO      ITEM-NAME   COUNT IN CTR-1
                      INV-NO DELIMITER IN DLTR-1 COUNT IN CTR-2
                      INV-CLASS
                      M-UNITS     COUNT IN CTR-3
                      FIELD A
                      DISPLAY-DOLS DELIMITER IN DLTR-2 COUNT IN CTR-4
            WITH POINTER CHAR-CT
            TALLYING IN FLDS-FILLED
            ON OVERFLOW GO TO UNSTRING-COMPLETE.
```

Before issuing the UNSTRING statement, the programmer places the value 3 in CHAR-CT (the POINTER field) to avoid working with the two control characters in INV-RCD. A period (.) is placed in DBY-1 for use as a delimiter, and the value 0 (zero) is placed in FLDS-FILLED (the TALLYING field). The data is then read into INV-RCD, as shown in the following:

```
Column

1        10        20        30        40        50        60
|        |         |         |         |         |         |
▼        ▼         ▼         ▼         ▼         ▼         ▼
ZYFOUR–PENNY–NAILS     707890/BBA 475120 00122 000379.50
```

UNSTRING Program Results

When the UNSTRING statement is performed, the following steps take place:

1. Positions 3 through 18 (FOUR-PENNY-NAILS) of INV-RCD are placed in ITEM-NAME, left-justified in the area, and the unused character positions are padded with spaces. The value 16 is placed in CTR-1.
2. Because ALL SPACES is coded as a delimiter, the 5 contiguous SPACE characters are considered to be one occurrence of the delimiter.
3. Positions 24 through 29 (707890) are placed in INV-NO. The delimiter character, /, is placed in DLTR-1 , and the value 6 is placed in CTR-2.
4. Positions 31 through 33 are placed in INV-CLASS. See Note at end of list.
5. Positions 35 through 40 (475120) are examined and placed in M-UNITS. The value 6 is placed in CTR-3. See Note at end of list.
6. Positions 42 through 46 (00122) are placed in FIELD-A and right-justified in the area. The high-order digit position is filled with a 0 (zero). See Note at end of list.
7. Positions 48 through 53 (000379) are placed in DISPLAY-DOLS. The period (.) delimiter character in DBY-1 is placed in DLTR-2, and the value 6 is placed in CTR-4.

8. Because all receiving fields have been acted on and 2 characters of data in INV-RCD have not been examined, the ON OVERFLOW exit is taken, and execution of the UNSTRING statement is completed.

SPACE Note: In steps 4, 5, and 6, the delimiter is a SPACE, but because no field has been defined as a receiving area for delimiters, the SPACE is bypassed.

After the UNSTRING statement is performed, the fields contain:

- DISPLAY-REC contains:

  ```
  707890 FOUR-PENNY-NAILS           000379
  ```

- WORK-REC contains:

  ```
  475120000122BBA
  ```

- CHAR-CT (the POINTER field) contains the value 55.
- FLDS-FILLED (the TALLYING field) contains the value 6.

Manipulating Null-Terminated Strings

Null-terminated strings are supported using syntax shown in the *IBM COBOL Language Reference*. You can construct and manipulate null-terminated strings passed to or from a C program, for example, by using string handling mechanisms such as:

- Using null terminated literal constants (Z" ... ").
- Using INSPECT statement to count number of characters in a null-terminated string:

  ```
  MOVE 0 TO char-count
  INSPECT source-field TALLYING char-count
                       FOR CHARACTERS
                       BEFORE X"00"
  ```

- Using UNSTRING statement to move characters in a null-terminated string to a target-field and get the character count:

  ```
  WORKING-STORAGE SECTION.
   ...
  01  source-field          PIC X(1001).
  01  char-count     COMP-5 PIC 9(4).
  01  target-area.
      02 individual-char OCCURS 1 TO 1000 TIMES DEPENDING ON char-count
                            PIC X.
   ...
  PROCEDURE DIVISION.
   ...
      UNSTRING source-field DELIMITED BY X"00"
                            INTO target-area
                            COUNT IN char-count
        ON OVERFLOW
          DISPLAY "source not null terminated or target too short"
           ...
      END-UNSTRING
       ...
  ```

- Using a SEARCH statement to locate trailing null or space characters (Define the string being examined as a table of single characters.)
- Checking each character in a field in a loop (PERFORM) looking at each character of the field (Each character in the field can be examined using a reference modifier such as `source-field (I:1)`.)

The following example shows the use of several of these mechanisms:

```
 01 L pic X(20) value z'ab'.
 01 M pic X(20) value z'cd'.
 01 N pic X(20).
 01 N-Length pic 99 value zero.
 01 Y pic X(13) value 'Hello, World!'.

* Display null-terminated string
     Inspect N tallying N-length
      for characters before initial x'00'
     Display 'N: ' N(1:N-length) ' Length: ' N-length

* Move null-terminated string to alphanumeric, strip null
     Unstring N delimited by X'00' into X

* Create null-terminated string
     String Y     delimited by size
            X'00' delimited by size
            into N.

* Concatenate two null-terminated strings
     String L     delimited by x'00'
            M     delimited by x'00'
            X'00' delimited by size
            into N.
```

Figure 22. Handling Null-Terminated Strings

Referencing Substrings of Data Items (Reference Modifiers)

Reference a substring of character-string data item items (including ASCII data items) with reference modifiers. Intrinsic functions that return character-string values are also considered alphanumeric data items, and can include a reference modifier.

The following example shows how to use a reference modifier to reference a substring of a data item:

```
Move Customer-Record(1:20) to Orig-Customer-Name
```

As this shows, in parentheses immediately following the data item you code the ordinal position (from the left) of the character you want the substring to start with and the length of the desired substring, separated by a colon.

The length is optional. If you omit the length, the substring created will automatically extend to the end of the item. Omitting the length, when possible, is recommended as a simpler, less error-prone coding technique.

These values can be variables or expressions.

Common Reference Modification Mistakes

If the leftmost character position or the length value is a fixed-point non-integer, truncation will occur to create an integer; if it is a floating-point non-integer, rounding will occur to create an integer.

Both numbers in the reference modifier must be at least 1, and their sum should not exceed the total length of the data item.

The following options detect out-of-range reference modifiers and flag violations with a run-time message:

- SSRANGE compiler option, discussed on page 193.
- CHECK run-time option, discussed on page "CHECK" on page 240.

For additional information on reference modification, see *IBM COBOL Language Reference*.

Benefits of Reference Modification

Assume that you want to retrieve the current time from the system and display its value in an expanded format. You can retrieve the current time value from the system with the ACCEPT statement, which returns the hours, minutes, seconds, and hundredths of seconds in this format:

```
HHMMSSss
```

However, you might prefer to view the current time in this format:

```
HH:MM:SS
```

Without reference notification you would have to define data items for both formats, the one from the system and the one you want, and write code to convert from one format to the other.

With reference modification, you do not need to provide names for the subfields that describe the TIME elements. The only data definition needed is:

```
01  REFMOD-TIME-ITEM           PIC X(8).
```

The code to retrieve and expand the time value would appear as follows:

Referencing Substrings

```
   ACCEPT REFMOD-TIME-ITEM FROM TIME.
   DISPLAY "CURRENT TIME IS: "
  * Retrieve the portion of the time value that corresponds to
  *   the number of hours:
      REFMOD-TIME-ITEM (1:2)
      ":"
  * Retrieve the portion of the time value that corresponds to
  *   the number of minutes:
      REFMOD-TIME-ITEM (3:2)
      ":"
  * Retrieve the portion of the time value that corresponds to
  *   the number of seconds:
      REFMOD-TIME-ITEM (5:2).
```

Reference Modification of an Intrinsic Function

The simplest solution to our problem would be to reference a substring of the CURRENT-DATE function, which requires no DATA DIVISION entries and fewer lines of code.

```
   Display "Current Date is:  "
      Function Current-Date(9:2)
      ":"
      Function Current-Date(11:2)
      ":"
      Function Current-Date(13:2).
```

Using Arithmetic Expressions as Reference Modifiers

You can also use an arithmetic expression as either of the integers in a reference modifier. For example:

Suppose that a field contains some characters, right-justified, and you want to move the characters to another field, but justified to the left instead of the right. Using reference modification and an INSPECT statement, you could do that.

The program would have the following data:

```
   01  LEFTY                       PIC X(30).
   01  RIGHTY                      PIC X(30) JUSTIFIED RIGHT.
   01  I                           PIC 9(9)  USAGE BINARY.
```

The program would count the number of leading spaces and, using arithmetic expressions in a reference modification expression, move the right-justified characters into another field, left-justified:

```
   MOVE SPACES TO LEFTY
   MOVE ZERO TO I
   INSPECT RIGHTY
      TALLYING I FOR LEADING SPACE.
   IF I IS LESS THAN LENGTH OF RIGHTY THEN
      MOVE RIGHTY ( I + 1 : LENGTH OF RIGHTY - I ) TO LEFTY
   END-IF
```

The MOVE statement transfers characters from RIGHTY, beginning at the position computed in I + 1, for a length that is computed in LENGTH OF RIGHTY - I, into the field LEFTY.

Using Intrinsic Functions as Reference Modifiers

Because a numeric function-identifier can be used anywhere an arithmetic expression is allowed, it can be used as the leftmost character position and/or the length in the reference modifier.

For example:

```
05  WS-name          Pic x(20).
05  Left-posn        Pic 99.
05  I                Pic 99.
    .
    .
Move Customer-Record(Function Min(Left-posn I):Function Length(WS-name)) to WS-name
```

When performed, this statement causes a substring of Customer-Record to be moved into the variable WS-name; the substring is determined at run time.

If you want to use a numeric, non-integer function in a position requiring an integer function, you can use the INTEGER or INTEGER-PART function to convert the result to an integer. For example:

```
Move Customer-Record(Function Integer(Function Sqrt(I)): ) to WS-name
```

For a list that shows which numeric functions return integer and non-integer results, see *IBM COBOL Language Reference*.

Referencing Substrings of Table Items

You can also reference substrings of table entries, including variable-length entries. This is discussed in Chapter 4, “Handling Tables” on page 47.

Tallying and Replacing Data Items (INSPECT Statement)

The INSPECT statement is useful for:

- Filling selective portions of a data item with a value.
- Replacing portions with a corresponding portion of another data item.
- Counting the number of times a specific character (zero, space, asterisk, for example) occurs in a data item.

INSPECT Statement Examples

The following examples show some uses of the INSPECT statement. In all instances, the programmer has initialized the COUNTR field to zero before the INSPECT statement is performed.

Counting and Replacing Data Items

Example 1:

```
77  COUNTR                PIC   9   VALUE ZERO.
01  DATA-2                PIC X(11).
    .
    .
    INSPECT DATA-2
        TALLYING COUNTR FOR LEADING "0"
        REPLACING FIRST "A" BY "2" AFTER INITIAL "C"
```

DATA-2 Before	COUNTR After	DATA-2 After
00ACADEMY00	2	00AC2DEMY00
0000ALABAMA	4	0000ALABAMA
CHATHAM0000	0	CH2THAM0000

Example 2:

```
77  COUNTR                PIC   9   VALUE ZERO.
01  DATA-3                PIC X(8).
    .
    .
    INSPECT DATA-3
        REPLACING CHARACTERS BY ZEROS BEFORE INITIAL QUOTE
```

DATA-3 Before	COUNTR After	DATA-3 After
456"ABEL	0	000"ABEL
ANDES"12	0	00000"12
"TWAS BR	0	"TWAS BR

Example 3:

The following example shows the use of INSPECT CONVERTING with AFTER and BEFORE phrases. The table shows examples of the contents of DATA-4 before and after the conversion statement is performed.

```
01  DATA-4                PIC X(11).
    .
    .
    INSPECT DATA-4
        CONVERTING
           "abcdefghijklmnopqrstuvwxyz" TO
           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        AFTER INITIAL "/"
        BEFORE INITIAL"?"
```

DATA-4 Before	DATA-4 After
`a/five/?six`	`a/FIVE/?six`
`r/Rexx/RRRr`	`r/REXX/RRRR`
`zfour?inspe`	`zfour?inspe`

Converting Data Items (Intrinsic Functions)

Intrinsic functions are available to convert character-string data items to the following:

- Upper or lower case
- Reverse order
- Numbers

Besides using intrinsic functions to convert characters, you can also use the INSPECT statement. See the examples under “Tallying and Replacing Data Items (INSPECT Statement)” on page 83.

Converting to Uppercase or Lowercase (UPPER-CASE, LOWER-CASE)

This code:

```
01  Item-1            Pic x(30)  Value "Hello World!".
01  Item-2            Pic x(30).
    .
    .
    Display Item-1
    Display Function Upper-case(Item-1)
    Display Function Lower-case(Item-1)
    Move Function Upper-case(Item-1) to Item-2
    Display Item-2
```

would display the following messages on the terminal:

```
Hello World!
HELLO WORLD!
hello world!
HELLO WORLD!
```

The DISPLAY statements do not change the actual contents of `Item-1` and only affect how the letters are displayed. However, the MOVE statement causes uppercase letters to be moved to the actual contents of `Item-2`.

Converting to Reverse Order (REVERSE)

The following code:

```
Move Function Reverse(Orig-cust-name) To Orig-cust-name
```

would reverse the order of the characters in `Orig-cust-name`. For example, if the starting value was “JOHNSONƀƀƀ,” the value after the statement is performed would be “ƀƀƀNOSNHOJ.”

Converting Character Data Items

Converting to Numbers (NUMVAL, NUMVAL-C)

The NUMVAL and NUMVAL-C functions convert character strings to numbers. Use these functions to convert alphanumeric data items that contain free format character representation numbers to numeric form and process them numerically. For example:

```
01  R           Pic x(20)  Value "- 1234.5678".
01  S           Pic x(20)  Value "  $12,345.67CR".
01  Total       Usage is Comp-1.
    .
    .
    Compute Total = Function Numval(R) + Function Numval-C(S)
```

The difference between NUMVAL and NUMVAL-C is that NUMVAL-C is used when the argument includes a currency symbol and/or comma, as shown in the example. You can also place an algebraic sign in front or in the rear, and it will be processed. The arguments must not exceed 18 digits (not including the editing symbols). For exact syntax rules, see *IBM COBOL Language Reference*.

Numeric Result: Both NUMVAL and NUMVAL-C return long (double-precision) floating-point values. A reference to either of these functions, therefore, represents a reference to a numeric data item. For more information on characteristics of numeric data, see Chapter 3, “Numbers and Arithmetic” on page 29.

Why Use NUMVAL and NUMVAL-C

When you use NUMVAL or NUMVAL-C you don't need to statically declare numeric data in a fixed format and input data in a precise manner. For example, for this code:

```
01  X           Pic S999V99  leading sign is separate.
    .
    .
    Accept X from Console
```

The user of the application must enter the numbers exactly as defined by the PICTURE clause. For example:

```
+001.23
-300.00
```

However, using the NUMVAL function, you could code:

```
01  A           Pic x(10).
01  B           Pic S999V99.
    .
    .
    Accept A from Console
    Compute B = Function Numval(A)
```

and the input could be:

```
 1.23
 -300
```

Evaluating Data Items (Intrinsic Functions)

Several intrinsic functions can be used in evaluating data items:

- CHAR and ORD for evaluating integers and single alphanumeric characters with respect to the collating sequence used in your program.
- MAX, MIN, ORD-MAX, and ORD-MIN for finding the largest and smallest items in a series of data items.
- LENGTH for finding the length of data items.
- WHEN-COMPILED for finding the date and time the program was compiled.

Evaluating Single Characters for Collating Sequence (CHAR, ORD)

If you want to know the ordinal position of a certain character in the collating sequence, reference the ORD function using the character in question as the argument, and ORD will return an integer representing that ordinal position. One convenient way to do this is to use the substring of a data item as the argument to ORD:

```
IF Function Ord(Customer-record(1:1)) Is > 194 THEN ...
```

On the other hand, if you know what position in the collating sequence you want but don't know what character it corresponds to, then reference the CHAR function using the integer ordinal position as the argument, and CHAR will return the desired character:

```
INITIALIZE Customer-Name REPLACING ALPHABETIC BY Function Char(65)
```

Finding the Largest or Smallest Data Item (MAX, MIN, ORD-MAX, ORD-MIN)

If you have two or more alphanumeric data items and want to know which data item contains the largest value (evaluated according to the collating sequence), use the MAX or ORD-MAX function, supplying the data items in question as arguments. If you want to know which item contains the smallest value, you would use the MIN or ORD-MIN function.

MAX and MIN

The MAX and MIN functions simply return the contents of one of the variables you supply.

For example, with these data definitions:

```
05  Arg1        Pic x(10)  Value "THOMASSON ".
05  Arg2        Pic x(10)  Value "THOMAS    ".
05  Arg3        Pic x(10)  Value "VALLEJO   ".
```

the following statement:

```
Move Function Max(Arg1 Arg2 Arg3) To Customer-record(1:10)
```

would assign “VALLEJOƀƀƀ” to the first ten character positions of `Customer-record`.

If MIN were used instead, then “THOMASƀƀƀƀ” would be returned.

ORD-MAX and ORD-MIN

The functions ORD-MAX and ORD-MIN return an integer that represents the ordinal position of the argument with the largest or smallest value in the list of arguments you have supplied (counting from the left).

If the ORD-MAX function were used in the example above, you would receive a syntax error message at compile time, because you would be attempting to reference a numeric function in an invalid place (see *IBM COBOL Language Reference*). The following is a valid example of the ORD-MAX function:

```
Compute x = Function Ord-max(Arg1 Arg2 Arg3)
```

This would assign the integer 3 to x, if the same arguments were used as in the previous example. If ORD-MIN were used instead, the integer 2 would be returned.

Notes on MAX, MIN, ORD-MAX, ORD-MIN

This group of functions can also be used for numbers, in which case the algebraic values of the arguments are compared. For more information, see the appropriate section of Chapter 3, “Numbers and Arithmetic” on page 29.

The above examples would probably be more realistic if Arg1, Arg2 and Arg3 were instead successive elements of an array (table). For information on using table elements as function arguments, see the section on “Processing Table Items (Intrinsic Functions)” on page 63 in Chapter 4, “Handling Tables.”

Returning Variable-Length Results with Alphanumeric Functions

The results of alphanumeric functions might be of varying lengths and values depending on the function arguments.

In the following example, the amount of data moved to R3 and the results of the COMPUTE statement depend on the values and sizes of R1 and R2:

```
01  R1                    Pic x(10) value "e".
01  R2                    Pic x(05) value "f".
01  R3                    Pic x(20) value spaces.
01  L                     Pic 99.
    .
    .
    Move Function Max(R1 R2) to R3
    Compute L = Function Length(Function Max(R1 R2))
```

Here, R2 is evaluated to be larger than R1. Therefore, assuming that the symbol ƀ represents a blank space, the string “fƀƀƀƀ” would be moved to R3 (the unfilled character positions in R3 are padded with spaces), and L evaluates to the value 5. If R1 were the value “g” then R1 would be larger than R2, and the string “gƀƀƀƀƀƀƀƀƀ” would be moved to R3 (the unfilled character positions in R3 would be padded with spaces); the value 10 would be assigned to L.

You might be dealing with variable-length output from alphanumeric functions. Plan your program code accordingly. For example, you might need to think about using variable-length record files when it is possible that the records you will be writing might be of different lengths:

```
File Section.
FD  Output-File.
    01  Customer-Record   Pic X(80).

Working-Storage Section.
01  R1                    Pic x(50).
01  R2                    Pic x(70).
    .
    .
    Write Customer-Record from Function Max(R1 R2)
```

Finding the Length of Data Items (LENGTH)

The LENGTH function is useful in many programming contexts for determining the length of string items. The following COBOL statement shows moving a data item such as a customer name into the particular field in a record that is for customer names:

```
Move Customer-name To Customer-record(1:Function Length(Customer-name))
```

Numeric & Table: The LENGTH function can also be used on a numeric data item or a table entry. Numeric data and tables are discussed in Chapter 3, “Numbers and Arithmetic” on page 29 and in Chapter 4, “Handling Tables” on page 47.

LENGTH OF Special Register

In addition to the LENGTH function, another technique to find the length of a data item is to use the LENGTH OF special register. Coding either `Function Length(Customer-Name)` or `LENGTH OF Customer-Name` would return the same result— the length of `Customer-Name` in bytes.

Whereas the LENGTH function can only be used where arithmetic expressions are allowed, the LENGTH OF special register can be used in a greater variety of contexts. For example, the LENGTH OF special register can be used as an argument to an intrinsic function that allows integer arguments. (An intrinsic function cannot be used as an operand to the LENGTH OF special register.) The LENGTH OF special register can also be used as a parameter in a CALL statement.

Finding the Date of Compilation (WHEN-COMPILED)

If you want to know the date and time the program was compiled as provided by the system on which the program was compiled, you can use the WHEN-COMPILED function. The result returned has 21 character positions with the first 16 positions in the format:

```
YYYYMMDDhhmmsshh
```

to show the 4-digit year, month, day, and time (in hours, minutes, seconds, and hundredths of seconds) of compilation.

WHEN-COMPILED Special Register

The WHEN-COMPILED special register is another technique you can use to find the date and time of compilation. It has the format:

```
MM/DD/YYhh.mm.ss
```

The WHEN-COMPILED special register supports only a two-digit year and carries the time out only to seconds. This special register can only be used as the sending field in a MOVE statement.

Chapter 7. Processing Files

Reading and writing data to and from files is an essential part of every program. Your program retrieves information, processes it as you request, and then produces the results.

This chapter provides a brief introduction on file organization and access modes, describes the coding your COBOL programs need to identify and process files, and explains how files must be defined and identified to the operating system before your program can process them.

The topics in this chapter are:

Record-oriented files that are organized as sequential, relative, indexed, or line sequential (byte stream) files are accessed through a *file system*. An application can use file-system functions to create and manipulate the records in any of these types of files.

VisualAge COBOL supports the following file systems:

- The STL file system, which provides the basic facilities for local files. It is provided with VisualAge COBOL and supports sequential, relative, and indexed files.
- The VSAM file system, which allows you to read and write files on remote systems such as OS/390. It is provided with VisualAge COBOL and supports sequential, relative, and indexed files.

  OS/2 ▶ On OS/2, the VSAM file system supports local files as well as remote.
  ◀ OS/2

- The Btrieve file system, which allows you to access Btrieve files. Btrieve is a separate product available from Btrieve Technologies, Inc. (BTI).

  Note: By using the Btrieve file system you can access files created by VisualAge CICS Enterprise Application Development and CICS for OS/2.

Most programs will get the same results on all file systems. However, files written using one file system cannot be read using a different file system.

Two ways you can select a file system are by setting the assignment-name environment variable or by using the FILESYS run-time option. See “Accessing Files” on page 97 for futher details. All the file systems allow you to use COBOL statements to read or write COBOL files.

File Organization

If you have more complex requirements which are not covered in this book, or are going to be a frequent user of file systems, you should review the *Btrieve Programmer's Manual* and the publications for the SMARTdata Utilities for OS/2 or Windows, which are provided as part of the on-line documentation.

File Input/Output Overview

This section describes file organization and file access modes. You should decide on the file types you will use when you design your program. Your file management system handles the input/output requests and record retrieval from the input/output devices.

Figure 23 summarizes file organization, access modes, and record lengths for COBOL files.

Figure 23. File Organizations and Access Modes

File Organization	Sequential Access	Random Access	Dynamic Access
Sequential	Yes	No	No
Line sequential	Yes	No	No
Indexed	Yes	Yes	Yes
Relative	Yes	Yes	Yes

File I/O Limitations:

- For line sequential files, the maximum record size is 64K.
- For VSAM files:
  - Minimum record size: 1 byte
  - Maximum record size: 64,000 bytes
  - Maximum record key length: 255 bytes
  - Maximum relative key value: 2**32-2
  - Maximum number of bytes allocated for a file: 2**32
- For STL files:
  - Minimum record size: 1 byte
  - Maximum record size: 65536 bytes
  - Maximum record key length: 255 bytes
  - Maximum number or alternate keys: 253 bytes
  - Maximum relative key value: 2**32-1
  - Maximum number of bytes allocated for a file: 2**32-1

Additional or more restrictive limits might be applicable depending on the platform on which the target file is located. See the appropriate books for the file system of the target platform for these limits.

File Organization

You can organize your files as sequential, line sequential, indexed, or relative.

Sequential File Organization

A sequential file contains records organized by the order in which they are entered. The order of the records is fixed.

Records in sequential files can only be read or written sequentially.

After you have placed a record into the file, you cannot shorten, lengthen, or delete it. However, you can update (REWRITE) a record if the length does not change. New records are added at the end of the file.

Line Sequential File Organization

Line sequential files are just like sequential files, except that the records can contain only characters as data. Line sequential files are supported by the native byte stream files of the operating system.

Line sequential files that are created with WRITE statements with the ADVANCING phrase can be directed to a printer (as well as a disk).

Indexed File Organization

An indexed file contains records ordered by a record key. Each record contains a field that contains the record key. The record key uniquely identifies the record and determines the sequence in which it is accessed with respect to other records. A record key for a record might be, for example, an employee number or an invoice number.

An indexed file can also use alternate indexes—record keys that let you access the file using a different logical arrangement of the records. For example, you could access the file through employee department rather than through employee number.

The record transmission (access) modes allowed for indexed files are sequential, random, or dynamic. When indexed files are read or written sequentially, the sequence is that of the key values. For a description of random and dynamic record transmission, see “File Access Modes” on page 94.

Relative File Organization

A relative record file contains records ordered by their relative key—the relative key being the relative record number representing the record's location relative to where the file begins. For example, the first record in the file has a relative record number of 1, the tenth record has a relative record number of 10, and so forth. The relative record number identifies the fixed-or variable-length record.

The record transmission modes allowed for relative files are sequential, random, or dynamic. When relative files are read or written sequentially, the sequence is that of the relative record number. For a description of random and dynamic record transmission, see “File Access Modes” on page 94.

Figure 24 (Page 1 of 2). Comparison of Different Files

Sequential	Line Sequential	Indexed	Relative
Records are in the order in which they are written.	Records are in the order in which they are written.	Records are in collating sequence by key field.	Records are in relative record number order.

Figure 24 (Page 2 of 2). Comparison of Different Files

Sequential	Line Sequential	Indexed	Relative
Access is sequential.	Access is sequential.	Access is by key through an index. Can have one or more alternate indexes.	Access is by relative record number, which is handled like a key.
A record cannot be deleted, but you can reuse its space for a record of the same length.	A record cannot be deleted or replaced.	Records can be deleted or replaced.	Records can be deleted or replaced.

File Access Modes

You can access records in sequential and line sequential files sequentially only.

You can access records in indexed and relative files in three ways: sequentially, randomly, or dynamically.

Sequential Access

Code ACCESS IS SEQUENTIAL in the FILE-CONTROL entry.

For indexed files, records are accessed in the order of the key field selected (either primary or alternate).

For relative files, records are accessed in the order of the relative record numbers.

Random Access

Code ACCESS IS RANDOM in the FILE-CONTROL entry.

For indexed files, records are accessed according to the value you place in a key field.

For relative files, records are accessed according to the value you place in the relative key.

Dynamic Access

Code ACCESS IS DYNAMIC in the FILE-CONTROL entry.

Dynamic access is a mixed sequential-random access in the same program. Using dynamic access, you can use one COBOL file definition to perform both sequential and random processing, accessing some records in sequential order and others by their keys.

For example, suppose you had an indexed file of employee records, and the employee's hourly wage formed the record key. Also, suppose your program was interested in those employees earning between $7.00 and $9.00 per hour and those earning $15.00 per hour and above. To do this, retrieve the first record randomly (with a random-retrieval READ) based on the key of 0700. Next, begin reading sequentially (using READ NEXT) until the salary field exceeds 0900. You would then switch back to a random read, this time based on a key of 1500. After this random read, switch back to reading sequentially until you reach the end of the file.

COBOL Coding for Files

Code your COBOL program according to the types of files you decide to use. The general format of input/output coding is shown in Figure 25. Explanations of user-supplied information (lowercase) follow the figure.

```
IDENTIFICATION DIVISION.
    .
    .
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT filename ASSIGN TO assignment-name  1  2
    ORGANIZATION IS org ACCESS MODE IS access  3  4
    FILE STATUS IS file-status  5
    .
    .
DATA DIVISION.
  FILE SECTION.
   FD filename
   01 recordname  6
      nn . . . fieldlength & type  7  8
      nn . . . fieldlength & type
    .
    .
WORKING-STORAGE SECTION
 01 file-status  PICTURE 99.
    .
    .
PROCEDURE DIVISION.
    .
    .
    OPEN iomode filename   9
    .
    .
    READ filename
    .
    .
    WRITE recordname
    .
    .
    CLOSE filename
    .
    .
  STOP RUN.
```

Figure 25. Overview of COBOL Input/Output Coding

The user-supplied information in Figure 25 can be explained as follows:

COBOL Coding for Files

1 filename
Any valid COBOL name. You must use the same filename on the SELECT and the FD statements, and on the OPEN, READ, START, DELETE, and CLOSE statements. This name is not necessarily the actual name of the file as known to the system. Each file requires its own SELECT, FD, and input/output statements. For WRITE and REWRITE, you specify a record defined for the file.

2 assignment-name
You can specify ASSIGN TO *assignment-name* to specify the target file-system ID and the file of the name as known to the system directly, or you can set a value indirectly by using an environment variable.

If you want to have the system file name identified at OPEN time, you can specify ASSIGN USING *data-name*. The value of *data-name* at the time of the execution of the OPEN statement for the file is used and has the system file identification optionally preceded by the file-system type identification.

The following example illustrates how `inventory-file` is dynamically (by way of a MOVE statement) associated with a file `d:\inventory\parts`.

```
SELECT inventory-file ASSIGN TO a-file
  ⋮
    MOVE "d:\inventory\parts" TO a-file
    OPEN INPUT inventory-file
```

For more information, see *IBM COBOL Language Reference*.

3 org
Indicates the organization: LINE SEQUENTIAL, SEQUENTIAL, INDEXED, or RELATIVE. If this clause is omitted, the default is ORGANIZATION SEQUENTIAL.

4 access
Indicates the access mode, SEQUENTIAL, RANDOM, or DYNAMIC. If this clause is omitted, the default is ACCESS SEQUENTIAL.

5 file-status
The 2-character COBOL FILE STATUS key.

6 recordname
The name of the record used in the WRITE and REWRITE statements. You can specify more than one record for a file.

7 fieldlength
The logical length of the field.

8 type
Must match the file's record format. If you break the record description entry beyond the level-01 description, each element should map accurately against the record's fields.

9 iomode
Specifies the open mode. For example, if you are only reading from a file, code INPUT. If you are only writing to it, code OUTPUT or EXTEND. If you are doing both, code I-O.

Line Sequential: I-O is not a valid parameter of OPEN for line sequential files.

Accessing Files

Your programs are able to access STL, VSAM, and Btrieve (Btrieve Technologies, Inc.) files.

Windows ▶ On Windows, only remote files are supported using VSAM. ◀ Windows

Use *assignment-name* to specify both the file you want to access and the file system to be used. For a detailed description of *assignment-name*, see the *IBM COBOL Language Reference*.

The general syntax involved in making an assignment to a file stored in an alternate file system is:

```
SELECT file ASSIGN TO FileSystemID-Filename
```

FileSystemID
Identifies the file system as one of the following:

STL For the STL file system.

VSAM For the VSAM file system. VSAM can be abbreviated to VSA.

Windows ▶ On Windows, `Filename` must start with "\\", indicating remote file access. ◀ Windows

BTR For the Btrieve file system.

If the file-system specification is not provided, then the run-time option FILESYS is used to select the file system. If a file system is not specified using FILESYS, the default is VSAM on OS/2 and STL on Windows.

Filename
The file you want to access. Alternatively, you can specify an environment variable to allow you to specify the file name at run time. For details, see “Run-Time Environment Variables” on page 137, and the *IBM COBOL Language Reference*.

Usage Note: The following file status indicators are not set for Btrieve:

02
21
39

Example—Accessing Btrieve Files

- To use the Btrieve file system, the following assignment would be valid:

  ```
  SELECT file1 ASSIGN USING 'BTR-MyFile'
  ```

- If the run-time option FILESYS(BTRIEVE) was in effect, the following assignment would be valid:

  ```
  SELECT file1 ASSIGN TO 'MyFile'
  ```

- Given that you have defined the environment variable MYFILE (for example, `SET MYFILE=BTR-MYFILE`), the following assignment would be valid:

  ```
  SELECT file1 ASSIGN TO MYFILE
  ```

Example—Accessing STL Files

- To use the STL file system, the following assignment would be valid:

  ```
  SELECT file1 ASSIGN USING 'STL-MyFile'
  ```

- If the run-time option FILESYS(STL) was in effect, the following assignment would be valid:

  ```
  SELECT file1 ASSIGN TO 'MyFile'
  ```

- Given that you have defined the environment variable MYFILE , (for example, `SET MYFILE=STL-MYFILE`), the following assignment would be valid:

  ```
  SELECT file1 ASSIGN TO MYFILE
  ```

Distributed File Access

Using the Distributed File feature of the SMARTdata Utilities, you can access a remote file (for example, OS/390 VSAM, SAM, or PDS) without any source program change.

In the following example, the SELECT clause is used to associate a file on OS/390 with a file in your workstation program:

```
SELECT myfile ASSIGN TO TARGETFILE
```

OS/2 ▶ On OS/2, you can associate `myfile` to an OS/390 file called MVSMAST by setting the TARGETFILE environment variable:

```
set TARGETFILE=m:MVSMAST
```

where the drive `m` is set to point to the specific OS/390 system and MVSMAST is the data set name on the OS/390 system. ◀ OS/2

See *VSAM in a Distributed Environment* for more information.

Coding Input/Output Statements for Files

After identifying and describing the files in the ENVIRONMENT DIVISION and DATA DIVISION, process the file records in the PROCEDURE DIVISION of your program.

Figure 26 shows the possible combinations of input/output statements for sequential files. The 'X' indicates that the statement can be used with the open mode given at the top of the column.

Figure 26 (Page 1 of 2). Valid COBOL Statements for Sequential Files

Access Mode	COBOL Statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Sequential	OPEN	X	X	X	X
	WRITE		X		X

Figure 26 (Page 2 of 2). Valid COBOL Statements for Sequential Files

Access Mode	COBOL Statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
	START				
	READ	X		X	
	REWRITE			X	
	DELETE				
	CLOSE	X	X	X	X

Figure 27 shows the possible combinations of input/output statements for line sequential files. The 'X' indicates that the statement can be used with the open mode given at the top of the column.

Figure 27. Valid COBOL Statements for Line Sequential Files

Access Mode	COBOL Statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Sequential	OPEN	X	X		X
	WRITE		X		X
	START				
	READ	X			
	REWRITE				
	DELETE				
	CLOSE	X	X		X

Figure 28 shows the possible combinations with indexed and relative files. The 'X' indicates that the statement can be used with the open mode given at the top of the column.

Figure 28 (Page 1 of 2). Valid COBOL Statements with Indexed Files and Relative Files

Access Mode	COBOL Statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Sequential	OPEN	X	X	X	X
	WRITE		X		X
	START	X		X	
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	X

Figure 28 (Page 2 of 2). Valid COBOL Statements with Indexed Files and Relative Files

Access Mode	COBOL Statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Random	OPEN	X	X	X	
	WRITE		X	X	
	START				
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	
Dynamic	OPEN	X	X	X	
	WRITE		X	X	
	START	X		X	
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	

File Position Indicator

The file position indicator marks the next record to be accessed for sequential COBOL requests. You do not set the file position indicator anywhere in your program; it is set by successful OPEN, START, READ, READ NEXT, and READ PREVIOUS statements. Subsequent READ, READ NEXT, or READ PREVIOUS requests use the established file position indicator location and update it.

The file position indicator is not used or affected by the output statements WRITE, REWRITE, or DELETE. The file position indicator has no meaning for random processing.

Opening a File

Before your program can use any WRITE, START, READ, REWRITE, or DELETE statements to process records in a file, it must first open the file with an OPEN statement. The OPEN processing is affected by whether or not the file exists, and whether or not the OPTIONAL attribute is specified on the file definition.

For example, an OPEN EXTEND of a file that is neither optional nor available results in file status 35, and the OPEN statement fails. If the file is OPTIONAL, the OPEN EXTEND will create the file and return file status 05.

COBOL Coding for Files

Figure 29 on page 101 shows the COBOL statements used when creating or extending a new or existing file.

Figure 29. Statements Used when Writing Records to a File

Division	Sequential	Line Sequential	Indexed	Relative
Environment Division	SELECT ASSIGN FILE STATUS ACCESS MODE	SELECT ASSIGN ORGANIZATION IS LINE SEQUENTIAL FILE STATUS ACCESS MODE	SELECT ASSIGN ORGANIZATION IS INDEXED RECORD KEY ALTERNATE RECORD KEY FILE STATUS ACCESS MODE	SELECT ASSIGN ORGANIZATION IS RELATIVE RELATIVE KEY FILE STATUS ACCESS MODE
Data Division	FD entry	FD entry	FD entry	FD entry
Procedure Division	OPEN OUTPUT OPEN EXTEND WRITE CLOSE	OPEN OUTPUT OPEN EXTEND WRITE CLOSE	OPEN OUTPUT OPEN EXTEND WRITE CLOSE	OPEN OUTPUT OPEN EXTEND WRITE CLOSE

Opening a File with Records

To open a file that already contains records, use OPEN INPUT, OPEN I-O, or OPEN EXTEND.

Line Sequential: OPEN I-O is not valid for line sequential files.

If you open a sequential, line sequential, or relative file as EXTEND, the added records are placed after the last existing records in the file.

If you open an indexed file as EXTEND, each record you add must have a record key higher than the highest record in the file.

Reading Records from a File

Use the READ statement to retrieve (read) records from a file. To read a record, you must have opened the file INPUT or I-O. Check the file status key after each READ.

Line Sequential: OPEN I-O is not valid for line sequential

Records in sequential and line sequential files can be retrieved only in the sequence in which they were written.

Records in indexed and relative record files can be retrieved:

Sequentially

According to the ascending order of the key you are using, the RECORD KEY or the ALTERNATE RECORD KEY, beginning at the current position of the file position indicator for indexed files, or according to ascending relative record locations for relative files.

Randomly
In any order, depending on how you set the RECORD KEY or ALTERNATE RECORD KEY or the RELATIVE KEY before your READ request.

Dynamically
Mixed sequential and random.

With dynamic access, you can switch between reading a specific record directly and reading records sequentially, by using READ NEXT and READ PREVIOUS for sequential retrieval, and READ for random retrieval (by key).

When you want to read sequentially, beginning at a specific record, use START before the READ NEXT or the READ PREVIOUS statements to set the file position indicator to point to a particular record (see “File Position Indicator” on page 100). When you code START followed by READ NEXT, the next record is read and the file position indicator is reset to the next record. When you code START followed by READ PREVIOUS, the previous record is read and the file position indicator is reset to the previous record. The file position indicator can be moved randomly by using START, but all reading is done sequentially from that point.

You can continue to read records sequentially, or you can use the START statement to move the file position indicator. For example:

```
START file-name KEY IS EQUAL TO ALTERNATE-RECORD-KEY
```

When a direct READ is performed for an indexed file, based on an alternate index for which duplicates exist, only the first record in the file (base cluster) with that alternate key value is retrieved. You need a series of READ NEXT statements to retrieve each of the data set records with the same alternate key. A FILE STATUS value of '02' is returned if there are more records with the same alternate key value to be read; a value of '00' is returned when the last record with that key value has been read.

Updating Records in a File

The COBOL language statements that can be used to update a file in the ENVIRONMENT DIVISION and DATA DIVISION are the same as those shown in Figure 29 on page 101.

Figure 30 shows the statements that you can use in the PROCEDURE DIVISION for sequential, line sequential, indexed, and relative files.

Figure 30. PROCEDURE DIVISION Statements Used to Update Files

Access Method	Sequential	Line Sequential	Indexed	Relative
ACCESS IS SEQUENTIAL	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE CLOSE	OPEN EXTEND WRITE CLOSE	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE DELETE CLOSE	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE DELETE CLOSE
ACCESS IS RANDOM	Not applicable	Not applicable	OPEN I-O READ WRITE REWRITE DELETE CLOSE	OPEN I-O READ WRITE REWRITE DELETE CLOSE
ACCESS IS DYNAMIC: Sequential Processing	Not applicable	Not applicable	OPEN I-O READ NEXT READ PREVIOUS START CLOSE	OPEN I-O READ NEXT READ PREVIOUS START CLOSE
ACCESS IS DYNAMIC: Random Processing	Not applicable	Not applicable	OPEN I-O READ WRITE REWRITE DELETE CLOSE	OPEN I-O READ WRITE REWRITE DELETE CLOSE

Adding Records to a File

The COBOL WRITE statement adds a record to a file, without replacing any existing records. The record to be added must not be larger than the maximum record size set when the file was defined. Check the file status key after each WRITE statement.

Adding Records Sequentially

Use ACCESS IS SEQUENTIAL and code the WRITE statement to add records sequentially to the end of a file that has been opened with either OUTPUT or EXTEND.

Sequential and line sequential files are always written sequentially.

For indexed files, new records must be written in ascending key sequence. If the file is opened EXTEND, the record keys of the records to be added must be higher than the highest primary record key on the file when the file was opened.

For relative files, the records must be in sequence. If you include a RELATIVE KEY data-item in the SELECT clause, the relative record number of the record to be written is placed in that data item.

Adding Records Randomly or Dynamically

When you write records to an indexed data set and ACCESS IS RANDOM or ACCESS IS DYNAMIC, the records can be written in any order.

Replacing Records in a File

To replace records in a file, use REWRITE on a file that you have opened for I-O. If you try to use REWRITE on a file that is not opened I-O, the record is not rewritten and the status key is set to 49. Check the file status key after each REWRITE statement.

- For sequential files, the length of the record you rewrite must be the same as the length of the original record.
- For indexed files, you can change the length of the record you rewrite.
- For variable-length relative files, you can change the length of the record you rewrite.

To replace records randomly or dynamically, the record to be rewritten need not be read by the COBOL program. Instead, to position the record you want to update:

- For indexed files, move the record key to the RECORD KEY data item, and then issue the REWRITE.
- For relative files, move the relative record number to the RELATIVE KEY data item, and then issue the REWRITE.

Deleting Records from a File

Open the file I-O and use the DELETE statement to remove an existing record from an indexed or relative file. You cannot use DELETE on a sequential file or a line sequential file.

When ACCESS IS SEQUENTIAL, the record to be deleted must first be read by the COBOL program. The DELETE then removes the record that was just read. If the DELETE is not preceded by a successful READ, the deletion is not done and the status key value is set to 92.

When ACCESS IS RANDOM or ACCESS IS DYNAMIC, the record to be deleted need not be read by the COBOL program. To delete a record, the key of the record to be deleted is moved to the RECORD KEY data item and the DELETE is issued. Check the file status key after each DELETE statement.

File Sharing and Record Locking (OS/2 Only)

Sometimes data needs to be accessible by many users of your COBOL program at the same time. If a file is to be accessed from multiple processes, you need to lock the file and/or individual records of the file.

The LOCK MODE clause is used to specify the locking technique used for a file. It is an optional clause of the file control entry. When LOCK MODE is specified, the file that is opened using the file connector[4] can be shared when it is opened. When LOCK MODE is omitted, opening the file causes it to become exclusive, unless the file is opened for input. The LOCK MODE IS AUTOMATIC clause is **only** supported by the VSAM file system running on OS/2. In all other cases, the LOCK MODE IS AUTOMATIC clause is ignored.

For OS/2 VSAM files, record locking is not supported for files that reside on an OS/2 LAN server. Files residing on OS/2 LAN servers are opened shared read or exclusive write.

When the LOCK MODE IS AUTOMATIC clause is specified for a file opened for I-O, a record lock is acquired when the READ statement is processed. It is released when a subsequent I/O statement for the file connector is processed.

Figure 31 shows the effect of using the LOCK MODE clause with other COBOL statements.

Figure 31 (Page 1 of 2). Using LOCK MODE with Other COBOL Statements

COBOL Statement	**Effect of LOCK MODE Clause**
OPEN	The file that is opened can be shared; that is, the file may be opened using more than one file connector. If the OPEN statement fails due to locking constraints, the file status value is set to 98 (file locked).
WRITE	1. If two or more file connectors for a sequential file add records by sharing the file after opening it in extended mode, the records are in unspecified order. 2. If two or more file connectors for a relative file add records by sharing the file after opening it in extended mode, the relative key values returned are ascending by not necessarily consecutive. 3. If two or more file connectors for a indexed file add records by sharing the file after opening it in extended mode, the order of alternate keys allowing for duplicates is unspecified. 4. A successive WRITE statement releases an existing record lock.
START	The START statement neither acquires nor detects a record lock. However, a successful START statement releases an existing record lock.

[4] You can think of a file connector as the connection of the COBOL declared file to the associated physical file.

Figure 31 (Page 2 of 2). Using LOCK MODE with Other COBOL Statements

COBOL Statement	Effect of LOCK MODE Clause
READ	1. For files opened for INPUT, READ statements will not acquire a record lock. 2. The READ statement will only be successful if no other file connector holds a lock on the record being accessed. If the record is locked, the statement is unsuccessful and the file status value is set to 99 (record locked). The file position indicator setting is unaffected for a sequential READ and unspecified for a random READ. 3. When no next record exists when the READ statement is processed, the AT END condition is returned regardless of whether the file is shared; for example, if the file is opened in extend mode by another file connector. 4. If the file is opened for I-O, each record is locked as it is read and released by the next I-O statement accessing the file connector.
REWRITE	A successful REWRITE statement releases an existing record lock. REWRITE is not successful when another file connector holds a lock on the record to be deleted.
DELETE	A successful DELETE statement releases an existing record lock. DELETE is not successful when another file connector holds a lock on the record to be deleted.
CLOSE	A successful CLOSE statement releases any record locks or file locks.
Restriction for SORT/MERGE: The LOCK MODE IS AUTOMATIC clause must not be specified if the file is specified in a USING or GIVING phrase of a SORT or MERGE statement.	

File Sorting and Merging

Arranging records in a particular sequence, a common requirement in data processing, can be done using sort or merge operations:

Sort operation Accepts input that is not in sequence and produces output in a requested sequence.

Merge operation Compares records from two or more sequenced files and combines them in order.

COBOL has special language features that assist in sort and merge operations. For information on the COBOL sort and merge language, see *IBM COBOL Language Reference*.

Basics of Sorting and Merging

To sort or merge files, do the following:

Figure 32. Preparing to Sort or Merge Files

<table>
<tr><th>Action</th><th>Code</th></tr>
<tr><td>Describe sort files and merge files.</td><td>SELECT statements in the FILE-CONTROL SECTION of the ENVIRONMENT DIVISION, and SD entries in the FILE SECTION of the DATA DIVISION.

SELECT statements and SD entries are always needed for sort files and merge files, even if you are only sorting or merging data items from Working-Storage.

The files described in an SD entry is the working file used for a sort or merge operation. You cannot perform any input/output operations on this file.

Every SD entry must contain a record description. For example:
<pre>
SD SORT-WORK-1
    RECORD CONTAINS 100 CHARACTERS.
01 SORT-WORK-1-AREA.
    05 SORT-KEY-1   PIC X(10).
    05 SORT-KEY-2   PIC X(10).
    05 FILLER   PIC X(80).
</pre></td></tr>
<tr><td>Describe the input and output files, if any, for sorting or merging.</td><td>SELECT statements in the FILE-CONTROL SECTION of the ENVIRONMENT DIVISION, and FD entries in the FILE SECTION of the DATA DIVISION.</td></tr>
<tr><td>Request the sort or merge operation.</td><td>SORT or MERGE statements in the PROCEDURE DIVISION.

The SORT or MERGE statement specifies the key fields in the record upon which the sort or merge is to be sequenced. You can code a key or keys as ascending or descending, or when you code more than one key, as a mixture of the two.

You can mix SORT and MERGE statements in the same program. A COBOL program can contain any number of sort or merge operations.

In your COBOL program, you can perform the same sort or merge multiple times, or perform multiple sorts or merges. However, one operation must be completed before another can begin.

For more information,see “The SORT Statement” on page 108 and “The MERGE Statement” on page 112.</td></tr>
</table>

Figure 33 on page 108 is an example of the ENVIRONMENT DIVISION and DATA DIVISION entries needed to describe sort files and an input file.

```
  ID Division.
    Program-ID. SmplSort.
  Environment Division.
    Input-Output Section.
    File-Control.
*
* Assign Name For A Sort File Is
* Treated As Documentation.
*
      Select Sort-Work-1 Assign To SortFile.
      Select Sort-Work-2 Assign To SortFile.
      Select Input-File  Assign To InFile.

  Data Division.
    File Section.
    SD  Sort-Work-1
        Record Contains 100 Characters.
    01  Sort-Work-1-Area.
        05  Sort-Key-1                Pic  X(10).
        05  Sort-Key-2                Pic  X(10).
        05  Filler                    Pic  X(80).

    SD  Sort-Work-2
        Record Contains 30 Characters.
    01  Sort-Work-2-Area.
        05  Sort-Key                  Pic  X(5).
        05  Filler                    Pic  X(25).

    FD  Input-File
    01  Input-Record                  Pic X(100).
     .
     .
     .
    Working-Storage Section.
    01  EOS-Sw                        Pic  X.
    01  Filler.
        05  Table-Entry Occurs 100 Times
              Indexed By X1           Pic X(30).
     .
     .
     .
```

Figure 33. ENVIRONMENT DIVISION and DATA DIVISION Entries for a Sort Program

The SORT Statement

You can use the SORT statement to do the following:

- Use input or output procedures to add, delete, change, edit, or otherwise change records.
  - To request that input procedures be performed on the sort records before they are sorted, use SORT . . . INPUT PROCEDURE.

    See "Coding the Input Procedure" on page 110 for more information on input procedures.

  - To request that output procedures be performed on the sort records after they are sorted, use SORT . . . OUTPUT PROCEDURE.

    See “Coding the Output Procedure” on page 111 for more information on output procedures.

- Sort data items (including tables) in Working-Storage.
- Read records directly into a new file without any preliminary processing (SORT . . . USING).
- Transfer sorted records from the sort program directly to another file without any further processing (SORT . . . GIVING).

SORT Program Organization

A COBOL program containing a sort operation can be organized so that one or more input files are read and operated on by an input procedure. In the input procedure, a RELEASE statement (analogous to the WRITE statement) releases a record to be sorted. If you do not want to change or process the records in the files before the sorting operation begins, the SORT statement USING option releases the records in the files unchanged to the new file.

The sort operation then arranges the entire set of records in the sequence specified by the key(s).

After the sort, sorted records can be made available one at a time through a RETURN statement to an output procedure. If you want to put the sorted records in files without changing or processing these records, the SORT statement GIVING option names the output files and writes the sorted records to the output files.

Setting the Sort Criteria

To set sort criteria:

1. In the record description of the file to be sorted, define the key or keys on which it will be sorted.

   The key used in the SORT statement cannot be variably located. (For more information on variably located data items, see Appendix D, “Complex OCCURS DEPENDING ON” on page 553.)

2. In the SORT statement, code the key on which the file will be sorted.

   To sort on more than one key, list the keys in descending order of importance.

In the example below, `SORT-GRID-LOCATION` and `SORT-SHIFT` are defined in the DATA DIVISION before they are used in the SORT statement.

The example also shows the use of an input and an output procedure. Use an input procedure if you want to process the records before you sort them, and use an output procedure if you want to further process the records after you sort them.

Coding the Input Procedure

```
DATA DIVISION.
     .
     .
     .
SD  SORT-FILE
    RECORD CONTAINS 115 CHARACTERS
    DATA RECORD SORT-RECORD.

01  SORT-RECORD.
    05  SORT-KEY.
        10  SORT-SHIFT                PIC X(1).
        10  SORT-GRID-LOCATION        PIC X(2).
        10  SORT-REPORT               PIC X(3).
    05  SORT-EXT-RECORD.
        10  SORT-EXT-EMPLOYEE-NUM     PIC X(6).
        10  SORT-EXT-NAME             PIC X(30).
        10  FILLER                    PIC X(73).

PROCEDURE DIVISION.
     .
     .
     .
    SORT SORT-FILE
        ON ASCENDING KEY SORT-GRID-LOCATION SORT-SHIFT
        INPUT PROCEDURE 600-SORT3-INPUT
        OUTPUT PROCEDURE 700-SORT3-OUTPUT.
     .
     .
     .
```

Alternate Collating Sequences

You can sort records on a user specified collating sequence for single byte character keys. The default collating sequence is the collating sequence specified by the locale setting in effect at compile time. To override the PROGRAM COLLATING SEQUENCE specified either explicitly or by the default, use the COLLATING SEQUENCE option of the SORT statement. You can use different collating sequences for multiple sorts in your program.

For DBCS keys, the collating sequence is that specified by the locale setting in effect at execution time.

Coding the Input Procedure

If you want to process the records in an input file before they are released to the sort program, use the INPUT PROCEDURE option of the SORT statement. You might use an input procedure to:

- Release data items to the new file from Working-Storage.
- Release records that have already been read in elsewhere in the program.
- Read records from an input file, select or process them, and release them to the new file.

Each input procedure must be contained in either paragraphs or sections. For example, to release records from Working-Storage (a table) to the new file:

```
    SORT SORT-WORK-2
      ON ASCENDING KEY SORT-KEY
      INPUT PROCEDURE 600-SORT3-INPUT-PROC
      .
      .
      .

600-SORT3-INPUT-PROC SECTION.
    PERFORM WITH TEST AFTER
       VARYING X1 FROM 1 BY 1 UNTIL X1 = 100
      RELEASE SORT-WORK-2-AREA  FROM  TABLE-ENTRY (X1)
       END-PERFORM.
```

Transferring Records to the Sort Program

To transfer records to the new file, all input procedures must contain at least one RELEASE or RELEASE FROM statement. To release A from X, for example, you can enter:

```
MOVE X TO A.
RELEASE A.
```

Figure 34 compares the RELEASE and RELEASE FROM statements.

Figure 34. Comparison of RELEASE and RELEASE FROM

RELEASE	RELEASE FROM
MOVE EXT-RECORD	
TO SORT-EXT-RECORD	
PERFORM RELEASE-SORT-RECORD	PERFORM RELEASE-SORT-RECORD
.	.
.	.
.	.
RELEASE-SORT-RECORD.	RELEASE-SORT-RECORD.
RELEASE SORT-RECORD	RELEASE SORT-RECORD FROM SORT-EXT-RECORD

Coding the Output Procedure

If you want to select, edit, or otherwise change sorted records before writing them from the sort work file into another file, use the OUTPUT PROCEDURE option of the SORT statement.

Each output procedure must be contained in either a section or a paragraph and must include:

- At least one RETURN or RETURN INTO statement.

  The RETURN statement makes each sorted record available to your output procedure (the RETURN statement for a sort file is similar to a READ statement for an input file).

Success of Sort and Merge

You can use the AT END and END-RETURN phrases with the RETURN statement. The imperative statements on the AT END phrase will be performed after all the records have been returned from the sort file. The END-RETURN explicit scope terminator serves to delimit the scope of the RETURN statement.

If you use the RETURN INTO statement, instead of RETURN, your records will be returned to Working-Storage or to an output area.

- Any statements necessary to process the records that are made available, one at a time, by the RETURN statement.

Restrictions on Input and Output Procedures

The following restrictions apply to the procedural statements in input and output procedures:

- The input/output procedure must not contain any SORT or MERGE statements.
- The input/output procedure must not contain any STOP RUN, EXIT PROGRAM, or GOBACK statements.
- You can use ALTER, GO TO, and PERFORM statements in the input/output procedure to refer to procedure-names outside the input/output procedure. However, you must return to the input/output procedure after a GO TO or PERFORM statement.
- The remainder of the PROCEDURE DIVISION must not contain any transfers of control to points inside the input/output procedure (with the exception of the return of control from a Declarative Section).
- In a SORT or MERGE input or output procedure, calling a program is permitted, but the called program cannot issue a SORT or MERGE statement and the called program must return to the caller.

The MERGE Statement

You cannot specify an input procedure in the MERGE statement; use MERGE . . . USING.

The MERGE statement combines the files you name into one sequenced file. The files to be merged must be already be in the same sorted sequence.

The merge operation compares keys in the records of the input files, and passes the sequenced records one-by-one to the RETURN statement of an output procedure or to the file named in the GIVING phrase.

Determining Whether the Sort or Merge Was Successful

The SORT or MERGE statement returns one of the following completion codes after a sort is finished:

0 Successful completion of sort/merge

16 Unsuccessful completion of sort/merge

The return code or completion code is stored in a SORT-RETURN special register. The contents of SORT-RETURN change after each SORT or MERGE statement is performed.

You should test for successful completion after each SORT or MERGE statement:

```
    SORT SORT-WORK-2
        ON ASCENDING KEY SORT-KEY
        INPUT PROCEDURE IS 600-SORT3-INPUT-PROC
        OUTPUT PROCEDURE IS 700-SORT3-OUTPUT-PROC.
    IF SORT-RETURN NOT=0
      DISPLAY "SORT ENDED ABNORMALLY. SORT-RETURN = "
      SORT-RETURN.
    .
    .
    .
600-SORT3-INPUT-PROC SECTION.
    .
    .
    .
700-SORT3-OUTPUT-PROC SECTION.
    .
    .
    .
```

Prematurely Stopping a Sort or Merge Operation

To stop a sort or merge operation, use the SORT-RETURN special register. Move the integer 16 into the register in:

- An input or output procedure.

  Sort or merge processing will be stopped immediately after the next RELEASE or RETURN statement is performed.

- A Declarative Section entered during processing of a USING or GIVING file.

  Sort or merge processing will be stopped on exit from the declarative section.

Control then returns to the statement following the SORT or MERGE statement.

If you do not reference SORT-RETURN anywhere in your program, COBOL will test the return code. If the code is 16, COBOL issues a run-time diagnostic message and terminates the run unit (or the thread, in a multithread environment). If you test SORT-RETURN for one or more (but not necessarily all) SORT or MERGE statements, COBOL will not check the return code.

SORT Special Registers

You can use the SORT-RETURN and SORT-CONTROL special registers to get or test values related to sort behavior.

The SORT-CONTROL special register is implicitly defined as

```
01 SORT-CONTROL GLOBAL PICTURE X(160) VALUE='file name'.
```

where `file name` is used as the system file identifier for the options file for the sort product. You can assign to SORT-CONTROL the *file name* of a file that contains your sort control statements. See *SMARTsort for OS/2 and AIX* for information about the SMARTsort options file.

The STL File System

The STL file system supports sequential, indexed, and relative files on the local system. It provides the basic file facilities that you need for accessing local files. It gives conformance to ANSI standards, good performance, and the ability to port easily between AIX, OS/2, and Windows systems.

Line sequential files are the only files not supported.

The file system is safe for use with threads; it is your responsibility to ensure that multiple threads do not access COBOL buffers at the same time. Multiple threads can perform operations on the same STL file, but you must use an operating system call (for example, `DosRequestMuteSem` on OS/2 or `WaitForSingleObject` on Windows) to force all but one of them to wait for the file access to complete on the active thread.

With the STL file system you can easily read and write files to be shared with PL/I programs.

File Status and the STL File System

In the FILE STATUS clause of the FILE-CONTROL paragraph, you can specify one or two names:

```
FILE STATUS data-name-1
```

or

```
FILE STATUS data-name-1 data-name-2
```

After an input/output operation, *data-name-1* will contain a status code which is independent of the file system used. If you specify *data-name-2*, it will contain a status code that is file-system specific. In the case of the STL file system, *data-name-2* will contain one of the STL file system return codes shown in Figure 35.

See *IBM COBOL Language Reference* for additional information on the FILE STATUS clause.

Figure 35 (Page 1 of 3). The STL file system Return Codes

Code	Meaning	Notes
0	Successful completion	The input/output operation completed successfully.
1	Invalid operation	This return code should not occur; it indicates an error in the file system.
2	I/O error	A call to an operating system I/O routine returned an error code.
3	File not open	Attempt to do an operation (other than OPEN) on a file that is not open.

Figure 35 (Page 2 of 3). The STL file system Return Codes

Code	Meaning	Notes
4	Key value not found	Attempt to read a record using key which is not in the file.
5	Duplicate key value	Attempt to use a key a second time for a key which does not allow duplicates.
6	Invalid key number	This return code should not occur; it indicates an error in the file system.
7	Different key number	This return code should not occur; it indicates an error in the file system.
8	Invalid flag for the operation	This return code should not occur; it indicates an error in the file system.
9	End of file	An end of file was detected. This is not an error.
10	I/O operation must be preceed by I/O GET op	The operation is looking for the current record and the current record has not been defined.
11	Error return from get space routine	The operating system indicates that not enough memory is available.
12	Duplicate key accepted	The operation specified a key and the key is a duplicate. See the description of File Status 2 in *IBM COBOL Language Reference*.
13	Sequential access and key sequence bad	Sequential access was specified but the records are not in sequential order.
14	Record length < max key	The record length does not allow enough space for the all of the keys.
15	Access to file denied	The operation system reported that it cannot access the file. Either the file does not exist or the user does not have the proper permission of the operating system to access the file.
16	File Already exists	You appempted to open a new file, but the operating system reports that the file already exists.
17	(Reserved)	
18	File locked	Attempt to open a file which is already open in exclusive mode.
19	File table full	The operating system reports that its file table is full.
20	Handle table full	The operating system reports that it cannot allocate any more file handles.
21	Title does not say STL.	Files opened for reading by the STL file system must contain a header record that contains “STL” at a certain offset in the file.
22	Bad indexcount arg for create	This return code should not occur; it indicates an error in the file system.

Figure 35 (Page 3 of 3). The STL file system Return Codes

Code	Meaning	Notes
23	Index or rel record > 64K	Index and relative records are limted to a length of 64K.
24	Error found in file header or data in open of existing file	STL files begin with a header. The header or its associated data has inconsistent values.
25	Indexed open on seq file	Attempt to open a sequential file as an indexed or relative file.
Note: The following are errors detected in the adapter open routines.		
1000	Sequential open on indexed/rel file	Attempt to open an indexed or relative file as a sequential file.
1001	Relative open of indexed file	Attempt to open a relative file as an indexed file.
1002	Index open of rel file	Attempt to open an indexed file as a sequential file.
1003	File does not exist	The operating system reports that the file does not exist.
1004	Number of keys differ	Attempt to open a file with a different number of keys.
1005	Record lengths differ	Attempt to open a file with a different record length.
1006	Record types differ	Attempt to open a file with a different record type.
1007	Key position or length differ	Attempt to open a file with a different key position or length.

SMARTdata Utilities for OS/2

This section gives you tips and hints for using the SMARTdata Utilities on OS/2. For equivalent information under Windows, see *SMARTdata Utilities for Windows User's Guide*.

Quick Start for Remote File Access

This section gives you the steps to get started quickly with accessing remote files.

1. Install and configure Communications Manager/2 (CM/2) on your OS/2 for APPC. Define an LU alias for the remote host. You will need a user ID and password on that host.
2. Copy the file CONFIG.DFM from the samples subdirectory to the directory defined by the EHNDIR environmental variable in your CONFIG.SYS file.
3. Edit CONFIG.DFM as follows:
   - Activate one of the target definitions by removing the appropriate semicolons.

- Edit the remote_lu line and replace the sample value with the LU alias defined in CM/2 for the target you wish to access.
- Edit the line containing local_lu and replace the existing value with the LU alias of your local machine as defined in CM/2.

4. From an OS/2 command line, issue the command STARTDFM.
5. Enter your user ID and password on the remote system when prompted.
6. Assign a drive to that host with the following command:

   ```
   DFMDRIVE ASSIGN x: //lualias
   ```

   Where x is a drive not currently accessed and lualias is the LU alias of the host system.
7. You can now access files on the remote host both through OS/2 commands and through VSAM applications through the drive letter assigned in the previous step.

Problems with Remote Files Access

The majority of problems that people encounter when installing Distributed File Manager (DFM) are related to configuring communications between the workstation and the host. Some of the common problems encountered when setting up DFM, and their solutions are:

STARTDRM cannot find CONFIG.DFM. STARTDRM looks for the configuration file in the following order:

1. Full path name, if provided on the command line
2. CONFIG.DFM in the current directory
3. CONFIG.DFM in the directory defined by the EHNDIR environmental variable

Problem in the configuration file.

- Ensure all semicolons are removed from the target definition.
- Ensure the remote_lu value for the target definition matches exactly the LU alias assigned to the target system in CM/2.
- Ensure your local_lu value matches the local LU alias in CM/2.
- Check for case mismatches between configuration file values and CM/2 values.

ID and password. Ensure you entered the correct user ID and password at the prompt.

Communication Manager setup problems. The most common problems are errors in the CM/2 setup that prevent DFM/2 from initializing an APPC conversation with the host. The APING utility available with the APPC Productivity Suite allows you to verify that you are able to make an LU 6.2 connection to the target system.

The DFM target is not available on the host system. For OS/390, the target DFM is part of the DFSMS/MVS Version 1.3 product. It is a startable procedure. Ask your

OS/390 administrator whether DFM/MVS is started. Also, have your administrator ensure that the SNA mode of QPCSUPP is defined on the OS/390 system.

Platform-Specific Behavior

It is important to remember that when accessing files on a remote system, you are constrained by the limitations of the remote file access method.

To better understand the limitations of the remote access methods, refer to the following documents:

Distributed FileManager/MVS Guide and Reference, SC26-4915

AS/400 Distributed Data Management Guide, SC41-9600

A list of known restrictions:

OS/390 Target

The OS/390 file access method associated with the remote file will limit the type of access you may use. For instance, if the remote file is a PDSE member, you can only access it sequentially. The same holds for SAM and VSAM ESDS files. You can only use keyed access if the remote file is a VSAM KSDS.

AS/400 Target

Specifying a SECURITY value other than the default PROGRAM will cause file space allocation problems.

File names appear to be limited to 10 characters including periods.

File names must be specified in upper case.

AS/400 allocates direct files with three extents with 1000 records per extent. All records are defined with unused records designated as inactive. This might cause problems if you access the file the return inactive flag (DDM_RTNINA).

AS/400 requires upper case for user ID and password.

Data Conversion

When accessing remote files, it is possible to have DFM convert the records between their native format on the target system and a local format used on the workstation. For instance, DFM can convert EBCDIC into ASCII or binary encoded decimal into byte-reversed binary. Refer to the SMARTdata Utilities documentation for the details of how to take advantage of this function. The following are some common problems that you might encounter:

Data conversion does not occur.

Note that the data conversion function applies only when files are accessed through the VSAM interface. This means when files are viewed or edited from the command line, they will not be converted; however, if they are opened and accessed by an application that invokes the VSAM APIs either directly or through COBOL, the conversion should occur.

A case mismatch between the file name specified in the FILE_DESCRIPTOR_MAP statement of the DFM configuration file and the file name specified when accessing the file will also prevent conversion. While specifying the file name `user1.ddm.file1` might open the correct file on OS/390, it will not match against USER1.DDM.FILE1 in the FILE_DESCRIPTOR_MAP.

Data conversion does not work when the DFMDRIVE GUI is used to assign a drive.

When using the directory mapping function of the DFMDRIVE GUI, the file name is appended to the value entered for the directory with a slash separating the directory name from the file name. The OS/390 target then removes the slash and replaces it with a dot before accessing the file. If the TARGET_FILENAME in the FILE_DESCRIPTOR_MAP definition contains a dot instead of a slash, the comparison will fail.

Note: Data conversion is not necessary if the program is expecting System/390 host data types and the appropriate compiler option, that is, BINARY(S390), CHAR(EBCDIC), or FLOAT(HEX) is in effect.

For additional information about remote file access and System/390 host data types, see Appendix B, “System/390 Host Data Type Considerations” on page 543.

File Conversion

When accessing a remote file, SdU converts the file name from the local character set (as defined by the CODEPAGE environmental variable in `CONFIG.SYS`) into codepage 0500 used by DFM target systems. The current release of SMARTdata Utilities is capable of converting between single-byte character sets only. If you are running the Kanji or Simplified Chinese version of SMARTdata Utilities (Codepage 0932 and 1381 respectively), use only single-byte characters when specifying remote file names.

LAN-Installed SMARTdata Utilities

If you are using a LAN-installed SMARTdata Utilities and wish to use the remote file access portion of SMARTdata Utilities when SMARTdata Utilities is installed on a LAN disk then the following steps must be taken:

- Change directory to the subdirectory of the LAN disk where SMARTdata Utilities is installed.
- Change directory into the DLL subdirectory.
- Copy the file DFMSFL0.DLL to a local disk.
- Locate the line of your CONFIG.SYS file that references the file DFMSFL0. Change the full path name to point to its location on your local disk.

Translation Tables

The environment variable FMTCDRA should be set to the name of the directory that contains the CDRA translation tables. To set this variable, issue the command: `FMTCDRA=[Root_directory]\BIN\CONVTABL`.

For example,

SMARTdata Utilities for OS/2

```
FMTCDRA=K:\IBMDDM\BIN\CONVTABL
```

Client Enhancement for Stream Data Conversion

The DFMDRIVE end user interface is modified to allow the specification of a parameter list which will be attached to filenames passed to the DFM target server.

One specific use for passing a parameter list to the Distributed FileManager/MVS target server is to trigger Stream Data Conversion, that is, to access System/390 EBCDIC data as ASCII data from the work station.

If the target server does not support Stream Data Conversion, the following message appears when using the DFMDRIVE ASSIGN or DFMDRIVE SETPARM line command:

```
EHN0252E: Drive %1 target system does not support parameters
```

Similarly, if the Graphical User Interface is used to send a parameter list to the target server, and it does not support a parameter list, a popup appears with the message:

```
Target system does not support parameters.
```

For details of how to pass a parameter list to the target system, see *VSAM in a Distributed Environment*.

Chapter 8. Error Handling

Anticipate the possibility of coding or system problems by putting code into your program to handle them. Error handling code can be thought of as built-in distress flares or lifeboats. If such code is not in your program, not only could output data and files be corrupted, but you might not even be aware of the problem.

The action taken by your error-handling code can vary from trying to handle the situation and continue, to issuing a message, to halting the running of the program. In any event, coding a warning message is a good idea.

You might be able to create your own error-detection routines for data-entry errors or for errors as your installation defines them.

COBOL contains special elements to help you anticipate and correct error conditions. These fall into the following main areas:

- "STRING and UNSTRING Operations."
- "Arithmetic Operations" on page 122.
- "Input/Output Error Handling Techniques" on page 123.
- "CALL Statements" on page 131.

STRING and UNSTRING Operations

When stringing or unstringing data, the pointer might fall out of the range of the receiving field. Here a potential overflow condition exists, but COBOL does not allow the overflow to happen; the STRING/UNSTRING operation will not be completed and the receiving field remains unchanged.

If you do not have an ON OVERFLOW clause on the STRING or UNSTRING statement, control passes to the next sequential statement, and you are not notified of the incomplete operation.

Consider the following statement:

```
String Item-1 space Item-2 delimited by Item-3
    into Item-4
    with pointer String-ptr
    on overflow
        Display "A string overflow occurred"
End-String
```

Figure 36. Data Values before and after the Statement is Performed

Data Item	PICTURE	Value Before	Value After
Item-1	X(5)	`AAAAA`	`AAAAA`
Item-2	X(5)	`EEEAA`	`EEEAA`
Item-3	X(2)	`EA`	`EA`
Item-4	X(8)	`ƀƀƀƀƀƀƀƀ`	`ƀƀƀƀƀƀƀƀ`
String-ptr	9(2)	`0`	`0`

Note: The symbol ƀ represents a blank space.

Because `String-ptr` has a value of zero that falls short of the receiving field, an overflow condition occurs and the STRING operation is not completed (a `String-ptr` greater than nine would cause the same result). If ON OVERFLOW had not been specified, you would not be notified that the contents of Item-4 remain unchanged.

Arithmetic Operations

When your program performs arithmetic operations, the results might be larger than the fixed-point field that is to hold them, or you might have tried a division by 0. In either case, the ON SIZE ERROR clause after the ADD, SUBTRACT, MULTIPLY, DIVIDE, or COMPUTE statement can handle the situation.

For ON SIZE ERROR to work correctly for fixed-point overflow and decimal overflow, you must specify the TRAP(ON) run-time option.

If you code the ON SIZE ERROR clause, the imperative statement of your clause will be performed and your result field will not be changed in the following five cases:

- Fixed-point overflow.
- Division by 0.
- Zero raised to the zero power.
- Zero raised to a negative number.
- A negative number raised to a fractional power.

Example of Checking for Division by Zero

Code your ON SIZE ERROR imperative statement so that it issues an informative message. For example:

```
DIVIDE-TOTAL-COST.
    DIVIDE TOTAL-COST BY NUMBER-PURCHASED
        GIVING ANSWER
        ON SIZE ERROR
          DISPLAY "ERROR IN DIVIDE-TOTAL-COST PARAGRAPH"
          DISPLAY "SPENT " TOTAL-COST, " FOR " NUMBER-PURCHASED
          PERFORM FINISH
    END-DIVIDE
   .
   .
   .
FINISH.
    STOP RUN.
```

In this example, if division by 0 occurs, the program will write out a message identifying the trouble and halt program execution.

Input/Output Error Handling Techniques

When a program encounters an error in processing a file, whether logical errors in the program or input/output errors on the disk, control returns to your COBOL program, except in the following cases:

- There is no file status specified for the file
- There is no applicable EXCEPTION/ERROR declarative
- There is no INVALID KEY/AT END phrase specified for the error condition

In these cases, a COBOL run-time message is written and the run unit ends.

When an input/output statement operation fails, COBOL will not perform corrective action for you. You choose whether your program will continue running after a less-than-severe input/output error occurs.

COBOL offers five techniques for intercepting and handling certain input/output errors.

End-of-file phrase (AT END)
EXCEPTION/ERROR declarative
FILE STATUS key
File System Return Code
INVALID KEY phrase

The most important thing to remember about input/output errors is that you choose whether your program will continue running after a less-than-severe input/output error occurs. COBOL does not perform corrective action. If you choose to have your program continue (by incorporating error-handling code into your design), you must also code the appropriate error-recovery procedure; for example, a procedure to check the file status key value.

Figure 37 on page 124 shows the flow of logic after a file system input/output error occurs:

I/O Error Handling

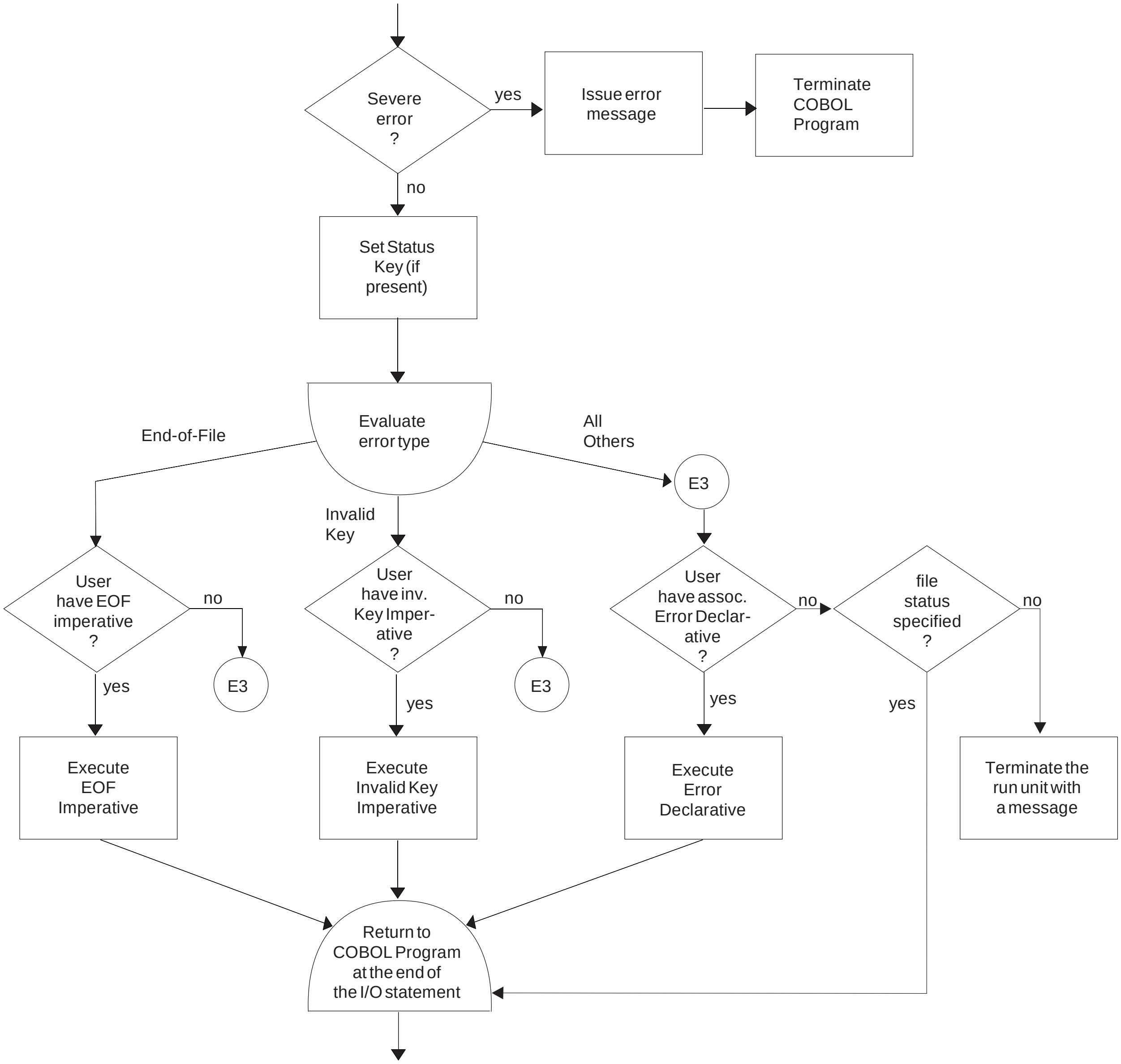

Severe error ?
yes
Issue error message
Terminate COBOL Program
no
Set Status Key (if present)
Evaluate error type
End-of-File
All Others
E3
Invalid Key
User have EOF imperative ?
no
E3
yes
User have inv. Key Imper-ative ?
no
E3
yes
User have assoc. Error Declar-ative ?
no
file status specified ?
no
yes
yes
Execute EOF Imperative
Execute Invalid Key Imperative
Execute Error Declarative
Terminate the run unit with a message
Return to COBOL Program at the end of the I/O statement

Figure 37. Flow of Logic after a File System I/O Error

End-of-File Phrase (AT END)

An end-of-file condition might or might not represent an error. In many designs, reading sequentially to the end of a file is done intentionally, and the AT END condition is expected.

For example, suppose you are processing a file containing transactions in order to update a master file:

```
PERFORM UNTIL TRANSACTION-EOF = "TRUE"
  READ UPDATE-TRANSACTION-FILE INTO WS-TRANSACTION-RECORD
    AT END
      DISPLAY "END OF TRANSACTION UPDATE FILE REACHED"
      MOVE "TRUE" TO TRANSACTION-EOF
  END READ
  .
  .
END-PERFORM
```

Sometimes, however, the condition will reflect an error. You code the AT END phrase of the READ statement to handle either case, according to your program design.

If you code an AT END phrase, on end-of-file the phrase is performed. If you do not code an AT END phrase, the associated ERROR declarative is performed.

Any NOT AT END phrase you code is performed only if the READ statement completes successfully. If the READ operation fails because of any condition other than end-of-file, neither the AT END nor the NOT AT END phrase is performed. Instead, control passes to the end of the READ statement after performing any associated declarative procedure.

If you have coded neither an AT END phrase nor an EXCEPTION declarative procedure, but have coded a status key clause for the file, control passes to the next sequential instruction after the input/output statement that detected the end-of-file (where presumably you have some coding to take appropriate action).

EXCEPTION/ERROR Declarative

You can code one or more ERROR declarative procedures in your COBOL program that will be given control if an input/output error occurs. You can have:

- A single, common procedure for the entire program.
- Group procedures for each file open mode (whether INPUT, OUTPUT, I-O, or EXTEND).
- Individual procedures for each particular file.

Place each such procedure in the declaratives section of your PROCEDURE DIVISION. (For the syntax detail, see *IBM COBOL Language Reference*.

I/O Error Handling

In your procedure, you can choose to try corrective action, retry the operation, continue, or end execution. You can use the ERROR declaratives procedure in combination with the file status key if you want a further analysis of the error.

If you continue processing a blocked file, you might lose the remaining records in a block after the record that caused the error.

Write an ERROR declarative procedure if you want the system to return control to your program after an error occurs. If you do not write an ERROR declarative procedure, your job could be canceled or abnormally terminated after an error occurs.

File Status Key

The system updates the FILE STATUS key after each input/output statement is performed on a file, placing values in the two digits of the file status key. In general, a zero in the first digit indicates a successful operation, and a zero in both digits means there is nothing abnormal to report. Possible file status codes are listed in the *IBM COBOL Language Reference*. Establish a FILE STATUS key using the FILE STATUS clause in the FILE-CONTROL paragraph and data definitions in the DATA DIVISION.

```
FILE STATUS IS data-name-1
```

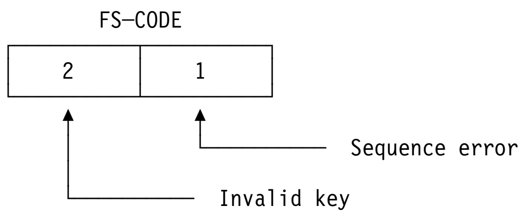
data-name-1
Specifies the 2-character COBOL FILE STATUS key that should be defined in the WORKING-STORAGE SECTION.

Restriction: The *data-name* in the FILE STATUS clause cannot be variably located. (For more information on variably located data items, see Appendix D, “Complex OCCURS DEPENDING ON” on page 553.)

Your program can check the COBOL FILE STATUS key to discover whether an error has been made and, if so, what general type of error it is. For example, if a FILE STATUS clause is coded like this:

```
FILE STATUS IS FS-CODE
```

FS-CODE is used by COBOL to hold status information like this:



Follow these rules for each file:

- Define a different FILE STATUS key for each file.

This is especially important since it allows you to determine the cause of a file input/output exception which might have occurred as a result of, for example, an application logic error or a disk error.

- Check the FILE STATUS key after every input/output request.

  After an input or output statement is performed, check the contents of the status key; if it contains a value other than 0, your program can issue an error message, or can act based on the value of the code placed in the status key.

  You do not have to reset the status key code, because it is set following each input/output attempt.

  For VSAM, STL, and Btrieve files, in addition to the file status key, you can code a second identifier in the FILE STATUS clause to get more detailed information on file system input/output requests. For further details, see “File System Return Code” on page 128.

  You can use the status key alone, or in conjunction with the INVALID KEY option, or to supplement the EXCEPTION/ERROR declarative. Using the status key in this way gives you precise information about the results of each input/output operation.

File Status Key Example

Figure 38 shows an example of the COBOL coding that performs a simple check on the status key after opening a file.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SIMCHK.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MASTERFILE ASSIGN TO AS-MASTERA
    FILE STATUS IS MASTER-CHECK-KEY
    .
    .
DATA DIVISION.
    .
    .
WORKING-STORAGE SECTION.
01  MASTER-CHECK-KEY       PIC X(2).
    .
    .
PROCEDURE DIVISION.
    .
    .
   OPEN INPUT MASTERFILE
   IF MASTER-CHECK-KEY NOT = "00"
       DISPLAY "Non-zero file status returned from OPEN " MASTER-CHECK-KEY
    .
    .
```

Figure 38. Using the Status Key to Check an OPEN Statement

I/O Error Handling

File System Return Code

Often the 2-character FILE STATUS code is too general to pinpoint the disposition of a request. You can get more detailed information about file system input/output requests by coding a second status area:

```
FILE STATUS IS data-name-1 data-name-2
```

data-name-1
Specifies the 2-character COBOL FILE STATUS key.

data-name-2
Specifies a data item that contains the file system return code when the COBOL FILE STATUS key is not 0. *data-name-2* is at least 6 bytes long.

STL and Btrieve File Systems
If *data-name-2* is 6 bytes in length, it will contain the return code. If it is greater than 6 bytes in length, it will also contain a message with further information. For example, given the definition

```
01  my-file-status-2.
    02 exception-return-value PIC 9(6).
    02 additional-info PIC X(100).
```

and an attempt to open a file with a different defintion than the one with which it was created, return code 39 would be returned in `exception-return-value` and a message telling you what keys you need to perform the open would be returned in `additional-info`.

VSAM File System
data-name-2 must be defined as PICTURE X(*n*) and USAGE DISPLAY attributes, where *n* is 6 or greater. The PICTURE string value represents the first *n* bytes of the VSAM reply message structure. If the size of the reply message structure, *m*, is less than *n*, only the first *m* bytes contain useful information.

For full details on the VSAM reply message structure, and VSAM file handling in general, refer to the SMARTdata Utilities documentation for your operating environment as listed in “Bibliography” on page 658.

See the *IBM COBOL Language Reference* for the rules for coding *data-name-2*.

For information about possible return codes from the STL file system, see “File Status and the STL File System” on page 114.

For information on interpreting the codes for other file systems, refer to the relevant file system documentation listed in “Bibliography” on page 658.

Checking File System Status Codes Example

Figure 39 on page 129 shows an example of a COBOL program that reads an indexed file (starting on the fifth record), checks the file status key after each input/output request, and displays the VSAM codes when the file status key is not zero. Figure 39 on page 129 also illustrates what the output from this program might look like, assuming that the file being processed contains six records.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FILESYSFILE ASSIGN TO FILESYSFILE
    ORGANIZATION IS INDEXED
    ACCESS DYNAMIC
    RECORD KEY IS FILESYSFILE-KEY
    FILE STATUS IS FS-CODE, FILESYS-CODE.

DATA DIVISION.
FILE SECTION.
FD  FILESYSFILE
    RECORD  30.
01  FILESYSFILE-REC.
    10 FILESYSFILE-KEY             PIC X(6).
    10 FILLER                   PIC X(24).
WORKING-STORAGE SECTION.
01  RETURN-STATUS.
    05 FS-CODE                  PIC XX.
    05 FILESYS-CODE             PIC X(6).

PROCEDURE DIVISION.
    OPEN  INPUT FILESYSFILE.
    DISPLAY "OPEN INPUT FILESYSFILE FS-CODE: " FS-CODE.

    IF FS-CODE NOT = "00"
       PERFORM FILESYS-CODE-DISPLAY
       STOP RUN
    END-IF.

    MOVE "000005" TO FILESYSFILE-KEY.
    START FILESYSFILE KEY IS EQUAL TO FILESYSFILE-KEY.
    DISPLAY "START FILESYSFILE KEY="  FILESYSFILE-KEY
            " FS-CODE: "  FS-CODE.

    IF FS-CODE NOT = "00"
       PERFORM FILESYS-CODE-DISPLAY
    END-IF.
```

Figure 39 (Part 1 of 2). Getting File System Code Information on Input/Output Requests

```
          IF FS-CODE = "00"
             PERFORM READ-NEXT UNTIL FS-CODE NOT = "00"
          END-IF.

          CLOSE FILESYSFILE.
          STOP RUN.

      READ-NEXT.
          READ FILESYSFILE NEXT.
          DISPLAY "READ NEXT FILESYSFILE FS-CODE: " FS-CODE.
          IF FS-CODE NOT = "00"
             PERFORM FILESYS-CODE-DISPLAY
          END-IF.
          DISPLAY FILESYSFILE-REC.

      FILESYS-CODE-DISPLAY.
          DISPLAY "FILESYS-CODE ==>", FILESYS-CODE.
```

Figure 39 (Part 2 of 2). Getting File System Code Information on Input/Output Requests

INVALID KEY Phrase

The INVALID KEY phrase will be given control if an input/output error occurs because of a faulty index key. You can include INVALID KEY phrases on READ, START, WRITE, REWRITE, and DELETE requests for indexed and relative files.

INVALID KEY and ERROR Declaratives

INVALID KEY phrases differ from ERROR declaratives in these ways:

- INVALID KEY phrases operate for only limited types of errors, whereas the ERROR declarative encompasses all forms.
- INVALID KEY phrases are coded directly onto the input/output verb, whereas ERROR declaratives are coded separately.
- INVALID KEY phrases are specific for one single input/output operation, whereas ERROR declaratives are more general.

If you code INVALID KEY in a statement that causes an INVALID KEY condition, control is transferred to the INVALID KEY imperative statement. Here, any ERROR declaratives you have coded are not performed.

NOT INVALID KEY

Any NOT INVALID KEY phrase that you code is performed only if the statement completes successfully. If the operation fails because of any condition other than INVALID KEY, neither the INVALID KEY nor the NOT INVALID KEY phrase is performed. Instead control passes to the end of the statement after performing any associated ERROR declaratives.

Using FILE STATUS and INVALID KEY Example

Use the FILE STATUS clause with INVALID KEY to evaluate the status key and determine the specific INVALID KEY condition.

For example, assume you have a file containing master customer records and need to update some of these records with information in a transaction update file. You will read each transaction record, find the corresponding record in the master file, and make the necessary updates. The records in both files each contain a field for a customer number, and each record in the master file has a unique customer number.

The FILE-CONTROL entry for the master file of customer records includes statements defining indexed organization, random access, MASTER-CUSTOMER-NUMBER as the prime record key, and CUSTOMER-FILE-STATUS as the file status key. The following example shows how you can use FILE STATUS with the INVALID KEY to more specifically determine the cause of an I/O statement failure.

```
.
.  (read the update transaction record)
.
MOVE "TRUE" TO TRANSACTION-MATCH
MOVE UPDATE-CUSTOMER-NUMBER TO MASTER-CUSTOMER-NUMBER
READ MASTER-CUSTOMER-FILE INTO WS-CUSTOMER-RECORD
  INVALID KEY
    DISPLAY "MASTER CUSTOMER RECORD NOT FOUND"
    DISPLAY "FILE STATUS CODE IS: " CUSTOMER-FILE-STATUS
    MOVE "FALSE" TO TRANSACTION-MATCH
END-READ
```

CALL Statements

When dynamically calling a separately compiled program, the program that you call might be unavailable to the system. For example, the system could run out of storage or it could be unable to locate the load module. If you do not have an ON EXCEPTION or ON OVERFLOW clause on the CALL statement, your application might abend.

Use the ON EXCEPTION clause to perform a series of statements and to perform your own error handling. For example:

```
MOVE "REPORTA" TO REPORT-PROG
CALL REPORT-PROG
  ON EXCEPTION
    DISPLAY "Program REPORTA not available, using REPORTB.'
    MOVE "REPORTB" TO REPORT-PROG
    CALL REPORT-PROG
    END-CALL
END-CALL
```

If program REPORTA is unavailable, control will continue with the ON EXCEPTION clause.

ON EXCEPTION Limitation: The ON EXCEPTION clause applies only to the availability of the called program. If an error occurs while the called program is running, the ON EXCEPTION clause will not be performed.

CALL Statements

Part 2. Compiling, Linking, and Running Your Program

This part of the book provides instructions for compiling your program on the personal workstation.

Chapter 9. Compiling, Linking, and Running Programs

This chapter explains how to complete the following tasks:

- Set compiler and run-time environment variables
- Compile and link programs
- Specify compiler options
- Understand and respond to compiler errors and messages
- Run compiled programs

Setting Environment Variables

Environment variables are used to set values that can be read by programs. For example, the COBOL run time reads the environment variable COBPATH when a program dynamically CALLs another program.

To specify environment variables, use the SET command. There are two ways to set environment variables:

- Temporarily, by defining the environment variable using the SET command at the command prompt (or as part of a command (.cmd) file).
- Persistently, by defining the environment variable using the SET command.

The environment variable definition using the SET command applies to programs run from the same window where the SET command is issued. For example, the following command syntax sets the COBPATH environment variable (which defines the locations in which the COBOL run time locates dynamically-accessed programs) to include two directories:

```
SET COBPATH=d:\cobdev\dll;d:\dev\dll;
```

However, if you open another window, programs run from the new window will not be affected by the definition you have SET for COBPATH.

Steps required to set an environment variable persistently vary depending on your operating system.

OS/2

To set an environment variable persistently, add the appropriate SET command to the OS/2 file named CONFIG.SYS. If you SET environment variables in CONFIG.SYS, the values of these variables are defined automatically whenever you boot your computer and apply to all OS/2 windows and graphical applications.

For example, the installation process sets up OS/2 environment variables to access the compiler and libraries. These variables are listed in CONFIG.SYS.

The value that you assign to an environment variable can include other environment variables or the variable itself. For example, to add a directory to the value of COBPATH, which has already been set, issue the command

```
SET COBPATH=%COBPATH%;d:\myown\dll;
```

◀OS/2

Windows▶

In Windows 95, environment variables are set in the AUTOEXEC.BAT file. In Windows NT, environment variables are set in the System window (to get there, in Main double-click on Control Panel, then double-click on System.)

To SET an environment variable persistently, add it in the System window (Windows NT) or add the appropriate SET command to the file named AUTOEXEC.BAT (Windows 95). If you set environment variables in this way, the values of these variables are defined automatically whenever you boot your computer and apply to all Windows windows and graphical applications.

On Windows NT changes made to user environment variables in the System window are stored, but you must restart your computer to make the values available to processes, including the command prompt.

For example, the installation process sets up environment variables to access the compiler and libraries. These variables are listed in AUTOEXEC.BAT (Windows 95) or the Registry (Windows NT).

The value that you assign to an environment variable can include other environment variables or the variable itself. For example, to add a directory to the value of COBPATH, which has already been set, issue the command

```
SET COBPATH=%COBPATH%;d:\myown\dll;
```

◀Windows

SET Notation: Throughout this book, the setting of environment variables is illustrated with a SET command. On Windows NT the setting is done in an analogous manner, with the variable name and the value, separated by an equal sign in the SET command, being entered on the two different fields in the System window.

Definitions of COBOL Environment Variables

If you do not specify environment variables, either a default value is applied or the variable is not defined. Environment variables are used by both the compiler and the run-time library.

COBOL Compiler Environment Variables

Compiler Environment Variables

The COBOL compiler uses the following environment variables:

COBOPT
Is one way of specifying COBOL compiler options. You can specify multiple options. Separate each option by a space or comma. For example:

```
SET COBOPT=TRUNC(OPT) TERMINAL
```

Defaults: Individual compiler option defaults apply (see “Default Values for Compiler Options” on page 161).

COBPATH
Specifies PATHs to be used for locating user defined compiler exit programs identified by the EXIT compiler option.

SYSLIB
Specifies **paths** to be used for COBOL COPY statements with *text-names* that are unqualified by *library names*. For a description of how SYSLIB is used for COPY statements, see the description of *Library-name* and *text-name* on page COPY statement on page 204. See “SQL INCLUDE Statement” on page 408 for use with SQL INCLUDE.

TEMPMEM
If TEMPMEM=ON, compiler work files will be memory files rather than disk files. This can significantly reduce compilation time.

In some cases with very large source programs, insufficient memory errors can occur. In this event, reset TEMPMEM to null.

Library-name
A user-defined word that specifies the **path** for the library text. For example:

```
SET MYLIB=D:\CPYFILES\COBCOPY
```

Defaults: If you do not specify a *library-name*, the compiler searches the library path(s) in the following order, the search ending when the file is found:

1. The current directory
2. The path(s) specified by the `-I`*`xxx`* option, if set (see “Options Supported by cob2” on page 142)
3. The paths specified by the SYSLIB environment variable

See the COPY statement on page 204 for the search rules for copy files.

Text-name
A user-defined word that specifies the path for the copybook text.

Defaults: If you do not set *text-name* as an environment variable, the compiler uses the default search described with the COPY statement on page 204.

DB2DBDFT
Is one way of specifying the database for compiling your programs with embedded SQL statements. See “Compiling with the DB2 Co-Processor” on page 406 for more information on connecting to the target database.

Object-Oriented Programming Environment Variables

When you create object-oriented programs, there are different environment variables you need to set. System Object Model (SOM) requires you to set SOM-specific environment variables. For more information on environment variables needed when using SOM to create object-oriented COBOL programs, see Chapter 15, “Using System Object Model (SOM)” on page 317.

Run-Time Environment Variables

The COBOL run-time library uses the following environment variables.

assignment-name

The *assignment-name* can be any COBOL file that you want to specify in an ASSIGN clause. This use of *assignment-name* follows the rules for a COBOL word. For example:

```
SET OUTPUTFILE=d:\january\results.car
```

You can then use the environment variable as a COBOL user-defined word in an ASSIGN clause. For example, based on the previous SET statement, your COBOL source program could include the following:

```
SELECT CARPOOL ASSIGN TO OUTPUTFILE
```

Because `OUTPUTFILE` was defined in the environment, this statement would result in data being written to the file `d:\january\results.car`.

Alternate File System: The general syntax involved in making an assignment to a file stored in an alternate file system is:

```
ASSIGN TO FileSystemID-Filename
```

FileSystemID

Identifies the file system as one of the following:

STL For the STL file system.

VSAM For the VSAM file system. VSAM can be abbreviated to VSA.

Windows ▶

On Windows, `Filename` must start with "\\", indicating remote file access.

◀ Windows

BTR For the Btrieve file system.

If the file system specification is not provided, the run-time option FILESYS is used to select the file system. If FILESYS is not set, the default is VSAM for OS/2 and STL for Windows.

COBOL Run-Time Environment Variables

Filename
The file you want to access.

Alternatively, you can specify an environment variable (for details, see the *IBM COBOL Language Reference*).

Defaults: None. You must set all assignment-names. If you make an assignment to a user-defined word that was not set as an environment variable, the assignment is made to a file with the literal name of the user-defined word (OUTPUTFILE in our earlier example). If the assginment is valid, this file is written to the current directory.

COBMSGS
Specifies the name of a file to which run-time error messages will be written. To capture run-time error messages into a file, use the SET command to set COBMSGS to a file name. If your program has a run-time error that terminates the application, the file that COBMSGS is set to will contain the error message indicating the reason for termination.

Defaults: None. If COBMSGS is not set, error messages are written to the terminal.

COBPATH
Specifies directory path(s) to be used by the COBOL run time to locate dynamically accessed programs, such as .DLL (Dynamic Link Library) files. This variable must be set to run programs that require dynamic loading. For example:

```
SET COBPATH=D:\pgmpath\pgmdll
```

Defaults: None.

COBRTOPT
Specifies the COBOL run-time options. The run-time options are separated by a comma or a colon. Use parentheses or equal signs (=) as the delimiters for suboptions. Options are not case sensitive.

For example:

```
SET COBRTOPT=TRAP=ON:errcount
```

Is equivalent to:

```
SET COBRTOPT=trap(on):ERRCOUNT
```

Defaults: Individual run-time option defaults apply (see Chapter 12, “Run-Time Options” on page 240).

EBCDIC_CODEPAGE
Specifies an EBCDIC code set applicable to the EBCDIC data being processed by programs compiled with the CHAR(EBCDIC) or CHAR(S390) compiler option.

To set the EBCDIC code set, issue the command:

```
SET EBCDIC_CODEPAGE=codepage
```

where *codepage* is the name of the code set to be used.

If EBCDIC_CODEPAGE is not set, it will default to the EBCDIC code page of the current locale. If multiple code pages are available for the current locale the

CHAR(EBCDIC) compiler option must be set, “CHAR” on page 165 discusses this option.

Refer to “Locales and Code Sets Supported” on page 477 for the supported code set translations.

LANG

Specifies the national language locale name in effect for message catalogs and help files. LANG must always be set and is given an initial value during installation. The run-time library uses LANG to access the message catalog. For example, the following command sets the language locale name to U.S. English:

```
SET LANG=En_US
```

If LANG is not SET correctly, run-time messages appear in an abbreviated form.

Defaults: Set to EN_US at installation time.

LC_COLLATE

Determines the locale to be used to define the behaviour of ranges, equivalence classes, and multicharacter collating elements.

Defaults: The locale specified by the LANG environment variable is used.

LC_MESSAGES

Determines the locale which defines the language in which messages are written.

Defaults: The locale specified by the LANG environment variable is used.

LC_TIME

Determines the locale for date and time formatting information.

Defaults: The locale specified by the LANG environment variable is used.

OS/2 ▶

LIBPATH

Specifies the full path name for the COBOL run-time library.

Defaults: Set at installation.

◀ OS/2

LOCPATH

Specifies the search path where the locale information database exists. It is a colon-separated list of directory names. It is used at the time of setting up locale for a process. It is also used to locate conversion tables for EBCDIC data support.

NLSPATH

Specifies the full path name of message catalogs and help files. NLSPATH must always be set and is given an initial value during installation. The run-time library uses NLSPATH to access the message catalog. If NLSPATH is not set correctly, run-time messages appear in an abbreviated form.

For example:

```
SET NLSPATH=C:\COBOL\MESSAGES\%L\%N;%NLSPATH%
```

Cautions: When you set NLSPATH, be sure to add to the NLSPATH, not replace it. Other programs might use this environment variable. Also, note that %L and %N must be upper case.

Defaults: Vary. Set at installation.

SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH

These COBOL environment names are used as the environment variable names corresponding to the mnemonic names used on ACCEPT and DISPLAY statements. Set them equal to files, not existing directory names. For example, the following command defines CONSOLE:

```
SET CONSOLE=c:\mypath\terminal.txt
```

CONSOLE could then be used in conjunction with the following COBOL source code:

```
SPECIAL-NAMES.
   CONSOLE IS terminal
      ...
   DISPLAY 'Hello World' UPON terminal
```

If you set the environment variables SYSIN and SYSOUT to files which have write permission, GUI applications can use ACCEPT and DISPLAY statements to communicate with the user.

Defaults: SYSIN and SYSIPT are directed to the logical input device (keyboard). SYSOUT, SYSLIST, SYSLST, and CONSOLE are directed to the system logical output device (screen). SYSPUNCH and SYSPCH are not assigned a value by default and are not valid unless you explicitly define them.

TEMP

Specifies the location of temporary work files (if needed) for SORT and MERGE functions. For example:

```
SET TEMP=c:\shared\temp
```

Defaults: Vary. Set by the sort utility installation program.

TZ

This variable is used to describe the time zone information to be used by the locale and has the following format:

```
SET TZ=SSS[+|-]nDDD[,sm,sw,sd,st,em,ew,ed,et,shift]
```

The values for the TZ variable are defined below.

Figure 40 (Page 1 of 2). TZ Environment Variable Parameters

Variable	Description	Default Value
SSS	Standard time zone identifier. This must be three characters, must begin with a letter, and can contain spaces.	EST

Figure 40 (Page 2 of 2). TZ Environment Variable Parameters

Variable	Description	Default Value
n	Difference (in hours) between the standard time zone and coordinated universal time (UTC), formerly Greenwich mean time (GMT). A positive number denotes time zones west of the Greenwich meridian, a negative number denotes time zones east of the Greenwich meridian.	5
DDD	Daylight saving time (DST) zone identifier. This must be three characters, must begin with a letter, and can contain spaces.	EDT
sm	Starting month (1 to 12) of DST.	4
sw	Starting week (-4 to 4) of DST.	1
sd	Starting day of DST: 0 to 6 if *sw* is not zero; 1 to 31 if *sw* is zero.	0
st	Starting time (in seconds) of DST.	3600
em	Ending month (1 to 12) of DST.	10
ew	Ending week (-4 to 4) of DST.	-1
ed	Ending day of DST: 0 to 6 if *ew* is not zero; 1 to 31 if *ew* is zero.	0
et	Ending time (in seconds) of DST.	7200
shift	Amount of time change (in seconds).	3600

For example:

```
SET TZ=CST6CDT
```

sets the standard time zone to CST, the daylight saving time to CDT, and sets a difference of 6 hours between CST and UTC. It does not set any values for the start and end of daylight saving time.

Other possible values are PST8PDT for Pacific United States and MST7MDT for Mountain United States.

When TZ is not present, the default is EST5EDT, the default locale value. When only the standard time zone is specified, the default value of *n* (difference in hours from GMT) is 0 instead of 5.

If you give values for any of *sm*, *sw*, *sd*, *st*, *em*, *ew*, *ed*, *et*, or *shift*, you must give values for all of them. If any of these values is not valid, the entire statement is considered not valid, and the time zone information is not changed.

Defaults: Depends on the current locale. See Figure 40 on page 140 for the default locale values.

Environment Variable Precedence

Some environment variables (such as COBPATH and NLSPATH) define directories in which to search for files. If multiple directory paths are listed, they are delimited by semi-colons. Paths defined by environment variables are evaluated in order, from the first path to the last in the SET statement. Therefore, if you have multiple files with the same name that are defined in the paths of an environment variable, be aware that the *first* located copy of the file is used.

Compiling and Linking Programs

The command `cob2` is the command-line utility which invokes the COBOL compiler and linker. For compiling Visual Builder projects see *Visual Builder User's Guide,* SC26-9053. For compiling using WorkFrame see the WorkFrame online help. `cob2` accepts options to control the compilation and link-edit in any order on the command line. Likewise, if you want to compile multiple files, the filenames can be specified at any position in the command syntax. However, all options and filenames should be separated by spaces.

Because OS/2 and Windows are not case-sensitive, `cob2` and its options do not need to be capitalized. The general syntax for `cob2` is:

```
►►──cob2──┬─────────┬──filenames──────────────────────────────►◄
          └─options─┘
```

For example, the following command:

```
cob2 -g filea.cbl fileb.cbl -v -qflag(w)
```

is equivalent to:

```
cob2 filea.cbl -qflag(w) -g -v fileb.cbl
```

Usage Notes

1. Any options specified apply to all files on the command line.
2. `cob2` passes all files with extensions named in “Filenames and Extensions Supported” on page 145 to the linker; all other files are passed to the compiler.
3. The default location for compiler input and output is the current directory.

Options Supported by cob2

-b"*xxx*" Pass the *xxx* string to the linker as parameters. *xxx* is a list of linker options separated by blank spaces. The `cob2` default parameters are also passed. There should be no spaces between `-b` and `"xxx"`.

Alternatively, linker options can be specified directly as individual `cob2` options. For example, to pass the /DE option to the linker:

```
cob2 /DE myprog.cbl
```

For information on linker options, see Chapter 11, “Setting Linker Options” on page 208.

-c Compile programs but do not link them.

-cmain Make a C or PL/I object file containing a main routine[5] the main entry point in the executable file (.EXE).

Warning: If a C or PL/I object file containing a main routine is linked with one or more COBOL object files, -cmain **must** be used to designate the C or PL/I routine as the main entry point in the executable file; a COBOL program **cannot** be the main entry point in an executable file containing a C or PL/I main. Unpredictable execution behavior will occur if this is attempted and no diagnostics are issued.

OS/2 ▶

Under OS/2, -cmain is only required if -host is also specified.

◀ OS/2

Example:

```
cob2 -cmain myCmain.obj myCOBOL.obj
```

and

```
cob2 -cmain myCOBOL.obj myCmain.obj -main:myCmain
```

both generate the executable file myCmain.exe with the main entry point being the C main() function contained in the myCmain.obj object file.

-comprc_ok=*n* Controls the cob2 behavior on the return code from the compiler. If the return code returned by the compiler is less than or equal to *n*, cob2 continues to the link step, or, in the compile only case, exits with a zero return code. If the return code returned by the compiler is greater than *n*, cob2 exits with the same return code returned by the compiler.

The default is: -comprc_ok=4.

-dll[:*xxx*] Causes cob2 to produce linker files (.LIB and .EXP) to create a DLL named *xxx*. If *xxx* is omitted, the name of the first object (.OBJ) or COBOL source (usually .CBL or .PPR) file specified in the cob2 command is the name of the DLL (and .LIB and .EXP files).

-g Produce symbolic information used by the debugger. This option is equivalent to compiling with the TEST compiler option and linking with the /DEBUG linker option.

[5] In C, a main routine is identified by the function name main(). In PL/I, a main routine is identified by the PROC OPTIONS(MAIN) statement.

-host
Set all host data compiler options:

BINARY(S390)
CHAR(EBCDIC)
COLLSEQ(EBCDIC)
FLOAT(S390)

Note: This option will present run-time command-line arguments in host data format, that is, EBCDIC for character data, and "big endian" for binary data.

-I*xxx*
Add a path *xxx* to the directories to be searched for COPY files if a *library-name* is not specified (see "Compiler Environment Variables" on page 136). Only a single path is allowed per `-I` option. To add multiple paths, use multiple `-I` options. There should be no spaces between `-I` and *xxx*. (This is upper-case "eye," not lower-case "el.")

Use LIB: If you use the COPY statement, you must ensure that the LIB compiler option is in effect.

For a description of the manner in which the compiler evaluates paths for COPY files, see the description of the *Library-name* environment variable in "Run-Time Environment Variables" on page 137.

-main:*xxx*
Make object file *xxx* the COBOL main program of the executable (`.EXE`) file. *xxx* must be the filename of an object (`.OBJ`) file or source file specified to `cob2`. For example, `cob2 -main:abc a1.cbl d:\cats\abc.obj b2.cbl` will result in abc being the main program. *xxx* cannot appear in a linker response file.

If `-main` is not specified, the first object or source file specified will, in the absence of a response file, be the COBOL main program.

If the syntax of `-main:`*xxx* is invalid, or *xxx* is not the filename of an object or source file processed by `cob2`, `cob2` will terminate.

-p
Include the profile hooks that allow the Performance Analyzer to monitor the application execution and create a trace file. This option is equivalent to compiling with the PROFILE compiler option and linking in the Performance Analyzer module IWZPAN40.OBJ.

-q*xxx*
Use the option *xxx* (where *xxx* is any compiler option) when calling the compiler. If a parenthesis is part of the compiler (sub)option, or a series of options are specified, they should be included in quotes. For multiple options, each option should be delimited by a blank or comma. There should be no spaces between `-q` and *xxx*. For example, you can use

```
-qoptiona,optionb
```

or

```
-q"optiona optionb"
```

Special Syntax

If you plan to use a batch or command file to automate your `cob2` tasks, a special syntax is provided for the `-q`*`xxx`* option. Use the following syntax to prevent the command shell from passing invalid syntax to `cob2`:

- Use "=" (equal sign) and ":" (colon) rather than "( )" (parentheses) to specify compiler suboptions. For example, use

  ```
  -qBINARY=NATIVE:,ENTRYINT=OPTLINK:
  ```

  rather than

  ```
  -qBINARY(NATIVE),ENTRYINT(OPTLINK)
  ```

- Use "_" (underscore) rather than " ' " (apostrophe) where a compiler option requires a suboption to be delimited by apostrophes. For example, use

  ```
  -qEXIT=INEXIT=_String_,MYMODULE::
  ```

  rather than

  ```
  -qEXIT(INEXIT('String',MYMODULE))
  ```

- Do not use any blanks in the option string.

-v Display compile and link steps, and execute them.

-# Display compile and link steps, but do not execute them.

Filenames and Extensions Supported

Files with @ as the first character and files with the following extensions are assumed to be linker parameters and are passed to the linker. Those with recognized file extensions are processed as follows:

.DEF The name of the module definition file. For more information about module definition files, see "Module Definition Files" on page 442.

.DLL The name of the generated dynamic link library (DLL). The default DLL is the first source file listed in the `cob2` command syntax with an extension of `.DLL`.

Windows ▶

.EXP The name of the export file, if required. For more information about export files, see "Export Files (Windows Only)" on page 442.

◀ Windows

.EXE The name of the generated executable file. If not specified, the name defaults to the name of the first COBOL source file listed in the `cob2` command with the file extension `.EXE`.

.IMP The name of the import library associated with a .DLL that contains symbols (usually names of external routines) referenced by your programs. This file is used by the linker to resolve those references.

.LIB The name of the import or standard library, which contains symbols (usually names of external routines) referenced by your programs. This file is used by the linker to resolve those references.

.MAP The name of the map file. If not specified, no map file is generated.

.OBJ The name of the object file(s) to be passed to the linker.

All other files are processed by the compiler. The file extension `.CBL` is most commonly used for COBOL source.

Examples using cob2

The following examples illustrate the use of `cob2`:

- To compile the file `alpha.cbl`, enter:

  ```
  cob2 -c alpha.cbl
  ```

  The compiler produces the files `alpha.obj` and `alpha.lst`.

- To compile `alpha.cbl` and `beta.cbl`, enter:

  ```
  cob2 -c alpha.cbl c:\mydir\beta.cbl
  ```

  The compiler produces the files `alpha.obj`, `beta.obj`, `alpha.lst`, and `beta.lst` in the current directory.

- To link two files together, compile them without the `-c` option. For example, to compile and link `alpha.cbl` and `beta.cbl` and generate `alpha.exe`, enter:

  ```
  cob2 alpha.cbl beta.cbl
  ```

  This command creates `alpha.obj` and `beta.obj`, then links `alpha.obj`, `beta.obj`, and the COBOL libraries. If the link step is successful, it produces an executable program named `alpha.exe`.

- In the following example:

  ```
  cob2 alpha.obj beta.cbl mylib.lib gamma.exe
  ```

  `beta.cbl` is compiled, and the string:

  ```
  alpha.obj beta.obj mylib.lib /out:gamma.exe
  ```

  is passed to the linker. If linking is successful, the executable `gamma.exe` is produced.

- In the following example:

  ```
  cob2 alpha.cbl alpha.def
  ```

  `alpha.dll` will be produced (assuming a valid `alpha.def` file).

- To compile with the LIST and NOADATA options, enter:

  ```
  cob2 -qlist,noadata alpha.cbl
  ```

  Note: There is no space between the `-q` and the options list.

  Options should be delimited by commas or blanks if enclosed in quotes:

  ```
  cob2 -q"list noadata" alpha.cbl
  ```

Alternative Ways to Specify Compiler Options

“Compiling and Linking Programs” on page 142 explains how to specify compiler options using the `cob2` command. There are other ways to select the options used to compile your programs. Here are three additional methods:

1. The COBOPT environment variable (See “Definitions of COBOL Environment Variables” on page 135)
2. Command-line specification of compile environment variables and `cob2` options, such as an OS/2 command (`.CMD`) file
3. Specification by way of *CONTROL (synonym, *CBL) or PROCESS (synonym, CBL) statements

These means of specification are listed in order of precedence. For example, an option specified using PROCESS overrides every other option specification *except for non-overridable options selected during product installation.*

Specifying Compiler Options with the PROCESS (CBL) Statement

You can code compiler options on the PROCESS statement in your COBOL source (`.CBL`) programs. The PROCESS statement is placed before the IDENTIFICATION DIVISION header and has the following format:

```
►►──┬─CBL─────┬──┬──────────────┬─────────────────────────────────►◄
    └─PROCESS─┘  └─options-list─┘
```

PROCESS Statement Rules

- Your programming organization can inhibit the use of PROCESS statements with the default options module of the COBOL compiler. When PROCESS statements are found in a COBOL program where not allowed by the organization, the COBOL compiler generates error diagnostics.
- One or more blanks must separate PROCESS and the first option in *options-list*. Separate options with a comma or a blank. Do not insert spaces between individual options and their suboptions.
- The PROCESS statement must be placed before any comment lines or compiler-directing statements.
- PROCESS can start in columns 1 through 66. A sequence field is allowed in columns 1 through 6. When used with a sequence field, PROCESS can start in columns 8 through 66. If used, the sequence field must contain six characters, and the first character must be numeric.

  You can use CBL as a synonym for PROCESS. CBL can start in columns 1 through 70. When used with a sequence field, CBL can start in columns 8 through 70.
- You can use more than one PROCESS statement. If multiple PROCESS statements are used, they must follow one another with no intervening statement of any other type.

- Options cannot be continued across multiple PROCESS statements.

Compiler-Detected Errors and Messages

As the compiler processes your source program, it checks for COBOL language errors you might have made. For each error discovered, the compiler issues a message. These messages are included in the compilation listing (subject to the FLAG option). The compiler listing file has the same name as the compiler source file, with the file extension `.LST`. For example, the compiler listing for `myfile.cbl` would be `myfile.lst`. The listing file is written to the directory from which `cob2` was run.

Each message in the listing does the following:

- Explains the nature of your error
- Identifies the compiler phase that detected the error
- Identifies the severity level of the error

Wherever possible, the message provides specific instructions for correcting the error.

Compiler Error Messages

The messages for errors found during processing of compiler options, CBL and PROCESS statements, or BASIS, COPY, and REPLACE statements are displayed near the top of your listing.

The messages for compilation errors found in your program (ordered by line number) are displayed near the end of the listing for each program.

A summary of all errors found during compilation is displayed near the bottom of your listing. Each message issued by the compiler is of the following form:

Format

nnnnnn **IGY***ppxxxx*-*l* *message-text*

nnnnnn
The number of the source statement of the last line the compiler was processing. Source statement numbers are listed on the source printout of your program. If you specified the NUMBER option at compile time, these are your original source program numbers. If you specified NONUMBER, the numbers are those generated by the compiler.

IGY
The prefix that identifies this message as coming from the COBOL compiler.

pp Two characters that identify which phase of the compiler discovered the error. As an application programmer, you can ignore this information. If you are diagnosing a suspected compiler error, contact IBM for support.

xxxx
A 4-digit number that identifies the error message.

l A character that indicates the severity level of the error: I, W, E, S, or U (see "Compiler Error Message Codes").

message-text
The message text itself which, in the case of an error message, is a short explanation of the condition that caused the error.

Caution: If you used the FLAG option to suppress messages, there might be additional errors in your program.

Compiler Error Message Codes

Errors the compiler can detect fall into five categories of severity:

I	Informational (Return Code=0)	An informational-level message is an aid to you. No action is required and the program executes correctly as it stands.
W	Warning (Return Code=4)	A warning-level message calls attention to a possible error. It is probable that the program executes correctly as written.
E	Error (Return Code=8)	An error-level message indicates a condition that is definitely an error. The compiler has attempted to correct the error but the results of program execution might not be what you expect. You should correct the error.
S	Severe (Return Code=12)	A severe-level message indicates a condition that is a serious error. The compiler was unable to correct the error. The program does not execute correctly, and execution should not be attempted. An .OBJ file might not be created.
U	Unrecoverable (Return Code=16)	An unrecoverable-level message indicates an error condition of such magnitude that the compilation was terminated.

In the following example, the part of the statement that caused the message to be issued is enclosed in quotes.

Compiler-Detected Errors and Messages

```
⋮
LineID  Message code  Message text

     2  IGYDS0009-E   "PROGRAM" should not begin in area "A".  It was processed as if found in area "B".

     2  IGYDS1089-S   "PROGRAM" was invalid.  Scanning was resumed at the next area "A" item, level-number,
                      or the start of the next clause.

     2  IGYDS0017-E   "ID" should begin in area "A".  It was processed as if found in area "A".

     2  IGYDS1003-E   A "PROGRAM-ID" paragraph was not found.  Program name "CBLPGM01" was assumed.

     2  IGYSC1082-E   A period was required.  A period was assumed before "ID".

     2  IGYDS1102-E   Expected "DIVISION", but found "ALONGPRO".  "DIVISION" was assumed before "ALONGPRO".

     2  IGYDS1082-E   A period was required.  A period was assumed before "ALONGPRO".

     2  IGYDS1089-S   "ALONGPRO" was invalid.  Scanning was resumed at the next area "A" item, level-number,
                      or the start of the next clause.

     2  IGYDS1003-E   A "PROGRAM-ID" paragraph was not found.  Program name "CBLPGM02" was assumed.

     3  IGYPS0017-E   "PROCEDURE" should begin in area "A".  It was processed as if found in area "A".

    34  IGYSC0137-E   Program-name "ALONGPRO" did not match the name of any open program.  The "END PROGRAM" statement
                      was assumed to have ended program "CBLPGM02".

    34  IGYSC0136-E   Program "CBLPGM01" required an "END PROGRAM" statement at this point in the program.
                      An "END PROGRAM" statement was assumed.
Messages    Total    Informational    Warning    Error    Severe    Terminating

Printed:      12                                  10        2
⋮
```

Correcting Errors in Your Source Program

Messages about source coding errors indicate where the error happened (LINEID) and the text of the message tells you what the problem is. With this information, you can correct the source program and re-compile.

Although you should try to correct errors, it is not necessary to fix all of them. A W-level or I-level message can be left in a program without much risk, and you might decide that the recoding and compilation needed to remove the error are not worth the effort. On the other hand, S-level and E-level errors indicate probable program failure and should be corrected.

U-level errors are in a class by themselves. Here, you have no choice but to correct the error, because the compiler is forced to end early and does not produce complete object code and listing. In contrast with the four lower levels of errors, a U-level error might not result from a mistake in the source program. It could come from a flaw in the compiler itself, or in the operating system.

After correcting the errors in your source program, re-compile the program. If this second compilation is successful, go on to the link-editing step. If the compiler still finds problems, repeat the above procedure until only informational messages are returned.

Generating a List of All Compiler Error Messages

You can generate a complete listing of compiler diagnostic messages, with their explanations, by compiling a program with a *program-name* of ERRMSG specified in the PROGRAM-ID paragraph. The rest of the program can be omitted. For example:

```
Identification Division.
Program-ID. ErrMsg.
```

The listing that is produced includes messages from other IBM COBOL platforms, such as AIX, OS/390, and VM. Some messages do not apply to OS/2 or Windows

Starting the Linker

Once the compiler has created object modules out of your source files, use the linker to link them together with the IBM VisualAge COBOL runtime libraries to create an .EXE file or .DLL file. By default, the IBM VisualAge COBOL compiler `cob2` invokes the linker for you.

There are several ways you can start the linker:

- From the popup menu of an object file in a WorkFrame project, or from the project popup menu as part of the make or build process.
- Through the compiler, which automatically invokes the linker.
- From the command line.
- Through a make file, which invokes both the compiler and the linker.

Linking within WorkFrame

To use the linker through WorkFrame, do the following:

1. Double-click on your project icon. The Project Window appears.

   OS/2 ▶

   At this point you can customize settings for the project, if the default settings for the project type are unacceptable. The **Options** menu contains choices that allow you to specify the actions available to the project, and compiler and linker options. Use **Build Smarts** to set options for a standard task. Use the Compiler and Linker Options dialogs to set options on an individual basis.

2. Select **Build** from the **Actions** menu. Your project is created, with the compiler and linker invoked as required.

   ◀ OS/2

   Windows ▶

3. Use the initial WorkFrame dialog to either open an existing project or create a new one. These actions are also choices on the **Project** pull-down menu. Once a project has been opened or created, its files are listed in the WorkFrame window.

4. Customize the linker options from the **Options** pull-down menu, if you do not want to use the defaults. The **Options** menu contains choices that allow you to specify options for other actions (for example, `compile`). You can also customize the project settings by selecting **Settings** from the **View** pull-down menu.
5. Select **Build** from either the **Project** pull-down menu or the project toolbar. Your project is built using the linker as required.

◀Windows

Linking through the Compiler

When you invoke the IBM VisualAge COBOL compiler, it compiles the object files from your source code and then automatically starts the linker, to link the object files into an .EXE or .DLL file. Use the `cob2` option `-b` to pass options to the linker.

If you do not want the compiler to start the linker, specify the `cob2` option `-c`. You can then invoke the linker in a separate step.

Windows▶

The compiler does not pass any default parameters to the linker.

◀Windows

OS/2▶

By default, the `cob2` compiler invokes the linker with the following options:

`/FREEFORMAT` Uses the free-format syntax, rather than the LINK386-compatible syntax.

`/NOLOGO` Specify no logo.

`/BASE:65536` Specify the starting address of the program. For .DLL files, this results in a smaller and potentially faster executable, if the specified address is free when the .DLL is loaded. For .EXE files, the OS/2 operating system always loads executable programs at 64K. You can give the linker the address 65536 (or `0x10000`) to let the linker know where the program will be loaded, so it can resolve relocation information at link time, resulting in a smaller .EXE file.

`/PMTYPE:VIO` Create program with standard I/O that is compatible with Presentation Manager.

In addition, the following `cob2` option generates the equivalent linker option:

`-g` Generates debugger information. Passes `/DEBUG` to the linker.

See "Linker Options for OS/2" on page 211 for more information on these linker options.

Passing Additional Options to the Linker

You can override these options, and pass additional options to the linker, using the cob2 option -b. For example, to generate a map file and override the default alignment, specify

```
cob2 -.b"/AL:256 /MAP"
```

If you do not want the compiler to start the linker, specify the cob2 option -c. You can then invoke the linker in a separate step.

◀ OS/2

Linking from a Make File

Use a make file to organize the sequence of actions (such as compiling and linking) required to build your project. You can then invoke all the actions in one step. The NMAKE utility saves you time by performing actions on only the files that have changed, and on the files that incorporate or depend on the changed files.

You can write the make file yourself, or you can use WorkFrame to manage the make file. When you build through WorkFrame, a make file is created and maintained automatically.

Optimized Linking (OS/2 Only)

OS/2 ▶

Removing Unreachable Functions

Just as the compiler can optimize your source code by removing or replacing instructions, the linker can optimize your object code, including code in libraries you are linking in, by removing unreferenced functions. When the function is removed, any code that was required only by that function is also removed, including any other functions that were referenced only by that function. This reduces the size of your output file.

Link with the option /OPTFUNC to remove functions that are:

- Not referenced in any input file
- Rendered unreferenced by the removal of other functions
- Not exported for use in other files

Performance Consideration

Optimized linking generally takes longer than regular linking, because of the extra processing that the linker performs. However, if the optimization is effective enough, it can actually speed up the linking process, because there is less information to write to file. Generally, you may want to link without the /OPTFUNC option, until your code is tested and stable.

Linker Input and Output

Packing Executables

Specify /EXEPACK to reduce the size of the executable by compressing pages in the file. The operating system automatically decompresses the pages when the program is loaded. If your program is intended to run only on OS/2 version 3.0 or later, then specify /EXEPACK:2 for best results. If your program is also intended to run on older versions of OS/2, specify /EXEPACK:1.

Specify /PACKCODE to produce slightly faster and more compact code by grouping neighboring code segments that have similar attributes.

Specify /PACKDATA to produce more compact files by grouping neighboring data segments that have similar attributes.

Specify /DBGPACK when you are debugging, to reduce the size of the executable file and potentially improve debugger performance.

See “Linker Options for OS/2” on page 211 for more information on these and other linker options.

◀ OS/2

Linker Input and Output Files

The linker takes object files, links them with each other and with any library files you specify, and produces an executable output file. The executable output can be either an executable program (extension .EXE) file or a dynamic link library (extension .DLL).

The linker optionally produces a map file, which provides information about the contents of the executable output.

Input	**Output**
options	*executable file* (.EXE or .DLL)
object files (*.OBJ)	*map file* (.MAP)
library files (*.LIB)	*return code*
import libraries (*.LIB)	
module definition file (.DEF)	

Linker Search Rules

When searching for an object (.OBJ), library (.LIB), or module definition (.DEF) file, the linker looks in the following locations in this order:

1. The directory you specified for the file, or the current directory, if you did not give a path. Default libraries do not include path specifications.

   Note: If you specify a path with the file, the linker searches only that path, and stops linking if the file cannot be found there.

2. Any directories entered by themselves on the command line must end with a slash (/) or backslash (\) character. See “Specifying Directories” on page 155 for more information.

OS/2 ▶ If you specified /NOFREE, they must be in the *libraries* parameter.
◀ OS/2

3. Any directories listed in the LIB environment variable.

If the linker cannot locate a file, it generates a fatal error message and stops linking.

Example

If you respond to linker prompts as follows:

OS/2 ▶

```
ILINK /NOFREE
Object Modules [.obj]: FUN TEXT TABLE CARE
Run File [fun.*]:
List File [fun.map]:
Libraries [.lib]: NEWLIBV2 C:\TESTLIB\
Definitions File [nul.def]:
```

◀ OS/2

Windows ▶

```
FUN.OBJ TEXT.OBJ TABLE.OBJ CARE.OBJ
NEWLIBV2.LIB
C:\TESTLIB\
```

◀ Windows

The linker links four object files to create an executable file named FUN.EXE. The linker searches NEWLIBV2.LIB before searching the default libraries to resolve references.

To locate NEWLIBV2.LIB and the default libraries, the linker searches the following locations in this order:

1. The current directory (because NEWLIBV2.LIB was entered without a path)
2. The C:\TESTLIB\ directory
3. The directories listed in the LIB environment variable

Specifying Directories

To have the linker search additional directories for input files, specify a drive or directory by itself on the command line. Specify the drive or directory with a slash (/) or backslash (\) character at the end for the linker to recognize it as a path.

The paths you specify are searched before the paths in the LIB environment variable. See “Linker Search Rules” on page 154 for more information.

Linker Input and Output

OS/2

Note: If you specified `/NOFREE`, then you can only specify directories in the *library* parameter at the command line, or in response to the `Libraries [.LIB]:` prompt. You must still end each directory with a slash (/) or or backslash (\) character.

OS/2

File Name Defaults

If you do not enter a file name, the linker assumes the defaults shown below.

OS/2 If you specify `/NOFREE`, the linker also assumes default file extensions for files without extensions. OS/2

Figure 41. Linker Filename Defaults

File	Default File Name	Default Extension
Object files	None. You must enter at least one object file name.	.OBJ
Output file	The base name of the first object file.	.EXE
Map file	The base name of the output file.	.MAP
Library files	The default libraries defined in the object files. Use compiler options to define the default libraries. Any additional libraries you specify are searched before the default libraries.	.LIB
Module definition file	None. The linker assumes you accept the default for all module statements.	.DEF

Specifying Object Files

When you invoke the linker from the command line, the linker assumes that any input it cannot recognize as other files, options, or directories must be a object file. Use a space or tab character to separate files.

OS/2

If you specified `/NOFREE` to use the LINK386-compatible syntax, then the first set of file names you give it are taken as object files, up to the first comma. Use a plus (+), space, or tab as a separator between the file names. If you do not specify an extension, the linker assumes the .OBJ extension.

OS/2

When you invoke the linker through the compiler, the compiler automatically passes the object files it creates to the linker, as well as passing any object files you specify on the compiler command line.

You must enter at least one object file.

The linker accepts object files compiled or assembled:

Windows ▶

- In 32-bit OMF format
- For Windows NT Version 3.5.1 (or higher) or Windows 95
- For the 80386, 80486, and Pentium microprocessors

◀ Windows

OS/2 ▶

- In 16- or 32-bit OMF format
- For OS/2 version 1.0 or higher
- For the 80286 (16-bit only), 80386, 80486, and Pentium microprocessors

Entering Library Files As Object Files

If you specify `/NOFREE` to use LINK386-compatible syntax, then you can enter library files in place of object files in the *object* parameter on the command line or at the `Object Modules [.OBJ]:`. prompt. Be sure to include the .LIB file name extension; otherwise, the linker assumes a `.obj` extension.

When you enter a library as an object file, all the modules in the library are added to your output file, just as if you had entered all of the library's modules as object files in the *object* parameter.

In contrast, when you enter a library in the *library* parameter, the linker links only to those modules needed to resolve external references.

If you are linking with the `/FREEFORMAT` option (the default), you cannot enter library files as object files.

◀ OS/2

Specifying Executable Output Type

You can use the linker to produce executable modules (with the extension .EXE) or dynamic link libraries (with the extension .DLL). The linker produces .EXE files by default.

Use linker options or statements in the module definition (.DEF) file, to specify what kind of output you want:

- To produce an .EXE, specify the `/EXEC` option, or include the module statement NAME. See “Static Linking Overview” on page 439 for more information.
- To produce a .DLL, specify the `/DLL` option, or include the module statement LIBRARY. See “Creating a DLL” on page 441 for more information.

Linker Errors

Linker Return Codes

The linker has the following return codes:

Code **Meaning**

0 The link was completed successfully. The linker detected no errors, and issued no warnings.

4 **Warnings** issued. There may be problems with the output file.

8 **Errors** detected. The linking might have completed, but the output file cannot be run successfully.

12 Both warnings issued and errors detected (see return codes 4 and 8)

16 **Severe errors** detected. Linking ended abnormally, and the output file cannot be run successfully.

20 Both warnings issued and severe errors detected (see return codes 4 and 16)

24 Both errors and severe errors issued (see return codes 8 and 16)

28 The linker issued warnings, detected errors, and detected severe errors (see return codes 4, 8, and 16)

If you invoke the linker through a makefile, you can force NMAKE to ignore warnings by putting `-7` before the `ILINK` command. If you invoke the linker through the compiler, then a return code of zero is issued for warnings.

Correcting Linker Errors

If you use the PGMNAME(UPPER) compiler option, then the names of subprograms referenced in CALL statements are translated to upper case. So, for example:

```
    Call "RexxStart"
```

will be translated by the compiler to

```
    Call "REXXSTART"
```

This affects the linker, which recognizes case-sensitive names. If the “real” name of the called program is RexxStart, the linker will not find it, and will produce an error message saying that REXXSTART is an unresolved external reference.

This type of error typically happens when you are calling API routines supplied by another software product. If the API routines have mixed-case names, you must:

- Use the PGMNAME(MIXED) compiler option, and
- Ensure that your CALL statements specify the correct names, with the correct mix of upper and lower case characters, of the API routines.

Windows Considerations

Windows ▶

Under Windows, the default linkage convention is SYSTEM(STDCALL), which is in effect when you use the compiler option CALLINT(SYSTEM). With this convention, the name of the called routine is expanded by:

- prefixing an underscore character (_), and
- suffixing an at symbol (@) and a one or two digit number signifying the length in bytes of the argument list.

This convention is known as “name decoration”. For example, if you code:

```
Call SubProg Using Parm-1 Parm-2.
```

the name of the called routine will be `_SubProg@8`. If, however, the SubProg routine itself is coded as:

```
Procedure Division Using Parm-1 Parm-2 Parm-3.
```

its system-generated name will be `_SubProg@12`. This will cause an error in the linker because it will not be able to resolve the call to `_SubProg@8`.

If you are using this linkage convention, you must ensure that the argument list in the calling program exactly matches the parameter list in the called subroutine.

◀ Windows

Running COBOL programs

To run a COBOL program, first make sure that any needed environment variables are SET (see “Setting Environment Variables” on page 134). Then type the name of the executable module on the command line or execute a command file which invokes the module. For example, if

```
cob2 alpha.cbl beta.cbl
```

is successful, you can execute the program by typing:

```
alpha
```

If your program uses an environment variable name to assign a value to a system file name, set the environment variable before typing the command which executes the program.

If the run-time routines detect an error, they send a message to the error unit.

If run-time messages are abbreviated or incomplete, one or both of the following environment variables might be incorrectly set:

- LANG
- NLSPATH

Chapter 10. Compiler Options

You can direct and control compilation with the following:

- Compiler options
- Compiler-directing statements (compile directives)

Compiler options are listed and described in alphabetical order in “Compiler Option Descriptions” on page 162. Compiler-directing statements are listed at the end of this chapter, on page 202.

Compiler Options Summary

Compiler options affect the aspects of your program listed in Figure 42.

Figure 42 (Page 1 of 2). List of Compiler Options

Aspect of Your Program	Compiler Option	Abbreviations	Found on Page
Source language	APOST	None	188
	CURRENCY	CURR|NOCURR	168
	LIB	None	182
	NUMBER	NUM|NONUM	185
	QUOTE	Q	188
	SEQUENCE	SEQ|NOSEQ	190
	SQL	None	192
	WORD	WD|NOWD	199
Date processing	DATEPROC	DP	170
	YEARWINDOW	YW	201
Maps and listings	LINECOUNT	LC	183
	LIST	None	183
	MAP	None	184
	SOURCE	S|NOS	191
	SPACE	None	192
	TERMINAL	TERM|NOTERM	193
	VBREF	None	198
	XREF	X|NOX	200
Object module generation	COMPILE	C|NOC	168
	OPTIMIZE	OPT|NOOPT	185
	PGMNAME	PGMN(U|M)	186
	SEPOBJ	None	189

Figure 42 (Page 2 of 2). List of Compiler Options

Aspect of Your Program	Compiler Option	Abbreviations	Found on Page
Object code control	BINARY	None	163
	CHAR	None	165
	FLOAT	None	180
	TRUNC	None	195
	ZWB	None	201
CALL statement behaviour	DYNAM	DYN|NODYN	171
Debugging and diagnostics	FLAG	F|NOF	177
	FLAGSTD	None	178
	TEST	None	194
	SSRANGE	SSR|NOSSR	193
Other	ADATA	None	162
	ANALYZE	None	162
	CALLINT	None	164
	COLLSEQ	None	167
	ENTRYINT	None	171
	EXIT	EX(INX,LIBX,PRTX,ADX)	172
	IDLGEN	IDL|NOIDL	181
	PROBE	None	188
	PROFILE	None	188
	SIZE	SZ	191
	THREAD	None	194
	TYPECHK	TC|NOTC	197
	WSCLEAR	None	199

Default Values for Compiler Options

The default options that were set up when your compiler was installed are in effect for your program unless you override them with other options. To find out the default compiler options in effect, run a test compilation without specifying any options; the output listing lists the default options specified by your installation.

Non-overridable Options: In some installations, certain compiler options are set up so that you cannot override them. If you have problems, see your system administrator.

Performance Considerations

The BINARY, CHAR, DYNAM, FLOAT, OPTIMIZE, SSRANGE, TEST, and TRUNC compiler options can all affect run-time performance.

Compiler Option Descriptions

The compiler option descriptions that follow are given in alphabetical order. For a list of compiler options by effect, refer to Figure 42 on page 160.

Installation Defaults: The defaults listed with the options below are the defaults shipped with the product. They might have been changed by your installation.

ADATA

```
►►──┬─ADATA───┬──────────────────────────────────────────────────►◄
    └─NOADATA─┘
```

Default is: ADATA

Abbreviations are: None

Use ADATA when you want the compiler to create a SYSADATA file, which contains records of additional compilation information. This information is used by other tools, which will set ADATA ON for their use. The size of this file generally grows with the size of the associated program.

You cannot specify ADATA in a PROCESS (CBL) statement; it can be specified only:

- On invocation of the compiler using an option list
- As a command option
- As an installation default

See “EXIT” on page 172 for information on using SYSADATA with your compiler-exit module.

ANALYZE

```
►►──┬─ANALYZE───┬────────────────────────────────────────────────►◄
    └─NOANALYZE─┘
```

Default is: NOANALYZE

Abbreviations are: None

Use ANALYZE when you want the compiler to check the syntax of embedded SQL and CICS statements in addition to native COBOL statements.

No executable code is generated when this compiler option is specified, regardless of the COMPILE|NOCOMPILE setting. The ANALYZE option also enables COPY/BASIS/REPLACE processing, regardless of the LIB|NOLIB setting.

You can specify the ADATA option with this option to create a SYSADATA file for later analysis by program understanding tools, such as the Year 2000 tool included with the *Professional Edition* of IBM VisualAge COBOL.

This option may be set as the installation default option or as a compiler invocation option, but may not be set on a CBL or PROCESS statement.

The specification of the ANALYZE option forces the handling of the following character strings as reserved words:

CICS
EXEC
END-EXEC
SQL

APOST

See “QUOTE/APOST” on page 188.

BINARY

```
►►──BINARY(──┬─NATIVE─┬─)────────────────────────────────────────────►◄
             ├─S370───┤
             └─S390───┘
```

Default is: NATIVE

Abbreviations are: None

Specifying NATIVE means that BINARY, COMP, and COMP-4 data items are represented in the native format of the platform or product. For example, binary data on a PC would be stored in Little-Endian format (least significant digit on the highest address). Binary data on AIX would be stored in Big-Endian format (most significant digit on the highest address).

Specifying S370 or S390 means that binary data is represented in the Big-Endian format. However, COMP-5 binary data and data items defined with the NATIVE keyword on the USAGE clause are not impacted by the BINARY(S390) option. These are always stored in the native format of the platform.

Visual Builder: Visual Builder applications require BINARY(NATIVE), which is the default specification in the GUI compile options notebook. Do not change this default setting.

Object-oriented programs: Do not specify BINARY(S370) or BINARY(S390) in object-oriented programs.

CALLINT Compiler Option

CALLINT

```
►►──CALLINT(──┬─SYSTEM───┬──┬─────────────────────┬─)──────────────►◄
              ├─OPTLINK──┤  └─,──┬─DESC─────────┬─┘
              ├─FAR16────┤       ├─DESCRIPTOR───┤
              ├─PASCAL16─┤       ├─NODESC───────┤
              └─CDECL────┘       └─NODESCRIPTOR─┘
```

Default is: CALLINT(SYSTEM,NODESC)

Abbreviations are: None

Use CALLINT to indicate the call interface convention applicable to CALLS.

This option may be overridden for specific call statements via the compiler directive >>CALLINT (see “Compiler-Directing Statements” on page 202)

See “ENTRYINT” on page 171 for the compiler option, ENTRYINT. ENTRYINT is used for the selection of the call interface convention for the program entry point or points.

- Selecting a call interface convention:

  SYSTEM
  The SYSTEM suboption specifies that the call convention is that of the standard system linkage convention of the platform.

  Windows On Windows, this is STDCALL, the linkage used by the system Windows APIs.

  Alert: This convention cannot be used in all cases when the called program has multiple entry points. See “Multiple Entry Points on Windows” on page 397 for details. Windows

  OPTLINK
  The OPTLINK suboption specifies that the call convention is that of the _OPTLINK convention of VisualAge for C++ for OS/2 and VisualAge for C++ for Windows.

  FAR16
  The FAR16 suboption specifies that the call convention is that of the _FAR16_Cdecl convention.

  PASCAL16
  The PASCAL16 suboption specifies that the call convention is that of the _FAR16_Pascal convention.

  CDECL
  Windows The CDECL suboption specifies that the call interface convention is that of the CDECL calling convention as defined by Microsoft Visual C++ for Windows. Windows

- Specifying if the argument descriptors are to be generated or not:

  DESC
  The DESC suboption specifies that an argument descriptor is passed for each argument on a CALL statement. See Figure 49 on page 203 for information on the passing of descriptors.

  Note: Do not specify the DESC suboption in object-oriented programs.

  DESCRIPTOR
  The DESCRIPTOR suboption is synonymous with the DESC suboption.

  NODESC
  The NODESC suboption specifies that no argument descriptors are passed for any arguments on a CALL statement.

  NODESCRIPTOR
  The NODESCRIPTOR suboption is synonymous with the NODESC suboption.

Visual Builder: Visual Builder applications require CALLINT(SYSTEM,NODESCRIPTOR), which is the default specification in the GUI compile options notebook. Do not change this default setting.

CHAR

```
►►──CHAR(──┬─NATIVE─┬─)────────────────────────────────────────────────►◄
           ├─EBCDIC─┤
           └─S390───┘
```

Default is: CHAR(NATIVE)

Abbreviations are: None

Specify CHAR(NATIVE) to use the native character representation format of the platform. For VisualAge COBOL, this is ASCII.

CHAR(EBCDIC) and CHAR(S390) are synonymous and indicate that DISPLAY data items are in the data representation of System/390 (EBCDIC).

The following are affected by the CHAR(EBCDIC) compiler option:

- **USAGE DISPLAY items**
  - Single byte characters with USAGE DISPLAY, and double byte characters with USAGE DISPLAY-1, are treated as EBCDIC:
    - ASCII data is converted to EBCDIC on ACCEPT from the terminal.
    - EBCDIC data is converted to ASCII on DISPLAY to the terminal.
    - The EBCDIC equivalent of an ASCII literal is used for assignment to EBCDIC character data. See Figure 43 on page 166 for the rules on the compares of character data with the CHAR(EBCDIC) option in effect.

- Editing is also done with EBCDIC characters.
- Any padding is done using EBCDIC spaces. This includes alphanumeric operations (For example, assignments and compares) on group items regardless of the definition of the elementary items in the group items.
- Figurative constant SPACE/SPACES used in a VALUE clause for, an assignment to, or in a relational condition with a DISPLAY item is treated as single byte EBCDIC spaces (that is, `X'40'`).
- CLASS tests are performed based on EBCDIC value ranges.
- The program name in CALL *identifier*, CANCEL *identifier*, or in the Format 6 SET statement is converted to ASCII characters if the identifier is EBCDIC.
- The file name in the data name in ASSIGN USING *data-name* is converted to ASCII characters if the data name is EBCDIC.
- The file name in SORT-CONTROL is converted to ASCII characters before being passed to the Sort/Merge function.

  Note that the SORT-CONTROL special register has the implicit USAGE DISPLAY definition.

– Zoned decimal data (numeric picture with USAGE DISPLAY) and external floating point data. For example, zoned decimal PIC S9 value “1” is treated as `X'C1'` instead of `X'31'`.

- **Group items**

  Group items are treated similar to USAGE DISPLAY items. Note that any USAGE clause on a group item applies to the elementary items within the group and not to the group itself.

Hexadecimal literals are assumed to represent EBCDIC characters if the literals are assigned to, or compared with, character data. For example, `X'C1'` will compare equal to an alphanumeric item with the value “A.”

Figurative constants, HIGH-VALUE, LOW-VALUE, SPACE/SPACES, ZERO/ZEROS, and QUOTE/QUOTES are treated logically as their EBCDIC character representations for assignments and/or comparisons with EBCDIC characters.

In comparisons between non-numeric DISPLAY items, the collating sequence is the ordinal sequence of the characters based on their binary (hexadecimal) values (as modified by the alternate collating sequence for the single byte characters, if specified). The collating sequence for EBCDIC characters is not affected by the locale setting or the COLLSEQ compiler option.

Figure 43 summarizes the conversion and the collating sequence applicable based on the types of data (ASCII, EBCDIC) and the COLLSEQ option in effect when PROGRAM COLLATING SEQUENCE is not specified. If it is specified, the source specification has precedence over the compiler option specification.

Figure 43. Collating Sequence Summary

Comparands	COLLSEQ(BIN)	COLLSEQ(NATIVE)	COLLSEQ(EBCDIC)
Both ASCII	No conversion is performed. The comparison is based on the binary value (ASCII).	No conversion is performed. The comparison is based on the current locale.	Both comparands are converted to EBCDIC. The comparison is based on the binary value (EBCDIC).
Mixed ASCII and EBCDIC	The EBCDIC comparand is converted to ASCII. The comparison is based on the binary value (ASCII).	The EBCDIC comparand is converted to ASCII. The comparison is based on the current locale.	The ASCII comparand is converted to EBCDIC. The comparison is based on the binary value (EBCDIC).
Both EBCDIC	No conversion is performed. The comparison is based on the binary value (EBCDIC).	The comparands are converted to ASCII. The comparison is based on the current locale.	No conversion is performed. The comparison is based on the binary value (EBCDIC).

Visual Builder: Visual Builder applications require CHAR(NATIVE), which is the default specification in the GUI compile options notebook. Do not change this default setting.

Object-oriented programs: Do not specify CHAR(EBCDIC) in object-oriented programs.

For additional information about the CHAR compiler option, see Appendix B, “System/390 Host Data Type Considerations” on page 543.

COLLSEQ

```
►►──COLLSEQ(──┬─NATIVE─┬──)──────────────────────────────────────────►◄
              ├─EBCDIC─┤
              └─BIN────┘
```

Default is: COLLSEQ(BIN)

Abbreviations are: None

Specify COLLSEQ(EBCDIC) to use the EBCDIC collating sequence rather than the ASCII collating sequence.

Specify COLLSEQ(BIN) to use the hex values of the characters; the locale setting has no effect. This setting will give better execution-time performance.

If you use the PROGRAM-COLLATING-SEQUENCE clause in your source with an alphabet-name of STANDARD-1, STANDARD-2, or EBCDIC, the COLLSEQ option will be ignored. If you specify PROGRAM COLLATING SEQUENCE IS NATIVE, the value of NATIVE is taken from the COLLSEQ option.

Otherwise, when the alphabet-name specified on the PROGRAM-COLLATING-SEQUENCE clause is defined with literals, the collating sequence used is that given by the COLLSEQ option, modified by the user-defined sequence given by alphabet-name.

The PROGRAM-COLLATING-SEQUENCE clause has no effect on DBCS data.

Visual Builder: Visual Builder applications require COLLSEQ(NATIVE), which is the default specification in the GUI compile options notebook. Do not change this default setting.

COMPILE

```
►►──┬─COMPILE──────────────┬──────────────────────────────────────►◄
    ├─NOCOMPILE────────────┤
    └─NOCOMPILE(──┬─W─┬──)─┘
                  ├─E─┤
                  └─S─┘
```

Default is: NOCOMPILE(S)

Abbreviations are: C|NOC

Use the COMPILE option only if you want to force full compilation even in the presence of serious errors. All diagnostics and object code will be generated. Do not try to run the object code generated if the compilation resulted in serious errors—the results could be unpredictable or an abnormal termination could occur.

Use NOCOMPILE without any suboption to request a syntax check (only diagnostics produced, no object code).

Use NOCOMPILE with W, E, or S for conditional full compilation. For meanings of error codes, see "Compiler-Detected Errors and Messages" on page 148. Full compilation (diagnosis and object code) will stop when the compiler finds an error of the level you specify (or higher), and only syntax checking will continue.

If you request an unconditional NOCOMPILE, the following options have no effect because no object code will be produced:

LIST
SSRANGE
OPTIMIZE
TEST

CURRENCY

```
►►──┬─CURRENCY(literal)─┬─────────────────────────────────────────►◄
    └─NOCURRENCY────────┘
```

Default is: NOCURRENCY

The default currency symbol is the dollar sign ($). You can use the CURRENCY option to provide an alternate default currency symbol to be used for the COBOL program.

NOCURRENCY specifies that no alternate default currency symbol will be used.

To change the default currency symbol, use the CURRENCY(*literal*) option where *literal* is a valid COBOL non-numeric literal (including a hex literal) representing a one-byte, printable ASCII character that must not be any of the following:

- Digits zero (0) through nine (9)
- Uppercase alphabetic characters A B C D P R S V X Z
- Lowercase alphabetic characters a through z
- The space
- Special characters * + - / , . ; ( ) " =
- A figurative constant
- The uppercase alphabetic character G, if the COBOL program defines an MBCS item with the PICTURE symbol G; the PICTURE clause will be invalid for that MBCS item because the symbol G is considered to be a currency symbol in the PICTURE clause.
- The uppercase alphabetic character N, if the COBOL program defines an MBCS item with the PICTURE symbol N; the PICTURE clause will be invalid for that MBCS item because the symbol N is considered to be a currency symbol in the PICTURE clause.
- The uppercase alphabetic character E, if the COBOL program defines an external floating-point item; the PICTURE clause will be invalid for the external floating-point item because the symbol E is considered to be a currency symbol in the PICTURE clause.

You can use the CURRENCY option as an alternative to the CURRENCY SIGN clause for selecting the currency symbol you will use in the PICTURE clause of a COBOL program.

When both the CURRENCY option and the CURRENCY SIGN clause are used in a program, the symbol coded in the CURRENCY SIGN clause will be considered the currency symbol in a PICTURE clause when that symbol is used.

When the NOCURRENCY option is in effect and you omit the CURRENCY SIGN clause, the dollar sign ($) is used as the PICTURE symbol for the currency sign.

Delimiter Note: The CURRENCY option literal can be delimited by either the quote or the apostrophe, regardless of the QUOTE|APOST compiler setting.

DATEPROC Compiler Option

DATEPROC

```
►►──┬─DATEPROC──┬─────────────────┬─┬─────────────────────────────────►◄
    │           └─(─┬─FLAG───┬─)─┘ │
    │               └─NOFLAG─┘      │
    └─NODATEPROC────────────────────┘
```

Default is: NODATEPROC, or DATEPROC(FLAG) if only DATEPROC is specified

Abbreviations are: DP|NODP

Use the DATEPROC option to enable the millennium language extensions of the COBOL compiler. For information on using these extensions, see Chapter 31, "Using the Millennium Language Extensions" on page 520.

DATEPROC(FLAG)
With DATEPROC(FLAG), the millennium language extensions are enabled, and the compiler will produce a diagnostic message wherever a language element uses or is affected by the extensions. The message will usually be an information-level or warning-level message that identifies statements that involve date-sensitive processing. Additional messages may be generated that identify errors or possible inconsistencies in the date constructs. For information on how to reduce these diagnostic messages, see "Eliminating Warning-Level Messages" on page 534.

Production of diagnostic messges, and their appearance in or after the source listing, is subject to the setting of the FLAG compiler option.

DATEPROC(NOFLAG)
With DATEPROC(NOFLAG), the millennium language extensions are in effect, but the compiler will not produce any related messages unless there are errors or inconsistencies in the COBOL source.

NODATEPROC
NODATEPROC indicates that the extensions are not enabled for this compilation unit. This affects date-related program constructs as follows:

- The DATE FORMAT clause is syntax-checked, but has no effect on the execution of the program.
- The DATEVAL and UNDATE intrinsic functions have no effect. That is, the value returned by the intrinsic function is exactly the same as the value of the argument.
- The YEARWINDOW intrinsic function returns a value of zero.

Notes:

1. Specification of the DATEPROC option requires that the NOCMPR2 option is also used.
2. NODATEPROC conforms to the COBOL 85 Standard.

DYNAM

```
►►──┬─DYNAM───┬──────────────────────────────────────────────►◄
    └─NODYNAM─┘
```

Default is: NODYNAM

Abbreviations are: DYN|NODYN

Use DYNAM to cause non-nested, separately compiled programs invoked through the CALL *literal* statement to be loaded (for CALL) and deleted (for CANCEL) dynamically at run time. CALL *identifier* statements always result in a run-time load of the target program and are not impacted by this option.

The condition for the ON EXCEPTION phrase can occur for a CALL statement using the literal name only when the DYNAM option is in effect.

With NODYNAM, the target program name is resolved through the linker.

With the DYNAM option, this statement

```
    CALL "myprogram" ...
```

has the identical behavior to these statements:

```
    MOVE "myprogram" to id-1
    CALL id-1 ...
```

See “Static, Dynamic, and Run-time Linking” on page 376 for information on subprogram calls.

DYNAM conforms to the COBOL 85 Standard.

ENTRYINT

```
►►──ENTRYINT(──┬─SYSTEM──┬─)─────────────────────────────────►◄
               ├─OPTLINK─┤
               └─CDECL───┘
```

Default is: ENTRYINT(SYSTEM)

Abbreviations are: None

Use ENTRYINT to indicate the call interface convention applicable to the program entry point(s) in the USING phrase of either the PROCEDURE DIVISION or ENTRY statement.

See “CALLINT” on page 164 for the compiler option, CALLINT. CALLINT is used for the selection of the call interface convention for CALLs.

SYSTEM

The SYSTEM suboption specifies that the call convention is that of the standard system linkage convention of the platform.

Windows On Windows, this is STDCALL, the linkage used by the system Windows APIs.

Alert: This convention cannot be used in all cases when the called program has multiple entry points. See "Multiple Entry Points on Windows" on page 397 for details. Windows

OPTLINK

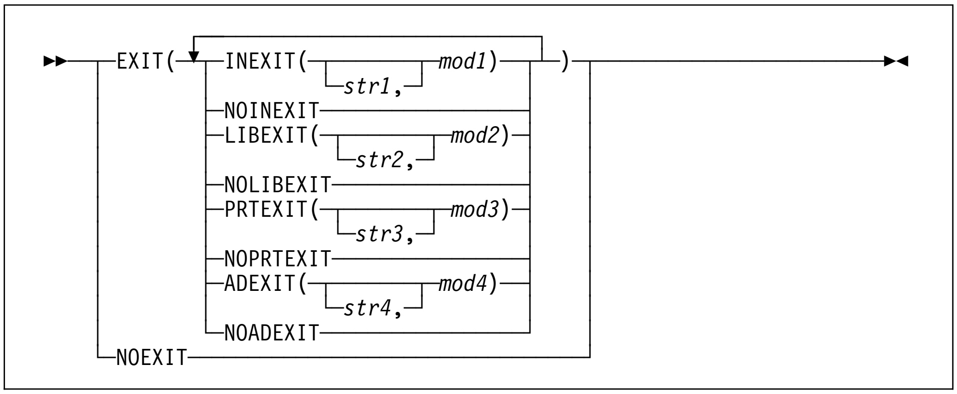
The OPTLINK suboption specifies that the call convention is that of the _OPTLINK convention of VisualAge for C++ for OS/2 and VisualAge for C++ for Windows.

CDECL

Windows The CDECL suboption specifies that the call interface convention is that of the CDECL calling convention as defined by Microsoft Visual C++ for Windows. Windows

Visual Builder: Visual Builder applications require ENTRYINT(SYSTEM), which is the default specification in the GUI compile options notebook. Do not change this default setting.

EXIT



Default is: EXIT(ADEXIT(IWZRMGUX))

Abbreviations are: EX(INX|NOINX,LIBX|NOLIBX,PRTX|NOPRTX,ADX|NOADX)

If you specify the EXIT option without providing at least one suboption, NOEXIT will be in effect. The suboptions can be specified in any order, separated by either commas or spaces. If you specify both the positive and negative form of a suboption (INEXIT|NOINEXT, LIBEXIT|NOLIBEXIT, PRTEXIT|NOPRTEXIT, OR ADEXIT|NOADEXIT), the form specified last takes effect. If you specify the same suboption more than one time, the one specified last takes effect.

EXIT Compiler Option

Use the EXIT option to allow the compiler to accept user-supplied modules in place of SYSIN, SYSLIB (or copy library), and SYSPRINT. When creating your EXIT module, ensure that the module is linked as a DLL module before you run it with the COBOL compiler. EXIT modules are invoked with the system linkage convention of the platform.

For SYSADATA, the ADEXIT suboption provides a module that will be called for each SYSADATA record immediately after the record has been written out to the file.

No PROCESS: The EXIT option cannot be specified in a PROCESS(CBL) statement; it can be specified only via the environment variable COBOPT, via the `cob2` command option, or at installation time.

INEXIT(['str1',]mod1)
The compiler reads source code from a user-supplied load module (where *mod1* is the module name), instead of SYSIN.

LIBEXIT(['str2',]mod2)
The compiler obtains copy code from a user-supplied load module (where *mod2* is the module name), instead of *library-name* or SYSLIB. For use with either COPY or BASIS statements.

PRTEXIT(['str3',]mod3)
The compiler passes printer-destined output to the user-supplied load module (where *mod3* is the module name), instead of SYSPRINT.

ADEXIT(['str4',]mod4)
The compiler passes the SYSADATA output to the user-supplied load module (where *mod4* is the module name).

The module names *mod1*, *mod2*, *mod3*, and *mod4*, can refer to the same module.

The suboptions *'str1'*, *'str2'*, *'str3'*, and *'str4'*, are character strings that are passed to the load module. These strings are optional; if you use them, they can be up to 64 characters in length and must be enclosed in apostrophes. Any character is allowed, but included apostrophes must be doubled, and lowercase characters are folded to uppercase.

Character String Formats

If *'str1'*, *'str2'*, *'str3'*, or *'str4'*, is specified, the string is passed to the appropriate user-exit module with the following format:

LL	string

where **LL** is a halfword (on a halfword boundary) containing the length of the string. See Figure 44 on page 174 for the location of the character string in the parameter list.

User-Exit Work Area

When an exit is used, the compiler provides a user-exit work area that can be used to save the address of storage allocated by the exit module. This allows the module to be reentrant.

The user-exit work area is four fullwords, residing on a fullword boundary, that is initialized to binary zeroes before the first exit routine is invoked. The address of the work area is passed to the exit module in a parameter list. After initialization, the compiler makes no further reference to the work area. So, you will need to establish your own conventions for using the work area if more than one exit is active during the compilation. For example, the INEXIT module uses the first word in the work area, the LIBEXIT module uses the second word, and the PRTEXIT module uses the third word.

Linkage Conventions

Your EXIT modules should use standard linkage conventions between COBOL programs, between library routines, and between COBOL programs and library routines. You need to be aware of these conventions in order to trace the call chain correctly.

Parameter List for Exit Modules

The following table shows the format of the parameter list used by the compiler to communicate with the exit module.

Figure 44 (Page 1 of 2). Parameter List for LIBEXIT

Offset	Contains Address of	Description of Item
00	User-exit type	Halfword identifying which user exit is to perform the operation. 1=INEXIT; 2=LIBEXIT; 3=PRTEXIT; 4=ADEXIT
02	Operation code	Halfword indicating the type of operation. 0=OPEN; 1=CLOSE; 2=GET; 4=FIND
04	Return code	Fullword, placed by the exit module, indicating status of the requested operation. 0=Successful; 4=End-of-data; 12=Failed
08	Data length	Fullword, placed by the exit module, specifying the length of the record being returned by the GET operation.
12	Data	Fullword, placed by the exit module, containing the address of the record in a user-owned buffer, for the GET operation.
	or 'str2'	'str2' applies only to OPEN. The first halfword (on a halfword boundary) contains the length of the string; the string follows.
16	User-exit work area	Four-fullword work area provided by the compiler for use by user-exit module.
32	Text-name	Fullword containing the address of a a null-terminated string containing the fully qualified text-name. Applies only to FIND.

Figure 44 (Page 2 of 2). Parameter List for LIBEXIT

Offset	Contains Address of	Description of Item
36	User exit parameter string	Fullword containing the address of a four-element array, each element of which is a structure that contain a two-byte length field followed by a 64 characters string that contain the exit parameter string.

Note: Only the second element of the parameter string array is used for LIBEXIT, to store the length of the LIBEXIT parameter string followed by the parameter string.

Using INEXIT

When INEXIT is specified, the compiler loads the exit module (*mod1*) during initialization, and invokes the module using the OPEN operation code (op code). This allows the module to prepare its source for processing and then pass the status of the OPEN request back to the compiler. Subsequently, each time the compiler requires a source statement, the exit module is invoked with the GET op code. The exit module then returns either the address and length of the next statement or the end-of-data indication (if no more source statements exist). When end-of-data is presented, the compiler invokes the exit module with the CLOSE op code so that the module can release any resources that are related to its input.

The compiler uses a parameter list to communicate with the exit module. The parameter list consists of 10 fullwords. The return code, data length, and data parameters are placed by the exit module for return to the compiler; and the other items are passed from the compiler to the exit module.

Figure 44 on page 174 shows the contents of the parameter list and a description of each item.

Using LIBEXIT

When LIBEXIT is specified, the compiler loads the exit module (*mod2*) during initialization. Calls are made to the module by the compiler to obtain copy text whenever COPY or BASIS statements are encountered.

Use LIB: If LIBEXIT is specified, the LIB compiler option must be in effect.

The first call invokes the module with an OPEN op code. This allows the module to prepare the specified library-name for processing. The OPEN op code is also issued the first time a new library-name is specified. The exit module returns the status of the OPEN request to the compiler by passing a return code.

Once the exit invoked with the OPEN op code returns, the exit module is then invoked with a FIND op code. The exit module establishes positioning at the requested text-name (or basis-name) in the specified library-name. This becomes the “active copy source”. When positioning is complete, the exit module passes an appropriate return code to the compiler.

The compiler then invokes the exit module with a GET op code, and the exit module passes the compiler the length and address of the record to be copied from the active copy source. The GET operation is repeated until the end-of-data indicator is passed to the compiler.

When end-of-data is presented, the compiler will issue a CLOSE request so that the exit module can release any resources related to its input.

Nested COPY Statements: Any record from the active copy source can contain a COPY statement. (However, nested COPY statements cannot contain the REPLACING phrase, and a COPY statement with the REPLACING phrase cannot contain nested copy statements.) When a valid nested COPY statement is encountered, the compiler issues a request based on the following:

- If the requested library-name from the nested COPY statement was not previously opened, the compiler invokes the exit module with an OPEN op code, followed by a FIND for the new text-name.
- If the requested library-name is already open, the compiler issues the FIND op code for the new requested text-name (an OPEN is not issued here).

The compiler does not allow recursive calls to text-name. That is, a COPY member can be named only once in a set of nested COPY statements until the end-of-data for that copy member is reached.

When the exit module receives the OPEN or FIND request, it should push its control information concerning the active copy source onto a stack and then complete the requested action (OPEN or FIND). The newly requested text-name (or basis-name) now becomes the active copy source.

Processing continues in the normal manner with a series of GET requests until the end-of-data indicator is passed to the compiler.

At end-of-data for the nested active copy source, the exit module should pop its control information from the stack. The next request from the compiler will be a FIND, so that the exit module can reestablish positioning at the previous active copy source.

The compiler now invokes the exit module with a GET request, and the exit module must pass the same record that was passed previously from this copy source. The compiler verifies that the same record was passed, and then the processing continues with GET requests until the end-of-data indicator is passed.

Figure 44 on page 174 shows the contents of the parameter list used for LIBEXIT and a description of each item.

Using PRTEXIT

When PRTEXIT is specified, the compiler loads the exit module (`mod3`) during initialization. The exit module is used in place of the SYSPRINT data set.

The compiler invokes the module using the OPEN operation code (op code). This allows the module to prepare its output destination for processing and then pass the status of the OPEN request back to the compiler. Subsequently, each time the compiler has a line to be printed, the exit module is invoked with the PUT op code. The compiler supplies the address and length of the record that is to be printed, and the exit module returns the status of the PUT request to the compiler by a return code. The first byte of the record to be printed contains an ANSI printer control character.

Before the compilation is ended, the compiler invokes the exit module with the CLOSE op code so that the module can release any resources that are related to its output destination.

Figure 44 on page 174 shows the contents of the parameter list used for PRTEXIT and a description of each item.

Using ADEXIT

When ADEXIT is specified, the compiler loads the exit module (`mod4`) during initialization. The exit module is called for each record written to the SYSADATA data set.

The compiler invokes the module using the OPEN operation code (op code). This allows the module to prepare for processing and then pass the status of the OPEN request back to the compiler. Subsequently, each time the compiler has written a SYSADATA record, the exit module is invoked with the PUT op code. The compiler supplies the address and length of the SYSADATA record, and the exit module returns the status of the PUT request to the compiler by a return code.

Before the compilation is ended, the compiler invokes the exit module with the CLOSE op code so that the module can release any resources.

Figure 44 on page 174 shows the contents of the parameter list used for ADEXIT and a description of each item.

FLAG

```
►►──┬─FLAG(x──┬────┬─)─┬──────────────────────────────────────►◄
    │         └─,y─┘   │
    └─NOFLAG───────────┘
```

Default is: FLAG(I)

Abbreviations are: F|NOF

x and *y* can be either I, W, E, S, or U. (See “Compiler Error Message Codes” on page 149 for descriptions of error codes.)

Use FLAG(*x*) to produce diagnostic messages for errors of a severity level *x* or above at the end of the source listing.

Use FLAG(*x*,*y*) to produce diagnostic messages for errors of severity level *x* or above at the end of the source listing, with error messages of severity *y* and above to be embedded directly in the source listing. The severity coded for *y* must not be lower than the severity coded for *x*. To use FLAG(*x*,*y*), you must also specify the SOURCE compiler option.

Error messages in the source listing are set off by embedding the statement number in an arrow that points to the message code. The message code is then followed by the message text. For example:

```
  000413     MOVE CORR WS-DATE TO HEADER-DATE

==000413==>    IGYPS2121-S       " WS-DATE " was not defined as a data-name.  ...
```

With FLAG*(x,y)* selected, messages of severity *y* and above will be embedded in the listing following the line that caused the message. (Refer to the notes below for exceptions.)

Use NOFLAG to suppress error flagging. NOFLAG will not suppress error messages for compiler options.

Embedded Messages:

1. Specifying embedded level-U messages is accepted, but will not produce any messages in the source. Embedding a level-U message is not recommended.
2. The FLAG option does not affect diagnostic messages produced before the compiler options are processed.
3. Diagnostic messages produced during processing of compiler options, CBL and PROCESS statements, or BASIS, COPY, and REPLACE statements, are never embedded in the source listing. All such messages appear at the beginning of the compiler output.
4. Messages produced during processing of the *CONTROL (*CBL) statement are not embedded in the source listing.

FLAGSTD

```
►►──┬─FLAGSTD(x──┬────┬──┬────┬──)─┬──────────────────────────────►◄
    │            └─yy─┘  └─,O─┘    │
    └─NOFLAGSTD────────────────────┘
```

Default is: NOFLAGSTD

x specifies the level or subset of COBOL 85 Standard to be regarded as conforming:

M Language elements that are *not* from the minimum subset are to be flagged as “nonconforming standard”.

I Language elements that are *not* from the minimum or the intermediate subset are to be flagged as “nonconforming standard”.

H The high subset is being used and elements will not be flagged by subset. And, elements in the IBM extension category will be flagged as “non-conforming Standard, IBM extension”.

yy specifies, by a single character or combination of any two, the optional modules to be included in the subset:

D Elements from Debug module level 1 are *not* flagged as “non-conforming standard”.

N Elements from Segmentation module level 1 are *not* flagged as “non-conforming standard”.

S Elements from Segmentation module level 2 are *not* flagged as “non-conforming standard”.

If S is specified, N is included (N is a subset of S).

O specifies that obsolete language elements are flagged as “obsolete”.

Use FLAGSTD to get informational messages about the COBOL 85 Standard elements included in your program. You can specify any of the following items for flagging:

- A selected Federal Information Processing Standard (FIPS) COBOL subset
- Any of the optional modules
- Obsolete language elements
- Any combination of subset and optional modules
- Any combination of subset and obsolete elements
- IBM extensions (these are flagged any time FLAGSTD is specified and are identified as “non-conforming non-standard”)

  This includes the new language syntax for object-oriented COBOL and for improved interoperability, the PGMNAME(MIXED) compiler option, and the Millennium Language Extensions.

The informational messages appear in the source program listing and contain the following information:

- Identify the element as “obsolete”, “non-conforming standard”, or “non-conforming non-standard” (a language element that is both obsolete and non-conforming is flagged as obsolete only).
- Identify the clause, statement, or header that contains the element.
- Identify the source program line and beginning location of the clause, statement, or header that contains the element.
- Identify the subset or optional module to which the element belongs.

FLAGSTD requires the standard set of reserved words.

In the following example, the line number and column where a flagged clause, statement, or header occurred are shown, as well as the message code and text. At the bottom is a summary of the total of the flagged items and their type.

```
  LINE.COL CODE       FIPS MESSAGE TEXT

           IGYDS8211  Comment lines before "IDENTIFICATION DIVISION":
                      nonconforming nonstandard, IBM extension to
                      ANS/ISO 1985.

     11.14 IGYDS8111  "GLOBAL clause":  nonconforming standard, ANS/ISO
                      1985 high subset.

     59.12 IGYPS8169  "USE FOR DEBUGGING statement":  obsolete element
                      in ANS/ISO 1985.


FIPS MESSAGES TOTAL             STANDARD      NONSTANDARD      OBSOLETE

                 3                   1              1              1
```

FLOAT

```
 ►►──FLOAT(──┬─NATIVE─┬─)──────────────────────────────────────────►◄
             ├─HEX────┤
             └─S390───┘
```

Default is: FLOAT(NATIVE)

Abbreviations are: None

Specify FLOAT(NATIVE) to use the native floating point data representation format of the platform. For VisualAge COBOL, this is the IEEE format.

FLOAT(HEX) and FLOAT(S390) are synonymous and indicate that COMP-1 and COMP-2 data items are represented consistently with System/390 (that is, in the hex floating point format):

- Hex floating point values are converted to IEEE format prior to any arithmetic operations (computations or comparisons).
- IEEE floating point values are converted to hex format prior to being stored in floating point data fields.
- Assignment to a floating point item is done by converting the source floating point data (for example, external floating point) to hex floating point as necessary.

Object-oriented programs: Do not specify FLOAT(S390) in object-oriented programs.

For additional information about the FLOAT compiler option, see Appendix B, “System/390 Host Data Type Considerations” on page 543.

IDLGEN

```
 ►►──┬─IDLGEN───┬──────────────────────────────────────────────────►◄
     └─NOIDLGEN─┘
```

Default is: NOIDLGEN

Abbreviations are: IDL|NOIDL

Use the IDLGEN option to indicate whether SOM Interface Definition Language (IDL) should be generated for COBOL class definitions contained in the COBOL source file.

Use IDLGEN to request that in addition to the normal compile of the COBOL source file, IDL definitions for classes defined in the file are generated.

Use NOIDLGEN to request that no IDL definitions are generated.

The IDL file has the same name as the compiler source file, with the file extension IDL. For example, IDL file generated for `myfile.cbl` would be `myfile.idl`. The IDL file is written to the directory from which `cob2` was run.

When a class definition includes references to other classes (such as on the INHERITS or METACLASS IS phrases, or typed object references as method parameters) that are defined in separate source files, the generated IDL will contain include statements for the IDL files of the referenced classes. The COBOL compiler will attempt to obtain the file name (referred to as the *filestem* in the SOM documentation) for a referenced class from the SOM interface repository (IR). If the referenced class does not have an IR entry, then the external class-name of the referenced class is assumed as the filestem. An include is then generated of the form: `#include <filestem..idl>` This may be adequate for classes where external class-names are the same as the original source file name. However, in many cases this include statement will need to either be updated to reflect the correct filestem or preferably, the entire IDL file should be regenerated after the missing definition has been added to the IR.

When a COBOL source file contains more than one class definition (batch compile) and the IDLGEN option is used, the COBOL class definitions must be sequenced in an appropriate order within the source file. The generated IDL for such a batch compile will contain multiple class interfaces with the IDL interfaces in the same order as the COBOL classes were defined in the COBOL source file. The SOM IDL compiler requires that interfaces be defined before they are referenced, so if there are references between the classes in the COBOL batch compile, the referenced classes must precede the referencing classes in the COBOL source file.

The mapping of COBOL to IDL is designed to balance two (conflicting) objectives, namely enablement of object-oriented COBOL type checking and enabling COBOL classes to operate with other SOM-based programming languages. At a high level:

- COBOL classes map to IDL interfaces.
- COBOL methods map to IDL operation declarations.

- Where possible, the data types of COBOL method parameters are mapped to corresponding native IDL types. These cases include binary integer, floating point, pointer, object reference, and character types.

  All elementary USAGE DISPLAY types and fixed-length COBOL groups are mapped to IDL as array of character.

  Remaining COBOL types that do not naturally map to any native IDL data type are mapped to COBOL-specific “foreign” IDL types. These cases include packed-decimal, scaled binary, DBCS, and variable-length groups.
- Method formal-parameters that specify BY REFERENCE on the method PROCEDURE DIVISION header are given the IDL parameter attribute *inout* and parameters that specify BY VALUE are given the IDL parameter attribute *in*.

The IDL generated for the same COBOL class by the IBM COBOL compiler on OS/390, OS/2, Windows, and AIX might differ; hence, the IDL should be regenerated for the target platform rather than ported between platforms. For example, the procedure-pointer data type in COBOL for OS/390 & VM is an 8-byte data item that does not map to any native IDL type, hence a COBOL specific mapping is used. On OS/2, Windows, and AIX, procedure-pointers are 4-byte data items that map to IDL pointers. Another example is that on OS/390 or AIX, a PIC S9(8) BINARY data item maps naturally to an IDL “long” type, while on OS/2 and Windows, the same data item may map either to an IDL *long* or to a COBOL-specific data type that emulates System/390 binary format, depending on the compilation options used.

No PROCESS: The IDLGEN options cannot be specified on the PROCESS(CBL) statement.

See Chapter 14, “Writing Object-Oriented Programs” on page 270 and Chapter 16, “Using SOM IDL-Based Class Libraries” on page 323 for more information on IDL and SOM.

LIB

```
►►──┬─LIB───┬──────────────────────────────────────────────────►◄
    └─NOLIB─┘
```

Default is: LIB

Abbreviations are: None

If your program uses COPY, BASIS, or REPLACE statements, the LIB compiler option must be in effect.

For more information, see the discussion of the *library-name* user-defined variable on page 136.

In addition, for COPY and BASIS statements, you need to define the library or libraries from which the compiler can take the copied code:

- If the library-name is specified with a user-defined word (not a literal), you must set the corresponding environment variable to point to the desired directory/path for the copy file.
- If the library-name is omitted for a COPY statement, the path to be searched can be specified via the `-Ixxx` option on the `cob2` command.
- If the library-name is specified with a literal, the literal value is treated as the actual path name.

Visual Builder: Visual Builder applications require LIB, which is the default specification in the GUI compile options notebook. Do not change this default setting.

LIB conforms to the COBOL 85 Standard.

LINECOUNT

```
►►──LINECOUNT(nnn)──────────────────────────────────────────►◄
```

Default is: LINECOUNT(60)

Abbreviations are: LC

nnn must be an integer between 10 and 255, or 0.

Use LINECOUNT(*nnn*) to specify the number of lines to be printed on each page of the compilation listing, or use LINECOUNT(0) to suppress pagination.

If you specify LINECOUNT(0), no page ejects are generated in the compilation listing.

The compiler uses three lines of *nnn* for titles. For example, if you specify LINECOUNT(60), 57 lines of source code are printed on each page of the output listing.

LIST

```
►►──┬─LIST───┬──────────────────────────────────────────────►◄
    └─NOLIST─┘
```

Default is: NOLIST

Abbreviations are: None

Use LIST to produce a listing of the assembler-language expansion of your source code.

You will also get these in your output listing:

- Global tables
- Literal pools
- Information about Working-Storage

- Size of the program's Working-Storage

If you want to limit the assembler listing output, use *CONTROL LIST or NOLIST statements in your PROCEDURE DIVISION. Your source statements following a *CONTROL NOLIST are not included in the listing until a *CONTROL LIST statement switches the output back to normal LIST format. For a description of the *CONTROL (*CBL) statement, see *IBM COBOL Language Reference*.

Batch Compiles: The number of and names of the resulting .asm files depend on the SEPOBJ option:

SEPOBJ The file for the first program in the source file has the name of the source file. The files for all subsequent programs in the source file have the names of the corresponding PROGRAM IDs.

NOSEPOBJ The one file for all programs in the source file has the name of the source file.

For information on using LIST output, see “Data Map Listing” on page 258.

MAP

```
►►──┬─MAP───┬──────────────────────────────────────────────────►◄
    └─NOMAP─┘
```

Default is: NOMAP

Abbreviations are: None

Use MAP to produce a listing of the items you defined in the DATA DIVISION. Map output includes:

- DATA DIVISION map
- Global tables
- Literal pools
- Nested program structure map, and program attributes
- Size of the program's Working-Storage

If you want to limit the MAP output, use *CONTROL MAP or NOMAP statements in the PROCEDURE DIVISION. Source statements following a *CONTROL NOMAP are not included in the listing until a *CONTROL MAP statement switches the output back to normal MAP format. For a description of the *CONTROL (*CBL) statement, see the *IBM COBOL Language Reference*.

For information on using LIST output, see “Data Map Listing” on page 258.

By selecting the MAP option, you can also print an embedded MAP report in the source code listing. The condensed MAP information is printed to the right of data-name definitions in the FILE SECTION, WORKING-STORAGE SECTION, and LINKAGE SECTION of the DATA DIVISION.

NUMBER

```
►►──┬─NUMBER───┬──────────────────────────────────────────────►◄
    └─NONUMBER─┘
```

Default is: NONUMBER

Abbreviations are: NUM|NONUM

Use NUMBER if you have line numbers in your source code and want those numbers to be used in error messages and MAP, LIST, and XREF listings.

If you request NUMBER, columns 1 through 6 are checked to make sure that they contain only numbers, and the sequence is checked according to numeric collating sequence. (In contrast, SEQUENCE checks them according to ASCII collating sequence.) When a line number is found to be out of sequence, the compiler assigns to it a line number with a value one number higher than the line number of the preceding statement. Sequence-checking continues with the next statement, based on the newly assigned value of the previous line.

If you use COPY statements and NUMBER is in effect, be sure that your source program line numbers and the COPY member line numbers are coordinated.

Use NONUMBER if you do not have line numbers in your source code, or if you want the compiler to ignore the line numbers you do have in your source code. With NONUMBER in effect, the compiler generates line numbers for your source statements and uses those numbers as references in listings.

NONUMBER conforms to the COBOL 85 Standard.

OPTIMIZE

```
►►──┬─OPTIMIZE─┬────────────────┬─┬───────────────────────────►◄
    │          └─(──┬─STD──┬─)──┘ │
    │               └─FULL─┘      │
    └─NOOPTIMIZE──────────────────┘
```

Default is: NOOPTIMIZE

Abbreviations are: OPT|NOOPT

Use OPTIMIZE to reduce the run time of your object program; optimization might also reduce the amount of storage your object program uses. Optimizations performed include the propagation of constants and the elimination of computations whose results are never used. Because OPTIMIZE increases compile time, and can change the order of statements in your program, it should not be used when debugging.

If OPTIMIZE is specified without any suboptions, OPTIMIZE(STD) will be in effect.

The FULL suboption requests that in addition to the optimizations performed under OPT(STD), that the compiler discard unreferenced data items from the DATA DIVISION, and suppress generation of code to initialize these data items to their VALUE clauses. If the OPT(FULL) and MAP options are specified, then a BL number of XXXX in the data map information indicates that the data item was discarded.

Recomendation: Use OPTIMIZE(FULL) for database applications; it can make a huge performace improvement, because unused constants included by the associated COPY statements will be eliminated.

However:

Do not use OPT(FULL) if your programs depend on making use of unused data items. Two common ways this has been done in the past are:

1. A technique sometimes used in OS/VS COBOL programs is to place an unreferenced table after a referenced table and use out-of-range subscripts on the first table to access the second table. To see if your programs have this problem, use the SSRANGE compiler option with the CHECK(ON) run-time option. To work around this problem, use the ability of COBOL to code large tables and use just one table.
2. The second technique utilizing unused data items is to place eyecatcher data items in the WORKING-STORAGE section to identify the beginning and end of the program data, or to mark a copy of a program for a library tool that uses the data to identify a version of a program. To solve this problem, initialize these items with PROCEDURE DIVISION statements rather than VALUE clauses. With this method, the compiler will consider these items as used, and will not delete them.

The OPTIMIZE option is turned off in the case of a severe-level error or higher. The OPTIMIZE and TEST options are mutually exclusive; if you use both, OPTIMIZE will be ignored.

PGMNAME

```
►►──PGMNAME(──┬─MIXED─┬──)──────────────────────────────────────►◄
              └─UPPER─┘
```

Default is: PGMNAME(UPPER), or
PGMNAME(MIXED) for Visual Builder GUI applications

Abbreviations are: PGMN(LU|LM)

For compatibility with IBM COBOL for OS/390 & VM, LONGMIXED and LONGUPPER are also supported.

LONGUPPER can be abbreviated as UPPER, LU, or U. LONGMIXED can be abbreviated as MIXED, LM, or M.

COMPAT: If you specify PGMNAME(COMPAT), PGMNAME(UPPER) will be set, and you will receive a warning message.

PGMNAME Compiler Option

The PGMNAME option controls the handling of names used in the following contexts:

- Program names defined in the PROGRAM-ID paragraph.
- Program entry point names on the ENTRY statement.
- Program name references in:
  - CALL statements
  - CANCEL statements
  - SET *procedure-pointer* TO ENTRY statements

PGMNAME(UPPER)

With PGMNAME(UPPER), program names that are specified in the PROGRAM-ID paragraph as COBOL user-defined words must follow the normal COBOL rules for forming a user-defined word:

- The program name can be up to 30 characters in length.
- All the characters used in the name must be alphabetic, digits, or the hyphen.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last character.

When a program or method name is specified as a literal, in either a definition or a reference, then:

- The program name can be up to 160 characters in length.
- All the characters used in the name must be alphabetic, digits, or the hyphen.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last character.

External program names are processed with alphabetic characters folded to upper case.

PGMNAME(MIXED)

With PGMNAME(MIXED), program names are processed as is, without truncation, translation, or folding to upper case.

With PGMNAME(MIXED), all program name definitions must be specified using the literal format of the program name in the PROGRAM-ID paragraph or ENTRY statement.

| **Visual Builder:** Visual Builder applications require PGMNAME(MIXED), which is the
| default specification in the GUI compile options notebook. Do not change this default
| setting.

PROBE

```
►►──┬─PROBE───┬──────────────────────────────────────────────────►◄
    └─NOPROBE─┘
```

Default is: PROBE

Abbreviations are: None

PROBE requests the generation of stack probes. This extra code causes a protection exception if there is not enough storage available on the stack.

Use PROBE if the program might be executed in a multithreading environment. For information about multithreading, see Chapter 26, “Preparing COBOL Programs for Multithreading” on page 467.

NOPROBE produces more efficient code and is appropriate for non-threading environments.

PROFILE

```
►►──┬─PROFILE───┬────────────────────────────────────────────────►◄
    └─NOPROFILE─┘
```

Default is: PROFILE

Abbreviations are: None

PROFILE instructs the compiler to generate the profile hooks that allow the Performance Analyzer to monitor application execution and generate a trace file. This option should be used with the `-p` option of the `cob2` command (see “Compiling and Linking Programs” on page 142 for details.)

QUOTE/APOST

```
►►──┬─QUOTE─┬────────────────────────────────────────────────────►◄
    └─APOST─┘
```

Default is: QUOTE

Abbreviations are: Q|APOST

Use QUOTE if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more quotation mark (") characters. QUOTE conforms to the COBOL 85 Standard.

Use APOST if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more apostrophe (') characters.

Delimiters: Either quotes or apostrophes can be used as literal delimiters, regardless of whether the APOST or QUOTE option is in effect. The delimiter character used as the opening delimiter for a literal must be used as the closing delimiter for that literal.

SEPOBJ

```
►►──┬─SEPOBJ───┬──────────────────────────────────────────────────►◄
    └─NOSEPOBJ─┘
```

Default is: SEPOBJ, or
NOSEPOBJ for Visual Builder GUI applications

Abbreviations are: None

The option specifies whether or not each of the outermost COBOL programs in a batch compilation is to be generated as a separate object file rather than a single object file.

Batch Compilation

When multiple outer-most programs (non-nested programs) are compiled with a single invocation of the compiler (*batch* compiled), how many separate files are produced for the object program output of the batch compilation varies on the compiler option SEPOBJ.

Assume that the COBOL source file, `pgm.cbl`, contains three outer-most COBOL programs named `pgm1`, `pgm2`, and `pgm3`. The following figures illustrate whether the object program output is generated as two (with NOSEPOBJ) or three (with SEPOBJ) files.

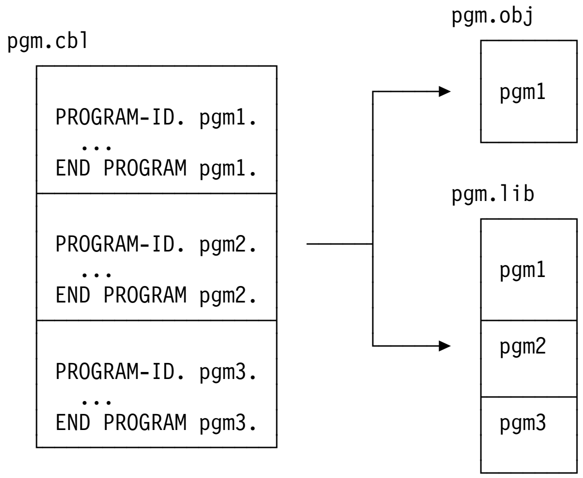


Figure 45. Batch Compilation with NOSEPOBJ

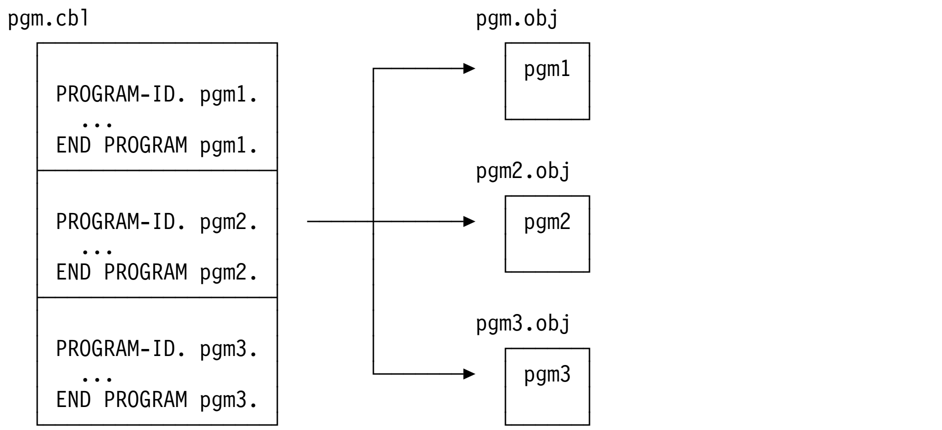


Figure 46. Batch compilation with SEPOBJ

Considerations:

1. The SEPOBJ option is required to conform to the ANSI COBOL standard where `pgm2` or `pgm3` in the above example is called via CALL *identifier* from another program.
2. If the NOSEPOBJ option is in effect, the name(s) of object module file(s) are named with the name of the source file with `.o`, `.OBJ`, and/or `.LIB` extensions. If the SEPOBJ option is in effect, the names of the object files (except for the first one) are based on the PROGRAM-ID name with the `.o` or `.OBJ` extension.
3. The programs called via CALL *identifier* must be referred to by the names of the object files (rather than the PROGRAM ID names) where PROGRAM ID and the object file name do not match.

You are responsible for giving the object file a valid file name for the platform and the file system. For example, if the FAT file system is used for OS/2 or Windows, the length of the PROGRAM ID name must be eight characters or fewer *except* when the object file name(s) are created from the source file name (as in the case with NOSEPOBJ option) as described above.

Visual Builder: Visual Builder applications require NOSEPOBJ, which is the default specification in the GUI compile options notebook. Do not change this default setting.

SEQUENCE

```
►►──┬─SEQUENCE───┬──────────────────────────────────────────────►◄
    └─NOSEQUENCE─┘
```

Default is: SEQUENCE

Abbreviations are: SEQ|NOSEQ

When you use SEQUENCE, the compiler examines columns 1 through 6 of your source statements to check that the statements are arranged in ascending order according to their ASCII collating sequence. The compiler issues a diagnostic message if any statements are not in ascending sequence (source statements with blanks in columns 1 through 6 do not participate in this sequence check and do not result in messages).

If you use COPY statements and SEQUENCE is in effect, be sure that your source program sequence fields and the copy member sequence fields are coordinated.

If you use NUMBER and SEQUENCE, the sequence is checked according to numeric, rather than ASCII, collating sequence.

Use NOSEQUENCE to suppress this checking and the diagnostic messages.

NOSEQUENCE conforms to the COBOL 85 Standard.

SIZE

```
►►──SIZE(──┬─nnnnn─┬──)────────────────────────────────────────────►◄
           └─nnnK──┘
```

Default is: 2097152 bytes (approximately 2 Meg)

Abbreviations are: SZ

nnnnn specifies a decimal number that must be at least 778240.

*nnn*K specifies a decimal number in 1K increments. The minimum acceptable value is 782K.

Use SIZE to indicate the amount of main storage available for compilation (where 1K = 1024 bytes decimal).

SOURCE

```
►►──┬─SOURCE───┬───────────────────────────────────────────────────►◄
    └─NOSOURCE─┘
```

Default is: SOURCE

Abbreviations are: S|NOS

Use SOURCE to get a listing of your source program. This listing will include any statements embedded by PROCESS or COPY statements.

SOURCE must be specified if you want embedded messages in the source listing.

Use NOSOURCE to suppress the source code from the compiler output listing.

If you want to limit the SOURCE output, use *CONTROL SOURCE or NOSOURCE statements in your PROCEDURE DIVISION. Your source statements following a *CONTROL NOSOURCE are not included in the listing at all, unless a *CONTROL SOURCE statement switches the output back to normal SOURCE format. For a description of the *CONTROL (*CBL) statement, see *IBM COBOL Language Reference*.

For information on using LIST output, see "Data Map Listing" on page 258.

SPACE

```
►►──SPACE(──┬─1─┬──)──────────────────────────────────────────►◄
            ├─2─┤
            └─3─┘
```

Default is: SPACE(1)

Abbreviations are: None

Use SPACE to select single, double, or triple spacing in your source code listing.

SPACE has meaning only when the SOURCE compiler option is in effect.

SQL

```
►►──SQL(──┬─"─┬──suboptions for DB2 SQL──┬─"─┬──)─────────────►◄
          └─'─┘                          └─'─┘
```

Default is: SQL("")

Abbreviations are: None

Use this option when you have SQL statements embedded in your COBOL source. It allows you to specify options to be used in handling the SQL statements in your program and is required if the suboption string, which gives SQL options, is to be specified explicitly to DB2.

The syntax shown can be used on either the CBL or PROCESS statements. If the SQL option is given on the `cob2` command, only ' is allowed for the string delimiter: `-q"SQL('options')"`.

See Chapter 21, "Programming for a DB2 Environment" on page 406 for more information.

SSRANGE

```
►►──┬─SSRANGE───┬──────────────────────────────────────────────►◄
    └─NOSSRANGE─┘
```

Default is: NOSSRANGE

Abbreviations are: SSR|NOSSR

Use SSRANGE to generate code that checks if subscripts (including ALL subscripts) or indexes try to reference an area outside the region of the table. Each subscript or index is not individually checked for validity; rather, the effective address is checked to ensure that it does not cause a reference outside the region of the table. Variable-length items will also be checked to ensure that the reference is within their maximum defined length.

Reference modification expressions will be checked to ensure that:

- The reference modification starting position is greater than or equal to 1.
- The reference modification starting position is not greater than the current length of the subject data item.
- The reference modification length value (if specified) is greater than or equal to 1.
- The reference modification starting position and length value (if specified) do not reference an area beyond the end of the subject data item.

If SSRANGE is in effect at compile time, the range-checking code is generated; range checking can be inhibited at run time by specifying CHECK(OFF) as a run-time option. This leaves range-checking code dormant in the object code. The range-checking code can then be optionally used to aid in resolving any unexpected errors without recompilation.

If an out-of-range condition is detected, an error message will be displayed and the program will be terminated.

Remember: You will get range-checking only if you compile your program with the SSRANGE option and run it with the CHECK(ON) run-time option.

TERMINAL

```
►►──┬─TERMINAL───┬─────────────────────────────────────────────►◄
    └─NOTERMINAL─┘
```

Default is: NOTERMINAL

Abbreviations are: TERM|NOTERM

Use TERMINAL to send progress and diagnostic messages to the terminal.

Use NOTERMINAL if this additional output is not desired.

TEST

```
►►──┬─TEST───┬──────────────────────────────────────────────────────►◄
    └─NOTEST─┘
```

Default is: NOTEST

Abbreviations are: None

Use TEST to produce object code that contains symbol and statement information that enables the debugger to perform symbolic source-level debugging.

Use NOTEST if you do not want to generate object code with debugging information.

Programs compiled with NOTEST execute with the debugger, but there is limited debugging support.

The TEST option will be turned off if you use the WITH DEBUGGING MODE clause. The TEST option will appear in the list of options, but a diagnostic message will be issued to advise you that because of the conflict, TEST will not be in effect.

THREAD

```
►►──┬─THREAD───┬────────────────────────────────────────────────────►◄
    └─NOTHREAD─┘
```

Default is: NOTHREAD

Abbreviations are: None

THREAD indicates if the COBOL application is to be enabled for execution in a run unit with multiple threads. All programs within a run unit must be compiled with either the THREAD or NOTHREAD option.

When the THREAD option is in effect, the following language elements are not supported. If encountered, they are diagnosed as errors:

- STOP RUN
- ALTER statement
- DEBUG-ITEM special register
- GO TO statement without procedure-name
- RERUN
- STOP literal statement
- Segmentation module
- USE FOR DEBUGGING statement
- WITH DEBUGGING MODE clause

- INITIAL phrase in PROGRAM-ID paragraph

Caution: RERUN is flagged as an error with THREAD, but is accepted as a comment with NOTHREAD.

Visual Builder: Visual Builder applications require NOTHREAD, which is the default specification in the GUI compile options notebook. Do not change this default setting.

See Chapter 26, “Preparing COBOL Programs for Multithreading” on page 467 for a discussion of COBOL support for multithreading.

TRUNC

```
►►──TRUNC(──┬─STD─┬──)──────────────────────────────────────────────►◄
            ├─OPT─┤
            └─BIN─┘
```

Default is: TRUNC(STD)

Abbreviations are: None

TRUNC(STD) conforms to the COBOL 85 Standard, while TRUNC(OPT) and TRUNC(BIN) are IBM extensions.

TRUNC has no effect on COMP-5 data items; COMP-5 items are handled as if TRUNC(BIN) were in effect, regardless of the TRUNC option specified.

TRUNC(STD)

Use TRUNC(STD) to control the way arithmetic fields are truncated during MOVE and arithmetic operations. TRUNC(STD) applies only to USAGE BINARY receiving fields in MOVE statements and arithmetic expressions. When TRUNC(STD) is in effect, the final result of an arithmetic expression, or the sending field in the MOVE statement, is truncated to the number of digits in the PICTURE clause of the BINARY receiving field.

TRUNC(OPT)

TRUNC(OPT) is a performance option. When TRUNC(OPT) is specified, the compiler assumes that the data conforms to PICTURE and USAGE specifications of the USAGE BINARY receiving fields in MOVE statements and arithmetic expressions. The results are manipulated in the most optimal way, either truncating to the number of digits in the PICTURE clause, or to the size of the binary field in storage (halfword, fullword, or doubleword).

Caution: You should use the TRUNC(OPT) option only if you are sure that the data being moved into the binary areas will not have a value with larger precision than that defined by the PICTURE clause for the binary item. Otherwise, unpredictable results might occur. This truncation is performed in the most efficient manner possible; therefore, the results will be dependent on the particular code sequence generated. It is not possible to predict the truncation without seeing the code sequence generated for a particular statement.

TRUNC(BIN)

The TRUNC(BIN) option applies to all COBOL language that processes USAGE BINARY data. When TRUNC(BIN) is in effect:

- BINARY receiving fields are truncated only at halfword, fullword, or doubleword boundaries.
- BINARY sending fields are handled as halfwords, fullwords, or doublewords when the receiver is numeric; TRUNC(BIN) has no effect when the receiver is not numeric.
- The full binary content of the field is significant.
- DISPLAY will convert the entire content of the binary field, with no truncation.

Recommendation: TRUNC(BIN) is the recommended option for programs that use binary values set by other products. These other products, such as C/C++, FORTRAN, and PL/I, might place values in COBOL binary data items that do not conform to the PICTURE clause of the data item. For CICS considerations, see “Selecting Compiler Options” on page 413.

TRUNC(BIN) should never be used as an install default, only for specific programs, due to the performance cost. A better approach is to use COMP-5 for binary data items passed to non-COBOL programs or other products and subsystems. The use of COMP-5 is not affected by the TRUNC option in effect.

TRUNC Example 1

```
01  BIN-VAR     PIC 99 USAGE BINARY.
.
.
    MOVE 123451 to BIN-VAR
```

Figure 47. Values of the Data Items after the MOVE

	Decimal	**Hex**[1]	**Display**
Sender	123451	3B|E2|01|00	123451
Receiver TRUNC(STD)	51	33|00	51
Receiver TRUNC(OPT)	-7621	3B|E2	2J
Receiver TRUNC(BIN)	-7621	3B|E2	762J

Note: [1] Values are shown using the default BINARY compiler option.

A halfword of storage is allocated for BIN-VAR. The result of this MOVE statement if the program is compiled with the TRUNC(STD) option is 51; the field is truncated to conform to the PICTURE clause.

If the program is compiled with the TRUNC(BIN) option, the result of the MOVE statement is -7621. The reason for the unusual looking answer is that nonzero high-order digits were truncated. Here, the generated code sequence would merely move the lower halfword quantity X'E23B' to the receiver. Because the new truncated value overflowed into the sign bit of the binary halfword, the value becomes a negative number.

This MOVE statement should not be compiled with the TRUNC(OPT) option because 123451 has greater precision than the PICTURE clause for BIN-VAR. If TRUNC(OPT) was used, however, the results again would be -7621. This is because the best performance was gained by not doing a decimal truncation.

Assumption: The preceding example assumes that the BINARY(S390) option is in effect.

TRUNC Example 2

```
01  BIN-VAR     PIC 9(6)  USAGE BINARY
.
.
    MOVE 1234567891 to BIN-VAR
```

Figure 48. Values of the Data Items after the MOVE

	Decimal	**Hex**[1]	**Display**
Sender	1234567891	D3|02|96|49	1234567891
Receiver TRUNC(STD)	567891	53|AA|08|00	567891
Receiver TRUNC(OPT)	567891	00|08|AA|53	567891
Receiver TRUNC(BIN)	1234567891	D3|02|96|49	1234567891

Note: [1] Values are shown using the default BINARY compiler option.

When TRUNC(STD) is specified, the sending data is truncated to six integer digits to conform to the PICTURE clause of the BINARY receiver.

When TRUNC(OPT) is specified, the compiler assumes the sending data is not larger than the PICTURE clause precision of the BINARY receiver. The most efficient code sequence in this case performed truncation as if TRUNC(STD) had been specified.

When TRUNC(BIN) is specified, no truncation occurs because all of the sending data will fit into the binary fullword allocated for BIN-VAR.

Assumption: The preceding example assumes that the BINARY(S390) option is in effect.

TYPECHK

```
►►──┬─TYPECHK───┬──────────────────────────────────────────────►◄
    └─NOTYPECHK─┘
```

Default is: NOTYPECHK

Abbreviations are: TC|NOTC

Use TYPECHK to have the compiler enforce the rules for object-oriented type checking, and generate diagnostics for any violations.

Use NOTYPECHK to turn off the checking for typing violations.

The type conformance requirements are covered in the *IBM COBOL Language Reference* under the appropriate language elements. Type checking requirements include:

- The method being invoked on an INVOKE statement must be supported by the class of the referenced object.
- Method parameters on an INVOKE and the corresponding method PROCEDURE DIVISION USING must conform.
- The SET *object-reference-1* TO *object-reference-2* statement requires that the classes of the objects be of appropriate derivation relationships.
- A method override must have a conforming interface to the corresponding method in the parent class.

When TYPECHK is specified, there must be entries in the SOM Interface Repository (IR) for each class that is referenced in the COBOL source being compiled.

For COBOL classes, these IR entries can be created by using the COBOL IDLGEN option (see “IDLGEN” on page 181) when compiling the class definitions, to create an IDL file that describes the interface of the COBOL class. Compile the IDL using the SOM compiler with its "ir" emitter.

Note that if the COBOL program references classes that are provided by the SOM product itself (such as the SOMObject class), then the pre-generated IR for these classes that is provided as part of the OS/390 SOMobjects product may be used to verify that the COBOL usage conforms to the class interfaces.

See Chapter 15, “Using System Object Model (SOM)” on page 317, *SOMobjects Developer's Toolkit User's Guide*, and *SOMobjects Developer's Toolkit Programmer's Reference Manual* (available online) for further details on interface repositories, SOM, and type checking.

VBREF

```
►►──┬─VBREF───┬──────────────────────────────────────────────►◄
    └─NOVBREF─┘
```

Default is: NOVBREF

Abbreviations are: None

Use VBREF to get a cross-reference among all verb types used in the source program and the line numbers in which they are used. VBREF also produces a summary of how many times each verb was used in the program.

Use NOVBREF for more efficient compilation.

WORD

```
►►──┬─WORD(xxxx)─┬──────────────────────────────────────────────►◄
    └─NOWORD─────┘
```

Default is: NOWORD

Abbreviations are: WD|NOWD

xxxx specifies the ending characters of the name of the reserved-word table (IGYC*xxxx*) to be used in your compilation. IGYC are the first 4 standard characters of the name, and *xxxx* can be 1 to 4 characters in length.

Use WORD(*xxxx*) to specify that an alternate reserved-word table is to be used during compilation.

NOWORD conforms to the COBOL 85 Standard.

WSCLEAR

```
►►──┬─WSCLEAR───┬───────────────────────────────────────────────►◄
    └─NOWSCLEAR─┘
```

Default is: NOWSCLEAR

Abbreviations are: None

Use WSCLEAR to clear your program's working storage to binary zeros when the program is initialized. The storage is cleared before any VALUE clauses are applied.

Use NOWSCLEAR to bypass the storage clearing process.

If you use WSCLEAR and you are concerned about the size or performance of the object program, then you should also use OPTIMIZE(FULL). This instructs the compiler to eliminate all unreferenced data items from the DATA DIVISION, which will speed up the initialization process.

XREF Compiler Option

XREF

```
►►──┬─XREF──┬─────────────────┬─┬──────────────────────────────────────►◄
    │       └─(──┬─SHORT─┬──)─┘ │
    │            └─FULL──┘      │
    └─NOXREF────────────────────┘
```

Default is: NOXREF

Abbreviations are: X|NOX

You can choose XREF, XREF(FULL), or XREF(SHORT).

Use XREF to get a sorted cross-reference listing. Names are listed in the order of the collating sequence indicated by the locale setting. This applies whether the names are in single-byte characters or contain multi-byte characters (such as DBCS).

Also included is a section listing all the program names that are referenced in your program, and the line number where they are defined. External program names are identified as such.

If you use XREF and SOURCE, cross-reference information will also be printed on the same line as the original source in the listing. Line number references or other information, will appear on the right hand side of the listing page. On the right of source lines that reference intrinsic functions, the letters 'IFN' will appear with the line numbers of the location where the function's arguments are defined. Information included in the embedded references lets you know if an identifier is undefined or defined more than once (UND or DUP will be printed); if an item is implicitly defined (IMP), such as special registers or figurative constants; and if a program name is external (EXT).

If you use XREF and NOSOURCE, you'll get only the sorted cross-reference listing.

XREF(SHORT) will print only the explicitly referenced variables in the cross-reference listing. XREF(SHORT) applies to MBCS data names and procedure-names as well as ASCII names.

NOXREF suppresses this listing.

Observe:

1. Group names used in a MOVE CORRESPONDING statement are in the XREF listing. In addition, the elementary names in those groups are also listed.
2. In the data-name XREF listing, line numbers preceded by the letter “M” indicate that the data item is explicitly modified by a statement on that line.
3. XREF listings take additional storage.

See Chapter 13, “Debugging Techniques” on page 244 for sample listings.

YEARWINDOW

```
►►──YEARWINDOW──(base-year)────────────────────────────────────►◄
```

Default is: YEARWINDOW(1900)

Abbreviation is: YW

Use the YEARWINDOW option to specify the first year of the 100-year window (the *century window*) to be applied to windowed date field processing by the COBOL compiler. For information on using windowed date fields, see Chapter 31, “Using the Millennium Language Extensions” on page 520.

base-year represents the first year of the 100-year window, and must be specified as one of the following:

- An unsigned decimal number between 1900 and 1999.

  This specifies the starting year of a fixed window. For example, YEARWINDOW(1930) indicates a century window of 1930-2029.

- A negative integer from -1 through -99.

  This indicates a sliding window, where the first year of the window is calculated from the current run-time date. The number is subtracted from the current year to give the starting year of the century window. For example, YEARWINDOW(-80) indicates that the first year of the century window is 80 years before the current year at the time the program is run.

Notes:

1. The YEARWINDOW option has no effect unless the DATEPROC option is also in effect.
2. At run time, two conditions must be true:
   - The century window must have its beginning year in the 1900s
   - The current year must lie within the century window for the compilation unit

   For example, running a program in 1998 with YEARWINDOW(-99) violates the first condition, and would result in a run-time error.

ZWB

```
►►──┬─ZWB───┬──────────────────────────────────────────────────►◄
    └─NOZWB─┘
```

Default is: ZWB

Abbreviations are: None

With ZWB, the compiler removes the sign from a signed external decimal (DISPLAY) field when comparing this field to an alphanumeric elementary field during execution.

If the external decimal item is a scaled item (contains the symbol 'P' in its PICTURE character-string), its use in comparisons is not affected by ZWB. Such items always have their sign removed before the comparison is made to the alphanumeric field.

ZWB affects how the program runs; the same COBOL source program can give different results, depending on the option setting.

ZWB conforms to the COBOL 85 Standard.

Use NOZWB if you want to test input numeric fields for SPACES.

Compiler-Directing Statements

Several statements help you to direct the compilation of your program. For the definition of these statements, see *IBM COBOL Language Reference*.

BASIS statement
This extended source program library statement provides a complete COBOL program as the source for a compilation.

***CONTROL (*CBL) statement**
This compiler-directing statement selectively suppresses or allows output to be produced. The names *CONTROL and *CBL are synonymous. This statement is described in the *IBM COBOL Language Reference*.

>>**CALLINTERFACE statement**
This compiler directing statement specifies the interface convention for calls, including whether argument descriptors are to be generated. The convention specified using >>CALLINT is in effect until another >>CALLINT specification is made. >>CALLINT can be used only in the PROCEDURE DIVISION.

The syntax and usage of the >>CALLINT statement is similar to the CALLINT compiler option. Exceptions are:

- CALLINT is a valid abbreviation in the statement syntax
- The statement syntax does not include parentheses
- The statement form can be used to apply to selective calls as described below.
- The statement syntax includes the keyword **DESCRIPTOR** and its variants.

See the *IBM COBOL Language Reference* for the full syntax. See “CALLINT” on page 164 for details of the option form.

If you specify >>CALLINT with no suboptions, the call convention used is determined by the CALLINT compiler option. For example, if PROG1 is an IBM C program whose default call interface convention is _OPTLINK, or it is a COBOL program compiled with the ENTRYINT(OPTLINK) option, use the >>CALLINT directive to change the interface for this call only:

```
>>CALLINT OPTLINK DESC
CALL "PROG1" USING PARM1 PARM2.
>>CALLINT
CALL "PROG2" USING PARM1.
```

If you specify >>CALLINT with an invalid argument, the entire directive will be ignored.

The >>CALLINT statement can be specified anywhere that a COBOL procedure statement can be specified. For example, the following is valid COBOL syntax:

```
     MOVE 3 TO
>>CALLINTERFACE SYSTEM
     RETURN-CODE.
```

The effect of >>CALLINT is limited to the current program. A nested program or a program compiled in the same batch inherits the calling convention specified with the CALLINT compiler option, not the >>CALLINT compiler directive.

If you are writing a routine that is to be called with >>CALLINT SYSTEM, DESCRIPTOR, this is the argument-passing mechanism:

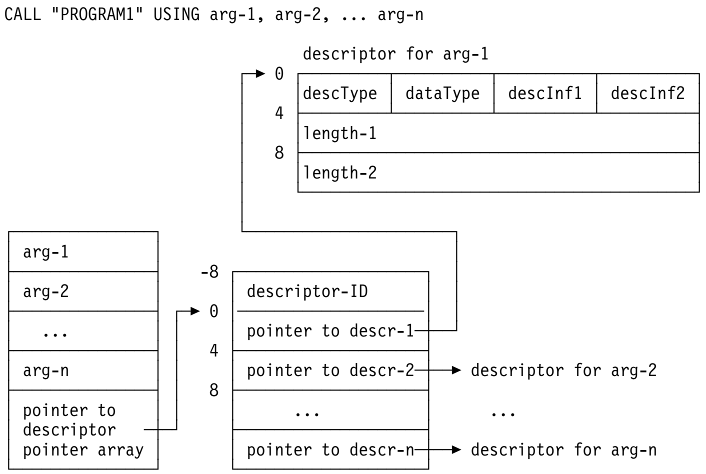


Figure 49. Argument Passing with SYSTEM(DESC)

pointer to descr-n Points to the descriptor for the specific argument; 0 if no descriptor exists for the argument.

descriptor-ID Set to COBDESC0 to identify this version of the descriptor, allowing for a possible change to the descriptor entry format in the future.

descType Set to X'02' (descElmt) for an elementary data item of USAGE DISPLAY with PICTURE X(n) or USAGE DISPLAY-1 with PICTURE G(n) or N(n). For all others (numeric fields, structures, tables), set to X'00'.

dataType
Set as follows:

- descType = X'00': dataType = X'00'
- descType = X'02' and the USAGE is DISPLAY: dataType = X'02' (typeChar)
- descType = X'02' and the USAGE is DISPLAY-1: dataType = X'09' (typeGChar)

descInf1
Always set to X'00'

descInf2
Set as follows:

- If descType = X'00'; descInf2 = X'00'
- If descType = X'02':
  - If the CHAR(EBCDIC) option is in effect and the argument is not defined with the NATIVE option in the USAGE clause: descInf2 = X'40'
  - Else: descInf2 = X'00'

length-1
In the argument descriptor is the length of the argument for a fixed length argument or the current length for a variable length item.

length-2
The maximum length of the argument, if the argument is a variable length item.

For a fixed length argument `length-2` is equal to `length-1`.

COPY statement

```
►►──COPY──┬─text-name─┬──┬──────────────┬──────────────────────────►◄
          └─literal-1─┘  ├─library-name─┤
                         └─literal-2────┘
```

This library statement places pre-written text into a COBOL program.

The uniqueness of text-name and library-name is determined after the formation and conversion rules for a system-dependent name have been applied. A user-defined word can be the same as a text-name or a library-name. If more than one COBOL library is available during compilation, text-name need not be qualified. If text-name is not qualified, a library-name of SYSLIB is assumed. The following affects library-name and text-name:

library-name
If you specify library-name as a literal (*literal-2*), it is treated as the actual path. If you specify library-name with a user-defined word, the name is used as an environment variable and the value of the environment variable is used for the path to locate the COPY text. To specify multiple path names, delimit them by using a a semicolon (;).

If you don't specify library-name, the path used is as described under text-name.

text-name

The processing of text-name as a user-defined word depends on whether the environment variable corresponding to the text-name is set. If the the environment variable *is* set, the value of the environment variable is used as the file name, and possibly the path name, for the copybook.

A text-name is treated as both the path and file name if:

- library-name (or literal-2) is not given, *and*
- text-name is a literal (literal-1) or an environment variable, *and*
- The first character is '\' or the second character is ':'

For example,

```
COPY "\mycpylib\..." or COPY "d:\mycpylib\..."
```

If the environment variable corresponding to the text-name is *not* set, the copy text is searched for as the following names:

1. The text-name with the extension of `.cpy`
2. The text-name with the extension of `.cbl`
3. The text-name with the extension of `.cob`
4. The text-name without an extension

For example, `COPY MyCopy` searches in the following order:

- `MyCopy.cpy` (in all the specified paths, as described above)
- `MyCopy.cbl` (in all the specified paths, as described above)
- `MyCopy.cob` (in all the specified paths, as described above)
- `MyCopy` (in all the specified paths, as described above)

-I option

For other cases (when neither a library-name nor text-name indicates the path), the path searched is dependent on the -I option. For details, see "Options Supported by cob2" on page 142.

To have COPY A be equivalent to COPY A OF MYLIB specify -I%MYLIB%.

Based on the above rules, COPY "\X\Y" will be searched in the root directory, while COPY "X\Y" will be searched in the current directory.

COPY A OF SYSLIB is equivalent to COPY A. The -I option does not impact COPY statements with explicit library-name qualifications besides those with the library name of SYSLIB.

Environment Variable Notes If both library-name and text-name are environment variables the compiler will insert a path separator (\) between the two values. For example, COPY MYCOPY OF MYLIB with the settings of

```
SET MYCOPY=MYPDS(MYMEMBER)
SET MYLIB=MYFILE
```

results in MYFILE\MYPDS(MYMEMBER)

Compiler-Directing Statements

OS/2

Using a user-defined word as text-name enables you not only to access local files but to access PDS members on OS/390 without changing your mainframe source. For example:

```
COPY mycopybook
```

In this example, when the environment variable *mycopybook* is set to `H:mypds(mycopy)`, where:

H: is assigned to the specific host
`mypds` is the OS/390 PDS data set name
`mycopy` is the PDS member name

You can access OS/390 files from OS/2 using SdU (Smart Data Utilities), which allows OS/390 files to be accessed using an OS/2 pathname. However, note that it converts the path separator to "." to follow OS/390 naming conventions. You should keep this in mind when assigning values to your environment variables to ensure proper name formation. For example, these settings

```
SET MYCOPY=(MYMEMBER)
SET MYLIB=M:MYFILE\MYPDS
```

do not work because what is created is

M:MYFILE\MYPDS\(MYMEMBER)
which becomes
M:MYFILE.MYPDS.(MYMEMBER)

See the VSAM SMARTdata Utilities documentation for details on using DFMDRIVE to assign a drive letter to DFM.

OS/2

For more information on the COPY statement, see the discussion of the COPY statement in the *IBM COBOL Language Reference*.

DELETE statement
This extended source library statement removes COBOL statements from the BASIS source program.

EJECT statement
This compiler-directing statement specifies that the next source statement is to be printed at the top of the next page.

ENTER statement
The compiler handles this statement as a comment.

INSERT statement
This library statement adds COBOL statements to the BASIS source program.

PROCESS (CBL) statement
This statement, which is placed before the IDENTIFICATION DIVISION header of an outermost program, indicates which compiler options are to be used during compilation of the program. (See page 147 for the format of this statement).

For details on specifying compiler options with the PROCESS (CBL) statement and with other methods, see the discussion under “Compiling and Linking Programs” on page 142.

REPLACE statement
This statement is used to replace source program text.

SKIP1/2/3 statement
These statements indicate lines to be skipped in the source listing.

TITLE statement
This statement specifies that a title (header) be printed at the top of each page of the source listing. (See “Changing Header of Source Listing” on page 13.)

USE statement
The USE statement provides *declaratives* to specify the following:

Error-handling procedures—EXCEPTION/ERROR
Debugging lines and sections—DEBUGGING

Chapter 11. Setting Linker Options

Linker options vary depending on the operating system you are using. For a complete list of linker options on OS/2, see “Summary of OS/2 Linker Options” on page 211 For a complete list of linker options on Windows, see “Summary of Windows Linker Options” on page 226

Linker options are not case sensitive, so you can specify them in lower-, upper-, or mixed case. You can also substitute a dash (-) for the slash (/) preceding the option. For example, -DEBUG is equivalent to /DEBUG. You can specify options in either a short or long form. For example, /DE, /DEB, and /DEBU are all equivalent to /DEBUG. See “Summary of OS/2 Linker Options” on page 211 or “Summary of Windows Linker Options” on page 226 for the shortest acceptable form for each option. Lower- and uppercase, short and long forms, dashes, and slashes can all be used on one command line, as in:

```
ilink /de -DBGPACK -Map /NOI prog.obj
```

Separate options with a space or tab character. You can specify linker options in the following ways:

- On the command line
- In the ILINK environment variable
- In WorkFrame

Options specified on the command line override the options in the ILINK environment variable.

Some linker options take numeric arguments. You can enter numbers in decimal, octal, or hexadecimal format using standard C-language syntax. See “Specifying Numeric Arguments” on page 209 for more information.

Setting Options on the Command Line

Linker options specified on the command line override any previously specified in the ILINK environment variable (as described in “Setting Options in the ILINK Environment Variable” on page 209).

You can specify options anywhere on the command line. Separate options with a space or tab character.

For example, to link an object file with the /MAP option, enter:

```
ilink /M myprog.obj
```

Setting Options in the ILINK Environment Variable

Store frequently used options in the ILINK environment variable. This method is useful if you find yourself repeating the same command-line options every time you link. You cannot specify file names in the environment variable, only linker options.

The ILINK environment variable can be set either from the command line, in a command (`.CMD`) file, or in the CONFIG.SYS file. If it is set on the command line or by running a command file, the options will only be in effect for the current session (until you reboot your computer). If it is set in the CONFIG.SYS file, the options are set when you boot your computer, and are in effect every time you use the linker unless you override them using a `.CMD` file or by specifying options on the command line.

Example

In the following example, options on the command line override options in the environment variable. If you enter the following commands:

```
SET ILINK=/NOI /AL:256 /DE
ILINK test
ILINK /NODEF /NODEB prog
```

The first command sets the environment variable to the options /NOIGNORECASE, /ALIGNMENT:256, and /DEBUG

The second command links the file `test.obj`, using the options specified in the environment variable, to produce `test.exe`

The last command links the file `prog.obj` to produce `prog.exe`, using the option /NODEFAULTLIBRARYSEARCH, in addition to the options /NOIGNORECASE and /ALIGNMENT:256. The /NODEBUG option on the command line overrides the /DEBUG option in the environment variable, and the linker links without the /DEBUG option.

Setting Options in the WorkFrame Environment

If you have installed the WorkFrame product, you can set linker options using the options dialogs. You can use the dialogs when you create or modify a project.

Options you select while creating or changing a project are saved with that project.

Specifying Numeric Arguments

Some linker options and module statements take numeric arguments. You can specify numbers in any of the following forms:

Decimal Any number **not** prefixed with 0 or 0x is a decimal number. For example, 1234 is a decimal number.

Octal Any number prefixed with 0 (but not 0x) is an octal number. For example, 01234 is an octal number.

Specifying Numeric Arguments

Hexadecimal Any number prefixed with 0x is a hexadecimal number. For example, 0x1234 is a hexadecimal number.

Summary of OS/2 Linker Options

Figure 50. OS/2 Linker Options Summary

Option	Description	Default
/?	Display help	None
/ALIGNMENT	Set alignment factor	/A:512
/BASE, /NOBASE	Set preferred loading address	/BAS:0x00010000
/CODEVIEW, NOCODEVIEW	Include debugging information	/NOC
/DBGPACK, /NODBGPACK	Pack debugging information	/NODB
/DEBUG, /NODEBUG	Include debugging information	/NODEB
/DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH	Search default libraries	/DEF
/DLL	Generate DLL	/EXEC
/EXEC	Generate .EXE file	/EXEC
/EXEPACK, /NOEXEPACK	Compress data	/NOEXE
/EXTDICTIONARY, /NOEXTDICTIONARY	Use extended dictionary to search libraries	/EXT
/FORCE	Create executable output file even if errors	/NOFO
/FREEFORMAT, /NOFREEFORMAT	Use free format command line syntax	/FR
/HELP	Display help	None
/IGNORECASE, /NOIGNORECASE	Ignore capitalization in identifiers	/NOI
/INFORMATION, /NOINFORMATION	Display status of linking process	/NOIN
/LINENUMBERS, /NOLINENUMBERS	Include line numbers in map file	/NOLI
/LOGO, /NOLOGO	Display logo, echo response file	/LO
/MAP, /NOMAP	Generate map file	/NOM
/OPTFUNC, /NOOPTFUNC	Remove unreferenced functions	/NOOPTF
/OUT	Name output file	Name of first .OBJ file
/PACKCODE, /NOPACKCODE	Pack neighboring code segments with similar attributes	/PACKC: 0xFfffFfff
/PACKDATA, /NOPACKDATA	Pack neighboring data segments with similar attributes	/PACKD: 0xFfffFfff
/PMTYPE	Specify application type	None
/SECTION	Set attributes for segment	Accept default attributes
/SEGMENTS	Set maximum number of segments	/SE:128
/STACK	Set stack size of application	/ST:32768

Linker Options for OS/2

This section describes the linker options in alphabetical order.

For each option, the description includes:

- The syntax for specifying the option.
- The default setting.

- Any accepted abbreviations.
- A description of the option and its parameters, and any interaction it may have with other options.

/?

```
►►──/?──────────────────────────────────────────────────────────►◄
```

Use `/?` to display a list of valid linker options. This option is equivalent to `/HELP`.

/ALIGNMENT

```
►►──/ALIGNMENT:factor───────────────────────────────────────────►◄
```

Default is: /ALIGNMENT:512

Abbreviation is: /A

Use `/ALIGNMENT` to set the alignment factor in the .EXE or .DLL file.

The alignment factor determines where pages in the .EXE or .DLL file start. From the beginning of the file, the start of each page is aligned at a multiple (in bytes) of the alignment factor. The alignment factor must be a power of 2, from 1 to 4096.

/BASE, /NOBASE

```
►►──┬─/BASE:address─┬───────────────────────────────────────────►◄
    └─/NOBASE───────┘
```

Default is: /BASE:0X00010000

Abbreviations are: /BAS

Use `/BASE` to specify the preferred load address for the first load segment of a .DLL file. The number you specify in *address* is rounded up to the nearest multiple of 64K. The second load segment is then loaded at the next available multiple of 64K, and so on.

If the file's load segments cannot be loaded beginning at this preferred address, then the preferred address is ignored and the objects are loaded according to the internal relocation records retained in the file data.

For .EXE files, use the default base address of 64K (`/BASE:0x00010000`). Specifying this address explicitly can slightly reduce the size of the executable. Any other address will result in a warning, and 64K will be used anyway.

This option has the same effect as the BASE module definition file statement. If you specify both the BASE statement and the `/BASE` option, the statement value overrides the option value.

Specify `/NOBASE` to retain relocation records and emit internal fixups, when you generate an .EXE file. This does not affect the actual base address, or interfere with any value you specified with `/BASE`. You can specify both options.

/CODEVIEW, NOCODEVIEW

```
►►──┬─/CODEVIEW───┬──────────────────────────────────────────────►◄
    └─/NOCODEVIEW─┘
```

Default is: /NOCODEVIEW

Abbreviations are: /C|/NOC

Obsolete: These options will not be available in future releases of the linker. Use /DEBUG, /NODEBUG instead.

Use `/CODEVIEW` to include debug information in the output file, so you can debug the file with the debugger, or trace its execution with the Performance Analyzer. The linker will embed symbolic data and line number information in the output file.

For debugging, compile the object files with `cob2` option `-g`.

For the Performance Analyzer, compile the object files with `cob2` option `-p`.

`/CODEVIEW` provides the same functionality as `/DEBUG` and is provided only for purposes of compatibility.

Note: Linking with `/CODEVIEW` or `/DEBUG` increases the size of the executable output file.

/DBGPACK, /NODBGPACK

```
►►──┬─/DBGPACK───┬───────────────────────────────────────────────►◄
    └─/NODBGPACK─┘
```

Default is: /NODBGPACK

Abbreviations are: /DB|/NODB

Use `/DBGPACK` to eliminate redundant debug type information. The linker takes the debug type information from all object files and needed library components, and reduces the information to one entry per type. This results in a smaller executable output file, and can improve debugger performance.

Performance Consideration: Generally, linking with /DBGPACK slows the linking process, because it takes time to pack the information. However, if there is enough redundant debug type information, /DBGPACK can actually speed up your linking, because there is less information to write to file.

You can only pack debug information in objects created with version 3.0 of the compiler or later. If you use /DBGPACK with older object files, the linker generates a warning and does not pack the debug information.

When you specify /DBGPACK, /DEBUG is turned on by default.

/DEBUG, /NODEBUG

```
►►──┬─/DEBUG───┬──────────────────────────────────────────►◄
    └─/NODEBUG─┘
```

Default is: /NODEBUG

Abbreviations are: /DE|/NODEB

Use /DEBUG to include debug information in the output file, so you can debug the file with the debugger, or analyze its performance with the performance analyzer. The linker will embed symbolic data and line number information in the output file.

For debugging, compile the object files with cob2 option -g.

For the Performance Analyzer, compile the object files with cob2 option -p.

Note: Linking with /DEBUG increases the size of the executable output file.

/DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH

```
►►──┬─/DEFAULTLIBRARYSEARCH──────────────┬────────────────►◄
    └─/NODEFAULTLIBRARYSEARCH──┬──────────┬┘
                               └─:library─┘
```

Default is: /DEFAULTLIBRARYSEARCH

Abbreviations are: /DEF|/NOD

Use /DEFAULTLIBRARYSEARCH to have the linker search the default libraries of object files when resolving references. The default libraries for an object file are defined at compile time, and embedded in the object file. The linker searches the default libraries by default.

Use /NODEFAULTLIBRARYSEARCH to tell the linker to ignore default libraries when it resolves external references. If you specify a *library* with the option, the linker ignores that default library, but searches any others that are defined in the object files.

If you specify /NODEFAULTLIBRARYSEARCH, then you must explicitly specify all the libraries you want to use, including IBM VisualAge COBOL runtime libraries and any OS/2 libraries you need.

/DLL

```
►►──┬─/DLL──┬──────────────────────────────────────────────────────►◄
    └─/EXEC─┘
```

Default is: /EXEC

Abbreviations are: None

Use /DLL to identify the output file as a dynamic link library (.DLL file). You can also identify the output file as a DLL with the LIBRARY statement in a module definition file.

If you specify /DLL with /EXEC, then only the last specified of the options takes effect.

If you do not specify /DLL, then by default the linker produces an .EXE file (/EXEC).

/EXEC

```
►►──┬─/DLL──┬──────────────────────────────────────────────────────►◄
    └─/EXEC─┘
```

Default is: /EXEC

Abbreviations are: None

Use /EXEC to identify the output file as an executable program (.EXE file). The linker generates .EXE files by default. You can also identify the output as an .EXE file with the NAME statement in a module definition file. If you specify /EXEC with /DLL, only the last specified of the options takes effect.

If you do not specify /EXEC, the linker produces an .EXE file by default.

/EXEPACK, /NOEXEPACK

```
►►──┬─/EXEPACK──┬────┬─┬───────────────────────────────────────────►◄
    │           ├─:1─┤ │
    │           └─:2─┘ │
    └─/NOEXEPACK───────┘
```

Default is: /NOEXEPACK

Abbreviations are: /E|/NOEXE

/EXT, /NOE Options

Use /EXEPACK to reduce the size of the executable by compressing pages in the file. The operating system automatically decompresses the pages when the program runs.

Specify /EXEPACK[:1] to compress data segments in your output file, using run-length encoding compression. If compression does not reduce the size of the segment, the linker does not compress that segment.

Specify /EXEPACK:2 to compress both data and code segments, as follows:

- For data segments, the linker tries both LZW compression and run-length encoding compression, and uses the method with the more efficient result.
- For code segments, the linker uses LZW compression.

Segments are evaluated one page at a time. If compression does not reduce the size of the page, the page is not compressed.

OS/2 V3.0 only: Only set /EXEPACK:2 if you are developing for OS/2 version 3.0 or later. OS/2 version 2.1 or earlier cannot run programs that have been compressed with /EXEPACK:2.

Linking and compressing generally takes longer than linking alone, because of the extra time spent compressing. However, if the compression is effective enough, it can actually speed up the linking process, because there is less information to write to file.

By default, the linker does not compress the output file.

/EXTDICTIONARY, /NOEXTDICTIONARY

```
►►──┬─/EXTDICTIONARY───┬──────────────────────────────────────────────►◄
    └─/NOEXTDICTIONARY─┘
```

Default is: /NOEXTDICTIONARY

Abbreviations are: /EXT|/NOE

Use /EXTDICTIONARY to have the linker search the extended dictionaries of libraries when it resolves external references. The extended dictionary is a list of module relationships within a library. When the linker pulls in a module from the library, it checks the extended dictionary to see if that module requires other modules in the library, and then pulls in the additional modules automatically.

The linker searches the extended dictionary by default, to speed up the linking process.

Use /NOEXTDICTIONARY if you are defining a symbol in your object code that is also defined in one of the libraries you are linking to. Otherwise the linker issues error L2044 because you have defined the same symbol in two different places. When you link with /NOEXTDICTIONARY, the linker searches the dictionary directly, instead of searching the extended dictionary. This results in slower linking, because references must be resolved individually.

/FORCE

```
►►──┬─/FORCE───┬──────────────────────────────────────────────────►◄
    └─/NOFORCE─┘
```

Default is: /NOFORCE

Abbreviations are: /FO|/NOFO

Use `/FORCE` to produce an executable output file even if there are errors during the linking process.

By default, the linker does not produce an executable output file if it encounters an error.

/FREEFORMAT, /NOFREEFORMAT

```
►►──┬─/FREEFORMAT───┬─────────────────────────────────────────────►◄
    └─/NOFREEFORMAT─┘
```

Default is: /FREEFORMAT

Abbreviations are: /FR|/NOFR

Use `/FREEFORMAT` to allow free placement of files, options, and directories on the command line, separated by space or tab characters. Use the `/OUT` option to name the executable output file. Use the `/MAP` option to name the map file. Library and definition files are identified by their extension.

`/FREEFORMAT` is in effect by default.

Use `/NOFREEFORMAT` to allow a LINK386-compatible command line syntax, in which different types of file are grouped and separated by commas. If you specify `/NOFREEFORMAT`, then you cannot specify `/OUT`. Instead, specify a name for the executable output file in the appropriate place in the command line syntax.

/HELP

```
►►──/HELP─────────────────────────────────────────────────────────►◄
```

Default is: None

Abbreviation is: /H

Use `/HELP` to display a list of valid linker options. This option is equivalent to `/?`.

/L, /NOLI Options

/IGNORECASE, /NOIGNORECASE

```
►►──┬─/IGNORECASE───┬──────────────────────────────────────────────►◄
    └─/NOIGNORECASE─┘
```

Default is: /NOIGNORECASE

Abbreviations are: /IG|/NOI

Use /IGNORECASE to turn off case sensitivity, ignoring capitalization in identifiers. For example, with this option on, the linker treats ABC, abc, and Abc as equivalent.

By default, the linker is case sensitive, and would treat ABC, abc, and Abc as unique names.

/INFORMATION, /NOINFORMATION

```
►►──┬─/INFORMATION───┬─────────────────────────────────────────────►◄
    └─/NOINFORMATION─┘
```

Default is: /NOINFORMATION

Abbreviations are: /I|/NOIN

Use /INFORMATION to have the linker display information about the linking process as it occurs, including the phase of linking and the names and paths of the object files being linked.

If you are having trouble linking because the linker is finding the wrong files or finding them in the wrong order, use /INFORMATION to determine the locations of the object files being linked and the order in which they are linked.

The output from this option is sent to **stdout**. You can redirect the output to a file using OS/2 redirection symbols.

/LINENUMBERS, /NOLINENUMBERS

```
►►──┬─/LINENUMBERS───┬─────────────────────────────────────────────►◄
    └─/NOLINENUMBERS─┘
```

Default is: /NOLINENUMBERS

Abbreviations are: /L|/NOLI

Use /LINENUMBERS to include source file line numbers and associated addresses in the map file. For this option to take effect, there must already be line number information in the object files you are linking.

When you compile, use the cob2 option -qNUMBER to include line numbers in the object file (or the cob2 option -g, to include all debugging information).

If you give the linker an object file without line number information, the /LINENUMBERS option has no effect.

The /LINENUMBERS option forces the linker to create a map file, even if you specified /NOMAP.

By default, the map file is given the same name as the output file, plus the extension .map. You can override the default name by specifying a map file name.

/LOGO, /NOLOGO

```
►►──┬─/LOGO───┬─────────────────────────────────────────────────►◄
    └─/NOLOGO─┘
```

Default is: /LOGO

Abbreviations are: /LO|/NOL

Use /NOLOGO to suppress the product information that appears when the linker starts. /NOLOGO also stops the contents of the response file from being echoed to the screen.

Specify /NOLOGO before the response file on the command line, or in the ILINK environment variable. If the option appears in or after the response file, it is ignored.

By default, the linker displays product information at the start of the linking process, and displays the contents of the response file as it reads the file.

/MAP, /NOMAP

```
►►──┬─/MAP──┬───────────────────────────┬─┬─────────────────────────►◄
    │       └─:──┬─────┬────┬──────┬────┘ │
    │            └─dir─┘    └─name─┘      │
    └─/NOMAP──────────────────────────────┘
```

Default is: /NOMAP

Abbreviations are: /M|/NOM

Use /MAP to generate a map file with the name *name*, and in the directory *dir*, that lists the composition of each segment, and the public (global) symbols defined in the object files. The symbols are listed twice: in order of name, and in order of address.

If you do not specify *dir*, the map file is generated into the current working directory. If you do not specify *name*, the map file has the same name as the executable output file, with the extension `.map`.

For compatibility with LINK386, you can specify `/MAP:full`. With the IBM VisualAge COBOL linker, this is the same as specifying `/MAP`.

Note: If you are linking with the `/NOFREE` option, you can specify a name for the map file in the *map* parameter. Any name you specify with the `/MAP` option is ignored.

By default, the linker does not produce a map file.

/OPTFUNC, /NOOPTFUNC

```
►►──┬─/OPTFUNC───┬───────────────────────────────────────────────►◄
    └─/NOOPTFUNC─┘
```

Default is: /NOOPTFUNC

Abbreviations are: /OPTF|/NOOPTF

Use `/OPTFUNC` to remove unreachable functions. The linker removes functions that are:

- Not referenced anywhere in the object code
- Rendered unreferenced by the removal of other functions
- Not exported for use in other files

When the function is removed, any additional functions that were required only by that function are also removed. Removing the functions and code reduces the size of your .EXE or .DLL output file.

By default, the linker does not remove unreachable functions.

Performance Consideration: Optimized linking generally takes longer than regular linking, because of the extra processing that the linker performs. However, if the optimization is effective enough, it can actually speed up the linking process, because there is less information to write to file. Generally, you may want to link without the `/OPTFUNC` option, until your code is tested and stable.

/OUT

```
►►──/OUT:name────────────────────────────────────────────────────►◄
```

Default is: Name of first .OBJ file with appropriate extension.

Abbreviation is: /O

Use /OUT to specify a name for the executable output file. To use /OUT, you must be using the default command line syntax (/FREEFORMAT). If you are using the /NOFREE (LINK386-compatible) format, then you cannot use the /OUT option.

If you do not provide an extension with *name*, then the linker provides an extension based on the type of file you are producing:

File produced	**Default extension**
Executable program	.EXE
Dynamic link library	.DLL

/PACKCODE, /NOPACKCODE

```
►►──┬─/PACKCODE──┬─────────┬─┬──────────────────────────────────►◄
    │            └─:number─┘ │
    └─/NOPACKCODE────────────┘
```

Default is: /PACKCODE:0XFFFFFFFF

Abbreviations are: /PACKC|/NOP

Use /PACKCODE to produce slightly faster and more compact code. The linker groups neighboring code segments that have similar attributes, and assigns them to the same load segment. The linker adjusts offsets to each routine upward as required.

Specify *number* to set the maximum size for a load segment. The linker will start new load segments as necessary to avoid exceeding the maximum.

For 16-bit segments, *number* is ignored, and 65500 is used instead.

By default, the linker sets a maximum of 0xFfffFfff.

Use /NOPACKCODE to turn off code segment packing.

Use the /OPTFUNC option to reduce the size of your output files even further.

/PACKDATA, /NOPACKDATA

```
►►──┬─/PACKDATA──┬─────────┬─┬──────────────────────────────────►◄
    │            └─:number─┘ │
    └─/NOPACKDATA────────────┘
```

Default is: /NOPACKDATA

Abbreviations are: /PACKD|/NOPACKD

Use /PACKDATA to produce more compact files by grouping neighboring data segments that have similar attributes, and assigning them to the same load segment.

Specify *number* to set the maximum size for a load segment. The linker will start new load segments as necessary to avoid exceeding the maximum. By default, the linker sets a maximum of `0xFfffFfff`.

By default, the linker does not pack data segments.

/PMTYPE

```
►►──/PMTYPE:type──────────────────────────────────────────►◄
```

Default is: None

Abbreviation is: /PM

Use `/PMTYPE` to specify the type of .EXE file that the linker generates. Do not use this option when generating dynamic link libraries (DLLs). The option is equivalent to the NAME module statement, but uses different type names.

Figure 51. /PMTYPE Parameters

Type	Description	Equivalent NAME Statement Parameter
PM	Presentation Manager application. The application uses the API provided by the Presentation Manager, and must run in the Presentation Manager environment.	WINDOWAPI
VIO	Application compatible with Presentation Manager. The application can run inside the Presentation Manager, or it can run in a separate screen group. An application can be of this type if it uses the proper subset of OS/2 video, keyboard, and mouse functions supported in the Presentation Manager applications.	WINDOWCOMPAT
NOVIO	Application that is not compatible with the Presentation Manager and must run in a separate screen group from the Presentation Manager.	NOTWINDOWCOMPAT

/SECTION

```
                    ┌────────────┐
►►──/SECTION:name,──▼─attribute──┴────────────────────────►◄
```

Default is: Depends on segment type

Abbreviation is: /SEC

Use `/SECTION` to specify memory-protection attributes for the *name* segment. You can specify the following attributes:

Letter	Sets Attribute
E	EXECUTE
R	READ
S	SHARED
W	WRITE

For example,

```
/SEC:dseg1,RS
```

sets the READ and SHARED attributes, but not the EXECUTE, or WRITE attributes, for the segment `dseg1` in an .EXE file.

Defaults

Segments are assigned attributes by default, as follows:

Segment	Default Attributes
Code segments	EXECUTE, READ (ER) Correspond to the `SEGMENTS` attribute `EXECUTEREAD`.
Data segments (in .EXE file)	READ, WRITE (RW) Correspond to the `SEGMENTS` attribute `READWRITE`.
Data segments (in .DLL file)	READ, WRITE, SHARED (RWS) Correspond to the `SEGMENTS` attributes `READWRITE` and `SHARED`.
CONST32_RO segment	READ, SHARED (RS) Correspond to the `SEGMENTS` attributes `READONLY` and `SHARED`.

You can also set these attributes, and other attributes, to segments using statements in a module definition file:

`CODE`	Sets default attributes for CODE segments
`DATA`	Sets default attributes for DATA segments
`SEGMENTS`	Sets attributes for specific segments

Assignments given in a module definition file override any assignments made with `/SECTION`.

/ST Option

/SEGMENTS

```
►►──/SEGMENTS:number──────────────────────────────────────────────►◄
```

Default is: /SEGMENTS:256

Abbreviation is: /SE

Use /SEGMENTS to set the number of logical segments a program can have. You can set *number* to any value in the range 1 to 16375. See "Specifying Numeric Arguments" on page 209.

For each logical segment, the linker must allocate space to keep track of segment information. By using a relatively low segment limit as a default (256), the linker is able to link faster and allocate less storage space.

When you set the segment limit higher than 256, the linker allocates more space for segment information. This results in slower linking, but allows you to link programs with a large number of segments.

For programs with fewer than 256 segments, you can improve link time and reduce linker storage requirements by setting *number* to the actual number of segments in the program.

/STACK

```
►►──/STACK:size───────────────────────────────────────────────────►◄
```

Default is: /STACK:32768 (32K)

Abbreviation is: /ST

Use /STACK to set the stack size (in bytes) of your program. The size must be an even number from 0 to 0xFffffffe. If you specify an odd number, it is rounded up to the next even number.

You cannot specify a stack size in which the second most significant byte is either 02 or 04 (in hex), because of a restriction in OS/2 2.0. The linker issues a warning, and adds 64k to the specified stack size to avoid this restriction.

For example, if you specify /STACK:0x00020000 the linker adds 64k, which results in /STACK:0x00030000

Similarly, if you specify /STACK:0x11041111 the linker adds 64k, which results in /STACK:0x11051111

If your program generates a stack-overflow message, use /STACK to increase the size of the stack.

If your program uses very little stack space, you can save space by decreasing the stack size.

If the executable is a visual application containing more than 10 windows, you should add about 10K to the stack size for each additional window.

If your program uses a visual part containing more than 10 windows, then add about another 10K to the stack size for each additional window in that part. For example, if the most windows contained in any one part is 18, then specify `/ST:1134688` (that is, (1024 × 10 × 8) + 32768).

Note: Once the executable is produced, you can still change its stack size, using the EXEHDR utility in the Warp toolkit.

`/STACK` is equivalent to the STACKSIZE statement in a module definition (.DEF) file. If you specify both the statement and the option, the statement value overrides the option value.

Windows Linker Options

Summary of Windows Linker Options

Figure 52. Windows Linker Options Summary

Option	Description	Default
/?	Display help	None
/ALIGNADDR	Set address alignment	/A:0x00010000
/ALIGNFILE	Set file alignment	/A:512
/BASE	Set preferred loading address	/BAS:0x00400000
/CODE	Set section attributes for executable	/CODE:RX
/DATA	Set section attributes for data	/DATA:RW
/DBGPACK, /NODBGPACK	Pack debugging information	/NODB
/DEBUG, /NODEBUG	Include debugging information	/NODEB
/DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH	Search default libraries	/DEF
/DLL	Generate DLL	/EXEC
/DLL	Specify an entry point in an executable file	None
/EXECUTABLE	Generate .EXE file	/EXEC
/EXTDICTIONARY, /NOEXTDICTIONARY	Use extended dictionary to search libraries	/EXT
/EXTDICTIONARY, /NOEXTDICTIONARY	Do not relocate the file in memory	/NOFI
/FORCE	Create executable output file even if errors are detected	/NOFO
/HEAP	Set the size of the progam heap	/HEAP:0x100000,0x1000
/HELP	Display help	None
/INCLUDE	Forces a reference to a symbol	None
/INFORMATION, /NOINFORMATION	Display status of linking process	/NOIN
/LINENUMBERS, /NOLINENUMBERS	Include line numbers in map file	/NOLI
/LOGO, /NOLOGO	Display logo, echo response file	/LO
/MAP, /NOMAP	Generate map file	/NOM
/OUT	Name output file	Name of first .obj file
/PMTYPE	Specify application type	/PMTYPE:VIO
/SECTION	Set attributes for section	Set by `/CODE` and `/DATA`
/SEGMENTS	Set maximum number of segments	/SE:256
/STACK	Set stack size of application	/STACK: 0x100000,0x1000
/STUB	Specify the name of the DOS stub file	None
/SUBSYSTEM	Specify the required subsystem and version	/SUBSYSTEM: WINDOWS,4.0
/VERBOSE	Display status of linking process	/NOV
/VERSION	Write a version number in the run file	/VERSION:0.0

Windows Linker Options

This section describes the linker options in alphabetical order.

For each option, the description includes:

- The syntax for specifying the option.
- The default setting.
- Any accepted abbreviations.
- A description of the option and its parameters, and any interaction it may have with other options.

/?

```
►►──/?──────────────────────────────────────────────────────►◄
```

Use /? to display a list of valid linker options. This option is equivalent to /HELP.

/ALIGNADDR

```
►►──/ALIGNADDR:factor───────────────────────────────────────►◄
```

Default is: /ALIGNADDR:0X00010000

Abbreviation is: /ALIGN

Use /ALIGNADDR to set the address alignment for segments.

The alignment factor determines where segments in the .EXE or .DLL file start. From the beginning of the file, the start of each segment is aligned at a multiple (in bytes) of the alignment factor. The alignment factor must be a power of 2, from 512 to 256M.

/ALIGNFILE

```
►►──/ALIGNFILE:factor───────────────────────────────────────►◄
```

Default is: /ALIGNFILE:512

Abbreviation is: /A

Use /ALIGNFILE to set the file alignment for segments.

The alignment factor determines where segments in the .EXE or .DLL file start. From the beginning of the file, the start of each segment is aligned at a multiple (in bytes) of the alignment factor. The alignment factor must be a power of 2, from 512 to 64K.

/CODE Option

/BASE

```
►►──/BASE:──┬─address─────────┬──────────────────────────────────────────►◄
            └─@filename,key───┘
```

Default is: /BASE:0X00400000

Abbreviations are: /BAS

Use /BASE to specify the preferred load address for the first load segment of a .DLL file.

Specifying *@filename, key*, in place of *address*, bases a set of programs (usually a set of DLLs) so they do not overlap in memory. *filename* is the name of a text file that defines the memory map for a set of files. *key* is a reference to a line in *filename* beginning with the specified key. Each line in the memory-map file has the syntax:

key address maxsize

Separate the elements with one or more spaces or tabs. The *key* is a unique name in the file. The *address* is the location of the memory image in the virtual address space. The *maxsize* is an amount of memory within which the image must fit. The linker will issue a warning when the memory image of the program exceeds the specified size. A comment in the memory-map file begins with a semicolon (;) and runs to the end of the line.

/CODE

```
               ┌─────────────────┐
►►──/CODE:─────▼─attribute───────┴───────────────────────────────────────►◄
```

Default is: /CODE:RX

CODE Description Abbreviations are: None

Use /CODE to specify the default attributes for all code sections. Letters can be specified in any order.

Letter	**Attribute**
E or X	EXECUTE
R	READ
S	SHARED
W	WRITE

The default is /CODE:RX.

/DATA

```
         ┌────────────┐
►►──/DATA:─▼─attribute─┴──────────────────────────────────────────────►◄
```

Default is: /DATA:RW

Abbreviations are: None

Use /DATA to specify the default attributes for all data sections. Letters can be specified in any order.

Letter	**Attribute**
E or X	EXECUTE
R	READ
S	SHARED
W	WRITE

The default is /DATA:RW.

/DBGPACK, /NODBGPACK

```
►►──┬─/DBGPACK───┬──────────────────────────────────────────────────►◄
    └─/NODBGPACK─┘
```

Default is: /NODBGPACK

Abbreviations are: /DB|/NODB

Use /DBGPACK to eliminate redundant debug type information. The linker takes the debug type information from all object files and needed library components, and reduces the information to one entry per type. This results in a smaller executable output file, and can improve debugger performance.

Performance Consideration: Generally, linking with /DBGPACK slows the linking process, because it takes time to pack the information. However, if there is enough redundant debug type information, /DBGPACK can actually speed up your linking, because there is less information to write to file.

When you specify /DBGPACK, /DEBUG is turned on by default.

/DEF, /NOD Options

/DEBUG, /NODEBUG

```
►►──┬─/DEBUG───┬──────────────────────────────────────────►◄
    └─/NODEBUG─┘
```

Default is: /NODEBUG

Abbreviations are: /D|/NODEB

Use /DEBUG to include debug information in the output file, so you can debug the file with the debugger, or analyze its performance with Performance Analyzer. The linker will embed symbolic data and line number information in the output file.

For debugging, specify the cob2 option -g.

For the Performance Analyzer, compile the object files with the cob2 option -p.

Note: Linking with /DEBUG increases the size of the executable output file.

/DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH

```
►►──┬─/DEFAULTLIBRARYSEARCH───┬──┬──────────┬──────────────►◄
    └─/NODEFAULTLIBRARYSEARCH─┘  └─:library─┘
```

Default is: /DEFAULTLIBRARYSEARCH

Abbreviations are: /DEF|/NOD

Use /DEFAULTLIBRARYSEARCH to have the linker search the default libraries of object files when resolving references.

If you specify a *library* with the option, the linker adds the library name to the list of default libraries. The default libraries for an object file are defined at compile time, and embedded in the object file. The linker searches the default libraries by default.

Use /NODEFAULTLIBRARYSEARCH to tell the linker to ignore default libraries when it resolves external references. If you specify a *library* with the option, the linker ignores that default library, but searches the rest of the default libraries (and any others that are defined in the object files).

If you specify /NODEFAULTLIBRARYSEARCH without specifying *library*, then you must explicitly specify all the libraries you want to use, including IBM VisualAge COBOL runtime libraries.

/DLL

```
►►──┬─/DLL────────┬──────────────────────────────────────────────────►◄
    └─/EXECUTABLE─┘
```

Default is: /EXECUTABLE

Abbreviation is: /EXEC

Use /DLL to identify the output file as a dynamic link library (.DLL file). The object files should be compiled with the cob2 option -dll.

If you specify /DLL with /EXEC, only the last specified of the options takes effect.

If you do not specify /DLL, or any of the other options above, then by default the linker produces an .EXE file (/EXEC).

/ENTRY

```
►►──/ENTRY:name────────────────────────────────────────────────────►◄
```

Default is: None

Abbreviation is: /EN

Use /ENTRY to specify an entry point (name of a routine or function) in an executable.

/EXECUTABLE

```
►►──┬─/DLL────────┬──────────────────────────────────────────────────►◄
    └─/EXECUTABLE─┘
```

Default is: /EXECUTABLE

Abbreviation is: /EXEC

Use /EXEC to identify the output file as an executable program (.EXE file). The linker generates .EXE files by default.

If you specify /EXEC with /DLL, only the last specified of the options takes effect.

If you do not specify /EXEC or /DLL, then by default the linker produces an .EXE file.

/FO Option

/EXTDICTIONARY, /NOEXTDICTIONARY

```
►►──┬─/EXTDICTIONARY───┬───────────────────────────────────────────►◄
    └─/NOEXTDICTIONARY─┘
```

Default is: /EXTDICTIONARY

Abbreviations are: /EXT|/NOE

Use /EXTDICTIONARY to have the linker search the extended dictionaries of libraries when it resolves external references. The extended dictionary is a list of module relationships within a library. When the linker pulls in a module from the library, it checks the extended dictionary to see if that module requires other modules in the library, and then pulls in the additional modules automatically.

The linker searches the extended dictionary by default, to speed up the linking process.

Use /NOEXTDICTIONARY if you are defining a symbol in your object code that is also defined in one of the libraries you are linking to. Otherwise the linker issues an error because you have defined the same symbol in two different places. When you link with /NOEXTDICTIONARY, the linker searches the dictionary directly, instead of searching the extended dictionary. This results in slower linking, because references must be resolved individually.

/FIXED, /NOFIXED

```
►►──┬─/FIXED───┬───────────────────────────────────────────────────►◄
    └─/NOFIXED─┘
```

Default is: /NOFIXED

Abbreviations are: /FI|/NOFI

Use /FIXED to tell the loader not to relocate a file in memory when the specified base address is not available.

For more information on base addresses, see the /BASE linker option.

/FORCE

```
►►──┬─/FORCE───┬───────────────────────────────────────────────────►◄
    └─/NOFORCE─┘
```

Default is: /NOFORCE

Abbreviations are: /FO|/NOFO

Use /FORCE to produce an executable output file even if there are errors during the linking process.

By default, the linker does not produce an executable output file if it encounters an error.

/HEAP

```
►►──/HEAP:reserve──┬────────┬──────────────────────────────────────►◄
                   └─,commit─┘
```

Default is: /HEAP:0X100000,0X1000

Abbreviation is: /HEA

Use /HEAP to set the size of the program heap in bytes. The *reserve* argument sets the total virtual address space reserved. The *commit* sets the amount of physical memory to allocate initially. When *commit* is less than *reserve*, memory demands are reduced, but execution time can be slower.

/HELP

```
►►──/HELP──────────────────────────────────────────────────────────►◄
```

Default is: None

Abbreviation is: /H

Use /HELP to display a list of valid linker options. This option is equivalent to /?.

/INCLUDE

```
►►──/INCLUDE:symbol────────────────────────────────────────────────►◄
```

Default is: None

Abbreviation is: /INC

Use /INCLUDE to force a reference to a symbol. The linker searches for an object module that defines the symbol.

/INFORMATION, /NOINFORMATION

```
►►──┬─/INFORMATION───┬──────────────────────────────────────────►◄
    └─/NOINFORMATION─┘
```

Default is: /NOINFORMATION

Abbreviations are: /I|/NOIN

See the description of the /VERBOSE linker option.

/LINENUMBERS, /NOLINENUMBERS

```
►►──┬─/LINENUMBERS───┬──────────────────────────────────────────►◄
    └─/NOLINENUMBERS─┘
```

Default is: /NOLINENUMBERS

Abbreviations are: /L|/NOLI

Use /LINENUMBERS to include source file line numbers and associated addresses in the map file. For this option to take effect, there must already be line number information in the object files you are linking.

When you compile, use the cob2 option -qNUMBER to include line numbers in the object file (or the cob2 option -g, to include all debugging information).

If you give the linker an object file without line number information, the /LINENUMBERS option has no effect.

The /LINENUMBERS option forces the linker to create a map file, even if you specified /NOMAP.

By default, the map file is given the same name as the output file, plus the extension .map. You can override the default name by specifying a map filename.

/LOGO, /NOLOGO

```
►►──┬─/LOGO───┬─────────────────────────────────────────────────►◄
    └─/NOLOGO─┘
```

Default is: /LOGO

Abbreviations are: /LO|/NOL

Use /NOLOGO to suppress the product information that appears when the linker starts.

Specify /NOLOGO before the response file on the command line, or in the ILINK environment variable. If the option appears in or after the response file, it is ignored.

By default, the linker displays product information at the start of the linking process, and displays the contents of the response file as it reads the file.

/MAP, /NOMAP

```
►►──┬─/MAP──┬─────────┬─┬──────────────────────────────────────────────►◄
    │       └─:name───┘ │
    └─/NOMAP────────────┘
```

Default is: /NOMAP

Abbreviations are: /M|/NOM

Use /MAP to generate a map file called *name*. The file lists the composition of each segment, and the public (global) symbols defined in the object files. The symbols are listed twice: in order of name and in order of address.

If you do not specify a directory, the map file is generated into the current working directory. If you do not specify *name*, the map file has the same name as the executable output file, with the extension .map.

By default, the linker does not produce a map file.

/OUT

```
►►──/OUT:name──────────────────────────────────────────────────────────►◄
```

Default is: Name of first .OBJ file with appropriate extension.

Abbreviation is: /O

Use /OUT to specify a name for the executable output file.

If you do not provide an extension with *name*, then the linker provides an extension based on the type of file you are producing:

File produced	**Default extension**
Executable program	.EXE
Dynamic link library	.DLL

If you do not use the /OUT option, then the linker uses the filename of the first object file you specified, with the appropriate extension.

/SEC Option

/PMTYPE

```
►►──/PMTYPE:type──────────────────────────────────────────────►◄
```

Default is: /PMTYPE:VIO

Abbreviation is: /PM

Use /PMTYPE to specify the type of .EXE file that the linker generates. Do not use this option when generating dynamic link libraries (DLLs).

One of the following types must be specified:

PM The executable must be run in a window.

VIO The executable can be run either in a window or in a full screen.

NOVIO The executable must not be run in a window; it must use a full screen.

/SECTION

```
                    ┌──────────┐
►►──/SECTION:name,──▼─attribute─┴─────────────────────────────►◄
```

Default is: Depends on segment type

Abbreviation is: /SEC

Use /SECTION to specify memory-protection attributes for the *name* section. *name* is case sensitive. You can specify the following attributes:

Letter	**Sets Attribute**
E or **X**	EXECUTE
R	READ
S	SHARED
W	WRITE

For example,

```
/SEC:dseg1,RS
```

sets the READ and SHARED attributes, but not the EXECUTE, or WRITE attributes, for the section dseg1 in an .EXE file.

Defaults

Sections are assigned attributes by default, as follows:

Segment	Default Attributes
Code sections	EXECUTE, READ (ER)
Data sections (in .EXE file)	READ, WRITE (RW), not shared
Data sections (in .DLL file)	READ, WRITE, not shared
CONST32_RO section	READ, SHARED (RS)

/SEGMENTS

```
►►──/SEGMENTS:number──────────────────────────────────────────►◄
```

Default is: /SEGMENTS:256

Abbreviation is: /SE

Use /SEGMENTS to set the number of logical segments a program can have. You can set *number* to any value in the range 1 to 16375. See “Specifying Numeric Arguments” on page 209.

For each logical segment, the linker must allocate space to keep track of segment information. By using a relatively low segment limit as a default (256), the linker is able to link faster and allocate less storage space.

When you set the segment limit higher than 256, the linker allocates more space for segment information. This results in slower linking, but allows you to link programs with a large number of segments.

For programs with fewer than 256 segments, you can improve link time and reduce linker storage requirements by setting *number* to the actual number of segments in the program.

/STACK

```
►►──/STACK:reserve──┬─────────┬───────────────────────────────►◄
                    └─,commit─┘
```

Default is: /STACK:0X100000,0X1000

Abbreviation is: /ST

Use /STACK to set the stack size (in bytes) of your program. The size must be an even number from 0 to 0xFfffFffe. If you specify an odd number, it is rounded up to the next even number.

reserve indicates the total virtual address space reserved. *commit* sets the amount of physical memory to allocate initially. When *commit* is less than *reserve*, memory demands are reduced, although execution time may be slower.

/STUB

```
►►──/STUB:filename─────────────────────────────────────────────────►◄
```

Default is: None

Abbreviation is: /STU

Use /STUB to specify the name of the DOS executable at the beginning of the output file created.

By default, the linker defines its own stub.

/SUBSYSTEM

```
►►──/SUBSYSTEM:subsystem──┬─────────────────────┬──────────────────►◄
                          └─,major──┬────────┬──┘
                                    └─.minor─┘
```

Default is: /SUBSYSTEM:WINDOWS,4.0

Abbreviation is: /SU

Use /SUBSYSTEM to specify the subsystem and version required to run the program. The *major* and *minor* arguments are optional and specify the minimum required version of the subsystem. The *major* and *minor* arguments are integers in the range 0 to 65535.

Subsystem	**Major.Minor**	**Description**
WINDOWS	3.10	A graphical application that uses the Graphical Device Interface (GDI) API.
CONSOLE	3.10	A character-mode application that uses the Console API.

/VERBOSE

```
►►──┬─/VERBOSE───┬─────────────────────────────────────────────────►◄
    └─/NOVERBOSE─┘
```

Default is: /NOVERBOSE

Abbreviations are: /VERB|/NOV

Use /VERBOSE to have the linker display information about the linking process as it occurs, including the phase of linking and the names and paths of the object files being linked.

If you are having trouble linking because the linker is finding the wrong files or finding them in the wrong order, use /VERBOSE to determine the locations of the object files being linked and the order in which they are linked.

The output from this option is sent to **stdout**. You can redirect the output to a file using Windows redirection symbols.

/VERBOSE is the same as /INFORMATION.

/VERSION

```
►►──/VERSION:major──┬────────┬──────────────────────────────────────────►◄
                    └─.minor─┘
```

Default is: /VERSION:0.0

Abbreviation is: /VER

Use /VERSION to write a version number in the header of the run file. The *major* and *minor* arguments are integers in the range 0 to 65535.

Chapter 12. Run-Time Options

The following run-time options are supported:

- CHECK
- DEBUG
- ERRCOUNT
- FILESYS
- TRAP
- UPSI

Syntax

Syntax of the run-time options follows. See “Definitions of COBOL Environment Variables” on page 135 to see where to specify them.

CHECK

CHECK flags checking errors in an application. In COBOL, index, subscript, and reference modification ranges are checking errors.

```
                  ┌─ON──┐
►►──CHECK──(──────┤     ├──)──────────────────────────────────────►◄
                  └─OFF─┘
```

Default is: CHECK(ON).

Abbreviation is: CH

ON
Specifies that run-time checking is performed.

OFF
Specifies that run-time checking is not performed.

Usage Note

CHECK(ON) has no effect if NOSSRANGE was in effect at compile time.

Performance Consideration

If your COBOL program was compiled with SSRANGE, and you are not testing or debugging an application, performance improves when you specify CHECK(OFF).

DEBUG

DEBUG specifies whether the COBOL debugging sections specified by the USE FOR DEBUGGING declarative are active.

```
      ┌─DEBUG───┐
►►────┴─NODEBUG─┴──────────────────────────────────────────────────────►◄
```

Default is: NODEBUG.

DEBUG
Activates the debugging sections.

NODEBUG
Suppresses the debugging sections.

Performance Consideration

To improve performance, use this option only while debugging.

ERRCOUNT

ERRCOUNT specifies how many conditions of severity 1 (W-level) can occur before the run-unit terminates abnormally. Any severity 2 (E-level) or higher will result in termination of the run-unit independent of the ERRCOUNT option.

```
►►──ERRCOUNT──(──┬────────┬──)──────────────────────────────────────►◄
                 └─number─┘
```

Default: ERRCOUNT(20).

number
The number of severity 1 conditions per individual thread that can occur while this run-unit is running. If the number of conditions exceeds *number*, the run-unit terminates abnormally.

FILESYS

FILESYS specifies the file system used for files for which no explicit file system selections are made, either through an ASSIGN or an evironment variable. The option applies to sequential, relative, and indexed files. For details about using FILESYS for access to CICS files, see “Accessing Btrieve Data” on page 417.

```
                  ┌─VSA─┐
►►──FILESYS──(──┼─BTR─┼──)──────────────────────────────────────────►◄
                  └─STL─┘
```

Default is : FILESYS(VSA) for OS/2 and FILESYS(STL) for Windows.

VSA The file system is VSAM.

BTR The file system is Btrieve.

STL The file system is STL.

Only the first three characters of the file system identifier are used and the identifier is case insensitive. For example, the following examples are all valid specifications for VSAM:

- FILESYS(VSA)
- FILESYS(vSAM)
- FILESYS(vsa)

TRAP

TRAP indicates whether COBOL intercepts exceptions.

```
          ┌─ON──┐
►►──TRAP──(─┼─────┼──)──────────────────────────────────────────►◄
          └─OFF─┘
```

Default is : TRAP(ON).

Note: If TRAP(OFF) is in effect and you do not supply your own trap handler to handle exceptional conditions, the conditions will result in a default action by the operating system. For example, if your program attempts to store into an illegal location, the default system action is to issue a message and terminate the process.

ON
Activates COBOL interception of exceptions.

OFF
Deactivates COBOL interception of exceptions.

Usage Notes

- Use TRAP(OFF) only when you need to analyze a program exception before COBOL handles it.
- When you specify TRAP(OFF) in a non-CICS environment, no exception handlers are established.
- Running with TRAP(OFF) (for exception diagnosis purposes) can cause many side effects, because COBOL requires TRAP(ON). When you run with TRAP(OFF), you can get side effects even if you do not encounter a software-raised condition, program check, or abend. If you do encounter a program check or an abend with TRAP(OFF) in effect, the following side effects can occur:
  - Resources obtained by COBOL are not freed.
  - Files opened by COBOL are not closed, so records might be lost.
  - No messages or dump output are generated.

  The run-unit terminates abnormally if such conditions are raised.

UPSI

UPSI sets the eight UPSI switches on or off for applications that use COBOL routines.

```
►►──UPSI──(──┬──────────┬──)──────────────────────────────────►◄
             └─nnnnnnnn─┘
```

Default is : UPSI(00000000).

nnnnnnnn
n represents one UPSI switch between 0 and 7, the leftmost *n* representing the first switch. Each *n* can either be 0 (off) or 1 (on).

Chapter 13. Debugging Techniques

COBOL provides several language elements and facilities to help you determine the cause of problems in program behavior. This chapter focuses on how to use source language for debugging and describes the compiler options that make debugging easier.

This chapter describes only COBOL *source language* debugging techniques. The IDBUG Debugger is a graphical debugging tool you will find useful for debugging programs. For help with the debugger, refer to its online help and information.

Using Source Language to Debug

You can use several COBOL language features to pinpoint the cause of a failure in your program. If the program is part of a large application already in production, you will not want to re-compile and run the program again to debug. Instead, you can write a small test case to simulate the part of the program that failed and code some of these debugging features of the COBOL language in the test case to help detect the exact cause of the problem:

- DISPLAY statements.
- USE EXCEPTION/ERROR declaratives.
- Class test.
- INITIALIZE or SET verbs.
- Scope terminators.
- File status keys.
- USE FOR DEBUGGING declaratives.

The rules for using each of these language features are explained in *IBM COBOL Language Reference*.

Tracing Program Logic (DISPLAY Statements)

You can use the interactive debugger available on your platform to step through your program (compiled with the TEST option), or adding DISPLAY statements can help you trace through the logic of the program. If, for example, you determine that the problem appears in an EVALUATE statement or in a set of nested IF statements, DISPLAY statements in each path will show you how the logic flow is working. If you determine that the problem is being caused by the way a numeric value is calculated, you can use DISPLAY statements to check the value of some of the interim results.

For example, to determine whether a particular routine started and finished you might insert code like this into your program:

```
DISPLAY "ENTER CHECK PROCEDURE"
    .
    .  (checking procedure routine)
    .
DISPLAY "FINISHED CHECK PROCEDURE"
```

After you are sure that the routine works correctly, you can put asterisks in column 7 of the DISPLAY statement lines, which converts them to comment lines. Alternatively, you might put a 'D' in column 7 of your DISPLAY (or any other debugging) statements. If you include the WITH DEBUGGING MODE clause in the ENVIRONMENT DIVISION, the 'D' in column 7 will be ignored and the DISPLAY statements will be implemented. Without the DEBUGGING MODE clause, the 'D' in column 7 makes the statement a comment.

Before you put the program into production, delete all the debugging aids you used and re-compile the program. The program will run more efficiently and use less storage.

CICS: The DISPLAY statement cannot be used in programs running under CICS.

Handling Input/Output Errors (USE EXCEPTION/ERROR Declaratives)

If you have determined that the problem lies in one of the I/O procedures in your program, you can include the USE EXCEPTION/ERROR declarative to help debug the problem.

If the file fails to open for some reason, the appropriate EXCEPTION/ERROR declarative will be performed. The appropriate declarative might be a specific one for the file or one provided for the different open attributes—INPUT, OUTPUT, I/O, or EXTEND.

Each USE AFTER STANDARD ERROR statement must be coded in a separate section. This section(s) must be coded immediately after the DECLARATIVE SECTION keyword of the PROCEDURE DIVISION. The rules for coding these statements are provided in *IBM COBOL Language Reference*.

Validating Data (Class Test)

If you suspect that your program is trying to perform arithmetic on non-numeric data or is somehow receiving the wrong type of data on an input record, you can use the class test to validate the type of data. The class test checks whether data is alphabetic, alphabetic-lower, alphabetic-upper, MBCS, KANJI, or numeric.

Assessing Switch Problems (INITIALIZE or SET Statements)

Using INITIALIZE or SET statements to initialize a table or variable is useful when you suspect that the problem might be caused by residual data left in those fields. If the problem you are having happens intermittently and not always with the same data, the problem could be that a switch is not initialized but generally is set to the right value (0 or 1) by accident. By including a SET statement to ensure that the switch is initialized, you can either determine that the uninitialized switch is the problem or remove that as a possible cause.

Improving Program Readability (Explicit Scope Terminators)

Scope terminators can help you in debugging because they indicate clearly the end of a statement. The logic of your program will become more apparent, and therefore easier to trace, if you use scope terminators.

Using Source Language to Debug

Finding Input/Output Errors (File Status Keys)

File status keys can help you determine if your program errors are due to the logic of your program or if they are I/O errors occurring on the storage media.

To use file status keys as a debugging aid, include a test after each I/O statement to check for a value other than zero in the status key. If the value is other than zero, you can expect that you will receive an error message. You can use a nonzero value as an indication that you should look at the way the I/O procedures in the program were coded. You can also include procedures to correct the error based on the value of the status key.

The status key values and their associated meanings are described in the *IBM COBOL Language Reference*.

Generating Information about Procedures (USE FOR DEBUGGING Declaratives)

USE FOR DEBUGGING declaratives are another way to generate information about your program or test case and how it is running. The declarative allows you to include statements in the program and indicate when they should be performed when you run your compiled program. For example, if you want to check how many times a procedure is run, you could include a debugging procedure in the USE FOR DEBUGGING declarative and use a counter to keep track of the number of times control passes to that procedure.

Rules for Debugging Statements and Debugging Lines

Each USE FOR DEBUGGING declarative must be coded in a separate section. This section or these sections must be coded in the DECLARATIVES SECTION of the PROCEDURE DIVISION. The rules for coding them are provided in *IBM COBOL Language Reference*.

You can have either debugging lines or debugging statements or both in your program. Debugging lines are statements in your program that are identified by a 'D' in column 7. Debugging statements are the statements coded in the DECLARATIVES SECTION of the PROCEDURE DIVISION.

- The debugging statements in a USE FOR DEBUGGING declarative must:
  - Be only in a DECLARATIVE SECTION.
  - Follow the header USE FOR DEBUGGING.
  - Be only in the outermost program; they are not valid in nested programs. Debugging sections are also never triggered by procedures contained in nested programs.
- Debugging lines must have a D in column 7 to identify them.

To use debugging lines in your program, you must include the WITH DEBUGGING MODE clause on the SOURCE-COMPUTER line in the ENVIRONMENT DIVISION.

To use debugging sections in your program, you must include both:

- The WITH DEBUGGING MODE clause

- The DEBUG run-time option

See the *IBM COBOL Language Reference* appendix on source-language debugging for more details.

Options Note: The TEST compiler option and the WITH DEBUGGING MODE phrase are mutually exclusive, with the WITH DEBUGGING MODE phrase taking precedence.

USE FOR DEBUGGING Example

The program segments in Figure 53 on page 248 show what kind of statements are needed to use a DISPLAY statement and a USE FOR DEBUGGING declarative to test a program. The DISPLAY statement is used to generate information on the terminal or on the output file The USE FOR DEBUGGING declarative is used with a counter to show how many times a routine was actually run.

Use the adding-to-a-counter technique to check:

1. How many times a PERFORM was executed. You will know whether a particular routine is being used and whether the control structure you are using is correct.
2. How many times a loop routine actually executes. This will tell you whether the loop is executing and whether the number you have used for the loop is accurate.

```
Environment Division
   .
   .
   .
Data Division.
   .
   .
   .
Working-Storage Section.
   .
   .   (other entries your program needs)
   .

01  Trace-Msg    PIC X(30) Value "  Trace for Procedure-Name : ".
01  Total        PIC 9(9)  Value 1.
   .
   .
   .
Procedure Division.
Declaratives.
Debug-Declaratives Section.
    Use For Debugging On Some-Routine.
Debug-Declaratives-Paragraph.
    Display Trace-Msg, Debug-Name, Total.
End Declaratives.

Main-Program Section.
   .
   .  (source program statements)
   .
  Perform Some-Routine.
   .
   .  (source program statements)
   .
  Stop Run.
Some-Routine.
   .
   .  (whatever statements you need in this paragraph)
   .
  Add 1 To Total.
Some-Routine-End
```

Figure 53. Example of Using the USE FOR DEBUGGING EXAMPLE

In Figure 53, the DISPLAY statement coded in the DECLARATIVES SECTION will issue this message:

```
Trace For Procedure-Name : Some-Routine 22
```

every time the procedure SOME-ROUTINE is run. The number at the end of the message, 22, is the value accumulated in the data-item, TOTAL; it shows the number of times SOME-ROUTINE has been run. The statements in the debugging declarative are performed before the named procedure is run.

Another Use for the DISPLAY Statement: You can also use the DISPLAY statement technique shown above to trace program execution and show the flow through your program. You do this by changing the USE FOR DEBUGGING declarative in the DECLARATIVES SECTION to:

```
USE FOR DEBUGGING ON ALL PROCEDURES.
```

and dropping the word TOTAL from the DISPLAY statement. Now, a message will be displayed before every non-debugging procedure in the outermost program is run.

Using Compiler Options for Debugging

The FLAG Option

This section discusses the compiler options that generate information to help you find coding mistakes and other errors in your program.

The FLAG option lets you select the level of error to be diagnosed during compilation and indicate where the syntax-error messages appear in the listing. Use FLAG(I) or FLAG(I,I) to be notified of all errors in your program.

Code in the first parameter the lowest severity level of the syntax-error messages to be issued. You can code in the optional second parameter the lowest level of the syntax messages to be embedded in the source listing.

If you specify:

I (informational)
You get all the messages. I-level messages generate a return code of zero; RC=0.

W (warning)
You get all the warning messages and those of a higher severity. W-level errors generate a return code of four; RC=4.

E (error)
You get all error messages and those of a higher severity. E-level errors generate a return code of eight; RC=8.

S (severe)
You get all severe and U (unrecoverable) messages. S-level errors generate a return code of twelve; RC=12.

U (unrecoverable)
You get only unrecoverable messages. U-level errors generate a return code of sixteen; RC=16.

When you specify the second parameter, the syntax-error messages are embedded in the source listing at the point where the compiler had enough information available to detect the error. All embedded messages, except those issued by the library compiler phase, will directly follow the statement to which they refer. The number of the statement containing the error is also included with the message. Embedded messages are repeated with the rest of the diagnostic messages following the source listing.

Embedded Messages:

1. If NOSOURCE is one of your options, the syntax-error messages are included only in the list at the end of the listing.
2. U-level errors are not embedded in the source listing, as an error of this severity terminates the compilation.

For an illustration of how messages identifying syntax errors are imbedded in the source listing, see Figure 54 on page 250.

Note that some messages in the summary apply to more than one COBOL statement.

```
DATA VALIDATION AND UPDATE PROGRAM                               FLAGOUT   Date 02/27/1998  Time 12:26:53   Page    26
  LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Map and Cross Reference
.
.
.
  000977               /
  000978               *****************************************************************
  000979               ***          I N I T I A L I Z E   P A R A G R A P H           **
  000980               ***  Open files. Accept date, time and format header lines.    **
  000981         IA4690***   Load location-table.                                     **
  000982               *****************************************************************
  000983                  100-initialize-paragraph.
  000984                    move spaces to ws-transaction-record                                   IMP 339
  000985                    move spaces to ws-commuter-record                                      IMP 315
  000986                    move zeroes to commuter-zipcode                                        IMP 326
  000987                    move zeroes to commuter-home-phone                                     IMP 327
  000988                    move zeroes to commuter-work-phone                                     IMP 328
  000989                    move zeroes to commuter-update-date                                    IMP 332
  000990                    open input update-transaction-file                                     203

==000990==> IGYPS2052-S An error was found in the definition of file "LOCATION-FILE".  The
                        reference to this file was discarded.

  000991                        location-file                                                      192
  000992                        i-o commuter-file                                                  180
  000993                        output print-file                                                  216
  000994                    if loccode-file-status not = "00" or                                   248
  000995                       update-file-status not = "00" or                                    247
  000996                       updprint-file-status not = "00"                                     249
  000997      1                display "Open Error ..."
  000998      1                display "  Location File Status = " loccode-file-status             248
  000999      1                display "  Update   File Status = " update-file-status              247
  001000      1                display "  Print    File Status = " updprint-file-status            249
  001001      1               perform 900-abnormal-termination                                     1433
  001002                    end-if
  001003         IA4760     if commuter-file-status not = "00" and not = "97"                      240
  001004      1               display "100-OPEN"
  001005      1               move 100 to comp-code                                                230
  001006      1               perform 500-vsam-error                                               1387
  001007      1               display "Commuter File Status (OPEN) = "
  001008      1                        commuter-file-status                                        240
  001009      1               perform 900-abnormal-termination                                     1433
  001010         IA4790     end-if
```

Figure 54 (Part 1 of 3). FLAG(I,I) Output

```
  001011                 accept ws-date from date                                        UND

==001011==> IGYPS2121-S "WS-DATE" was not defined as a data-name.  The statement was discarded.

  001012        IA4810   move corr ws-date to header-date                                UND 463

==001012==> IGYPS2121-S "WS-DATE" was not defined as a data-name.  The statement was discarded.

  001013                 accept ws-time from time                                        UND

==001013==> IGYPS2121-S "WS-TIME" was not defined as a data-name.  The statement was discarded.

  001014        IA4830   move corr ws-time to header-time                                UND 457

==001014==> IGYPS2121-S "WS-TIME" was not defined as a data-name.  The statement was discarded.

  001015        IA4840   read location-file                                              192
.
.
.
DATA VALIDATION AND UPDATE PROGRAM                              FLAGOUT   Date 02/27/1998  Time 12:26:53   Page    69
LineID  Message code  Message text

   192  IGYDS1050-E   File "LOCATION-FILE" contained no data record descriptions.  The file definition was discarded.

   899  IGYPS2052-S   An error was found in the definition of file "LOCATION-FILE".  The reference to this file was discarded.

                      Same message on line:    990

  1011  IGYPS2121-S   "WS-DATE" was not defined as a data-name.  The statement was discarded.

                      Same message on line:   1012

  1013  IGYPS2121-S   "WS-TIME" was not defined as a data-name.  The statement was discarded.

                      Same message on line:   1014

  1015  IGYPS2053-S   An error was found in the definition of file "LOCATION-FILE".  This input/output statement was discarded.

                      Same message on line:   1027

  1026  IGYPS2121-S   "LOC-CODE" was not defined as a data-name.  The statement was discarded.

  1209  IGYPS2121-S   "COMMUTER-SHIFT" was not defined as a data-name.  The statement was discarded.

                      Same message on line:   1230

  1210  IGYPS2121-S   "COMMUTER-HOME-CODE" was not defined as a data-name.  The statement was discarded.

                      Same message on line:   1231

  1212  IGYPS2121-S   "COMMUTER-NAME" was not defined as a data-name.  The statement was discarded.

                      Same message on line:   1233

  1213  IGYPS2121-S   "COMMUTER-INITIALS" was not defined as a data-name.  The statement was discarded.

                      Same message on line:   1234

  1223  IGYPS2121-S   "WS-NUMERIC-DATE" was not defined as a data-name.  The statement was discarded.
```

Figure 54 (Part 2 of 3). FLAG(I,I) Output

```
Messages    Total    Informational    Warning    Error    Severe    Terminating
Printed:      19                                    1        18
* Statistics for COBOL program FLAGOUT:
*    Source records = 1755
*    Data Division statements = 279
*    Procedure Division statements = 479
End of compilation 1,  program FLAGOUT,  highest severity: Severe.
Return code 12
```

Figure 54 (Part 3 of 3). FLAG(I,I) Output

The NOCOMPILE Option

Use the NOCOMPILE option to produce a listing that will help you find your COBOL coding mistakes, such as missing definitions, improperly defined data names, and duplicate data names. You can use NOCOMPILE two ways: with or without parameters.

Using NOCOMPILE with Parameters

When you use NOCOMPILE(*x*), where *x* is one of the error levels, your program will be compiled, if all the errors are of a lower severity than the *x* level. If an error of *x* level or higher occurs, the compilation stops and your program will be syntax-checked only. You will receive a source listing if you have specified the SOURCE option.

Using NOCOMPILE without Parameters

When you use NOCOMPILE without parameters, the compiler only syntax-checks the source program. If you have also specified the SOURCE option, the compiler will produce a listing after the syntax-checking is completed. The compiler does not produce object code when NOCOMPILE without parameters is in effect.

The following compiler options are suppressed when you use NOCOMPILE without parameters: LIST, OBJECT, OPTIMIZE, SSRANGE, and TEST.

The SEQUENCE Option

The SEQUENCE option tells the compiler to check your source program and flag statements that are out of sequence. You can use this option to tell you if a section of your source program was moved or deleted accidentally.

When you use SEQUENCE, the compiler checks the source statement numbers you have supplied to see if they are in ascending order. Two asterisks are placed alongside any statement numbers out of sequence, and the total number of these statements is printed out as the first line of the diagnostics following the source listing.

The XREF Option

The XREF(FULL) option tells the compiler to generate a sorted cross-reference listing of data-names, procedure-names, and program-names. The cross-reference will include the line number where the data-name, procedure-name, or program-name was defined as well as the line numbers of all references.

You can use the cross-reference listing produced by the XREF option to find out where a data-name, procedure-name, or program-name was defined and referenced.

The XREF(SHORT) option allows you to control the cross-reference listing by printing only the explicitly referenced variables.

When you use both the XREF (with FULL or SHORT) and the SOURCE options, you will get a modified cross-reference printed to the right of the source listing. This embedded cross-reference gives the line number where the data-name or procedure-name was defined.

For more information on the XREF option and some example listings, see “A Data-Name, Procedure-Name, and Program-Name Cross-Reference Listing” on page 263.

The MAP Option

Use the MAP option to produce a listing of the items you defined in the DATA DIVISION, plus all items implicitly declared.

For more information on the MAP option, see “Data Map Listing” on page 258.

Embedded Map Summary

When you use the MAP option, an embedded MAP summary is generated to the right of the COBOL source data declaration. An embedded MAP summary contains condensed data MAP information. For more information, see “Embedded MAP Summary” on page 259.

The SSRANGE Option

Use the SSRANGE compiler option to check:

- Subscripted or indexed data references.

  The subscripted or indexed data reference is checked to determine if the effective address of the desired element is within the maximum boundary of the specified table.

- Variable-length data references (a reference to a data item that contains an OCCURS DEPENDING ON clause).

  The variable-length data reference is checked to determine if the actual length is positive and within the maximum defined length for the group data item.

- Reference-modified data references.

  The reference-modified data reference is checked to determine if the offset and length are positive and the sum of the offset and length are within the maximum length for the data item.

When the SSRANGE option is specified, checking is not performed until run time and then, only if both of the following are true:

- The COBOL statement containing the indexed, subscripted, variable-length, or reference-modified data item is actually performed.
- The CHECK run-time option is ON at run time.

If any check finds that an address is generated that is outside of the address range of the group data item containing the referenced data, an error message will be generated and the program will stop running. The error message identifies the table or identifier that was being referenced and the line number in the program where the error happened. Additional information is provided depending on the type of reference that caused the error.

If all subscripts, indices, or reference modifiers are literals in a given data reference and they result in a reference outside of the data item, the error will be diagnosed at compile time, regardless of the setting of the SSRANGE compiler option.

Performance Note: SSRANGE can cause the performance of your program to diminish somewhat because of the extra overhead needed to check each subscripted or indexed item.

The TEST Option

On the mainframe, you select the TEST option to prepare your program for use with the debugger. IBM VisualAge COBOL includes a graphical debugger. However, rather than use the TEST option to prepare your executable COBOL program for use with the debugger, you can use the `-g` option of the `cob2` command (see “Compiling and Linking Programs” on page 142).

Getting Useful Listing Components

This section introduces the different types of compiler listings produced by COBOL. The type of listing produced by the compiler depends on which compiler options you use.

Note: The listing produced by the compiler is not a programming interface and is subject to change.

After reading this section you should be familiar with each type of output; you will know how to request each type and what kind of information is provided in the output.

A Short Listing—the Bare Minimum

If you do not specify any compiler options and the default options are NOSOURCE, NOXREF, NOVBREF, NOMAP, NOOFFSET, and NOLIST, or if all the compiler diagnostic options have been turned off, you will get a short listing.

The short listing contains diagnostic messages about the compilation, a list of the options in effect for the program, and statistics about the content of the program. Figure 55 on page 255 is an example of a short listing.

The listing is explained after Figure 55, and the numbers used in the explanation correspond to those in the figure. (For illustrative purposes, some errors that cause diagnostic messages to be issued were deliberately introduced.)

```
PP 5639-B92 IBM VisualAge COBOL (OS/2)  2.2             Date 02/27/1998  Time 12:26:53   Page     1 1
Invocation parameters:    2
quote
PROCESS(CBL) statements:
CBL FLAG(I,I),MAP,TEST
CBL NONUMBER,QUOTE,SEQ,XREF,VBREF     3
Options in effect:    4
      ADATA
      QUOTE
      BINARY(NATIVE)
      CALLINT(SYSTEM,NODESCRIPTOR)
      CHAR(NATIVE)
    NOCICS
      COLLSEQ(BINARY)
    NOCOMPILE(S)
    NOCURRENCY
    NODATEPROC
    NODYNAM
      ENTRYINT(SYSTEM)
      EXIT(NOINEXIT,NOPRTEXIT,NOLIBEXIT,ADEXIT(IWZRMGUX))
      FLAG(I,I)
    NOFLAGSTD
      FLOAT(NATIVE)
    NOIDLGEN
      LIB
      LINECOUNT(60)
    NOLIST
      MAP
    NONUMBER
    NOOPTIMIZE
      PGMNAME(LONGUPPER)
      PROBE
    NOPROFILE
      SEPOBJ
      SEQUENCE
      SIZE(2097152)
      SOURCE
      SPACE(1)
      SQL
    NOSSRANGE
      TERM
      TEST
    NOTHREAD
    NOTILED
      TRUNC(STD)
    NOTYPECHK
      VBREF
    NOWORD
      XREF(FULL)
      YEARWINDOW(1900)
      ZWB
```

Figure 55 (Part 1 of 2). Example of a Short Listing

```
DATA VALIDATION AND UPDATE PROGRAM 5                    SLISTING  Date 02/27/1998  Time 12:26:53   Page     2

LineID  Message code  Message text    6

        IGYDS0139-W   Diagnostic messages were issued during processing of compiler options.  These messages are
                      located at the beginning of the listing.

   193  IGYDS1050-E   File "LOCATION-FILE" contained no data record descriptions.  The file definition was discarded.

   889  IGYPS2052-S   An error was found in the definition of file "LOCATION-FILE".  The reference to this file
                      was discarded.

                      Same message on line:    983

   993  IGYPS2121-S   "WS-DATE" was not defined as a data-name.  The statement was discarded.

                      Same message on line:    994

   995  IGYPS2121-S   "WS-TIME" was not defined as a data-name.  The statement was discarded.

                      Same message on line:    996

   997  IGYPS2053-S   An error was found in the definition of file "LOCATION-FILE".  This input/output statement
                      was discarded.

                      Same message on line:   1009

  1008  IGYPS2121-S   "LOC-CODE" was not defined as a data-name.  The statement was discarded.

  1219  IGYPS2121-S   "COMMUTER-SHIFT" was not defined as a data-name.  The statement was discarded.

                      Same message on line:   1240

  1220  IGYPS2121-S   "COMMUTER-HOME-CODE" was not defined as a data-name.  The statement was discarded.

                      Same message on line:   1241

  1222  IGYPS2121-S   "COMMUTER-NAME" was not defined as a data-name.  The statement was discarded.

                      Same message on line:   1243

  1223  IGYPS2121-S   "COMMUTER-INITIALS" was not defined as a data-name.  The statement was discarded.

                      Same message on line:   1244

  1233  IGYPS2121-S   "WS-NUMERIC-DATE" was not defined as a data-name.  The statement was discarded.


Messages    Total    Informational    Warning    Error   Severe   Terminating    7

Printed:      21                         2        1       18

* Statistics for COBOL program SLISTING:     8
    Source records = 1765
    Data Division statements = 277
    Procedure Division statements = 513

End of compilation 1,  program SLISTING,  highest severity: Severe.    9
Return code 12
```

Figure 55 (Part 2 of 2). Example of a Short Listing

1 COBOL default page header, including compiler level information from the LVLINFO installation time compiler option.

2 Message about options passed to the compiler at compiler invocation. This message does not appear if no options were passed.

3 Options coded in the PROCESS (or CBL) statement.

4 Status of options at the start of this compilation.

5 Customized page header resulting from the COBOL program TITLE statement.

6 Program diagnostics. The first message will refer you to the library phase diagnostics, if there were any. Diagnostics for the library phase are always presented at the beginning of the listing.

7 Count of diagnostic messages in this program, grouped by severity level.

8 Program statistics for the program SLISTING.

9 Program statistics for the compilation unit. When you perform a batch compilation (multiple outermost COBOL programs in a single compilation), the return code is the highest message severity level for the entire compilation.

A Listing of Your Source Code—for Historical Records

By using the SOURCE compiler option, you request a copy of your source code to be included with the compiler output. You will want this output for testing and debugging your program—and as an historical record when the program is completely debugged. Figure 56 shows an example of SOURCE output.

Using Your Own Line Numbers

The NUMBER compiler option tells the compiler to use your line numbers in the compiled program. When you use the NUMBER option, the compiler does a sequence check of your source statement line numbers in columns 1 through 6 as the statements are read in. When a line number is found to be out of sequence, the compiler assigns to it a number with a value one higher than the line number of the preceding statement. The new value is flagged with two asterisks. A diagnostic message indicating an out of sequence error is also included in the compilation listing.

Figure 56 shows an example of the output produced by the NUMBER compiler option. In the portion of the listing shown, the programmer numbered two of the statements out of sequence.

```
DATA VALIDATION AND UPDATE PROGRAM  1                           IGYTCARA  Date 02/27/1998  Time 12:26:53   Page    22
  LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Cross-Reference  2
     3      4      5
                 087000/****************************************************************
                 087100***                 D O   M A I N   L O G I C                   **
                 087200***                                                             **
                 087300*** Initialization. Read and process update transactions until **
                 087400*** EOE. Close files and stop run.                              **
                 087500*****************************************************************
                 087600 procedure division.
```

Figure 56 (Part 1 of 2). Example of SOURCE and NUMBER Output

```
               087700   000-do-main-logic.
               087800     display "PROGRAM SRCOUT - Beginning"
               087900     perform 050-create-vsam-master-file.
               088150     display "perform 050-create-vsam-master finished".
088151**       088125     perform 100-initialize-paragraph
               088200     display "perform 100-initialize-paragraph finished"
               088300     read update-transaction-file into ws-transaction-record
               088400         at end
            1  088500       set transaction-eof to true
               088600     end-read
               088700     display "READ completed"
               088800     perform until transaction-eof
            1  088900       display "inside perform until loop"
            1  089000       perform 200-edit-update-transaction
            1  089100       display "After perform 200-edit   "
            1  089200       if no-errors
            2  089300         perform 300-update-commuter-record
            2  089400         display "After perform 300-update "
            1  089650       else
089651**    2  089600         perform 400-print-transaction-errors
            2  089700         display "After perform 400-errors "
            1  089800       end-if
            1  089900       perform 410-re-initialize-fields
            1  090000       display "After perform 410-reinitialize"
            1  090100       read update-transaction-file into ws-transaction-record
            1  090200           at end
            2  090300         set transaction-eof to true
            1  090400       end-read
            1  090500       display "After '2nd READ'   "
               090600     end-perform
```

Figure 56 (Part 2 of 2). Example of SOURCE and NUMBER Output

1 Customized page header resulting from the COBOL program TITLE statement.

2 The scale line labels Area A, Area B, and source code column numbers.

3 Source code line number assigned by compiler.

4 Program (PL) and statement (SL) nesting level.

5 Columns 1 through 6 of program (the sequence number area).

Data Map Listing

The MAP compiler option provides you with a mapping of all Data Division items, plus all implicitly declared variables, of your program. You can see an example of MAP output in Figure 57 on page 259. The numbers used in the explanation below correspond to the numbers used in Figure 57. The terms and symbols used in MAP output are described in Figure 59 on page 261.

```
DATA VALIDATION AND UPDATE PROGRAM                               IGYTCARA  Date 02/27/1998  Time 12:26:53   Page    22

Data Division Map


1
Data Definition Attribute codes (rightmost column) have the following meanings:
    D = Object of OCCURS DEPENDING    G = GLOBAL                             LSEQ= ORGANIZATION LINE SEQUENTIAL
    E = EXTERNAL                      O = Has OCCURS clause                  SEQ= ORGANIZATION SEQUENTIAL
    VLO=Variably Located Origin       OG= Group has own length definition    INDX= ORGANIZATION INDEXED
    VL= Variably Located              R = REDEFINES                          REL= ORGANIZATION RELATIVE


2         3   4                                          5       6   7         8
Source   Hierarchy and                                                                              Data Def
LineID   Data Name                                        Length(Displacement) Data Type      Attributes
     4  PROGRAM-ID IGYTCARA----------------------------------------------------------------------------------*
   180   FD COMMUTER-FILE . . . . . . . . . . . . . . . .                      File            INDX
   182   1  COMMUTER-RECORD . . . . . . . . . . . . . . .         80           Group
   183     2  COMMUTER-KEY. . . . . . . . . . . . . . . .         16(0000000)  Display
   184     2  FILLER. . . . . . . . . . . . . . . . . . .         64(0000016)  Display
   186   FD COMMUTER-FILE-MST . . . . . . . . . . . . . .                      File            INDX
   188   1  COMMUTER-RECORD-MST . . . . . . . . . . . . .         80           Group
   189     2  COMMUTER-KEY-MST. . . . . . . . . . . . . .         16(0000000)  Display
   190     2  FILLER. . . . . . . . . . . . . . . . . . .         64(0000016)  Display
   192   FD LOCATION-FILE . . . . . . . . . . . . . . . .                      File            SEQ
   203   FD UPDATE-TRANSACTION-FILE . . . . . . . . . . .                      File            SEQ
   208   1  UPDATE-TRANSACTION-RECORD . . . . . . . . . .         80           Display
   216   FD PRINT-FILE. . . . . . . . . . . . . . . . . .                      File            SEQ
   221   1  PRINT-RECORD. . . . . . . . . . . . . . . . .        121           Display
   228   1  WORKING-STORAGE-FOR-IGYCARA . . . . . . . . .          1           Display
```

Figure 57. Example of MAP Output

1 Explanations of the data definition attribute codes.

2 Source line number where the data item was defined.

3 Level definition or number. The compiler generates this number in the following way:

- First level of any hierarchy is always 01. Increase 1 for each level— any item you coded as 02 through 49.
- Level numbers 66, 77, and 88, and the indicators FD and SD, are not changed.

4 Data-name that is used in the source module.

5 Length of data item. Base locator value.

6 Hexadecimal displacement from the beginning of the containing structure.

7 The data type and usage. These terms are explained in Figure 59 on page 261.

8 Data definition attribute codes. The definitions are explained at the top of the DATA DIVISION Map.

Embedded MAP Summary

An embedded MAP summary is printed by specifying the MAP option when generating a listing. The summary appears in the right margin of the listing for lines in the DATA DIVISION that contain data declarations. Figure 58 on page 260 describes the fields included in the embedded map summary.

When both XREF data and an embedded MAP summary exist on the same line, the embedded MAP summary is printed first.

```
000002                Identification Division.
000003
000004                Program-id.  EMBMAP.

 ⋮
000176                Data division.
000177                File section.
000178
000179
000180                FD  COMMUTER-FILE
000181                    record 80 characters.                                                     1     2
000182                  01 commuter-record.                                                        80
000183                    05 commuter-key                 PIC x(16).                                     16(0000000)
000184                    05 filler                       PIC x(64).                                     64(0000016)

 ⋮
000221         IA1620   01 print-record                   pic x(121).                                   121

 ⋮
000227                Working-storage section.
000228                  01 Working-storage-for-EMBMAP     pic x.                                          1
000229
000230                  77 comp-code                      pic S9999 comp.                                 2
000231                  77 ws-type                        pic x(3)   value spaces.                        3
000232
000233
000234                  01 i-f-status-area.                                                               2
000235                    05 i-f-file-status              pic x(2).                                       2(0000000)
000236                      88 i-o-successful             value zeroes.                          IMP
000237
000238
000239                  01 status-area.                                                                   8
000240                    05 commuter-file-status         pic x(2).                            3         2(0000000)
000241                      88 i-o-okay                   value zeroes.                          IMP
000242                    05 commuter-vsam-status.                                                        6(0000002)
000243                      10 vsam-r15-return-code       pic 9(2) comp.                                  2(0000002)
000244                      10 vsam-function-code         pic 9(1) comp.                                  2(0000004)
000245                      10 vsam-feedback-code         pic 9(3) comp.                                  2(0000006)
000246
000247                  77 update-file-status             pic xx.                                         2
000248                  77 loccode-file-status            pic xx.                                         2
000249                  77 updprint-file-status           pic xx.                                         2

 ⋮
000877                procedure division.
000878                  000-do-main-logic.
000879                    display "PROGRAM EMBMAP - Beginning".
000880                    perform 050-create-vsam-master-file.                                   931

 ⋮
```

Figure 58. Example of an Embedded MAP Summary

1 Decimal length of data item

2 Hexadecimal displacement from the beginning of the base locator value

3 Special definition symbols. These symbols are:

UND The user-name is undefined

DUP The user-name is defined more than once

IMP An implicitly defined name, such as special registers and figurative constants

IFN An intrinsic function reference

EXT An external reference

***** The program-name is unresolved because the NOCOMPILE option is in effect

Figure 59. Terms Used in MAP Output

Usage	Description
ALPHA-EDIT	Alphabetic-Edited
ALPHABETIC	Alphabetic
AN-EDIT	Alphanumeric-Edited
BINARY	Binary (Computational)
COMP-1	Internal floating-point (single-precision)
COMP-2	Internal floating-point (double-precision)
DBCS	DBCS (Display-1)
DBCS-EDIT	DBCS Edited
DISP-NUM	External Decimal
DISPLAY	Alphanumeric
File processing method (VSAM)	File (FD)
GROUP	Group Fixed-Length
GRP-VARLEN	Group Variable-Length
INDEX	Index
INDX-NAME	Index-name
Level name	Condition (77)
Level name for condition name	Condition (88)
Level name for RENAMES	Condition (66)
NUM-EDIT	Numeric-Edited
OBJECT REFERENCE	Object reference
PACKED-DEC	Internal Decimal (Computational-3)
POINTER	Pointer
PROCEDURE-POINTER	Pointer to an externally invocable program (or function)
Sort file definition	Sort Definition (SD)

Nested Program Map

The MAP compiler option also supplies you with a nested program map if your program contains nested programs. The nested program map shows where the programs are defined and provides program attribute information.

```
Nested Program Map

   1

Program Attribute codes (rightmost column) have the following meanings:
   C = COMMON
   I = INITIAL
   U = PROCEDURE DIVISION USING...

  2     3        4                                                      5

Source Nesting                                                          Program
LineID Level   Program Name from PROGRAM-ID paragraph                   Attributes
     2         NESTED. . . . . . . . . . . . . . . . . . . . . . . . . .
    12     1     X1. . . . . . . . . . . . . . . . . . . . . . . . . . .
    20     2       X11 . . . . . . . . . . . . . . . . . . . . . . . . .
    27     2       X12 . . . . . . . . . . . . . . . . . . . . . . . . .
    35     1     X2. . . . . . . . . . . . . . . . . . . . . . . . . . .
```

Figure 60. Example of Nested Program Map

1 Explanations of the program attribute codes.

2 The source line number where the program was defined.

3 Depth of program nesting.

4 The program name.

5 The program attribute codes.

A PROCEDURE DIVISION Listing with Assembler Expansion (LIST Output)

The LIST compiler option provides you with a listing of the PROCEDURE DIVISION along with the assembler coding produced by the COBOL compiler. This type of output can be especially helpful when you are trying to find the failing verb in a program. You can also use this output to find the address in storage of a data item that was moved during the program.

Note: The listing produced by the compiler is not a programming interface and is subject to change.

Getting LIST Output

You request LIST output from the compiler by using the LIST compiler option when you compile your program.

The assembler listing will be written to a file with the same name as the source program with the extension ".asm," except for batch compiles with the SEPOBJ option; see "LIST" on page 183 for the file names generated in that case.

A Verb Cross-Reference Listing

The VBREF compiler-time option produces an alphabetic listing of all the verbs in your program and shows where each is referenced. The output includes each verb used, a count of the number of times it is used, and the line numbers where the verb is used. You can use VBREF output as a handy lookup when you need to find an instance of a particular verb.

```
1         2                              3

2       ACCEPT . . . . . . . . . . . .  101  101
2       ADD. . . . . . . . . . . . . .  129  130
1       CALL . . . . . . . . . . . . .  140
5       CLOSE. . . . . . . . . . . . .  90  94  97  152  153
20      COMPUTE. . . . . . . . . . . .  150  164  164  165  166  166  166  166  167  168  168  169  169  170  171  171
                                        171  172  172  173
2       CONTINUE . . . . . . . . . . .  106  107
2       DELETE . . . . . . . . . . . .  96  119
47      DISPLAY. . . . . . . . . . . .  88  90  91  92  92  93  94  94  94  95  96  96  97  99  99  100  100  100  100
                                        103  109  117  117  118  119  138  139  139  139  139  139  139  140  140  140
                                        140  143  148  148  149  149  149  152  152  152  153  162
2       EVALUATE . . . . . . . . . . .  116  155
47      IF . . . . . . . . . . . . . .  88  90  93  94  94  95  96  96  97  99  100  103  105  105  107  107  107  109
                                        110  111  111  112  113  113  113  113  114  114  115  115  116  118  119  124
                                        124  126  127  129  132  133  134  135  136  148  149  152  152
183     MOVE . . . . . . . . . . . . .  90  93  95  98  98  98  98  98  99  100  101  101  102  104  105  105  106  106
                                        107  107  108  108  108  108  108  108  109  110  111  112  113  113  113  114
                                        114  114  115  115  116  116  117  117  117  118  118  118  119  119  120  121
                                        121  121  121  121  121  121  121  121  121  122  122  122  122  122  123  123
                                        123  123  123  123  123  124  124  124  125  125  125  125  125  125  125  126
                                        126  126  126  126  127  127  127  127  128  128  129  129  130  130  130  130
                                        131  131  131  131  131  132  132  132  132  132  132  133  133  133  133  133
                                        134  134  134  134  134  135  135  135  135  135  135  136  136  137  137  137
                                        137  137  138  138  138  138  141  141  142  142  144  144  144  144  145  145
                                        145  145  146  149  150  150  150  151  151  155  156  156  157  157  158  158
                                        159  159  160  160  161  161  162  162  162  168  168  168  169  169  170  171
                                        171  172  172  173  173
5       OPEN . . . . . . . . . . . . .  93  95  99  144  148
62      PERFORM. . . . . . . . . . . .  88  88  88  88  89  89  89  91  91  91  91  93  93  94  94  95  95  95  95  96
                                        96  96  97  97  97  100  100  101  102  104  109  109  111  116  116  117  117
                                        117  118  118  118  118  119  119  119  120  120  124  125  127  128  133  134
                                        135  136  136  137  150  151  151  153  153
8       READ . . . . . . . . . . . . .  88  89  96  101  102  108  149  151
1       REWRITE. . . . . . . . . . . .  118
4       SEARCH . . . . . . . . . . . .  106  106  141  142
46      SET. . . . . . . . . . . . . .  88  89  101  103  104  105  106  108  108  136  141  142  149  150  151  152  154
                                        155  156  156  156  156  157  157  157  157  158  158  158  158  159  159  159
                                        159  160  160  160  160  161  161  161  161  162  162  164  164
2       STOP . . . . . . . . . . . . .  92  143
4       STRING . . . . . . . . . . . .  123  126  132  134
33      WRITE. . . . . . . . . . . . .  94  116  129  129  129  129  129  130  130  130  130  145  146  146  146  146  147
                                        147  151  165  165  166  166  167  174  174  174  174  174  174  174  175  175
```

Figure 61. Example of VBREF Compiler Output

The numbers in the explanation below correspond to Figure 61.

1 Number of times the verb is used in the program.

2 Verb.

3 Line numbers where verb is used.

A Data-Name, Procedure-Name, and Program-Name Cross-Reference Listing

The XREF compiler option provides you with sorted cross-reference listings of data-names, procedure-names, and program-names. The listings also tell you the location of all references to a particular data-, procedure-, or program-name. This output will help you find, quickly, a reference to a particular data-, procedure-, or program-name in your program.

User-defined words in your program are sorted using the locale that is active. Hence, the collating sequence will determine the order for the cross-reference listing, including MBCS words.

Group Names: Group names in a MOVE CORRESPONDING statement are listed in the XREF listing. The cross-reference listing includes the group names and all the elementary names involved in the move.

Using a Sorted Cross-Reference Listing

You can use XREF output to find where you have used a particular data- or procedure-name. If you need to find all of the statements that modify a particular data item, you can use the output to determine what line(s) referenced or modified a data item. With the XREF output, you can also determine the context in which a procedure is referenced in your program. For example, you can determine whether a verb was used in a PERFORM block or as part of a USE FOR DEBUGGING declarative. (The context of the procedure reference is indicated by the characters preceding the line number.)

```
An "M" preceding a data-name reference indicates that the data-name is modified by this reference.

     1        2                              3

 Defined   Cross-reference of data names   References

     264   ABEND-ITEM1
     265   ABEND-ITEM2
     347   ADD-CODE . . . . . . . . . . .  1126 1192
     381   ADDRESS-ERROR. . . . . . . . .  M1156
     280   AREA-CODE. . . . . . . . . . .  1266 1291 1354 1375
     382   CITY-ERROR . . . . . . . . . .  M1159

        4

Context usage is indicated by the letter preceding a procedure-name reference.
These letters and their meanings are:
    A = ALTER (procedure-name)
    D = GO TO (procedure-name) DEPENDING ON
    E = End of range of (PERFORM) through (procedure-name)
    G = GO TO (procedure-name)
    P = PERFORM (procedure-name)
    T = (ALTER) TO PROCEED TO (procedure-name)
    U = USE FOR DEBUGGING (procedure-name)

     5        6                              7

 Defined   Cross-reference of procedures   References

     877   000-DO-MAIN-LOGIC
     943   050-CREATE-VSAM-MASTER-FILE. .  P879
     995   100-INITIALIZE-PARAGRAPH . . .  P881
    1471   1100-PRINT-I-F-HEADINGS. . . .  P926
    1511   1200-PRINT-I-F-DATA. . . . . .  P928
    1573   1210-GET-MILES-TIME. . . . . .  P1540
    1666   1220-STORE-MILES-TIME. . . . .  P1541
    1682   1230-PRINT-SUB-I-F-DATA. . . .  P1562
    1706   1240-COMPUTE-SUMMARY . . . . .  P1563
    1052   200-EDIT-UPDATE-TRANSACTION. .  P890
    1154   210-EDIT-THE-REST. . . . . . .  P1145
    1189   300-UPDATE-COMMUTER-RECORD . .  P893
    1237   310-FORMAT-COMMUTER-RECORD . .  P1194 P1209
    1258   320-PRINT-COMMUTER-RECORD. . .  P1195 P1206 P1212 P1222
    1318   330-PRINT-REPORT . . . . . . .  P1208 P1232 P1286 P1310 P1370 P1395 P1399
    1342   400-PRINT-TRANSACTION-ERRORS .  P896
```

Figure 62. Example of XREF Output—Data-Name Cross-References

The numbers used in explanation below correspond to the numbers in Figure 62.

Cross-Reference of Data-Names

1 Line number where the name was defined.

2 Data-name.

3 Line numbers where the name was used. If an “M” precedes the line number, the data-item was explicitly modified at the location.

Cross-Reference of Procedure References

4 Explanations of the context usage codes for procedure references.

5 Line number where the procedure-name is defined.

6 Procedure-name.

7 Line numbers where the procedure is referenced and the context usage code for the procedure.

The XREF compiler option also provides you with a sorted cross-reference listing of program names in your main program.

```
PP 5639-B92 IBM VisualAge COBOL (OS/2)  2.2            Date 02/27/1998  Time 12:26:53   Page     4

    1         2                               3

 Defined   Cross-reference of programs     References


 EXTERNAL   EXTERNAL1. . . . . . . . . . . 25
       2   X. . . . . . . . . . . . . . .  41
      12   X1 . . . . . . . . . . . . . .  33 7
      20   X11. . . . . . . . . . . . . .  25 16
      27   X12. . . . . . . . . . . . . .  32 17
      35   X2 . . . . . . . . . . . . . .  40 8
```

Figure 63. Example of XREF Output - Program Cross-Reference

1 The line number where the program-name was defined. If the program is external, the word EXTERNAL will be displayed instead of a definition line number.

2 The program name.

3 Line numbers where the program is referenced.

Using an Embedded Cross-Reference

The XREF compiler option also provides you with a modified cross-reference embedded in the source listing. This embedded cross-reference provides the line number where the data-name or procedure-name was defined.

```
LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Map and Cross Reference
  ⋮
000878               procedure division.
000879                 000-do-main-logic.
000880                   display "PROGRAM IGYTCARA - Beginning".
000881                   perform 050-create-vsam-master-file.                                    932  1
000882                   perform 100-initialize-paragraph.                                       984
000883                   read update-transaction-file into ws-transaction-record                 204 340
000884                       at end
000885       1                  set transaction-eof to true                                      254
000886                   end-read.

  ⋮

000984                 100-initialize-paragraph.
000985                   move spaces to ws-transaction-record                                    IMP 340  2
000986                   move spaces to ws-commuter-record                                       IMP 316
000987                   move zeroes to commuter-zipcode                                         IMP 327
000988                   move zeroes to commuter-home-phone                                      IMP 328
000989                   move zeroes to commuter-work-phone                                      IMP 329
000990                   move zeroes to commuter-update-date                                     IMP 333
000991                   open input update-transaction-file                                      204
000992                       location-file                                                       193
000993                       i-o commuter-file                                                   181
000994                       output print-file                                                   217

  ⋮

001442               1100-print-i-f-headings.
001443
001444                   open output print-file.                                                 217
001445
001446                   move function when-compiled to when-comp.                               IFN 698  2
001447                   move when-comp (5:2) to compile-month.                                  698 640
001448                   move when-comp (7:2) to compile-day.                                    698 642
001449                   move when-comp (3:2) to compile-year.                                   698 644
001450
001451                   move function current-date (5:2) to current-month.                      IFN 649
001452                   move function current-date (7:2) to current-day.                        IFN 651
001453                   move function current-date (3:2) to current-year.                       IFN 653
001454
001455                   write print-record from i-f-header-line-1                               222 635
001456                         after new-page.                                                   138

  ⋮
```

Figure 64. Example of an Embedded Cross-Reference

1 The line number of the definition of the data-name or procedure-name in the program.

2 Special definition symbols. These symbols are:

UND The user-name is undefined

DUP The user-name is defined more than once

IMP An implicitly defined name, such as special registers and figurative constants

IFN An intrinsic function reference

EXT An external reference

***** The program-name is unresolved because the NOCOMPILE option is in effect

Debugging User Exit Modules

To debug a user exit routine, you must invoke the debugger on the main compiler module rather than COB2.EXE. This is because the main compiler module is a separate process started by cob2, and the debugger can debug only one process.

To do this, first invoke cob2 with the -# option to see how cob2 invokes the main compiler module and what options it passes. For example, given the following cob2 invocation for compiling PGMNAME.CBL with the IWZRMGUX user exit and linking it:

```
cob2 -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
```

modify the cob2 invocation as follows:

```
cob2 -# -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
```

This is what you'll see:

```
igyccob2 -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
ilink  /free /nol /pm:vio pgmname.obj
```

You are interested in the IGYCCOB2 invocation because it is what actually calls your user exit.

You can debug the user exit as follows:

```
idbug igyccob2 -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
```

The debugger will automatically stop at the beginning of your user exit, assuming you built the exit with debug information.

Debugging Assembler Routines

The debugger will automatically go to the Disassembly view if the module being debugged does not have debug information. If the module is an assembler routine, of course you can debug only in this view. You can set a breakpoint at a disassembled statement in the Disassembly view by double clicking in the prefix area. It should be noted that by default, during startup the debugger will run until it hits the first debuggable statement it finds. If instead you would like the debugger to stop at the very first instruction in the application (debuggable or not), you must use the "-i" option. For example:

```
IDBUG -i progname
```

Resolution to Common Problems

This section outlines common problems that you might encounter and how to resolve them.

System Message SYS1808

OS/2

Problem

During execution my program terminates with the following system message:

SYS1808: The process has stopped. The software diagnostic code (exception code) is 0005.

When the program is run under the debugger, the exception XCPT_ACCESS_VIOLATION occurs in the program initialization code (at the first step from the PROGRAM-ID statement).

Solution

Your program probably has a very large LOCAL-STORAGE SECTION or LINKAGE SECTION which is causing a stack overflow. Recompile the program with the STACK linker option. See “Compiling and Linking Programs” on page 142 for a full description.

Example

```
cob2 -B"/STACK:4000000"  MYPROG.CBL
```

OS/2

Part 3. Object-Oriented Programming Topics

This part of the book covers object-oriented programming topics. Object-oriented programs are built from some new syntax plus the basic programming topics covered in Part 1, “Coding Your Program” on page 1.

Chapter 14. Writing Object-Oriented Programs

Object-oriented programs are based on classes and methods for objects. A class is a template defining the data structure and capabilities of an object. The data structure is commonly called instance data and the capabilities are commonly called methods. Usually, a program creates and works with multiple *object instances* of a class. Each instance has its own instance data and uses the methods defined for its class.

Consider a mail-order catalog business in which customers call service representatives to place orders. The service representatives are working with a user interface on the computer and creating an order. Therefore, in this situation there are two classes: user interface and order. Because there are many service representatives each processing a different customer's order, there are multiple instances of the two classes existing simultaneously.

Once classes are determined, the next step is to determine the methods the classes need to do their work. The order class must provide the following services:

- Add items to the order
- Delete items from the order
- Calculate the cost of the order
- Provide the order number to the service representative
- Write the final order for later processing

The following methods for the order class meet the above need:

AddItem
Add an item to the order

DeleteItem
Delete an item from the order

CalculateCost
Calculate the cost of the order

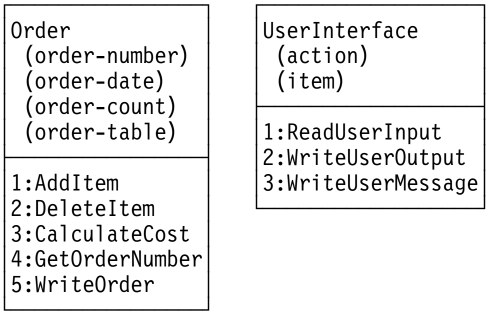
GetOrderNumber
Provide the order number

WriteOrder
Write the final order

As you design your class and its methods, you discover the need for the class to keep some instance data. Typically, an order class needs the following instance data:

- Order number
- Order date
- Number of items in the order
- Table of items ordered

Diagrams are very helpful when designing classes and their methods. The following diagrams depict the order and user interface classes.

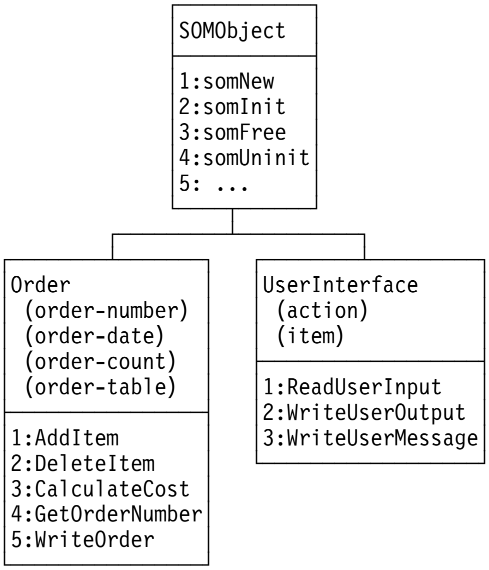


The words in parentheses are instance data and the words after the number and colon are methods.

The class structure of this object-oriented system is a tree structure. This structure shows how classes are related to each other and is known as the *inheritance hierarchy*. Order and user interface are basic classes, so they inherit from the System Object Model (SOM) base class, SOMObject.

Multiple Inheritance: All classes in COBOL inherit directly or indirectly from SOMObject. When multiple inheritance is used, the class structure might not be a tree—it may be an graph. However, the SOMObject class will always be at the root of the tree or graph.

The complete class structure for the mail-order catalog system is diagramed as follows:



More Methods: SOMObject has many methods other than the four listed here. See *SOMobjects Developer's Toolkit User's Guide* and *SOMobjects Developer's Toolkit Programmer's Reference Manual* for a complete description of all the SOM methods.

Writing a Class Definition

Like a COBOL program, a COBOL class definition consists of four divisions:

- IDENTIFICATION DIVISION

  The class name and class inheritance information are defined in this division.

- ENVIRONMENT DIVISION

  Associations between COBOL class names and SOM class names are defined in this division.

- DATA DIVISION

  Instance data is defined in this division.

- PROCEDURE DIVISION

  Methods are defined in this division.

Class IDENTIFICATION DIVISION: Required

In the IDENTIFICATION DIVISION of a class, you name a class and provide inheritance information for it. Optionally, you may give other identifying information. For example:

```
Identification Division.              Required
Class-Id.  Order INHERITS SOMObject.  Required
```

The AUTHOR, INSTALLATION, DATE-WRITTEN, and DATE-COMPILED paragraphs are optional and are treated as documentation.

CLASS-ID Paragraph

Use the CLASS-ID paragraph to:

- Name a class.

  In the example above, `Order` is the class name.

- Specify the System Object Model (SOM) base class or user-written class from which this class inherits its characteristics.

  In the example above, INHERITS `SOMObject` indicates `Order` inherits its basic characteristics from the base SOM class `SOMObject`.

- Name a metaclass.

  Discussed in “Writing a Metaclass Definition” on page 306.

`SOMObject` must be specified in the REPOSITORY paragraph in the ENVIRONMENT DIVISION (see “REPOSITORY Paragraph” on page 273). `Order` may optionally be specified in the REPOSITORY paragraph.

Class ENVIRONMENT DIVISION: Required

In the ENVIRONMENT DIVISION of a class, you describe the particular computer environment in which you are working and relate your class names to external SOM names. For example:

```
Environment Division.      Required
Configuration Section.     Required
Repository.                Required
    Class SOMObject is 'SOMObject'
    Class Order is 'Order'.
```

The SOURCE-COMPUTER, OBJECT-COMPUTER, and SPECIAL-NAMES paragraphs are optional. If they are specified in a class CONFIGURATION SECTION, they apply to the entire class definition, including all methods introduced by the class.

A class CONFIGURATION SECTION can consist of the same entries as a program CONFIGURATION SECTION, except the INPUT-OUTPUT SECTION. (See "CONFIGURATION SECTION" on page 13.)

REPOSITORY Paragraph

The REPOSITORY paragraph declares to the compiler that the specified user-defined word is a class name and optionally relates the class name to an external class name in the SOM interface repository. You must specify any class name you explicitly reference in your class definition in the REPOSITORY paragraph. For example:

- SOM base classes.

  In the example above, `CLASS SOMObject IS 'SOMObject'` indicates what you are calling `SOMObject` in your COBOL program is also called `SOMObject` in the SOM interface repository. All SOM names are mixed-case, so `SOMObject` spelled in mixed-case is required to properly handle SOM case sensitivity.

- User-written classes from which your class is inheriting.

  Discussed in "Writing a Subclass Definition" on page 290.

- Metaclass to which your class belongs.

  Discussed in "Writing a Metaclass Definition" on page 306.

- Any class referenced in methods introduced by the class.

You may optionally include the name of the class you are defining. If you do not include the name of your class, it is treated as all upper-case regardless of how you typed it on the CLASS-ID. In the example above, `Order` is stored in the SOM interface repository in mixed-case.

Class DATA DIVISION: Optional

In the DATA DIVISION of a class, you describe the instance data the class needs. For example:

```
Data Division.
Working-Storage Section.
01  order-number  PIC 9(5).
01  order-date    PIC X(8).
01  order-count   PIC 99.
01  order-table.
    02  order-entry  OCCURS 10 TIMES.
        03  order-item  PIC X(5).
```

A class DATA DIVISION contains only a WORKING-STORAGE SECTION.

WORKING-STORAGE SECTION

A class WORKING-STORAGE SECTION describes instance data that is statically allocated when the instance is created and exists until the instance is freed. By default, the data is global to all the methods introduced by the class. Instance data in a COBOL class is *private*. Thus, it cannot be referenced directly by any other class or subclass. See “Special Methods” on page 279 for an example of how to share instance data in COBOL.

Syntax of the class WORKING-STORAGE SECTION is generally the same as in a program. (described in “WORKING-STORAGE SECTION and LOCAL-STORAGE SECTION” on page 19).

Exceptions:

1. You cannot use the VALUE clause to initialize the data.

   Class instance data is initialized by overriding the 'somInit' method. See “somInit” on page 279 for an example using 'somInit'.

   Level-88 Note: You can have 88 level numbers with the VALUE clause.

2. You cannot use the EXTERNAL attribute.

3. You can use the GLOBAL attribute, but it has no effect.

Class PROCEDURE DIVISION: Optional

The class PROCEDURE DIVISION contains only method definitions. See “Writing a Method Definition” on page 276 for details about defining methods. A class definition must be properly terminated with an END CLASS statement. For example:

```
End Class Order.
```

marks the end of the `Order` class.

Complete Class Example

The class definition for the order class:

```
 IDENTIFICATION DIVISION.
*
*  Order is the name of the class
*  Order inherits from SOMObject (SOM base class)
*
 CLASS-ID.  Order INHERITS SOMObject.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
*
*  SOMObject is known as SOMObject in SOM repository
     CLASS SOMObject IS 'SOMObject'
*
*  Order is known as Order in SOM repository
     CLASS Order IS 'Order'.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
*  Instance data for Order class
*
 01  order-number  PIC 9(5).
 01  order-date    PIC X(8).
 01  order-count   PIC 99.
 01  order-table.
     02  order-entry  OCCURS 10 TIMES.
         03  order-item  PIC X(5).
 PROCEDURE DIVISION.
*
*  method definitions in here
*
 END CLASS Order.
```

The class definition for the user interface class:

```
 IDENTIFICATION DIVISION.
*
*  UserInterface is the name of the class
*  UserInterface inherits from SOMObject (SOM base class)
*
 CLASS-ID.  UserInterface INHERITS SOMObject.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
*
*  SOMObject is known as SOMObject in SOM repository
     CLASS SOMObject IS 'SOMObject'
*
*  UserInterface is known as UserInterface in SOM repository
     CLASS UserInterface IS 'UserInterface'.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
*  Instance data for UserInterface class
*
 01  uif-action  PIC X(10).
     88  uif-add     VALUE 'AddItem'.
     88  uif-delete  VALUE 'DeleteItem'.
     88  uif-quit    VALUE 'Quit'.
 01  uif-item    PIC X(5).
 PROCEDURE DIVISION.
*
*  method definitions in here
*
 END CLASS UserInterface.
```

Writing a Method Definition

A COBOL method can be defined only inside a class definition. Each method name within a class must be unique.

Like a COBOL program, a COBOL method definition consists of four divisions:

- IDENTIFICATION DIVISION

  The method name and whether it is overriding another method are defined in this division.

- ENVIRONMENT DIVISION

  Similar to a program ENVIRONMENT DIVISION.

- DATA DIVISION

  Similar to a program DATA DIVISION.

- PROCEDURE DIVISION

  Similar to a program PROCEDURE DIVISION.

Method IDENTIFICATION DIVISION: Required

Use the IDENTIFICATION DIVISION to name a method and indicate whether it is overriding another method from a superclass. Optionally, you can give other identifying information. For example:

```
Identification Division.     Required
Method-Id.  WriteOrder.      Required
```

The AUTHOR, INSTALLATION, DATE-WRITTEN, and DATE-COMPILED paragraphs are optional and are handled as documentation. You can use them for descriptive information about your method.

METHOD-ID Paragraph

Use the METHOD-ID paragraph to name the method. In the example above, `WriteOrder` is the method name. Other methods or programs use this name to invoke the method.

Method Override

Occasionally, a class defines a method whose name exists in a superclass. In this case, the superclass method must be overridden with the OVERRIDE clause on the METHOD-ID. System Object Model (SOM) provides two methods designed to be overridden. These SOM methods allow you to initialize instance data when an instance is created and save instance data when an instance is freed. The methods are called 'somInit' and 'somUninit' respectively. If you wish to override 'somInit', the IDENTIFICATION DIVISION is coded as follows:

```
Identification Division.          Required
Method-Id.  "somInit" Override.   Required
```

Method ENVIRONMENT DIVISION: Optional

The method ENVIRONMENT DIVISION has only one section, the INPUT-OUTPUT SECTION. The INPUT-OUTPUT SECTION relates your method files to the external file names known by the operating system. The syntax for a method INPUT-OUTPUT SECTION is the same as for a program INPUT-OUTPUT SECTION (see “INPUT-OUTPUT SECTION:” on page 16). For example:

```
Environment Division.
Input-Output Section.
File-Control.
    Select order-file Assign OrdrFile.
```

Method DATA DIVISION: Optional

A method DATA DIVISION consists of any of four sections:

- FILE SECTION

  A method FILE SECTION is the same as a program FILE SECTION except a method FILE SECTION can define only EXTERNAL files.

  (See “FILE SECTION (Using Data in Input/Output Operations)” on page 18 for more information.)

- LOCAL-STORAGE SECTION

  A separate copy of the data defined in the method LOCAL-STORAGE SECTION is allocated for each invocation of the method and is freed on the return from the method.

  If the VALUE clause is specified, the data item is initialized to the value on every invocation of the method.

  The method LOCAL-STORAGE SECTION is similar to a program LOCAL-STORAGE SECTION, except that the GLOBAL attribute has no effect.

  (See “WORKING-STORAGE SECTION and LOCAL-STORAGE SECTION” on page 19 for more information.)

- WORKING-STORAGE SECTION

  A single copy of the data defined in the method WORKING-STORAGE SECTION is allocated when the run-unit begins and persists in its last-used state until the run-unit terminates. The same single copy of the WORKING-STORAGE data is used whenever the method is invoked, regardless of the invoking object.

  If the VALUE clause is specified, the data item is initialized to the value on the first invocation of the method. The EXTERNAL clause may be specified for method WORKING-STORAGE data items (see “Sharing Data Using the EXTERNAL Clause” on page 399).

  A method WORKING-STORAGE SECTION is similar to a program WORKING-STORAGE SECTION except the GLOBAL attribute has no effect.

  (See “WORKING-STORAGE SECTION and LOCAL-STORAGE SECTION” on page 19 for more information.)

- LINKAGE SECTION

  A method LINKAGE SECTION is the same as a program LINKAGE SECTION except the GLOBAL attribute has no effect.

  (See “LINKAGE SECTION (Using Data from Another Program)” on page 21 for more information.)

See “Complete Class with Methods Example” on page 280 for a detailed example of a method DATA DIVISION.

If the same data item is defined in both the class DATA DIVISION and the method DATA DIVISION, a reference in the method to the data name refers to the data item in the method DATA DIVISION. The method DATA DIVISION takes precedence.

Method PROCEDURE DIVISION: Optional

In the PROCEDURE DIVISION of a method, you code the executable statements to complete the service the method is expected to provide. A method definition must be properly terminated with an END METHOD statement. For example:

```
 End Method WriteOrder.
```

marks the end of the 'WriteOrder' method.

The EXIT METHOD statement returns control to the invoking program or method. GOBACK has the same effect as EXIT METHOD. If the RETURNING clause is specified when the method is invoked, the EXIT METHOD or GOBACK returns the value of the data item to the invoking program or method. STOP RUN MAY be specified in a method; however, it terminates the run-unit.

An implicit EXIT METHOD is generated as the last statement of every method PROCEDURE DIVISION.

All COBOL statements that can be coded in a program PROCEDURE DIVISION can be coded in a method PROCEDURE DIVISION except:

- EXIT PROGRAM

- ENTRY statements
- The following obsolete elements of ANSI COBOL-85:
  - ALTER
  - GOTO without a specified procedure name
  - SEGMENTATION
  - USE FOR DEBUGGING

Special Methods

Simulated Attribute Methods: Instance variables in COBOL are all *private* in the sense that they are fully encapsulated by the class, and are accessible directly only by the methods that are introduced by the class that defines them. Normally, a well-designed object-oriented application does not need to access instance variables from outside the class.

The concept of a *public* instance variable, as defined in other object-oriented languages, and the concept of a class *attribute*, as defined by SOM and CORBA, are not directly supported by COBOL. (A CORBA attribute is an instance variable that has 'get' and/or 'set' methods to access and modify the value of the instance variable from outside the class definition.) A COBOL programmer can provide this capability by coding 'getX' and/or 'setX' methods for any instance variables X for which direct access from outside the class is required. The recommended naming convention for these methods is either 'getX' and 'setX' or perhaps 'get_X' and 'set_X'. Direct specification of method names (such as _get_X) is not recommended because such names are not valid in IDL, and use of such method names with the COBOL IDLGEN compiler option specified would result in an IDL file that will not compile with the SOM compiler. For example, this method

```
Identification Division.
Method-Id. 'getOrderNumber'.
Data Division.
Linkage Section.
01  ord-num  PIC 9(5).
Procedure Division returning ord-num.
    Move order-number To ord-num.
    Exit Method.
End Method 'getOrderNumber'.
```

passes the order number to any program that invokes 'getOrderNumber'.

somInit: The 'somInit' method is automatically invoked when an object instance is created. The default 'somInit' in SOM does nothing; however, you can override it to do your own initialization of instance variables. For example:

Method Example

```
 Identification Division.
 Method-Id. "somInit" Override.
 Procedure Division.
     Move Function Current-Date(1:8) To order-date.
     Move 0 To order-count.
     Initialize order-table.
     Exit Method.
 End Method "somInit".
```

somUninit: The 'somIninit' method is automatically invoked when an object instance is freed. The default 'somUninit' in SOM does nothing; however, you can override it if you wish. For example:

```
 Identification Division.
 Method-Id. "somUninit" Override.
 Data Division.
 Local-Storage Section.
 01  sub  Pic 99.
 Procedure Division.
     Display order-date.
     Perform varying sub from 1 by 1 until sub > order-count
        Display order-table (sub)
     End-Perform.
     Exit Method.
 End Method "somUninit".
```

The PROCEDURE DIVISION is discussed further in “PROCEDURE DIVISION” on page 22.

Complete Class with Methods Example

The class and method definitions for the order class:

```
 IDENTIFICATION DIVISION.
 CLASS-ID.  Orders INHERITS SOMObject.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
* Declare classes used in class defintion
 REPOSITORY.
     CLASS SOMObject IS 'SOMObject'
     CLASS Orders IS 'Orders'.

 DATA DIVISION.
* Define instance data
 WORKING-STORAGE SECTION.
 01  order-number  PIC 9(5).
 01  order-date    PIC X(8).
 01  order-count   PIC 99.
 01  order-table.
     02  order-entry  OCCURS 10 TIMES.
         03  order-item  PIC X(5).
 PROCEDURE DIVISION.

* Method to initialize instance data
*  - this overrides the default 'somInit' method
 IDENTIFICATION DIVISION.
 METHOD-ID.  'somInit' OVERRIDE.

 PROCEDURE DIVISION.
     MOVE FUNCTION CURRENT-DATE(1:8) TO order-date.
     COMPUTE order-number = FUNCTION RANDOM ( 99999 ).
     MOVE 0 TO order-count.
     INITIALIZE order-table.
     EXIT METHOD.
 END METHOD 'somInit'.

* Method to add an item to an order
 IDENTIFICATION DIVISION.
 METHOD-ID.  AddItem.

 DATA DIVISION.
* Use LOCAL-STORAGE for items that should be allocated
*  and initialized for each invocation of the method
 LOCAL-STORAGE SECTION.
 77  sub  PIC 99.
 01  found-flag  PIC 9  VALUE 1.
     88  found  VALUE 0.
 LINKAGE SECTION.
 01  in-item   PIC X(5).
 01  add-flag  PIC 9.

 PROCEDURE DIVISION USING in-item
                    RETURNING add-flag.
     MOVE 1 TO add-flag.
     PERFORM VARYING sub FROM 1 BY 1
             UNTIL (sub > 10) OR (found)
        IF order-item (sub) = SPACES
           MOVE in-item TO order-item (sub)
           ADD 1 TO order-count
           MOVE 0 TO add-flag
           SET found TO TRUE
        END-IF
     END-PERFORM.
     EXIT METHOD.
 END METHOD AddItem.

* Method to delete an item from an order
 IDENTIFICATION DIVISION.
 METHOD-ID.  DeleteItem.

 DATA DIVISION.
* Use LOCAL-STORAGE for items that should be allocated
*  and initialized for each invocation of the method
```

Method Example

```
 LOCAL-STORAGE SECTION.
 77  sub  PIC 99.
 01  found-flag  PIC 9  VALUE 1.
     88  found  VALUE 0.
 LINKAGE SECTION.
 01  out-item     PIC X(5).
 01  delete-flag  PIC 9.

 PROCEDURE DIVISION USING out-item
                    RETURNING delete-flag.
     MOVE 1 TO delete-flag.
     PERFORM VARYING sub FROM 1 BY 1
             UNTIL (sub > 10) OR (found)
        IF order-item (sub) = out-item
           MOVE SPACES TO order-item (sub)
           SUBTRACT 1 FROM order-count
           MOVE 0 TO delete-flag
           SET found TO TRUE
        END-IF
     END-PERFORM.
     EXIT METHOD.
 END METHOD DeleteItem.

* Method to compute the total cost of an order
 IDENTIFICATION DIVISION.
 METHOD-ID.  ComputeCost.

 DATA DIVISION.
* Use LOCAL-STORAGE for items that should be allocated
*  and initialized for each invocation of the method
 LOCAL-STORAGE SECTION.
 77  sub   PIC 99.
 77  cost  PIC 9(5)V99.
 LINKAGE SECTION.
 01  total-cost  PIC 9(7)V99.

 PROCEDURE DIVISION USING total-cost.
     MOVE 0 TO total-cost.
     PERFORM VARYING sub FROM 1 BY 1
             UNTIL sub > order-count
*  Call a subroutine
*   NOTE:  The subroutine code is not
*          included in this example.
        CALL 'InventoryGetCost'
           USING order-item (sub) cost
        ADD cost TO total-cost
     END-PERFORM.
     EXIT METHOD.
 END METHOD ComputeCost.

* Method to return the order number
 IDENTIFICATION DIVISION.
 METHOD-ID.  'getOrderNumber'.

 DATA DIVISION.
 LINKAGE SECTION.
 01  ord-num  PIC 9(5).

 PROCEDURE DIVISION RETURNING ord-num.
     MOVE order-number TO ord-num.
     EXIT METHOD.
 END METHOD 'getOrderNumber'.

* Method to write completed order to file
 IDENTIFICATION DIVISION.
 METHOD-ID.  WriteOrder.

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT order-file ASSIGN OrdrFile.

 DATA DIVISION.
 FILE SECTION.
* Methods support only EXTERNAL files
 FD  order-file  EXTERNAL.
 01  order-record  PIC X(80).
* Use LOCAL-STORAGE for items that should be allocated
*  and initialized for each invocation of the method
 LOCAL-STORAGE SECTION.
 01  print-line.
     02  print-order-number  PIC 9(5).
     02  print-order-date    PIC X(8).
     02  print-order-count   PIC 99.
     02  print-order-table.
         03  print-order-entry  OCCURS 10 TIMES.
             04  print-order-item  PIC X(5).

 PROCEDURE DIVISION.
     OPEN OUTPUT order-file.
     MOVE order-number TO print-order-number.
     MOVE order-date   TO print-order-date.
     MOVE order-table  TO print-order-table.
     MOVE order-count  TO print-order-count.
     WRITE order-record FROM print-line.
     CLOSE order-file.
     EXIT METHOD.
 END METHOD WriteOrder.

 END CLASS Orders.
```

The class and method definitions for the user interface class:

Method Example

```
 IDENTIFICATION DIVISION.
 CLASS-ID.  UserInterface INHERITS SOMObject.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
* Declare classes used in class definition
 REPOSITORY.
     CLASS SOMObject IS 'SOMObject'
     CLASS UserInterface IS 'UserInterface'.

 DATA DIVISION.
* Define instance data
 WORKING-STORAGE SECTION.
 01  uif-action  PIC X(10).
     88  uif-add     VALUE 'AddItem'.
     88  uif-delete  VALUE 'DeleteItem'.
     88  uif-quit    VALUE 'Quit'.
 01  uif-item    PIC X(5).

 PROCEDURE DIVISION.

*  Method to get input from customer - action and item
 IDENTIFICATION DIVISION.
 METHOD-ID.  ReadUserInput.

 DATA DIVISION.
 LINKAGE SECTION.
 01  action  PIC X(10).
 01  item    PIC X(5).

 PROCEDURE DIVISION USING item action.
     DISPLAY 'Enter the action:  add, delete, quit'.
     ACCEPT action FROM SYSIN.
     MOVE FUNCTION UPPER-CASE (action) TO action.
     EVALUATE TRUE
         WHEN action = 'ADD'
             SET uif-add TO TRUE
             PERFORM Get-Item
         WHEN action = 'DELETE'
             SET uif-delete TO TRUE
             PERFORM Get-Item
         WHEN action = 'QUIT'
             SET uif-quit TO TRUE
     END-EVALUATE.
     MOVE uif-action TO action.
     EXIT METHOD.

 Get-Item.
     DISPLAY 'Enter the item'.
     ACCEPT item FROM SYSIN.
     MOVE item TO uif-item.
 END METHOD ReadUserInput.

* Method to inform customer how action was completed
 IDENTIFICATION DIVISION.
 METHOD-ID.  WriteUserMessage.

 DATA DIVISION.
 LINKAGE SECTION.
 01  flag  PIC 9.

 PROCEDURE DIVISION USING flag.
     IF flag = 0
        DISPLAY uif-action
                ' successfully completed on '
                uif-item
     ELSE
        DISPLAY uif-action
                ' unsuccessfully completed on '
                uif-item
     END-IF.
     EXIT METHOD.

 END METHOD WriteUserMessage.

* Method to display final order information
 IDENTIFICATION DIVISION.
 METHOD-ID.  WriteUserOutput.

 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 77  formated-cost  PIC $Z,ZZZ,ZZ9.99.
 LINKAGE SECTION.
 01  total-cost    PIC 9(7)V99.
 01  order-number  PIC 9(5).

 PROCEDURE DIVISION USING total-cost order-number.
     MOVE total-cost TO formated-cost.
     DISPLAY 'Your order costs ' formated-cost.
     DISPLAY 'Your order number is ' order-number.
     EXIT METHOD.

 END METHOD WriteUserOutput.

 END CLASS UserInterface.
```

Writing a Client Definition

Any program that requests services from methods in a class is a client program. The client program consists of the usual four divisions:

- IDENTIFICATION DIVISION
- ENVIRONMENT DIVISION

- DATA DIVISION
- PROCEDURE DIVISION

Method Services: A method may request services from another method. Therefore, a method can be a client and use the statements discussed in this section.

Client IDENTIFICATION DIVISION: Required

The client IDENTIFICATION DIVISION is coded as usual.

Client ENVIRONMENT DIVISION: Required

In the ENVIRONMENT DIVISION of a client, you describe the particular computer environment in which you are working and relate your class names to external System Object Model (SOM) names. For example:

```
Environment Division.      Required
Configuration Section.     Required
Repository.                Required
    Client UserInterface is 'UserInterface'
    Client Orders is 'Orders'.
```

REPOSITORY Paragraph

The REPOSITORY paragraph declares to the compiler that the specified user-defined word is a class name and optionally relates the class name to an external class name in the SOM interface repository. You must specify any class name you explicitly reference in your program in the REPOSITORY paragraph. In the example above, `Orders` and `UserInterface` are the only two classes this program references.

Client DATA DIVISION: Optional

In the DATA DIVISION of a client, you describe the data the client needs. Since the client is using classes, it needs one or more special data items called *object references*. Object references are handles to instances of classes the program creates. All requests to a method are handled through an object reference to the instance of the class that defined the method. For example:

```
Data Division.
Working-Storage Section.
01  orderObj Usage Object Reference Orders.
01  userObj  Usage Object Reference UserInterface.
01  univObj  Usage Object Reference.
```

The phrase USAGE OBJECT REFERENCE indicates a data item is used as a handle for an instance.

In the above example, three object references are defined. The first two, `orderObj` and `userObj` are *typed* object references because a class name appears after the OBJECT REFERENCE phrase. Thus, `orderObj` can only be used to reference instances of the `Orders` class, or one of its subclasses. Likewise, `userObj` can only be used to reference instances of the `UserInterface` class, or one of its subclasses. The other object reference, `univObj`, does not have a class name after its OBJECT REFERENCE phrase. It is a *universal* object reference and can reference instances of any class.

Remember: Class names used on the OBJECT REFERENCE phrase must be defined in the REPOSITORY paragraph of the CONFIGURATION SECTION.

Client PROCEDURE DIVISION: Optional

The client PROCEDURE DIVISION contains code to create and free instances of classes, manipulate object reference data items, and invoke methods.

Creating and Freeing Instances of Classes

Before anything can be done with methods in a class, an instance of the class must be created. SOM provides a method, 'somNew', to create an instance of a class. For example:

```
Invoke Orders 'somNew' Returning orderObj.
```

creates an instance of the Orders class and assigns its handle to the object reference orderObj.

When 'somNew' executes it automatically invokes 'somInit', another SOM method, that you can override to initialize your instance data.

Remember: The class name, in this case Orders, must be defined in the REPOSITORY paragraph of the CONFIGURATION SECTION. And the object reference, in this case orderObj, must be defined as USAGE OBJECT REFERENCE in the DATA DIVISION.

When you finish with an instance of a class, you should free it. Again, SOM provides a method, 'somFree', to free the instance. For example:

```
Invoke orderObj 'somFree'.
```

frees the instance of orderObj; orderObj now has an undefined value. When 'somFree' executes it automatically invokes 'somUninit', another SOM method that you can override to save or display your instance data.

Manipulating Object References

Object references can be compared in conditional statements. For example:

```
If orderObj = Null ...
If orderObj = Nulls ...
If orderObj = univObj ...
```

are all valid uses of object references in an IF statement. The first and second IF statements check whether orderObj is a null object reference (refers to no instance). The third IF statement checks whether orderObj and univObj refer to the same instance.

Note: In a method there is a fourth form of object reference conditional:

```
If orderObj = Self ...
```

This checks whether the instance on which the method was invoked, SELF, refers to the same instance as orderObj.

It may be necessary to make an object reference null or make one object reference refer to the same instance as another object reference. The SET statement takes care of these situations:

Client PROCEDURE DIVISION

```
Set orderObj To Null.
Set univObj To orderObj.
```

In the first SET statement, orderObj is set to NULL.

In the second SET statement, univObj is made to refer to the instance to which orderObj refers. In this syntax, if the receiver (univObj) is a universal object reference then the sender (orderObj) can be either a universal or typed object reference. However, if the receiver is a typed object reference the sender must also be a typed object reference and typed to the same class or a subclass.

Note: In a method there is a third form of SET object reference:

```
Set orderObj To Self.
```

This makes the receiver (orderObj) refer to the same instance on which the method was invoked, SELF.

Invoking Methods

To receive service from a method, the method must be invoked with the INVOKE statement. For example:

```
Invoke Orders 'somNew' Returning orderObj.
Invoke orderObj 'AddItem' Using item Returning flag.
```

In the first INVOKE, a class name is used to create a new instance whose handle is returned in the object reference orderObj. The class name, Orders, must be defined in the REPOSITORY paragraph of the CONFIGURATION SECTION. The object reference, orderObj, must be define as either an universal object reference or a typed to class Orders object reference.

In the second INVOKE, an object reference, orderObj, is used to invoke the method AddItem. The general syntax of this form of INVOKE is one of the following:

```
Invoke objref 'literal-name'.
Invoke objref identifier-name.
```

In both cases the invoked method must be defined in the class for which the object reference, objref, is an instance. If the identifier-name form of the method is used, the object reference, objref, must by an universal object reference.

Conformance between the invoked method and the object reference is checked at compile time if the following three items are all true:

1. objref is a typed object reference.
2. The literal form of the method name is used in the INVOKE statement.
3. The TYPECHK compile option is specified.

Otherwise, conformance requirements are checked at run time. Run-time checking, however, is not as thorough as compile-time checking.

INVOKE has the optional scope terminator END-INVOKE. The USING and RETURNING phrases on the INVOKE work the same as they do on the CALL statement. Also, INVOKE has the optional ON EXCEPTION and NOT ON EXCEPTION phrases like the CALL statement. See *IBM COBOL Language Reference* for a discussion of USING, RETURNING, ON EXCEPTION, and NOT ON EXCEPTION.

The RETURN-CODE special register is not set by an INVOKE to a method.

Complete Client Example

A possible client program for the mail-order catalog using the Order and UserInterface classes:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  'PhoneOrders'.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*
*  Declare the classes used in the program
 REPOSITORY.
     CLASS Orders IS 'Orders'
     CLASS UserInterface IS 'UserInterface'.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
*  Declare the object references used in the program
 77  orderObj  USAGE OBJECT REFERENCE Orders.
 77  userObj  USAGE OBJECT REFERENCE UserInterface.
*
*  Declare other data items used in the program
 77  order-number  PIC 9(5).
 77  total-cost  PIC 9(7)V99.
 77  item  PIC X(5).
 77  action  PIC X(10).
 77  flag  PIC 9.

 PROCEDURE DIVISION.
*
*  Create an instance of the UserInterface class - userObj
     INVOKE UserInterface 'somNew' RETURNING userObj.
*
*  Create an instance of the Orders class - orderObj
     INVOKE Orders 'somNew' RETURNING orderObj.
*
*  Read customer input - action and item
     INVOKE userObj 'ReadUserInput' USING item action.

*
*  Begin customer driven loop based on action
     PERFORM UNTIL action = 'Quit'
*
*  Do appropriate action
```

```
         IF action (1:3) = 'Add'
            INVOKE orderObj 'AddItem' USING item
                                      RETURNING flag
         ELSE
            INVOKE orderObj 'DeleteItem' USING item
                                         RETURNING flag
         END-IF
*
*  Display result of action
         INVOKE userObj 'WriteUserMessage' USING flag
*
*  Read customer input - action and item
         INVOKE userObj 'ReadUserInput' USING item action
     END-PERFORM.
*  End customer driven loop based on action
*

*
*  Calculate the total cost of the order
     INVOKE orderObj 'ComputeCost' USING total-cost.
*
*  Determine the order number
     INVOKE orderObj 'getOrderNumber'
           RETURNING order-number.
*
*  Display information about the order
     INVOKE userObj 'WriteUserOutput'
         USING total-cost order-number.

*
*  Write the order to a file
     INVOKE orderObj 'WriteOrder'.

*
*  Free the object instances - orderObj and userObj
     INVOKE orderObj 'somFree'.
     INVOKE userObj 'somFree'.

     STOP RUN.
 END PROGRAM 'PhoneOrders'.
```

Writing a Subclass Definition

A subclass, sometimes called a child class, is a specialization of its superclass, sometimes called a parent class. The subclass is related to its superclass by an *is-a* type relationship. This means the phrase “Subclass S is a type of superclass P” makes sense within the application.

Subclassing has several advantages:

- Reuse of code.

A subclass can reuse methods already existing in another class through inheritance.

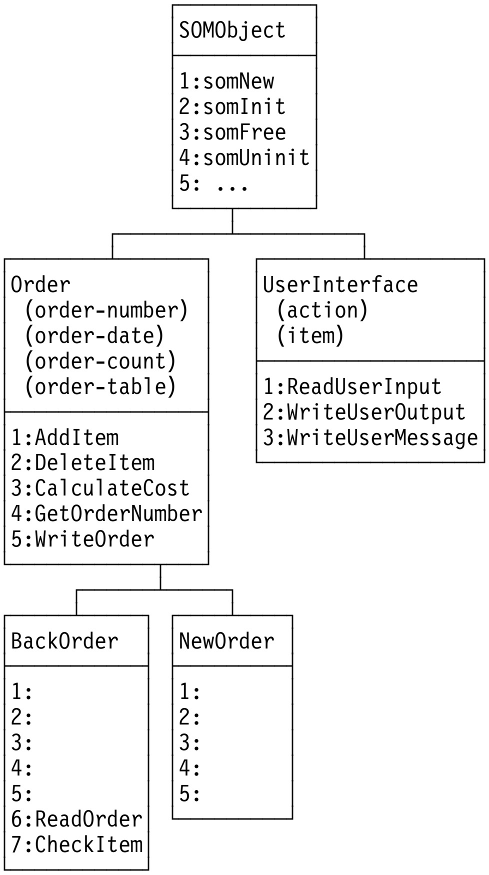
- More specific class.

  A subclass can add new methods to handle specific instances the superclass does not handle.

- Change in action.

  A subclass can override a method inherited from its superclass. Overriding can be anything from a few minor changes in how the method works to a complete overhaul of what the method does.

In the mail-order catalogue application, `Order` is a general class. One of the first things you discover working with `Order` is there are two kinds of orders: new order and back order. While both new order and back order have all the characteristics of order, back order also has the characteristic of requiring the order be read from the file and checking the status of the items. It might make sense to make new order and back order subclasses of order, diagramed as follows:



A number and colon with nothing after then represent a method inherited from a superclass.

In COBOL, a subclass inherits the methods from its superclass. A subclass may change, or override, one or more methods inherited from its superclass using the OVERRIDE clause on the METHOD-ID. Also, a subclass may add new methods it needs to perform its services.

In COBOL, instance data is private so the superclass must provide methods to allow the subclass to access instance data. A subclass can retrieve values from or store values in the instance data using the methods provided by the superclass. A subclass may also introduce new instance data of its own.

Multiple inheritance, inheriting from more than one superclass, is allowed in COBOL. Should there be a conflict in method names between two superclasses, the conflict is resolved according to the System Object Model (SOM) rules. See *SOMobjects Developer's Toolkit User's Guide* for an example.

Subclass IDENTIFICATION DIVISION: Required

In the IDENTIFICATION DIVISION of a subclass, you name the subclass and provide inheritance information for it. Optionally, you may give other identifying information. For example:

```
Identification Division.               Required
Class-Id.  BackOrder INHERITS Order.   Required
```

CLASS-ID Paragraph

The CLASS-ID paragraph names the subclass and indicates from what superclass or superclasses the subclass inherits. In the example above, `BackOrder` is the class name. It inherits all the methods from `Order`. Also, it can access `Order` instance data if `Order` provides methods to get and set its instance data.

The name(s) of the superclass(es) must be specified in the REPOSITORY paragraph in the ENVIRONMENT DIVISION (see “REPOSITORY Paragraph”). `BackOrder` may optionally be specified in the REPOSITORY paragraph.

Subclass ENVIRONMENT DIVISION: Required

In the ENVIRONMENT DIVISION of a subclass, you relate your subclass and class names to external System Object Model (SOM) names. For example:

```
Environment Division.     Required
Configuration Section.    Required
Repository.               Required
    Class BackOrder is 'BackOrder'
    Class Order is 'Order'.
```

REPOSITORY Paragraph

The REPOSITORY paragraph relates your subclass and class names to the subclass and class names in the SOM interface repository. You must include:

- User-written classes from which your subclass is inheriting.
- Metaclass to which your subclass belongs.

  Discussed in “Writing a Metaclass Definition” on page 306.
- Any class referenced in methods introduced by the subclass.

You may optionally include the name of the subclass you are defining. If you do not include the name of your subclass, it is treated as all upper-case regardless of how you typed it on the CLASS-ID. In the example above, `BackOrder` is stored in the SOM interface repository in mixed-case.

Subclass DATA DIVISION: Optional

In the DATA DIVISION of a subclass, you describe any extra instance data the subclass needs. For example:

```
Data Division.
Working-Storage Section.
01  order-status  PIC X(3).
```

A subclass DATA DIVISION contains only a WORKING-STORAGE SECTION.

WORKING-STORAGE SECTION

A subclass WORKING-STORAGE SECTION describes instance data that is statically allocated when the instance is created and exists until the instance is freed. By default, the data is global to all the methods introduced by the subclass. Instance data in a COBOL subclass is *private*. Thus, it cannot be referenced directly by any other class or subclass.

Subclass PROCEDURE DIVISION: Optional

The subclass PROCEDURE DIVISION contains only method definitions. A subclass definition must be properly terminated with an END CLASS statement. For example:

```
End Class BackOrder.
```

marks the end of the `BackOrder` subclass.

Subclass Method IDENTIFICATION DIVISION: Optional

Use the IDENTIFICATION DIVISION to name a method and to optionally give other identifying information. The name of each method in a subclass must be unique. For example:

```
Identification Division.
Method-ID.  ReadOrder.
```

METHOD-ID Paragraph

Use the METHOD-ID PARAGRAPH to name the method. Other methods or programs use this name to invoke the method.

If the subclass defines a method whose name exists in a superclass the OVERRIDE clause must be specified on the METHOD-ID. For example :

Subclass Example

```
 Identification Division.
 Method-Id.  AddItem Override.
```

When an object reference that is a handle to the BackOrder subclass invokes AddItem, this method is invoked rather than the method in the superclass Order.

Note: In a method, a subclass can invoke an overridden superclass method by using the INVOKE form:

```
 Invoke Super 'AddItem'.
```

This invokes the method AddItem defined in the superclass rather than the method AddItem defined in the subclass.

In the case of multiple inheritance, a subclass may inherit several methods with the same name from different parents. To specify precisely which method from which parent is invoked use the following INVOKE form:

```
 Invoke Class-A of Super 'AddItem'.
```

This invokes the method AddItem defined in the superclass Class-A rather than the method AddItem defined in any other superclass or in the subclass.

Subclass Method ENVIRONMENT DIVISION: Optional

The subclass method ENVIRONMENT DIVISION is coded in the same way a class method ENVIRONMENT DIVISION is coded. See “Method ENVIRONMENT DIVISION: Optional” on page 277 for a discussion of the class method ENVIRONMENT DIVISION.

Subclass Method DATA DIVISION: Optional

The subclass method DATA DIVISION is coded in the same way a class method DATA DIVISION is coded. See “Method DATA DIVISION: Optional” on page 277 for a discussion of the class method DATA DIVISION.

If the same data item is used in both the subclass DATA DIVISION and the method DATA DIVISION, a reference in the method to the data name refers to the data item in the method DATA DIVISION. The method DATA DIVISION takes precedence.

Subclass Method PROCEDURE DIVISION: Optional

In the PROCEDURE DIVISION of a subclass method, you code the executable statements to complete the service the method is expected to provide. A subclass method definition must be properly terminated with an END METHOD statement. See “Method PROCEDURE DIVISION: Optional” on page 278 for information about coding a method.

Complete Subclass with Methods Example

The new class and method definitions for the user interface class:

Subclass Example

```
 IDENTIFICATION DIVISION.
 CLASS-ID.  UserInterface INHERITS SOMObject.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
* Declare classes used in class definition
 REPOSITORY.
     CLASS SOMObject IS 'SOMObject'
     CLASS UserInterface IS 'UserInterface'.

 DATA DIVISION.
* Define instance data
 WORKING-STORAGE SECTION.
 01  uif-action  PIC X(10).
     88  uif-add     VALUE 'AddItem'.
     88  uif-delete  VALUE 'DeleteItem'.
     88  uif-quit    VALUE 'Quit'.
 01  uif-item    PIC X(5).
 PROCEDURE DIVISION.

* Method to read customer input - request
 IDENTIFICATION DIVISION.
 METHOD-ID.  ReadUserRequest.

 DATA DIVISION.
 LINKAGE SECTION.
 01  request  PIC X(6).

 PROCEDURE DIVISION USING request.
     DISPLAY 'Enter the request:  new, status'.
     ACCEPT request FROM SYSIN.
     MOVE FUNCTION UPPER-CASE (request) TO request.
     EXIT METHOD.
 END METHOD ReadUserRequest.

* Method to read customer input for new request - action and item
 IDENTIFICATION DIVISION.
 METHOD-ID.  ReadUserInput1.

 DATA DIVISION.
 LINKAGE SECTION.
 01  action  PIC X(10).
 01  item    PIC X(5).

 PROCEDURE DIVISION USING item action.
     DISPLAY 'Enter the action:  add, delete, quit'.
     ACCEPT action FROM SYSIN.
     MOVE FUNCTION UPPER-CASE (action) TO action.
     EVALUATE TRUE
         WHEN action = 'ADD'
             SET uif-add TO TRUE
             PERFORM Get-Item
```

Subclass Example

```
         WHEN action = 'DELETE'
             SET uif-delete TO TRUE
             PERFORM Get-Item
         WHEN action = 'QUIT'
             SET uif-quit TO TRUE
     END-EVALUATE.
     MOVE uif-action TO action.
     EXIT METHOD.

 Get-Item.
     DISPLAY 'Enter the item'.
     ACCEPT item FROM SYSIN.
     MOVE item TO uif-item.
 END METHOD ReadUserInput1.

* Method to read customer input for status request - order number
 IDENTIFICATION DIVISION.
 METHOD-ID.  ReadUserInput2.

 DATA DIVISION.
 LINKAGE SECTION.
 01  acct-numb  PIC 9(5).

 PROCEDURE DIVISION USING acct-numb.
     DISPLAY 'Enter the account number'.
     ACCEPT acct-numb FROM SYSIN.
     EXIT METHOD.
 END METHOD ReadUserInput2.

* Method to inform customer how action was completed
 IDENTIFICATION DIVISION.
 METHOD-ID.  WriteUserMessage.

 DATA DIVISION.
 LINKAGE SECTION.
 01  flag  PIC 9.

 PROCEDURE DIVISION USING flag.
     IF flag = 0
        DISPLAY uif-action
                ' successfully completed on '
                uif-item
     ELSE
        DISPLAY uif-action
                ' unsuccessfully completed on '
                uif-item
     END-IF.
     EXIT METHOD.
 END METHOD WriteUserMessage.

* Method to display order information
 IDENTIFICATION DIVISION.
 METHOD-ID.  WriteUserOutput.
 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 77  formated-cost  PIC $Z,ZZZ,ZZ9.99.
 LINKAGE SECTION.
 01  total-cost    PIC 9(7)V99.
 01  order-number  PIC 9(5).
 PROCEDURE DIVISION USING total-cost order-number.
     MOVE total-cost TO formated-cost.
     DISPLAY 'Your order costs ' formated-cost.
     DISPLAY 'Your order number is ' order-number.
     EXIT METHOD.
 END METHOD WriteUserOutput.

* Method to display out of stock items
 IDENTIFICATION DIVISION.
 METHOD-ID.  WriteUserStatus.

 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 77  sub  PIC 99.
 LINKAGE SECTION.
 01  out-table.
     02  out-entry  OCCURS 10 TIMES.
         03  out-item  PIC X(5).
 01  out-count  PIC 99.

 PROCEDURE DIVISION USING out-table out-count.
     IF out-count > 0
        PERFORM VARYING sub FROM 1 BY 1
                UNTIL sub > out-count
           DISPLAY 'Out of stock '
                   out-item (sub)
        END-PERFORM
     END-IF.
     EXIT METHOD.
 END METHOD WriteUserStatus.

 END CLASS UserInterface.
```

The new class and method definitions for the order class:

```
 IDENTIFICATION DIVISION.
 CLASS-ID.  Orders INHERITS SOMObject.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
* Declare classes used in program
 REPOSITORY.
     CLASS SOMObject IS 'SOMObject'
     CLASS Orders IS 'Orders'.
```

Subclass Example

```
 DATA DIVISION.
* Define instance data
 WORKING-STORAGE SECTION.
 01  order-number  PIC 9(5).
 01  order-date    PIC X(8).
 01  order-count   PIC 99.
 01  order-table.
     02  order-entry  OCCURS 10 TIMES.
         03  order-item  PIC X(5).
 PROCEDURE DIVISION.

* Method to intiialize instance data
*  - this overrides the default 'somInit' method
 IDENTIFICATION DIVISION.
 METHOD-ID.  'somInit' OVERRIDE.

 PROCEDURE DIVISION.
     MOVE FUNCTION CURRENT-DATE(1:8) TO order-date.
     COMPUTE order-number = FUNCTION RANDOM ( 99999 ).
     MOVE 0 TO order-count.
     INITIALIZE order-table.
     EXIT METHOD.
 END METHOD 'somInit'.

* Method to set instance data read by subclass
 IDENTIFICATION DIVISION.
 METHOD-ID.  'setInstanceData'.

 DATA DIVISION.
 LINKAGE SECTION.
 01  in-order.
     02  in-order-number  PIC 9(5).
     02  in-order-date    PIC X(8).
     02  in-order-count   PIC 99.
     02  in-order-table.
         03  in-order-entry  OCCURS 10 TIMES.
             04  in-order-item  PIC X(5).

 PROCEDURE DIVISION USING in-order.
     MOVE in-order-number TO order-number.
     MOVE in-order-date   TO order-date.
     MOVE in-order-count  TO order-count.
     MOVE in-order-table  TO order-table.
     EXIT METHOD.
 END METHOD 'setInstanceData'.

* Method to get instance data and give it to subclass
 IDENTIFICATION DIVISION.
 METHOD-ID.  'getInstanceData'.

 DATA DIVISION.
 LINKAGE SECTION.
 01  out-order.
     02  out-order-number  PIC 9(5).
     02  out-order-date    PIC X(8).
     02  out-order-count   PIC 99.
     02  out-order-table.
         03  out-order-entry  OCCURS 10 TIMES.
             04  out-order-item  PIC X(5).

 PROCEDURE DIVISION USING out-order.
     MOVE order-number TO out-order-number.
     MOVE order-date   TO out-order-date.
     MOVE order-count  TO out-order-count.
     MOVE order-table  TO out-order-table.
     EXIT METHOD.
 END METHOD 'getInstanceData'.

* Method to add an item to an order
 IDENTIFICATION DIVISION.
 METHOD-ID.  AddItem.
 DATA DIVISION.

 LOCAL-STORAGE SECTION.
 77  sub  PIC 99.
 01  found-flag  PIC 9  VALUE 1.
     88  found  VALUE 0.
 LINKAGE SECTION.
 01  in-item   PIC X(5).
 01  add-flag  PIC 9.

 PROCEDURE DIVISION USING in-item
                    RETURNING add-flag.
     MOVE 1 TO add-flag.
     PERFORM VARYING sub FROM 1 BY 1
             UNTIL (sub > 10) OR (found)
        IF order-item (sub) = SPACES
           MOVE in-item TO order-item (sub)
           ADD 1 TO order-count
           MOVE 0 TO add-flag
           SET found TO TRUE
        END-IF
     END-PERFORM.
     EXIT METHOD.
 END METHOD AddItem.

* Method to delete an item from an order
 IDENTIFICATION DIVISION.
 METHOD-ID.  DeleteItem.

 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 77  sub  PIC 99.
 01  found-flag  PIC 9  VALUE 1.
```

Subclass Example

```
     88  found  VALUE 0.
 LINKAGE SECTION.
 01  out-item     PIC X(5).
 01  delete-flag  PIC 9.

 PROCEDURE DIVISION USING out-item
                    RETURNING delete-flag.
     MOVE 1 TO delete-flag.
     PERFORM VARYING sub FROM 1 BY 1
             UNTIL (sub > 10) OR (found)
        IF order-item (sub) = out-item
           MOVE SPACES TO order-item (sub)
           SUBTRACT 1 FROM order-count
           MOVE 0 TO delete-flag
           SET found TO TRUE
        END-IF
     END-PERFORM.
     EXIT METHOD.
 END METHOD DeleteItem.

* Method to compute the total cost of an order
 IDENTIFICATION DIVISION.
 METHOD-ID.  ComputeCost.

 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 77  sub   PIC 99.
 77  cost  PIC 9(5)V99.
 LINKAGE SECTION.
 01  total-cost  PIC 9(7)V99.

 PROCEDURE DIVISION USING total-cost.
     MOVE 0 TO total-cost.
     PERFORM VARYING sub FROM 1 BY 1
             UNTIL sub > order-count
*  Call a subroutine
*   NOTE:  The subroutine code is not
*          included in this example.
        CALL 'InventoryGetCost'
           USING order-item (sub) cost
        ADD cost TO total-cost
     END-PERFORM.
     EXIT METHOD.
 END METHOD ComputeCost.

* Method to return the order number
 IDENTIFICATION DIVISION.
 METHOD-ID.  'getOrderNumber'.

 DATA DIVISION.
 LINKAGE SECTION.
 01  ord-num  PIC 9(5).
 PROCEDURE DIVISION RETURNING ord-num.
     MOVE order-number TO ord-num.
     EXIT METHOD.
 END METHOD 'getOrderNumber'.

* Method to write completed order to a file
 IDENTIFICATION DIVISION.
 METHOD-ID.  WriteOrder.

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT order-file ASSIGN OrdrFile.

 DATA DIVISION.
 FILE SECTION.
 FD  order-file  EXTERNAL.
 01  order-record  PIC X(80).
 LOCAL-STORAGE SECTION.
 01  print-line.
     02  print-order-number  PIC 9(5).
     02  print-order-date    PIC X(8).
     02  print-order-count   PIC 99.
     02  print-order-table.
         03  print-order-entry  OCCURS 10 TIMES.
             04  print-order-item  PIC X(5).

 PROCEDURE DIVISION.
     OPEN OUTPUT order-file.
     MOVE order-number TO print-order-number.
     MOVE order-date   TO print-order-date.
     MOVE order-count  TO print-order-count.
     MOVE order-table  TO print-order-table.
     WRITE order-record FROM print-line.
     CLOSE order-file.
     EXIT METHOD.
 END METHOD WriteOrder.

 END CLASS Orders.
```

The subclass and method definitions for the new order subclass:

```
 IDENTIFICATION DIVISION.
 CLASS-ID.  NewOrders INHERITS Orders.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
* Declare classes used in subclass defintion
 REPOSITORY.
     CLASS NewOrders IS 'NewOrders'
     CLASS Orders IS 'Orders'.
```

Subclass Example

```
 DATA DIVISION.

 PROCEDURE DIVISION.

* All methods are inherited from superclass

 END CLASS NewOrders.
```

The subclass and method definitions for the back order subclass:

```
 IDENTIFICATION DIVISION.
 CLASS-ID.  BackOrders INHERITS Orders.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
* Declare classes used in subclass definition
 REPOSITORY.
     CLASS BackOrders IS 'BackOrders'
     CLASS Orders IS 'Orders'.
 DATA DIVISION.

 PROCEDURE DIVISION.

* Method to read back order from file
 IDENTIFICATION DIVISION.
 METHOD-ID.  ReadOrder.

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT backorder-file ASSIGN BackFile.

 DATA DIVISION.
 FILE SECTION.
 FD  backorder-file  EXTERNAL.
 01  backorder-record  PIC X(80).
 LOCAL-STORAGE SECTION.
 01  backorder.
     02  backorder-number  PIC 9(5).
     02  backorder-date    PIC X(8).
     02  backorder-count   PIC 99.
     02  backorder-table.
         03  backorder-entry  OCCURS 10 TIMES.
             04  backorder-item  PIC X(5).
 77  eof-flag  PIC 9  VALUE 1.
     88  eof  VALUE 0.
 LINKAGE SECTION.
 01  order-number  PIC 9(5).

 PROCEDURE DIVISION USING order-number.
     OPEN INPUT backorder-file.
     PERFORM UNTIL eof
        READ backorder-file INTO backorder
           AT END
              SET eof TO TRUE
           NOT AT END
              IF order-number = backorder-number
                 INVOKE SELF 'setInstanceData' USING backorder
              END-IF
        END-READ
     END-PERFORM.
     CLOSE backorder-file.
     EXIT METHOD.
 END METHOD ReadOrder.

* Method to check whether item is still not in stock
 IDENTIFICATION DIVISION.
 METHOD-ID.  CheckItem.

 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 01  backorder.
     02  backorder-number  PIC 9(5).
     02  backorder-date    PIC X(8).
     02  backorder-count   PIC 99.
     02  backorder-table.
         03  backorder-entry  OCCURS 10 TIMES.
             04  backorder-item  PIC X(5).
 77  sub  PIC 99.
 77  status-flag  PIC 9.
     88  in-stock   VALUE 0.
     88  out-stock  VALUE 1.
 LINKAGE SECTION.
 01  out-table.
     02  out-entry  OCCURS 10 TIMES.
         03  out-item  PIC X(5).
 01  out-count  PIC 99.

 PROCEDURE DIVISION USING out-table out-count.
     INVOKE SELF 'getInstanceData' USING backorder.
     MOVE 0 TO out-count.
     PERFORM VARYING sub FROM 1 BY 1
             UNTIL sub > backorder-count
*  Call a subroutine
*   NOTE:  The subroutine code is not
*          included in this example.
        CALL 'InventoryGetItem'
            USING backorder-item (sub) status-flag
        IF out-stock
           ADD 1 TO out-count
           MOVE backorder-item (sub) TO out-item (out-count)
        END-IF
     END-PERFORM.
```

Subclass Example

```
     EXIT METHOD.
 END METHOD CheckItem.

 END CLASS BackOrders.
```

A possible new client program:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  'PhoneOrders'.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*
*  Declare the classes used in the program
 REPOSITORY.
     CLASS NewOrders IS 'NewOrders'
     CLASS BackOrders IS 'BackOrders'
     CLASS UserInterface IS 'UserInterface'.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
*  Declare the object references used in the program
*  Note:  univObj is a universal object reference
 77  univObj  USAGE OBJECT REFERENCE.
 77  userObj  USAGE OBJECT REFERENCE UserInterface.
*
*  Declare other data items used in the program
 77  order-number  PIC 9(5).
 77  total-cost  PIC 9(7)V99.
 77  out-count  PIC 9(2).
 77  request  PIC X(6).
 77  action  PIC X(10).
 77  flag  PIC 9.
 77  item  PIC X(5).
 01  item-table.
     02  item-entry  OCCURS 10 TIMES.
         03  item-element  PIC X(5).

 PROCEDURE DIVISION.
*
*  Create an instance of the UserInterface class - userObj
     INVOKE UserInterface 'somNew' RETURNING userObj.
*
*  Read customer input - request
     INVOKE userObj 'ReadUserRequest' USING request.
*
*  What is the customer's request?
     IF request = 'STATUS'
        PERFORM CheckBackOrder
     ELSE
        PERFORM CreateNewOrder
     END-IF.
*
*  Free the instance of the UserInterface class - userObj
     INVOKE userObj 'somFree'.

     STOP RUN.

 CreateNewOrder.
*
*  Create an instance of the NewOrders class - univObj
     INVOKE NewOrders 'somNew' RETURNING univObj.
*
*  Read customer input - action and item
     INVOKE userObj 'ReadUserInput1' USING item action.

*
*  Begin customer driven loop based on action
     PERFORM UNTIL action = 'Quit'
*
*  Do appropriate action
         IF action (1:3) = 'Add'
            INVOKE univObj 'AddItem' USING item
                                     RETURNING flag
         ELSE
            INVOKE univObj 'DeleteItem' USING item
                                        RETURNING flag
         END-IF
*
*  Display result of action
         INVOKE userObj 'WriteUserMessage' USING OMITTED flag
*
*  Read customer input - action and item
         INVOKE userObj 'ReadUserInput1' USING item action
     END-PERFORM.
*  End customer driven loop based on action
*

*
*  Calculate the total cost of the order
     INVOKE univObj 'ComputeCost' USING total-cost.
*
*  Determine the order number
     INVOKE univObj 'getOrderNumber'
           RETURNING order-number.
*
*  Display information about the order
     INVOKE userObj 'WriteUserOutput'
         USING total-cost order-number.

*
*  Write the order to a file
     INVOKE univObj 'WriteOrder'.
```

```
*
*  Free the NewOrders instance - univObj
     INVOKE univObj 'somFree'.

 CheckBackOrder.
*
*  Create an instance of the BackOrders class - univObj
     INVOKE BackOrders 'somNew' RETURNING univObj.
*
*  Read customer input - order number
     INVOKE userObj 'ReadUserInput2' USING order-number.
*
*  Read the back-ordered information from a file
     INVOKE univObj 'ReadOrder' USING order-number.
*
*  Check whether the back-ordered items are now in stock
     INVOKE univObj 'CheckItem' USING item-table out-count.
*
*  Display the status of the back-ordered items
     INVOKE userObj 'WriteUserStatus' USING item-table out-count.
*
*  Free the BackOrders instance - univObj
     INVOKE univObj 'somFree'.

 END PROGRAM 'PhoneOrders'.
```

Writing a Metaclass Definition

A metaclass is a special type of class whose instances are called class-objects. Class-objects are the run-time objects that represent SOM classes. Object-oriented COBOL applications either use the default metaclasses provided automatically by the SOM environment, or explicit metaclass definitions may be provided for specialized purposes.

Metaclasses have their own methods and can have their own instance data. The most common use of a metaclass is to control how an instance of a class is created. The method in the metaclass that creates the instance of a class is a *constructor* method. Metaclasses are also useful when multiple instances of a class are created and data must be gathered from all the instances. See the *SOMobjects Developer's Toolkit User's Guide* and *SOMobjects Developer's Toolkit Programmer's Reference Manual* (available online) for further details on metaclasses and their uses.

In the mail-order catalogue application, `BackOrder` required the reading of a file to establish its instance data. Reading the file cannot be done by `somInit` because an order number is needed as a parameter. This is a good place to use a metaclass with a constructor method to create the instance of `BackOrder` and read the file.

Metaclass IDENTIFICATION DIVISION: Required

In the IDENTIFICATION DIVISION of a metaclass, you name the metaclass and provide inheritance information for it. Optionally, you may give other identifying information. For example:

```
Identification Division.                      Required
Class-Id.  MetaBackOrder INHERITS SOMClass.    Required
```

CLASS-ID Paragraph

The CLASS-ID paragraph names the metaclass and indicates from what base System Object Model (SOM) class the metaclass inherits. In the example above, `MetaBackOrder` is the class name. It inherits from the base SOM class `SOMClass`. All metaclasses inherit directly or indirectly from `SOMClass`.

`SOMClass` must be specified in the REPOSITORY paragraph in the ENVIRONMENT DIVISION (see "REPOSITORY Paragraph"). `MetaBackOrder` may optionally be specified in the REPOSITORY paragraph.

Metaclass ENVIRONMENT DIVISION: Required

In the ENVIRONMENT DIVISION of a metaclass, you relate your metaclass names to external SOM names. For example:

```
Environment Division.     Required
Configuration Section.    Required
Repository.               Required
    Class MetaBackOrder is 'MetaBackOrder'
    Class SOMClass is 'SOMClass'.
```

REPOSITORY Paragraph

The REPOSITORY paragraph relates your metaclass and class names to the metaclass and class names in the SOM interface repository. You must include:

- SOM base classes.

  In the example above, `CLASS SOMClass IS 'SOMClass'` indicates what you are calling SOMClass in your COBOL program is also called SOMClass in the SOM repository.

- User-written classes from which your metaclass is inheriting.

  Discussed in "Writing a Subclass Definition" on page 290.

- Any class referenced in methods introduced by the metaclass.

You may optionally include the name of the metaclass you are defining. If you do not include the name of your metaclass, it is treated as all upper-case regardless of how you typed it on the CLASS-ID. In the example above, `MetaBackOrder` is stored in the SOM interface repository in mixed-case.

Metaclass Method PROCEDURE DIVISION

Metaclass DATA DIVISION: Optional

In THE DATA DIVISION of a metaclass, you describe any instance data the metaclass needs. For example:

```
Data Division.
Working-Storage Section.
01  total-orders  PIC X(3).
```

A metaclass DATA DIVISION contains only a WORKING-STORAGE SECTION.

WORKING-STORAGE SECTION

A metaclass WORKING-STORAGE SECTION describes instance data that is statically allocated when the first instance of an object in the metaclass is created and exists until the COBOL run-unit terminates. By default, the data is global to all the methods introduced by the metaclass. Instance data in a COBOL metaclass is *private*. Thus, it cannot be referenced directly by any other class or metaclass.

Metaclass PROCEDURE DIVISION: Optional

The metaclass PROCEDURE DIVISION contains only method definitions. A metaclass definition must be properly terminated with an END CLASS statement. For example:

```
End Class MetaBackOrder.
```

marks the end of the MetaBackOrder metaclass.

Metaclass Method IDENTIFICATION DIVISION: Optional

The metaclass method IDENTIFICATION DIVISION is coded in the same way a class method IDENTIFICATION DIVISION is coded. See “Method IDENTIFICATION DIVISION: Required” on page 276 for a discussion of the class method IDENTIFICATION DIVISION.

Metaclass Method ENVIRONMENT DIVISION: Optional

The metaclass method ENVIRONMENT DIVISION is coded in the same way a class method ENVIRONMENT DIVISION is coded. See “Method ENVIRONMENT DIVISION: Optional” on page 277 for a discussion of the class method ENVIRONMENT DIVISION.

Metaclass Method DATA DIVISION: Optional

The metaclass method DATA DIVISION is coded in the same way a class method DATA DIVISION is coded. See “Method DATA DIVISION: Optional” on page 277 for a discussion of the class method DATA DIVISION.

If the same data item is used in both the metaclass DATA DIVISION and the method DATA DIVISION, a reference in the method to the data name refers to the data item in the method DATA DIVISION. The method DATA DIVISION takes precedence.

Metaclass Method PROCEDURE DIVISION: Optional

In the PROCEDURE DIVISION of a metaclass method, you code the executable statements to complete the service the method is expected to provide. For the most part, a metaclass method PROCEDURE DIVISION is coded in the same way a class method

PROCEDURE DIVISION is coded. See “Method PROCEDURE DIVISION: Optional” on page 278 for a discussion of the class method PROCEDURE DIVISION.

Constructor Method

A metaclass constructor method is usually invoked with a class name so the use of the following INVOKE form is needed in the constructor method to create an instance of the class:

```
Invoke Self 'somNew' Returning anObj.
```

This creates an instance of the class on which the method was invoked, SELF, and returns the handle to that instance in the object reference anObj.

Method Only: SELF can be used only in a method.

Changes to Class or Subclass Definitions

When a class or subclass uses an explicit metaclass, the name of the metaclass must be specified with the METACLASS IS clause in the CLASS-ID paragraph. For example:

```
Identification Division.
Class-Id.  BackOrder Inherits Order
                     Metaclass is MetaBackOrder.
```

Also, the name of the metaclass must be specified in the REPOSITORY paragraph of the CONFIGURATION SECTION. For example:

```
Environment Division.
Configuration Section.
Repository.
     Class MetaBackOrder Is 'MetaBackOrder'
     Class BackOrder Is 'BackOrder'
     Class Order Is 'Order'.
```

Changes to the Client Program

To use the metaclass constructor method, the client program invokes the constructor method instead of 'somNew'. For example:

```
Invoke BackOrder 'CreateObject' Using order-number Returning anObj.
```

The method CreateObject is defined in the metaclass for BackOrder. This method invokes somNew to create an instance, reads the data from the file using the order number, and returns the handle to the instance in the object reference anObj. See “Complete Metaclass with Methods Example” on page 310 for a detailed example of using a metaclass constructor method.

Any method in a metaclass can be invoked with the class name. For example:

```
Invoke BackOrder 'CountOrders'.
```

Or, a metaclass object reference can be defined as a handle to the metaclass. For example:

```
Working-Storage Section.
01  metaObj Usage Object Reference Metaclass BackOrder.
```

Metaclass Example

The object reference metaObj is a handle to the metaclass for BackOrder, not a handle to BackOrder itself.

The metaclass object reference is used as follows:

```
Procedure Division.
   .
   .
Invoke backObj 'somGetClass' Returning metaObj.
Invoke metaObj 'CountOrders'.
```

The first INVOKE statement invokes a SOM method somGetClass which takes an object reference, backObj, to an instance and returns an object reference, metaObj, for the metaclass to which backObj belongs.

The second INVOKE statement uses the object reference to the metaclass, metaObj to invoke the method CountOrders which is defined in the metaclass. See “Complete Metaclass with Methods Example” for a detailed example of using a metaclass method.

Complete Metaclass with Methods Example

The metaclass and method definitions for the back order subclass:

```
 IDENTIFICATION DIVISION.
 CLASS-ID.  MetaBackOrders INHERITS SOMClass.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
* Declare classes used in metaclass definition
 REPOSITORY.
     CLASS MetaBackOrders IS 'MetaBackOrders'
     CLASS BackOrders IS 'BackOrders'
     CLASS SOMClass IS 'SOMClass'.

 DATA DIVISION.
* Define instance data
 WORKING-STORAGE SECTION.
 01  status-count  PIC 99.
 PROCEDURE DIVISION.

* Method to initialize instance data
 IDENTIFICATION DIVISION.
 METHOD-ID.  'somInit' OVERRIDE.

 PROCEDURE DIVISION.
     MOVE 0 TO status-count.
     EXIT METHOD.
 END METHOD 'somInit'.

* Method to create and initialize instances of BackOrders
 IDENTIFICATION DIVISION.
 METHOD-ID.  CreateBackOrders.
 DATA DIVISION.
 LINKAGE SECTION.
 01  order-number  PIC 9(5).
 01  anObj  USAGE OBJECT REFERENCE.

 PROCEDURE DIVISION USING order-number RETURNING anObj.
     INVOKE SELF 'somNew' RETURNING anObj.
     INVOKE anObj 'ReadOrder' USING order-number.
     ADD 1 TO status-count.
     EXIT METHOD.
 END METHOD CreateBackOrders.

* Method to return the number of back orders processed
 IDENTIFICATION DIVISION.
 METHOD-ID.  CountBackOrders.

 DATA DIVISION.
 LINKAGE SECTION.
 01  out-count  PIC 9(2).

 PROCEDURE DIVISION RETURNING out-count.
     MOVE status-count TO out-count.
     EXIT METHOD.
 END METHOD CountBackOrders.

 END CLASS MetaBackOrders.
```

The new subclass and method definitions for the back order subclass:

```
 IDENTIFICATION DIVISION.
 CLASS-ID.  BackOrders INHERITS Orders
                       METACLASS MetaBackOrders.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
* Declare classes used in subclass definition
 REPOSITORY.
     CLASS MetaBackOrders IS 'MetaBackOrders'
     CLASS BackOrders IS 'BackOrders'
     CLASS Orders IS 'Orders'.

 DATA DIVISION.

 PROCEDURE DIVISION.

* Method to read back order from file
 IDENTIFICATION DIVISION.
 METHOD-ID.  ReadOrder.

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
```

Metaclass Example

```
 FILE-CONTROL.
     SELECT backorder-file ASSIGN BackFile.

 DATA DIVISION.
 FILE SECTION.
 FD  backorder-file  EXTERNAL.
 01  backorder-record  PIC X(80).
 LOCAL-STORAGE SECTION.
 01  backorder.
     02  backorder-number  PIC 9(5).
     02  backorder-date    PIC X(8).
     02  backorder-count   PIC 99.
     02  backorder-table.
         03  backorder-entry  OCCURS 10 TIMES.
             04  backorder-item  PIC X(5).
 77  eof-flag  PIC 9  VALUE 1.
     88  eof  VALUE 0.
 LINKAGE SECTION.
 01  order-number  PIC 9(5).

 PROCEDURE DIVISION USING order-number.
     OPEN INPUT backorder-file.
     PERFORM UNTIL eof
        READ backorder-file INTO backorder
           AT END
              SET eof TO TRUE
           NOT AT END
              IF order-number = backorder-number
                 INVOKE SELF 'setInstanceData' USING backorder
              END-IF
        END-READ
     END-PERFORM.
     CLOSE backorder-file.
     EXIT METHOD.
 END METHOD ReadOrder.

* Method to check whether item is still not in stock
 IDENTIFICATION DIVISION.
 METHOD-ID.  CheckItem.

 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 01  backorder.
     02  backorder-number  PIC 9(5).
     02  backorder-date    PIC X(8).
     02  backorder-count   PIC 99.
     02  backorder-table.
         03  backorder-entry  OCCURS 10 TIMES.
             04  backorder-item  PIC X(5).
 77  sub  PIC 99  VALUE 0.
 77  status-flag  PIC 9.
     88  in-stock   VALUE 0.
     88  out-stock  VALUE 1.
 LINKAGE SECTION.
 01  out-table.
     02  out-entry  OCCURS 10 TIMES.
         03  out-item  PIC X(5).
 01  out-count  PIC 99.

 PROCEDURE DIVISION USING out-table out-count.
     INVOKE SELF 'getInstanceData' USING backorder.
     MOVE 0 TO out-count.
     PERFORM VARYING sub FROM 1 BY 1
             UNTIL sub > backorder-count
*  Call a subroutine
*   NOTE:  The subroutine code is not
*          included in this example.
        CALL 'InventoryGetItem'
            USING backorder-item (sub) status-flag
        IF out-stock
           ADD 1 TO out-count
           MOVE backorder-item (sub) TO out-item (out-count)
        END-IF
     END-PERFORM.
     EXIT METHOD.
 END METHOD CheckItem.

 END CLASS BackOrders.
```

A possible new client program:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  'PhoneOrders'.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*
*  Declare the classes used in the program
 REPOSITORY.
     CLASS NewOrders IS 'NewOrders'
     CLASS BackOrders IS 'BackOrders'
     CLASS UserInterface IS 'UserInterface'.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
*  Declare the object references used in the program
 77  univObj  USAGE OBJECT REFERENCE.
*  Note:  metaObj is an object reference to a metaclass
 77  metaObj  USAGE OBJECT REFERENCE METACLASS BackOrders.
 77  userObj  USAGE OBJECT REFERENCE UserInterface.
*
*  Declare other data items used in the program
 77  order-number  PIC 9(5).
 77  total-cost  PIC 9(7)V99.
```

Metaclass Example

```
 77  out-count  PIC 9(2).
 77  request  PIC X(6).
 77  action  PIC X(10).
 77  flag  PIC 9.
 77  item  PIC X(5).
 01  item-table.
     02  item-entry  OCCURS 10 TIMES.
         03  item-element  PIC X(5).

 PROCEDURE DIVISION.
*
*  Create an instance of the UserInterface class - userObj
     INVOKE UserInterface 'somNew' RETURNING userObj.
*
*  Read customer input - request
     INVOKE userObj 'ReadUserRequest' USING request.
*
*  What is the customer's request?
     IF request = 'STATUS'
        PERFORM CheckBackOrder
     ELSE
        PERFORM CreateNewOrder
     END-IF.
*
*  Free the instance of the UserInterface class - userObj
     INVOKE userObj 'somFree'.

     STOP RUN.

 CreateNewOrder.
*
*  Create an instance of the NewOrders class - univObj
     INVOKE NewOrders 'somNew' RETURNING univObj.
*
*  Read customer input - action and item
     INVOKE userObj 'ReadUserInput1' USING item action.

*
*  Begin customer driven loop based on action
     PERFORM UNTIL action = 'Quit'
*
*  Do appropriate action
         IF action (1:3) = 'Add'
            INVOKE univObj 'AddItem' USING item
                                     RETURNING flag
         ELSE
            INVOKE univObj 'DeleteItem' USING item
                                        RETURNING flag
         END-IF
*
*  Display result of action
```

```
         INVOKE userObj 'WriteUserMessage' USING OMITTED flag
*
*  Read customer input - action and item
         INVOKE userObj 'ReadUserInput1' USING item action
     END-PERFORM.
*  End customer driven loop based on action
*

*
*  Calculate the total cost of the order
     INVOKE univObj 'ComputeCost' USING total-cost.
*
*  Determine the order number
     INVOKE univObj 'getOrderNumber'
           RETURNING order-number.
*
*  Display information about the order
     INVOKE userObj 'WriteUserOutput'
         USING total-cost order-number.

*
*  Write the order to a file
     INVOKE univObj 'WriteOrder'.

*
*  Free the NewOrders instance - univObj
     INVOKE univObj 'somFree'.

 CheckBackOrder.
*
*  Read customer input - order number
     INVOKE userObj 'ReadUserInput2' USING order-number.

*
*  Begin customer driven loop based order number
     PERFORM UNTIL order-number < 0
*
*  Create an instance of the BackOrders class (univObj) and
*   read the back order from a file using a metaclass method
        INVOKE BackOrders 'CreateBackOrders'
              USING order-number RETURNING univObj
*
*  Check whether the back-ordered items are now in stock
        INVOKE univObj 'CheckItem'
              USING item-table out-count
*
*  Display the status of the back-ordered items
        INVOKE userObj 'WriteUserStatus'
              USING item-table out-count
*
*  Read customer input - order number
        INVOKE userObj 'ReadUserInput2'
```

Metaclass Example

```
              USING order-number
     END-PERFORM.
*  End customer driven loop based on order number
*

*
*  Get an object reference to the metaclass
*  Note:  'somGetClass' is a SOM method
     INVOKE univObj 'somGetClass' RETURNING metaObj.
*
*  How many back orders were processed?
*  Note: Metaclass object reference to invoke metaclass method
     INVOKE metaObj 'CountBackOrders' RETURNING out-count.
*
*  Display number of back orders processed
     INVOKE userObj 'WriteUserMessage' USING out-count OMITTED.

*
*  Free the metaclass instance - metaObj
*  Note:  This also frees all BackOrders instances
     INVOKE metaObj 'somFree'.

 END PROGRAM 'PhoneOrders'.
```

Others the Same: Other programs stay the same as the subclass example on page 294.

Chapter 15. Using System Object Model (SOM)

System Object Model (SOM) is an object-oriented programming technology that allows class implementers to describe the *interface* for a class in a standard language called the Interface Definition Language (IDL). Unlike the object model found in most other object-oriented programming languages, SOM is language-neutral. It preserves the key object-oriented programming characteristics of encapsulation, inheritance, and polymorphism without requiring the implementer of a SOM class and user of a SOM class to use the same programming language.

Note: The object-oriented COBOL language support is based on OS/390 SOMobjects. This support is not available on VM/CMS.

SOM Interface Repository

The SOM Interface Repository (IR) is a database in which the SOM Compiler optionally creates and maintains class interface definitions. The SOM IR is used by the COBOL compiler when compiling object-oriented COBOL programs. When compiling a class definition or client program with the IDLGEN or the TYPECHK option, the interface information for referenced classes must be present in the IR. (All referenced classes are declared in the REPOSITORY paragraph of the CONFIGURATION SECTION.)

Accessing the IR

The interface repository files to be used are specified outside the COBOL program using a SOM environment variable. The environment variable that specifies the IR is SOMIR. This environment variable is set as follows:

```
set SOMIR=c:\mydir\mycls.ir
```

If you do not set the SOMIR environment variable, the IR emitter creates a file named "som.ir" in the current directory.

```
set SOMIR=c:\som\som.ir;c:\dept\dept.ir;c:\work\work.ir
```

In this case, `som.ir` is SOM's IR that is not updated, `dept.ir` is a stable department IR that is not updated, and `work.ir` is the working IR that is updated.

Note: You may need to update the SOMIR environment variable if you delete and reinstall IBM VisualAge COBOL, or install another product that updates it.

You may set SOMIR at the time you use it; however, it is easier to put the statement in the above example into your CONFIG.SYS (OS/2) or AUTOEXEC.BAT (Windows 95) file or set it in the System window (Windows NT). For more information see the Interface Repository chapter of the *SOMobjects Developer's Toolkit User's Guide* (available online).

Populating the IR

The IR can be populated with interface information from COBOL classes via the following procedure:

1. Compile the COBOL class definition with the COBOL compiler, specifying the IDLGEN compiler option.
2. Compile the IDL source files with the SOM compiler, using the IR emitter.

Some COBOL class definitions with complex interdependencies may have to be compiled in two steps. For example, there may be circular compilation order dependencies, such as when two class definitions each contain references to the other. Such complex configurations may be compiled with the following procedure:

1. Compile all of the COBOL class definition source files with the IDLGEN, NOTYPECHK, and NOCOMPILE compiler options. This generates IDL files for the class interfaces, but does not perform type checking or generate an object file.
2. Compile the IDL files with the SOM compiler, using the IR emitter. This populates the IR with the class interface information.
3. Compile the COBOL class definitions again, with the NOIDLGEN and TYPECHK compiler options. This final compile performs full type checking and generates object files.

Compiling IDL Files

Compile IDL files and populate the IR using the SOM Compiler (for example, SC command or JCL procedure) with the `-usir` option. For example:

```
sc -usir myclass.idl
```

The SOM Compiler, sc, is started with the file `myclass.idl` and `-usir` option, which means update the IR. The rightmost IR file in the SOMIR list is the one updated.

SOM Environment Variables

The following environment variables specify information that is needed by the SOM compiler, interface repository framework, and run time. For full details, see *SOMobjects Developer's Toolkit User's Guide* (available online).

SMINCLUDE Specifies where to look for `#include` files included by the .idl file being compiled.

```
set SMINCLUDE=.;c:\toolkt20\include;c:\som\include
```

SMTMP Specifies where to put intermediate output files. This directory should not be the same as the ones where input and output files are located.

```
set SMTMP=c:\tmp\garbage
```

SMEMIT Specifies which emitters the SOM compiler runs.

For a COBOL class the most frequent emitter is the .h emitter which produces a header file for use by a C client of the COBOL class.

```
set SMEMIT="h"
```

For example, the following series of statements

```
set SMEMIT="h"
sc -usir myclass.idl
```

directs the SOM Compiler to produce 'myclass.h', and populate the IR from the 'myclass.idl' input specification.

SOMIR Specifies the location of the interface repositories.

```
set SOMIR=c:\mydir\mycls.ir
```

As with the SOMIR environment variable, you can type these environment variables when you need them. However, it is easier to put the above “set” statements in your CONFIG.SYS file (OS/2) or AUTOEXEC.BAT (Windows 95) file or set them in the System window (Windows NT). For more information see the SOM Compiler chapter of the *SOMobjects Developer's Toolkit User's Guide* (available online).

System Object Model (SOM) Services

IBM COBOL implements a subset of the ANSI Object-Oriented COBOL syntax based on the SOM object-oriented engine. Not all essential object-oriented capabilities are implemented in native COBOL syntax. Instead, SOM application programming interfaces, methods and functions are used. For example, native COBOL syntax is available for class definitions, object-reference datatype, and method invocation. However object creation, destruction, initialization, and termination are handled by invoking SOM methods provided by the `SOMObject` and `SOMClass` classes. Many other SOM facilities are available to COBOL programmers either for direct use or for overriding and customizing. These are described in *SOMobjects Developer's Toolkit User's Guide* (available online).

SOM Methods and Functions

The following SOM methods and function are especially important to COBOL programmers:

somNew A method in SOMClass to create a new object instance of a class. During creation, `somInit` is invoked for customized initialization of the object.

somFree A method in SOMObject to free an object instance releasing the storage used. Prior to freeing storage, “somUninit” is invoked for customized uninitialization.

`somFree` must not be invoked to destroy an active object, that is, an object upon which a method has been invoked that has not yet returned control to the invoker.

somInit A method in SOMObject that has no default function, but may be overridden explicitly in a COBOL class definition to perform customized initializations when an object is created.

somUninit A method in SOMObject that has no default function, but may be overridden explicitly in a COBOL class definition to perform customized uninitialization (typically the inverse of the function performed by a customized `somInit`).

SOM Services

somGetClass A method in SOMObject that returns an object reference for the class-object associated with the metaclass of an object.

somIsObj A function that determines whether an object-reference refers to a valid object.

"somIsObj" returns a Boolean. While COBOL has no BOOLEAN data type, COBOL programmers can declare the return value as PIC X and test the value using a symbolic character or hex literal.

```
 ...
 Data Division.
 Working-Storage Section.
 01  somBoolean  Pic X.
     88  invalid-obj  Value X'00'.
     88  valid-obj    Value X'01'.
 Procedure Division.
     ...
     Call 'somIsObj' Using By Value anObj Returning somBoolean.
     If invalid-obj
      Display 'Object reference does not refer to a valid object'
     End-if.
     ...
```

Function Note: When compiling a program that calls a SOM function, such as somIsObj:

- The PGMNAME(MIXED) compiler option must be specified, because the API names are case-sensitive. Otherwise, the compiler will translate `somIsObj` to `SOMISOBJ`, and you will get an unresolved external reference.
- The SOM API functions use the SYSTEM linkage convention. Hence the CALLINT(SYSTEM) compiler option or the >>CALLINT SYSTEM directive must be in effect for the CALL statement.

Your invocations of SOM methods does not require any special considerations; the correct linkage conventions are used automatically for method invocations.

SOM Initialization

During initialization of programs using object-oriented features, the COBOL run-time system automatically initializes the SOM environment and creates class-objects for classes referenced in the application. Application programmers do not have to perform these initializations manually.

Class Initialization

The SOM architecture specifies that every SOM class exports an initialization function `<classname>NewClass`. Normally COBOL programmers do not use this function directly, but the function is available on all COBOL classes. The COBOL run-time system automatically initializes all classes referenced within a COBOL program by calling their class initialization functions prior to the execution of the first user-written COBOL statement in the PROCEDURE DIVISION.

The class initialization function has a case-sensitive name, thus any COBOL program that explicitly calls a class-initialization function must be compiled with PGMNAME(LONGMIXED).

If an external class-name is specified in the REPOSITORY paragraph for a class, then the external class-name is used to form the initialization function name. If an external class-name is not specified in the REPOSITORY paragraph for a class, then the class-name declared is processed to form a CORBA-compliant external class name and this name is used to form the class initialization function. In a CORBA-compliant external class-name:

- The name is folded to upper case
- Hyphens in the name are translated to zero (0)
- If the first character in the name is a digit
  - 1 through 9 are translated to A through I
  - 0 is translated to J

For example:

```
Identification Division.
Class-Id.  Employee inherits SOMObject.
Environment Division.
Configuration Section.
Repository.
    Class SOMObject is class "SOMObject".
 ...
End-Class Employee.

The class initialization function names in the above cases are:
   EMPLOYEENewClass
   SOMObjectNewClass
```

and

```
Identification Division.
Class-Id.  Employee inherits SOMObject.
Environment Division.
Configuration Section.
Repository.
    Class Employee is class "Employee"
    Class SOMObject is class "SOMObject".
 ...
End-Class Employee.

The class initialization function names in the above cases are:
   EmployeeNewClass
   SOMObjectNewClass
```

Class Interface Evolution

One of the benefits of SOM is that classes can undergo changes over time and retain backward binary compatibility, that is, not require recompilation of programs and classes that reference the changed class. Changes that can be made to classes without recompilation requirements include:

SOM Services

1. Adding new methods.
2. Changing the size of an object by adding or deleting instance data.
3. Inserting new parent classes above a class in the inheritance hierarchy.
4. Relocating methods upward in the class hierarchy.

The SOM engine provides several alternative mechanisms for method resolution (see *SOMobjects Developer's Toolkit User's Guide* (available online) for details). IBM COBOL uses SOM name-lookup resolution to invoke methods. Thus when COBOL methods are invoked from COBOL code, the somewhat more stringent recompilation requirements of the SOM offset-resolution mechanism are not applicable. For example, COBOL methods that are invoked with COBOL INVOKE statements are not subject to the restriction in item four above. A COBOL method may be relocated anywhere in a class hierarchy without requiring recompilation of the COBOL programs that invoke the method.

Methods defined in COBOL classes may be invoked from other languages, such as C code built with the SOM C emitter, that use offset-resolution. In this case, the standard SOM requirements apply. Note that COBOL does not provide language comparable to the SOM “release-order” mechanism, which is used to ensure methods can be added to a class definition without requiring recompilation of code that invokes the methods using offset-resolution. When adding methods to an existing COBOL class, it is recommended that the new methods be added at the *end* of the PROCEDURE DIVISION of the class definition, *after* all of the existing methods. This will ensure that any existing code invoking the original methods does not require recompilation.

Chapter 16. Using SOM IDL-Based Class Libraries

This chapter is intended for COBOL programmers who want to use SOM IDL-based[6] class libraries, either as clients of the class, or by specializing the class using subclassing.

The chapter assumes that you understand the System Object Model (SOM), at least conceptually, and know where to find more detailed documentation about SOM when you need it. It also assumes that you have access to the documentation for the particular class library that you are intending to use.

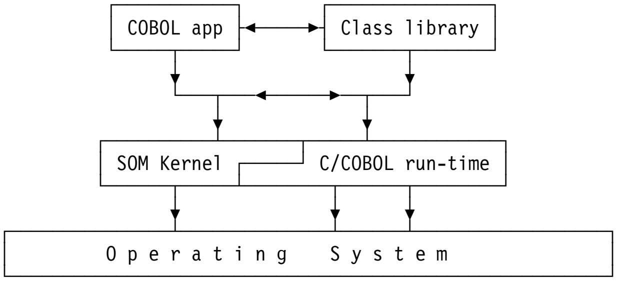
To get started, you need one of the object-enabled IBM COBOL products, together with the executables for the class library, plus its documentation, as above.

SOM Objects—a Refresher

A SOM class library consists of executable code and interface information that defines the operations that the library supports, including the parameters for invoking the operations—known as the operation “signatures.”

When the library is being used at run time, the components that are present in memory are illustrated in Figure 65.

Figure 65. Run-time Components of a SOM Class Library. The example also shows the COBOL components that are using the library.



[6] There are various forms of Interface Definition Language (IDL), such as those for the Distributed Computing Environment (DCE) or the Object Management Group's Common Object Request Broker Architecture (OMG CORBA), but in this chapter, we shall be concerned only with IBM's System Object Model Interface Definition Language (SOM IDL). SOM IDL is consistent with CORBA but also allows some additional data types, such as pointers, that are not included in CORBA.

Mapping IDL to COBOL

SOM IDL

The interface information for a SOM class library may be in various forms:

- IDL files;
- An Interface Repository (IR)—a machine-readable form of IDL, used during compilation;
- A book or on-line documentation describing the interfaces in IDL, together with operational descriptions of the methods.

IDL expresses the contract between the provider of object services, in this case the class library, and the user of these services: the COBOL program, method or subclass. The interface description is formally independent of the language in which either the user of the service or the service itself is implemented. This property is known as "language-neutrality." The separation of the interface from the implementation also allows flexibility in the deployment of the objects on the nodes of a network.

IDL data types have their origins in the C and C++ data model. Because many of them do not have an exact counterpart in the COBOL language, there needs to be a translation or "mapping" between IDL and COBOL. The mapping recommended here makes the explicit assumption that the data structures may be passed *directly*[7] between the COBOL and C/C++ mappings to SOM IDL.

Mapping IDL to COBOL

To use an IDL-based class library from COBOL, you must be able to map the elements of IDL in which it is expressed into the COBOL language. Typically, you find the description of the class library in a user's guide and reference, containing not only the guidelines for using the class library, but also the calling sequences for the individual methods expressed in SOM IDL. This, and the following sections, tell you how to map these IDL definitions into COBOL:

- "IDL Identifiers" on page 325 describes how to map IDL identifiers to COBOL names.
- "IDL Operations" on page 325 introduces IDL `operations`, which are described in more detail in "Operation Example" on page 342.
- "IDL Attributes" on page 326 explains how to express IDL `attributes` in COBOL.
- "Common IDL Types" on page 327 covers the "normal" IDL elements that virtually all interfaces use.

[7] The standard CORBA model presumes a "stub" routine between the invoking and invoked object to do argument translation, marshalling, and so on. Passing the structures directly yields very significant gains in efficiency, but it also means that some of the mappings may not seem as "natural" to the COBOL programmer as they would be if the transfer were mediated by a stub routine. It also means that you must ensure that you have the correct alignment and padding of any structures that are passed across an interface. In general the recommended way to achieve this for IDL-based interfaces is to specify the SYNCHRONIZED clause for COBOL mappings to any IDL `structs` or `arrays` that directly contain `structs`

- “Complex Types” on page 332 is provided for completeness; most interfaces do not use these complex IDL constructs.
- “Argument and Return Value Passing Conventions” on page 335 is a discussion of argument passing mechanisms.

IDL Identifiers

The only IDL names that must be *identical* in COBOL are the class (IDL interface) and method (IDL operation) names. These may be specified exactly by using literals:

- For a class, in the REPOSITORY paragraph using the CLASS IS clause.
- For a method, by using the literal form of the method name in the METHOD-ID paragraph. When you invoke a method, you use either the literal form of the name or a data name initialized with the exact method name.

The other identifiers, such as parameter, constant, and exception names, are internal to your program or class, and don't have to be identical to the IDL. However, it is a good idea to keep the spelling of these close to that of the IDL originals to enhance the readability and maintainability of your programs.

IDL Operations

IDL `operations` correspond with COBOL methods, and represent the services that an IDL interface provides. To use an operation, you code an INVOKE statement with the appropriate USING and RETURNING phrases that correspond with the parameters and the return value of the operation. If these parameters are all simple scalar types, the operation definition is self-contained. But if an operation uses one of the so-called “constructed” types, you may have to look elsewhere in the documentation for the definition of the parameter type in order to specify your INVOKE statement completely.

Consider the IDL operation definition:

```
void addColor(in color that);
```

The single input argument is of type `color`, which is not one of the basic scalar IDL types. Suppose that `color` is an IDL `enum` (see “enum” on page 328) with the following definition, typically found in a different section of the library documentation:

```
typedef enum color{red, white, blue};
```

Then the COBOL code to map the operation, adding the color `blue` to an object, could be written as follows:

```
1 color binary pic 9(9).
 88 red value 1.
 88 white value 2.
 88 blue value 3.
  ...
  Set blue to true
  Invoke anobject 'addColor' using by value evp color
```

The evp argument is the *environment pointer*, which precedes the explicit operation arguments. It is used for communicating back to the caller any exceptions that the operation encounters. See “Errors and Exceptions” on page 345 for a more detailed discussion.

IDL Attributes

An IDL attribute behaves like instance data that you can see outside the class definition (but note that there need not be any *actual* instance variable corresponding with it). Attributes are modeled as a pair of operations, one to set and one to get the attribute value. Attribute operations return errors by means of standard exceptions.

The mapping for attributes is best explained through an example. Consider the following IDL specification:

```
interface foo {
  struct position_t {
    float x, y;
  };

  attribute float radius;
  readonly attribute position_t position;
};
```

This is exactly equivalent to the following illegal[8] IDL specification:

```
interface foo {
  struct position_t {
    float x, y;
  };

  float _get_radius();
  void _set_radius(in float r);
  position_t _get_position();
};
```

The COBOL code to use these operations is straightforward:

```
         1 radius comp-1.
         1 position-t.
          2 x comp-1.
          2 y comp-1.
           ...
           Invoke a-foo '_get radius' using by value evp returning radius
           Invoke a-foo '_set_radius' using by value evp radius
           Invoke a-foo '_get_position' using by value evp
               returning position-t
```

[8] Illegal, because IDL identifiers are not permitted to start with an underscore (_) character.

Common IDL Types

These are the IDL types that you normally encounter in SOM IDL interfaces. The complex types are discussed in the next section.

Reference Summary

Figure 66. IDL Type to COBOL Mapping

IDL Type	COBOL Equivalent
`boolean`	`display picture x [+ level-88s...]`
`char`	`display picture x`
`double`	`computational-2`
`enum`	`binary picture 9(9) [+ level-88s...]`
`float`	`computational-1`
`interface`	`object reference`
`long`	`computational-5 or binary picture s9(9)`[1]
`octet`	`display picture x`
`pointer`	`pointer`
`short`	`computational-5 or binary picture s9(4)`[1]
`string`	`display pic x(n+1), z'value' or variable-length alphanumeric table`[2]
`unsigned long`	`computational-5 or binary picture 9(9)`[1]
`unsigned short`	`computational-5 or binary picture 9(4)`[1]

Note:

1. USAGE COMPUTATIONAL-5 data items are available only on the OS/2 and AIX versions of IBM COBOL. But if you map this IDL type to USAGE BINARY rather than COMP-5, you must either know that the expected range of values is accommodated by the PICTURE clause, or use the TRUNC(BIN) compiler option and, on the workstation or PC, ensure that the BINARY(NATIVE) compiler option is in effect.

   Also be aware that there are significant performance effects associated with the use of COMP-5 data items or the TRUNC(BIN) compiler option (which affects every USAGE BINARY data item in your program). So if you know that the picture accommodates the expected range of values, USAGE BINARY may be the better choice.

2. See “string” on page 330 for details.

boolean

The SOM IDL `boolean` type is mapped to a one-byte alphanumeric data item, together with suitable level-88 condition names. The condition names are recommended for convenience, but are not essential. For example, the following IDL:

```
boolean that;
```

could be written in COBOL as:

```
1 that display pic x.
 88 that-false value x'00'.
 88 that-true value x'01' thru x'ff'.
```

char

The SOM IDL `char` type maps to a one-byte alphanumeric data item. The IDL declaration:

```
char that;
```

could be written in COBOL as:

```
         1 that display pic x.
```

double

The SOM IDL `double` type represents 64-bit floating-point data, and is mapped to USAGE COMPUTATIONAL-2 in IBM COBOL. The IDL definition:

```
double that;
```

could be written in COBOL as:

```
         1 that comp-2.
```

enum

The closest COBOL equivalent to a SOM IDL `enum`[9] is an unsigned binary full-word, together with condition name entries for each of the enumeration members. For example, the following IDL:

```
enum that{red, white, blue, green};
```

could be written in COBOL as:

```
         1 that binary pic 9(9).
          88 that-red value 1.
          88 that-white value 2.
          88 that-blue value 3.
          88 that-green value 4.
```

In the unlikely event that any enumeration member exceeds 999,999,999, decimal picture considerations may apply—see the note in Figure 66 on page 327.

The way that you refer to a particular `enum` value in your PROCEDURE DIVISION depends on whether the value is supplied to an operation, or returned by it. See “Enum Type” on page 337 for more details.

[9] Note that a SOM IDL `enum` is different from a C/C++ `enum`:

1. it is always exactly four bytes long, whereas a C/C++ `enum` is one, two or four bytes long, depending on the maximum enum value and on compiler options;
2. the members are numbered sequentially starting from one, whereas a C/C++ `enum` starts at zero by default, or may optionally have specific values assigned to the enumeration members.

float

The SOM IDL `float` type represents 32-bit floating-point data and is mapped to COMPUTATIONAL-1 in IBM COBOL. For IDL:

```
float that;
```

you could write in COBOL:

```
       1 that comp-1.
```

interface

The use of an IDL `interface` as an argument to, or result of, an operation denotes an object reference to an instance of the class to which the interface has been mapped. In simple terms, if a method has an `interface` type as one of its parameters, specify an OBJECT REFERENCE to an instance of a class that supports that interface. Suppose you have the following skeletal IDL interface:

```
interface that {
  ...
  }
```

You would specify the corresponding class in the REPOSITORY paragraph as usual:

```
       Repository.
         class that 'that'.
```

Then you declare an instance of the `that` class:

```
       1 a-that object reference that.
```

and pass it according to the rules in “Argument and Return Value Passing Conventions” on page 335.

long

The SOM IDL `long` type describes 32-bit signed binary quantities, and is approximated by USAGE BINARY data items with a PICTURE clause of S9(9) in COBOL. On the workstation or PC, the type is exactly mapped by a USAGE COMPUTATIONAL-5 data item with a PICTURE clause of S9(5) through S9(9). Whichever mapping you use, be aware of the discussion in the note in Figure 66 on page 327.

For the IDL declaration:

```
long that;
```

you could write in COBOL:

```
       1 that binary pic s9(9).
```

or, on the workstation or PC:

```
       1 that comp-5 pic s9(9).
```

octet

The SOM IDL octet type represents an 8-bit quantity that is guaranteed to be unchanged during transmission between different objects. It is most closely matched in COBOL by a one-byte alphanumeric data item. Thus for IDL:

```
octet that;
```

you could write in COBOL:

```
         1 that display pic x.
```

pointer

A SOM IDL pointer corresponds with a COBOL POINTER data item. The IDL declaration:

```
pointer that;
```

would be written in COBOL as:

```
         1 that pointer.
```

short

The SOM IDL short type defines 16-bit signed binary data, and is most closely matched by COBOL data items with USAGE BINARY and a PICTURE clause of S9(4). On the workstation or PC, the type is exactly mapped by a USAGE COMPUTATIONAL-5 data item with a PICTURE clause of S9(1) through S9(4). Whichever mapping you use, be aware of the discussion in the note in Figure 66 on page 327.

The SOM IDL definition:

```
short that;
```

could be written in COBOL as:

```
         1 that binary pic s9(4).
```

or, on the workstation or PC:

```
         1 that comp-5 pic s9(4).
```

string

The SOM IDL type string is one of the most important types, since it is widely used in operations and interfaces. It is also one of the most difficult to match in COBOL, because SOM IDL strings are modeled on those of C and C++. These have a null terminator to determine the length of the string, and are passed via a typed pointer. IBM COBOL does not support such null-terminated strings as a native data type. However, the null-terminated literal — Z'string-value' — alleviates some of the problems and, when it can be passed BY CONTENT, is an exact match to a SOM IDL in `string`.[10] A null-terminated literal may also be used to set the initial VALUE of a data item to be used as a `string` argument.

In general though, IDL strings are mapped to pointers to the appropriate character string data or buffer. Except for `in` strings, what is actually passed in a method invocation is just the pointer. But, for this to work operationally, the pointer must be set to the address of the *underlying* character string data or buffer in PICTURE X format. There are several styles of data definition, depending on whether the parameter is an `in`, `inout`, or `out` argument, or a return value. These are discussed in more detail under “String Type” on page 338. For now, note that the declarations described below may be used to represent both the COBOL and SOM IDL view of a variable-length string simultaneously.

The main cases to distinguish are bounded and unbounded strings. Bounded strings have a fixed upper limit on their size. The IDL declaration:

```
string<100> that;
```

represents a SOM IDL string of no more than 100 characters in length, and may be approximated by the following COBOL data declarations:

```
1 that-l-max binary pic 9(9) value 101.
1 that-l binary pic 9(9).
1 that.
 2 that-v pic x occurs 1 to 101 depending that-l.
```

The “`-l`” suffix denotes the length of the string, the “`-v`” its value. Note the extra position to allow for the null terminator, and note that the data item `that`, in addition to being a valid SOM IDL string, is also variable-length in COBOL, because of the OCCURS DEPENDING clause.

For unbounded strings, the maximum length must be inferred from ancillary information about the interface and the semantics of its operations. Thus, for the IDL declaration:

```
string that;
```

which represents an unbounded SOM IDL string, you may, for example, know that the strings that are passed across the interface do not in practice exceed 4095 characters. Then the following COBOL declarations would be appropriate:

```
1 that-l-max binary pic 9(9) value 4096.
1 that-l binary pic 9(9).
1 that.
 2 that-v pic x occurs 1 to 4096 depending that-l.
```

See “Helper Routines Source Code” on page 356 for the C source code for a pair of “helper” routines to synchronize the two representations, for example, either the bounded or unbounded `that`, above:

- `'IDLStringToCOBOL' using that that-l`

[10] See the note in Figure 68 on page 336 about some restrictions on the use of BY CONTENT.

This routine sets the COBOL OCCURS subject `that-l` from the position of the mandatory null terminator.

If you prefer, you can achieve the same result in COBOL:

```
Move that-l-max to that-l
Move zero to tally
Inspect that tallying tally for characters before x'00'
Move tally to that-l
```

- `'IDLStringFromCOBOL' using that that-l`

  This routine inserts the null terminator at the string position indicated by the COBOL OCCURS object.

  If you prefer, you can do this quite easily yourself in COBOL:

  ```
  Move x'00' to that-v(that-l)
  ```

unsigned long and unsigned short

The unsigned forms of binary data are mapped as for SOM IDL types `long` and `short`, above, except that the picture clause does not specify the character “S.” Thus for the following IDL declarations:

```
unsigned long this;
unsigned short that;
```

you could write in COBOL:

```
1 this binary pic 9(9).
1 that binary pic 9(4).
```

or, on the workstation or PC:

```
1 this comp-5 pic 9(9).
1 that comp-5 pic 9(4).
```

void

When used as a return type for an operation, the IDL type `void` means that the operation doesn't return anything. You map this in COBOL merely by omitting the RETURNING phrase in the corresponding INVOKE statements or PROCEDURE DIVISION headers.

Complex Types

These are types that, although defined in SOM IDL, are rarely found as a type definition or as an argument to or result of an operation.

Reference Summary

Figure 67 (Page 1 of 2). IDL Type to COBOL Mapping

IDL Type	COBOL Equivalent
any	group + COBOL type
array	table

Figure 67 (Page 2 of 2). IDL Type to COBOL Mapping

IDL Type	COBOL Equivalent
sequence	group + variable-length table
struct	group
union	group + redefines

any

The IDL any type is a self-describing representation of any of the IDL types, including another IDL any. The descriptor is mapped to COBOL as a group item, which includes a pointer to the actual data item of the particular type. Suppose you want to map the following IDL declaration:

```
any that;
```

In COBOL, this would be represented by the following COBOL group item:

```
1 that.
 2 that-type pointer.
 2 that-value pointer.
```

The that-type field is a pointer to a “TypeCode” structure whose actual representation is opaque. SOM provides a set of functions to create and interrogate “TypeCode”s. A simple numeric type code is insufficient to describe an IDL type, because some types have additional information. For example, the type information for an IDL bounded string includes the size of the upper bound.

The that-value field points to the start of the data for the item that the any represents.

array

IDL arrays map to COBOL tables—groups whose subordinate items contain the OCCURS clause. The underlying IDL type can be any of the IDL types, including array itself, and is mapped according to the rules in this or the preceding section.

A simple instance of the IDL array type:

```
long that[4][5];
```

is represented in COBOL as:

```
1 that.
 2 occurs 4.
  3 that-v binary pic s9(9) occurs 5.
```

The “-v” suffix denotes the individual element values. The un-suffixed name is used to pass the entire array as an argument to a method; the suffixed name is used to refer to individual elements of the array.

You must specify the SYNCHRONIZED clause on the group item if the array contains a struct or union at any level of the array. This is to ensure that the subordinate items are aligned on their natural boundaries, in conformance with the default alignment of SOM IDL structures.

sequence

An IDL `sequence` is a one-dimensional array with a descriptor that specifies a maximum and current size for the sequence. If the maximum size is explicitly declared, the sequence is said to be "bounded." Otherwise, the sequence is "unbounded," and the maximum size is determined at run time (in an application-specific way) and is set prior to passing the sequence to an operation. There are no restrictions on the element type of a sequence. In particular, it is possible to have a sequence of another sequence type.

Let's look at a simple example, a bounded sequence of IDL type `long`:

```
sequence<long,10> that;
```

The descriptor for the maximum and current size and address of this sequence is represented in COBOL as a group item:

```
1 that.
 2 that-maximum binary pic 9(9).
 2 that-length binary pic 9(9).
 2 that-buffer pointer.
```

Then the element data are mapped as a variable-length table of the appropriate type, in this case, an array of IDL `long`s:

```
1 that-t.
 2 that-v binary pic s9(9) occurs 1 to 10
     depending that-length.
```

struct

An IDL `struct` type corresponds with a COBOL group item containing the individually mapped components of the `struct` as subordinate data items. Thus the following IDL `struct`:

```
struct that {
  long x;
  double y;
  };
```

could be represented in COBOL as:

```
1 that sync.
 2 x binary pic s9(9).
 2 y comp-2.
```

The SYNCHRONIZED clause is required so that the alignment of the subordinate items approximates the default alignment of SOM IDL structures. In most practical cases, the alignment would be correct either way, but specifying SYNCHRONIZED is a sensible precaution.

union

A SOM IDL union has a discriminator that indicates which format variant to use. In COBOL, this is mapped to a group item containing the discriminator, plus the union itself represented by using the REDEFINES clause. Then, in the PROCEDURE DIVISION, use an EVALUATE statement to determine which format is currently in effect.

An example should make all this more clear. Suppose you have the following IDL:

```
union that switch (long) {
  case 2:char x;
  case 5:long y;
  default:float z;
  };
```

The data declaration part of the COBOL mapping could be written as follows:

```
1 that sync.
 2 that-d binary pic s9(9).
 2 that-u display pic x(4).
 2 that-x redefines that-u display pic x.
 2 that-y redefines that-u binary pic s9(9).
 2 that-z redefines that-u comp-1.
```

The SYNCHRONIZED clause makes sure that COBOL mimics the default SOM IDL alignment rules. Thus, in the unlikely event that any IDL structures have “holes,” COBOL would insert slack bytes in the record as appropriate.

Notice the extra member of the union, that-u, whose size is the maximum of the sizes of the explicit union members. This is needed because of the COBOL restriction that a data item being redefined must be at least as large as the item redefining it. Alternatively, you could just declare the union members in order of decreasing size, although that may lose the correspondence between the COBOL declaration and the original IDL.

Whichever style you adopt, you can use an EVALUATE construct such as the following to determine which of the union members is currently in effect:

```
Evaluate that-d
  When 2
    Display 'case 2:IDL-char:  ' that-z
  When 5
    Display 'case 5:IDL-long:  ' that-x
  When other
    Display 'default case:IDL-float:  ' that-y
End-evaluate
```

Argument and Return Value Passing Conventions

This section describes how to write COBOL argument-passing constructs, such as BY REFERENCE or BY VALUE, to comply with the IDL access intent specifiers.

Argument/Result Passing Summary

Figure 68. Argument/Result Passing Conventions

IDL Type	in	inout/out	return value
`any`	`content`[1]	`reference`[1]	`type`[3]
`array`	`content`[1]	`reference`[1]	`pointer`[4]
`boolean`	`value`	`reference`[1]	`type`[3]
`char`	`value`	`reference`[1]	`type`[3]
`double`	`value`	`reference`[1]	`type`[3]
`enum`	`value`	`reference`[1]	`type`[3]
`float`	`value`	`reference`[1]	`type`[3]
`long`	`value`	`reference`[1]	`type`[3]
`object ref`	`value`	`reference`[1]	`type`[3]
`octet`	`value`	`reference`[1]	`type`[3]
`pointer`	`value`	`reference`[1]	`type`[3]
`short`	`value`	`reference`[1]	`type`[3]
`sequence`	`content`[1]	`reference`[1]	`type`[3]
`string`	`content`[1]	`pointer`[2]	`pointer`[4]
`struct`	`content`[1]	`reference`[1]	`type`[3]
`union`	`content`[1]	`reference`[1]	`type`[3]
`unsigned long`	`value`	`reference`[1]	`type`[3]
`unsigned short`	`value`	`reference`[1]	`type`[3]

Note:

1. For IBM COBOL for OS/390 & VM you can use BY CONTENT or BY REFERENCE only if the argument is not the last. This is due to the System/390 linkage conventions of the high-order bit of the last argument address being set ON to indicate the end of the argument list. This confuses C and C++ programs that attempt to manipulate the address as a pointer. An alternative to BY REFERENCE (for this situation and in general) is to pass BY VALUE a pointer that has previously been set to the address of the data item.
2. For `inout` and `out` strings, you must pass a pointer to the string data or buffer BY REFERENCE.
3. The term “type” means the COBOL equivalent of the IDL type, specified directly in the RETURNING phrase of the INVOKE statement.
4. The term “pointer” means a COBOL POINTER that has been set to the address of the equivalent data item or output buffer.

IDL Access Intent Specifiers

The IDL intent specifiers, `in`, `inout`, and `out`, do not correspond exactly with COBOL BY VALUE, BY CONTENT, and BY REFERENCE phrases. The IDL access intent determines only the semantics of the parameter, without necessarily implying a particular argument passing mechanism. In COBOL, both BY VALUE and BY CONTENT have input-only semantics but use different mechanisms, while BY REFERENCE parameters may have either input-output or output-only semantics, depending how they are used.

Some kinds of output parameters, IDL strings for example, cannot be expressed directly in COBOL, but must be mapped to pointers.

Literal Arguments

For return values, and for inout and out arguments, you must pass a data item. For input arguments however, you may be able to specify a literal, passed BY VALUE:

- Integer-valued fixed-point numeric literals and the figurative constant ZERO are formally equivalent to full-word binary data items, and thus match signed or unsigned long IDL types.
- Floating-point literals are formally equivalent to double-word floating-point (COMPUTATIONAL-2) data items, and thus match the IDL double type.
- Single-byte alphanumeric literals, symbolic characters, and figurative constants (other than ZERO) match boolean, char, and octet.

or BY CONTENT:

- Null-terminated literals of the form Z'value' match the IDL string type.

Note that literal arguments are not supported for enums, because of the risk of the source getting out of sync with the enumeration list.

Enum Type

The access intent of an enum parameter affects the way you refer to its value, not just on the INVOKE statement, but also elsewhere in your program. Consider an operation that expects an enum to be passed in both directions—as an input value and as the operation result:

```
that changeColor(in that hue);
typedef enum that{red, white, blue};
```

To supply a particular color to the operation, you use the corresponding condition name in a SET statement. For example, to pass the input value white, specify:

```
1 that-input binary pic 9(9).
 88 that-red-input value 1.
 88 that-white-input value 2.
 88 that-blue-input value 3.
  ...
  Set that-white-input to true
  Invoke anObject 'changeColor' using by value evp that-input
      returning that-output
```

Then, to inspect the returned color, use conditional statements:

```
1 that-output binary pic 9(9).
 88 that-red-output value 1.
 88 that-white-output value 2.
 88 that-blue-output value 3.
  ...
  Evaluate true
    When that-red-output
      Perform red-stuff
    ...
  End-evaluate
```

Complex Types

For the complex types (not including `string`, which is discussed separately below), you pass the level-1 group item. In the examples above, this is always the COBOL data name `that`. Where the conventions expect a pointer, this is set:

- For an argument, prior to executing the INVOKE statement
- For a return value, on return from the method

The rules for passing these types are quite involved and are described in some detail in the *SOMobjects Developer's Toolkit Programming Guide*. Generally, you provide the storage for all `in` and `inout` arguments, all modes of `struct` and `union` parameters, and, curiously enough, for `out array` arguments. The called method allocates some or all of the storage for all other `out` arguments and return values. You are not allowed to modify this returned storage, though you can, of course, use it otherwise, to copy it, for example. You must free it using `OMMFree` when you are finished with it. See “Helper Routines Source Code” on page 356.

If you have to supply `inout` arguments of any of the complex types, you would do well to allocate the storage dynamically, using `OMMAllocate`, and declare the COBOL equivalent type in the LINKAGE SECTION.[11] See “Memory Management” on page 352 for more information about `OMMAllocate` and `OMMFree`.

String Type

The examples in this section all presume the SOM IDL declaration:

```
string<100> that;
```

You pass `in string` types in several ways. Where the content is known, you could specify it as a null-terminated literal, either directly or as the value of a data item:

[11] This is because later versions of CORBA allow the called routine to *re-allocate* `inout` arguments when the output value is inconsistent with the type or size of the input data item. For this to work, a standard memory management protocol must be used by both the caller and the called routine.

```
 1 that pic x(101) value z'initial value'.
   ...
   Invoke anObject 'method'
       using by value evp by content z'this or that'
   ...
   Invoke anObject 'method' using by value evp by content that
```

For variable-content strings, you may find it convenient to use a plain (PICTURE X) alphanumeric data declaration for the string buffer. You can use reference modification if you want to see the valid part of the string:

```
 1 that-l binary pic 9(9).
 1 that pic x(101).
   ...
   Display 'Content of "that" = "' that(1:that-l) '"'
```

However, if you want the string to behave naturally as a variable-length string in both COBOL and the SOM IDL-based library, use the dual representations:

```
 1 that-l binary pic 9(9).
 1 that.
  2 that-v pic x occurs 1 to 101 depending that-l.
```

You can synchronize either the reference modified or the OCCURS DEPENDING form of the COBOL string representation with the IDL representation by using the `IDLStringToCOBOL` and `IDLStringFromCOBOL` helper routines given in “Helper Routines Source Code” on page 356.

For `inout` and `out` strings, you must pass the string buffer with an extra level of indirection. The way that you express the extra level of indirection in COBOL is to pass a pointer, which for `inout` strings has previously been set to the address of the string data. As usual, you have a choice of passing this pointer BY REFERENCE, or declaring a second pointer that you have set to the address of the first, then passing this second pointer BY VALUE:

```
   1 ptr1 pointer.
   1 ptr2 pointer.
   1 that-l binary pic 9(9).
     ...
 Linkage section.
   1 that.
    2 that-v pic x occurs 1 to 101 depending that-l.
     ...
     Set ptr1 to address of that
     Invoke anObject 'method' using by value evp by reference ptr1 ...
     ...
     Set ptr2 to address of ptr1
     Invoke anObject 'method' using by value evp ptr2
```

It's clear why the extra level of indirection is needed for `out` strings, given that they are allocated by the method and may be arbitrarily long. But the output size of an `inout` string is limited by the input size: the upper bound for a bounded string, and the actual input size for an unbounded string.[12]

For a return value, specify a pointer, which the method sets to the address of the output string before it returns.

For all the output modes of string, including `inout`, declare the string buffer itself in the LINKAGE SECTION, to allow the method to allocate, or re-allocate, the storage for the string. Storage for an `inout` string should be acquired by calling `OMMAllocate`, so that (in future) methods can re-size the string if necessary.

Be aware of your responsibilities for any storage allocated and returned by a SOM IDL-based library method. You can look at the storage, by declaring appropriate LINKAGE SECTION data items as usual, but do not attempt to modify it.[13] If you want to do that, make a copy of it and modify the copy. Also, note that you are responsible for freeing the storage for the original returned string when you have finished with it, by calling the `OMMFree` routine.

You can find the source code for `OMMAllocate` and `OMMFree` under “Helper Routines Source Code” on page 356.

Given the IDL definition:

```
interface this {
  string that (
    in string in_string,
    inout string inout_string,
    out string out_string
  );
};
```

and assuming an arbitrary limit of 99 characters on the string sizes, the following COBOL fragments illustrate these various techniques. This code is written in a very simple style, does not check for errors, and might not be complete.

[12] Because of this restriction, `inout` strings are not very useful, but you still need to pass the pointer rather than the string itself, because later versions of the CORBA argument conventions *do* allow reallocation of the string if the output is larger than the input.

[13] This is because the storage may be protected, and you cannot assume that you have appropriate write privileges.

```
Data division.
 Local-/Working-storage section.
  1 inout-string-p pointer.
  1 out-string-p pointer.
  1 return-string-p pointer.
    ...
  1 work-string-l binary pic 9(9).
  1 inout-string-l binary pic 9(9).
  1 out-string-l binary pic 9(9).
  1 return-string-l binary pic 9(9).
    ...
  1 work-string pic x(100).
  1 in-string pic(100) value z'Nothing strange for in strings'.
  1 evp pointer.
    ...
 Linkage section.
  1 inout-string.
   2 inout-string-v pic x occurs 1 to 100 depending inout-string-l.
  1 out-string.
   2 out-string-v pic x occurs 1 to 100 depending out-string-l.
  1 return-string.
   2 return-string-v pic x occurs 1 to 100 depending return-string-l.
  1 ev.
   2 major binary pic 9(9).
    88 no-exception value 0.
  ...
 Procedure division.
     ...
* Acquire storage for, and initialize, inout-string:
     Move 100 to inout-string-l
     Call 'OMMAllocate' using content length of inout-string
         returning inout-string-p
     Set address of inout-string to inout-string-p
     Move z'Initial value for inout-string' to inout-string
     Call 'IDLStringToCOBOL' using inout-string inout-string-l
* Invoke method 'that' on an instance of the 'this' class:
     Invoke a-this 'that' using by value evp
         by reference in-string inout-string-p out-string-p
         returning return-string-p
* De-reference the returned pointers and copy one string:
     Set address of inout-string to inout-string-p
     Call 'IDLStringToCOBOL' using inout-string inout-string-l
     Set address of out-string to out-string-p
     Call 'IDLStringToCOBOL' using out-string out-string-l
     Set address of return-string to return-string-p
     Call 'IDLStringToCOBOL' using return-string return-string-l
     Move out-string to work-string
* Operate on copy, and free allocated storage when done:
     Move function reverse(work-string) to work-string
     If work-string = out-string then
       Display '"' out-string '" is palindromic.'
     End-if
     ...
     Call 'OMMFree' using inout-string-p
     Call 'OMMFree' using out-string-p
     Call 'OMMFree' using return-string-p
     ...
```

Operation Example

This section illustrates the COBOL coding to use a very simple class library.

Let us begin by looking at the “documentation” for our class library, which provides a `bucket` class. A `bucket` is a container that lets you add or remove objects, and that can report the number of objects it contains. Buckets have no special initializer methods, and can thus be created and initialized correctly just by invoking the `somNew` method on the class. Normally, the documentation would define and describe each operation separately, but for this simple example, we will give the complete interface definition of a bucket:

```
interface Bucket {
  readonly attribute unsigned long count;
  void add(in SOMObject element) raises(BucketFull);
  SOMObject remove() raises(BucketEmpty);
};
```

The `raises` clause specifies the exceptions that the operation can incur.

The *things* that we put into our buckets have no external behavior beyond their existence. That is, they can be created and destroyed, and are identifiable by their object references, but they have no methods or attributes.

The COBOL program in Figure 69 shows how you might use this class library. It performs the following steps:

1. It creates an instance of a bucket.
2. It creates and adds some things to the bucket.
3. It prints the number of things in the bucket.
4. It removes a thing from the bucket.
5. It again prints the number of things in the bucket.

```
1   Process pgmname(longmixed)
    ******************************************************************
    * Client program for Bucket.                                     *
    ******************************************************************
     Identification division.
       Program-id. 'TryBucket'.
     Environment division.
      Configuration section.
       Repository.
         class thing 'Thing'
         class bucket 'Bucket'
         class somobject 'SOMobject'.
     Data division.
      Working-storage section.
2      1 evp pointer.
       1 abucket object reference bucket.
       1 athing object reference thing.
       1 asomobject redefines athing object reference somobject.
       1 cntnts binary pic 9(9).
      Linkage section.
3      1 ev.
        2 major binary pic 9(9).
         88 no-exception value 0.
         88 any-exception value 1 thru 999999999.
```

Figure 69 (Part 1 of 2). Complete Mapping Example

```
       Procedure division.
           display 'Trying Bucket...'
4          call 'somGetGlobalEnvironment' returning evp
           set address of ev to evp

5          invoke bucket 'somNew' returning abucket
           perform 5 times
6            invoke thing 'somNew' returning athing
7            invoke abucket 'add' using by value evp athing
             perform chkxcp
           end-perform

8          invoke abucket '_get_count' using by value evp returning cntnts
           perform chkxcp
           display 'Our bucket now has ' cntnts ' things in it.'
9          invoke abucket 'remove' using by value evp returning asomobject
           perform chkxcp
           invoke abucket '_get_count' using by value evp returning cntnts
           perform chkxcp
           display 'We took one out, so now it has only ' cntnts
               ' things in it.'

10         invoke abucket 'somFree'
           display 'Done with Bucket.'
           stop run.

         chkxcp.
           if any-exception
             display 'An exception occurred; quitting the program!'
             stop run
           end-if.

       End program 'TryBucket'.
```

Figure 69 (Part 2 of 2). Complete Mapping Example

Notice that the COBOL coding in this example is very simplistic. For example, it does not check for errors realistically, or even free all the objects that it creates. But it does cover most of the things that you have to do to start using a class library. Refer to the numbered keys in Figure 69 on page 343:

1 Regardless of what you call your program, you need to specify the PGMNAME(LONGMIXED) compiler option to be able to call SOM APIs such as `somGetGlobalEnvironment`. The option doesn't affect INVOKE statements, but it does apply to program names in CALL statements.

2, **3**, *and* **4** If not stated otherwise, SOM IDL class libraries use callstyle `idl`. With this convention, every operation has an implicit *environment pointer* preceding the explicit IDL arguments for the operation. Although this argument is implicit in the IDL, you code it explicitly on your INVOKE statements.

You must, at a minimum, define the environment pointer in the WORKING-STORAGE or LOCAL-STORAGE SECTION. If you want to examine any exceptions that are returned, you must also define the *exception type* in the LINKAGE SECTION, and set its base address to the value returned by somGetGlobalEnvironment. In the example, the exception type field is named major.

5 The somNew method creates an instance of the bucket class, and returns an object reference to the instance. Notice that this method does not take an environment pointer as its first argument.

6 Each time through the loop, a new thing is returned in the same variable. This is acceptable for the example, but normally, it would be very bad practice to lose addressability to one's objects. Among other reasons, the storage they use remains allocated and, without the object reference, cannot be freed.

7 For the methods that correspond to the IDL operations, the environment pointer is included as the first argument, evp, in the argument list. It's important to check for exceptions after invoking these methods. The ensuing PERFORM statement shows one way of doing that.

8 This statement shows how attributes are mapped to get/set methods. In this case, the attribute is readonly, so only the get method is defined.

9 It is a problem peculiar to container classes that they must allow arbitrary types for the elements that they contain. Thus the return type of the remove operation is specified as a SOMObject. We want to use the returned element with its proper description, to assure type safety. But coding a thing as the RETURNING value on the INVOKE statement would be a type violation. So the returned value, asomobject, is specified as a redefinition of athing. This allows the INVOKE statement to match the "signature" of the IDL operation. By using the redefined variable, athing, for any subsequent operations on the object, we can ensure that these operations are type safe.

10 This statement reminds us that all object instances that the program creates should be freed to avoid memory leaks. However, in this example none of the things in the bucket are freed.

Other SOM Topics

You will find it helpful to be familiar with the topics discussed in this section.

Errors and Exceptions

SOM uses two error or exception mechanisms: SOMError and CORBA-style exceptions.

SOMError is used for internal errors in the kernel classes, and is not really relevant to the average user. Methods of the kernel classes are used to create an object (somNew and somNewNoInit) or destroy it (somFree). The main implication of SOMError for these methods is that you don't have to provide an environment argument when you invoke them, and of course you don't have to check for exceptions after they return.

However you do need to know how to use the SOM exception mechanism, which is used for most other methods. Exceptions aren't necessarily errors, but errors *do* use the SOM exception mechanism.

Other SOM Topics

SOM exceptions are not the same as C++ exceptions, but instead set the value of an exception structure, which you can think of as a special kind of “return code,” accessed via the environment variable.

There are two ways of passing the environment variable, depending on the callstyle of the method you want to invoke, and check. For each of them, provide a global (per thread) environment variable, as shown in Figure 70 on page 347. For callstyle `oidl` methods, there's no explicit environment variable parameter. Such methods just use the global environment variable implicitly. Callstyle `idl` methods, on the other hand, use the same global environment variable, but passed explicitly, as the first argument to the method. Again, see Figure 70 on page 347 for an example of how this is accomplished in COBOL.

The environment variable is opaque, except for the exception type field (`major`) at the beginning of the structure. This is a four-byte C/C++ `enum`, origin zero, with three values: `NO_EXCEPTION`, `USER_EXCEPTION` and `SYSTEM_EXCEPTION` — and is coded in COBOL as BINARY PIC 9(9), with suitable level-88 condition names as illustrated in Figure 70 on page 347.

Every callstyle `idl` operation (that is, a method whose first parameter is an environment structure) can return one of the standard system exceptions[14], even if it does not declare any explicit exceptions with a `raises` expression in the `operation` declaration. This means that you must check the exception type field of the environment variable after *every* invocation of a method of a class defined with callstyle `idl`.

When one of these callstyle `idl` methods that you've invoked detects a condition that is to be expressed as an exception, it uses the `somSetException` function to supply the exception name and an exception structure.

Then, if you decide to handle the exception, perhaps by printing a message and continuing, you must reset the environment variable and free the associated exception structure by using the `somExceptionFree` function. Of course, there are other ways of handling exceptions. You may want to change the state of one of the input arguments to the method, then re-try it, or you might prefer to terminate your program rather than attempting to continue.

But in every case, at a minimum, check the successful completion of each method. It's not good programming practice to assume that a method completed successfully without checking it. You will not get a reliable implementation of your application unless you do so.

COBOL Example

The program fragments in Figure 70 on page 347 show in some detail how you can accomplish the foregoing in IBM COBOL. The data names are not mandatory; only suggestions.

[14] See the *SOMobjects Developer's Toolkit Programming Guide* for a list of the CORBA standard exceptions.

In the WORKING-STORAGE or LOCAL-STORAGE SECTION:

```
******************************************************************
* Declare the environment variable pointer:                      *
******************************************************************
   1 evp pointer.
```

In the LINKAGE SECTION:

```
******************************************************************
* Declare the environment variable itself:                       *
******************************************************************
   1 ev.
    2 major binary pic 9(9).
     88 no-exception value 0.
     88 any-exception value 1 thru 999999999.
     88 user-exception value 1.
     88 system-exception value 2.
```

In the PROCEDURE DIVISION:

```
******************************************************************
* Acquire a global environment variable                          *
******************************************************************
     Call 'somGetGlobalEnvironment' returning evp
     Set address of ev to evp
     ...
******************************************************************
* Check environment after invoking a method                      *
******************************************************************
     Invoke anObject 'op1' using by value evp other-args ...
     If any-exception then
*      respond to exception appropriately, perhaps by using:
       Call 'Print-ev' using evp by content z'op1 on anObject'
     End-if
     ...
```

Figure 70 (Part 1 of 3). Checking SOM Exceptions in COBOL

Here's a sample subroutine for printing out exceptions:

```
******************************************************************
* Subroutine for printing exceptions                             *
******************************************************************
 Identification division.
   Program-id.
     'Print-ev'.
 Data division.
  Working-storage section.
   1 counter binary pic 9(9) value 0.
  Local-storage section.
   1 d pic x(130).
   1 eip pointer.
   1 i binary pic 9(9).
   1 p binary pic 9(9).
   1 s pic 9(9).
  Linkage section.
   1 evp pointer.
   1 kind pic x(40).
   1 ev global.
    2 major binary pic 9(9).
     88 user-exception value 1.
     88 system-exception value 2.
   1 ei pic x(100).

 Procedure division using evp kind.
     Add 1 to counter
     Set address of ev to evp
     Call 'SLZ' using counter s i
     Move 1 to p
     String 'Check #' s(i : ) ':  method invocation "'
         delimited size into d pointer p
     Move 0 to i
     Inspect kind tallying i for characters before initial x'00'
     String kind(1 : i) '" returned '
         delimited size into d pointer p
     Evaluate true
       When user-exception
         String 'a user' delimited size into d pointer p
       When system-exception
         String 'a system' delimited size into d pointer p
       When other
         String 'an unknown' delimited size into d pointer p
     End-evaluate
```

Figure 70 (Part 2 of 3). Checking SOM Exceptions in COBOL

```
     Call 'SLZ' using major s i
     String ' exception (major = ' s(i : ) ')'
         delimited size into d pointer p
     Display d(1 : p - 1)
     Call 'somExceptionId' using by value evp returning eip
     Set address of ei to eip
     Move 0 to i
     Inspect ei tallying i for characters before initial x'00'
     Display '  Exception ID:  <' ei(1 : i) '>'
     Call 'somExceptionFree' using by value evp
     Goback
     .
 End program 'Print-ev'.

******************************************************************
* Subroutine to strip leading zeroes                             *
******************************************************************
 Identification division.
   Program-id.
     'SLZ'.
 Data division.
  Linkage section.
   1 uint binary pic 9(9).
   1 str pic x(9).
   1 pos binary pic 9(9).
 Procedure division using uint str pos.
     Move uint to str
     Move 0 to pos
     Inspect str(1 : length str - 1)
         tallying pos for leading '0'
     Add 1 to pos
     Goback
     .
 End program 'SLZ'.
```

Figure 70 (Part 3 of 3). Checking SOM Exceptions in COBOL

Initializers

IBM COBOL directly supports the existing somInit and somUninit protocols. For classes that use somInit, and this includes all pure COBOL classes, you can use the somNew method to create and initialize an object instance in one step. This is an appropriate technique when all instances have the same initial value, or do not have an explicit initial value at all. If, on the other hand, you want to parameterize object initialization, so that each object instance has a unique initial value, you may prefer the convenience of a metaclass; see the discussion of metaclasses later in this section.

You can execute the non-default initializer methods (as a client) of a class by using the documented technique of first invoking somNewNoInit, then invoking the appropriate initializer method explicitly. This is the recommended way of creating an instance of one of the SOM-enabled collections, for example.

You do need to know how to specify the so-called somInitCtrl structure that is used to control the progress of the initializer as it traverses the class hierarchy. For a client of a class initializer method (as opposed to a sub-class that provides its own initializer methods), this structure is initially null, represented in COBOL as an OMITTED argument. Suppose that the IDL for the initialization method is:

```
void ISHeap_withNumber(inout somInitCtrl ctrl, in long number);
```

then COBOL code for invoking this initializer might be:

```
1 a-heap object reference isheap.
  ...
  Invoke isheap 'somNewNoInit' returning a-heap
  Invoke a-heap 'ISHeap_withNumber'
      using by value evp by reference omitted by value 10000
```

For COBOL subclasses of classes that use explicit initializers, the recommended technique is to use metaclass methods to instantiate and initialize the COBOL object. After creating the instance, the metaclass method invokes the initializer for each parent (and with multiple inheritance, there may be several), then initializes any instance data introduced by the subclass itself. There is nothing to prevent you from doing this directly in the client code; but where it is possible to encapsulate the logic in a metaclass method, it is both more reliable and more convenient to do so, especially when the subclass inherits from multiple parents.

Using a metaclass method is also a good way of creating and initializing your own pure COBOL objects in a single step, particularly where each object may have a unique initial value.

If You Need to Look at the IDL File

Generally, the documentation for a class library has all the information you need to use (as a client) or specialize (subclass) the classes. In particular, you would expect to find the interfaces (types and operations or methods) expressed in IDL, together with the semantics of the operations and descriptions of the protocols for using the library. The operation definitions would include the required data types and argument passing conventions. The class library protocols are the rules for using the library: which objects must be instantiated, and in what order; what methods must be invoked to initialize the classes; what the relation between the classes is; and so on.

Sometimes, however, you may need more detailed information about a class library, for example when you are specializing the library by subclasssing. To get this additional information, you may need to look at the IDL or header files. It is then helpful to know their structure: what is relevant and what you can ignore. Consider the sample IDL file spred from the collection class library, shown in Figure 71 on page 351.

```
#ifndef _ISPRED_IDL  1
#define _ISPRED_IDL

#include <somobj.idl>  2

interface ISPredicate : SOMObject {  3

     boolean evaluateFor (in SOMObject element);  4

#ifdef __SOMIDL__  5
     implementation {
              releaseorder: evaluateFor;

              somDefaultInit: override,init;
              somDestruct:    override;

              callstyle      = idl;
              majorversion   = 1;
              minorversion   = 0;
              filestem       = spred;
              dllname        = "sccl.dll";
              functionprefix = sISPredicate_;

#ifdef __PRIVATE__  6
              passthru C_xh_before = "#include <ssglobal.xih>";
#endif
   };
#endif
};
#endif
```

Figure 71. Sample Collection Class IDL File

Typically, the IDL file consists of some IDL definitions, guarded so that they are processed only once per IDL compilation, plus some implementation-specific information, also guarded so that it is conditionally included. Refer to the numbered keys in Figure 71:

1 One of three conditional sections in the file; its purpose is to ensure that the IDL definitions in the file are processed no more than once during the IDL compilation. The matching `#endif` statement is at the end of the file.

2 The `#include` statement incorporates another IDL file that you may have to refer to.

3 The compound statement specifies the IDL element that this file defines, the `ISPredicate` interface.

4 This definition is for the (single) new operation `evaluateFor` that `ISPredicate` introduces.

5 The start of some SOM-specific implementation information, which needn't concern you; its matching `#endif` is the second to last.

6 A directive that is needed only by the implementation itself. Again it is not relevant to you, as a client of the class.

Memory Management

It is important to avoid memory leaks. This is particularly true with objects, because there are typically so many individual object instances created and destroyed. The idea is to ensure that, when an object is destroyed or assigned, all of its associated storage is also freed. “Helper Routines Source Code” on page 356 contains the source code for a pair of C routines that you can use to allocate and free dynamic storage for data that is pointed to by an object, for an `inout` argument to a method, and so on:

- `'OMMAllocate' using storage-size returning a-pointer`, to allocate storage, where `storage-size` is the 4-byte unsigned binary number of bytes to allocate;
- `'OMMFree' using a-pointer`, to free the previously allocated storage element that `a-pointer` addresses.

You *must* use `OMMFree` to free output storage allocated and returned to you by SOM class libraries. See “Complex Types” on page 338 and “String Type” on page 338 for details of how and when to do this.

You can also use these routines to manage dynamic storage (as opposed to instance data) for your own classes. Let's look at an example of a variable-length string class, where the string data is not an explicit part of the instance, but is instead a separate storage area that the instance *refers to*.

Here's the COBOL definition for the class:

```
******************************************************************
* COBOL variable-length string class definition.                 *
******************************************************************
 Identification division.
   Class-id.
     varstring inherits somobject.
 Environment division.
  Configuration section.
   Repository.
     Class varstring 'VarString'
     Class somobject 'SOMObject'
     .
```

Figure 72 (Part 1 of 4). COBOL Variable-Length String Class Example

```
      ******************************************************************
      * Variable-length string class instance data.                    *
      ******************************************************************
       Data division.
        Working-storage section.
         1 vstlen binary pic 9(9).
         1 vstptr pointer.

      ******************************************************************
      * Variable-length string class method:  default initialization;  *
      * set the instance to a predictable state.                       *
      ******************************************************************
       Identification division.
         Method-id.
           'somInit' override.
       Procedure division.
1          Set vstptr to null
           Move 0 to vstlen
           Goback
           .
       End method 'somInit'.
```

Figure 72 (Part 2 of 4). COBOL Variable-Length String Class Example

```
      *****************************************************************
      * Variable-length string class method:  assignment from a literal*
      *****************************************************************
       Identification division.
         Method-id.
           'SetVarstring'.
       Data division.
        Local-storage section.
         1 strsze pic 9(9) binary.
        Linkage section.
         1 valptr pointer.
         1 setval pic x(100).
         1 vstval pic x(100).
       Procedure division using by value valptr.
2          If vstptr not = null then
             Call 'OMMFree' using vstptr
           End-if
           Move 0 to vstlen
           Set address of setval to valptr
           Inspect setval tallying vstlen
               for characters before initial x'00'
           Add 1 to vstlen giving strsze
           Call 'OMMAllocate' using strsze returning vstptr
           Set address of vstval to vstptr
           Move setval(1:strsze) to vstval(1:strsze)
           Goback
           .
       End method 'SetVarstring'.

      *****************************************************************
      * Variable-length string class method:  return string (pointer). *
      *****************************************************************
       Identification division.
         Method-id.
           'GetVarstring'.
       Data division.
        Linkage section.
         1 valptr pointer.
       Procedure division returning valptr.
           Set valptr to vstptr
           Goback
           .
       End method 'GetVarstring'.
```

Figure 72 (Part 3 of 4). COBOL Variable-Length String Class Example

```
    ******************************************************************
    * Variable-length string class method: assign from another string*
    ******************************************************************
     Identification division.
       Method-id.
         'AssignVarstring'.
     Data division.
      Local-storage section.
       1 strsze binary pic 9(9).
       1 valptr pointer.
      Linkage section.
       1 str object reference varstring.
     Procedure division using by value str.
3        If self not = str then
           Invoke str 'GetVarstring' returning valptr
           Invoke self 'SetVarstring' using by value valptr
         End-if
         Goback
         .
     End method 'AssignVarstring'.

    ******************************************************************
    * Variable-length string class method:  free associated storage. *
    ******************************************************************
     Identification division.
       Method-id.
         'somUninit' override.
     Procedure division.
4        If vstptr not = null then
           Call 'OMMFree' using vstptr
           Set vstptr to null
           Move 0 to vstlen
         End-if
         Goback
         .
     End method 'somUninit'.

     End class varstring.
```

Figure 72 (Part 4 of 4). COBOL Variable-Length String Class Example

There are several points to notice about the use of storage in this example:

1 All VarString instances are created in a predictable initial state, with the length set to zero, and the string pointer set to null.

2 Before assigning a new value to an instance, the storage allocated for any current value is freed. If this weren't done, the storage would be “orphaned,” causing a memory leak.

3 When assigning one string to another, you have to check whether the sender is identical to the receiver before doing the assignment and thereby prematurely freeing the sender's storage.

4 Although somFree de-allocates the storage for the instance data, it does NOT free storage that the instance refers to. Thus it is critical to free any such storage when the instance is uninitialized.

Helper Routines Source Code

The C source in Figure 73 may be used to implement the helper functions for string representation and memory management discussed in this chapter. You can either statically link the functions into your application, or generate a dynamic load library (DLL) for the functions and bind your application to the DLL.

```
/*********************************************************************/
/* Helper functions for using SOM IDL-based class libraries.         */
/*********************************************************************/

/* OS/390 pragma to generate long, mixed-case names                  */
#pragma longname

/* Macro to clear the high-order bit of the argument address (OS/390  */
/* and VM)                                                          */
#define Clean(p,q) p=(void*)((int)q&0x7fffffff)
#include <som.h>

/* Object Memory Management:  allocate memory.                       */
somToken OMMAlloc(size_t *sze){
  size_t *s;
  Clean(s,sze);
  return SOMMalloc(*s);
}

/* Object Memory Management:  free allocated memory.                 */
void OMMFree(somToken *ptr){
  somToken *p;
  Clean(p,ptr);
  SOMFree(*p);
  return;
}

/* Set COBOL representation (ODO object) from IDL string length       */
void IDLStringToCOBOL(char *str, long *len) {
  char *s;
  long *l;
  Clean(s,str);
  Clean(l,len);
  (*l)=strlen(s);
  return;
}
```

Figure 73 (Part 1 of 2). C Source for IDL-based Library Helper Functions

```
/* Set IDL string length (null byte) from COBOL (ODO) representation  */
void IDLStringFromCOBOL(char *str, long *len) {
  char *s;
  long *l;
  Clean(s,str);
  Clean(l,len);
  s[*l]=0;
  return;
}
```

Figure 73 (Part 2 of 2). C Source for IDL-based Library Helper Functions

Chapter 17. Converting Procedure-Oriented Programs to Object-Oriented Programs

Conventional COBOL programs belong to one of three types:

- Batch
- Online
- Subprogram

Batch programs are often constructed to access files and/or databases and produce reports. In the access files and/or databases case, the file or database is the object of the action; however, the program is structured around the action—delete, insert, or update. In the produce reports case, the report can be viewed as an object; however, the program structure reflects the structure of the report.

Online transaction processing programs are built around the transactions which they process and these are reflected in the user interface maps or panels which make up the transaction. Online transactions may access several files or databases from one panel. In this case, there is a one to many relationship between the source of the action and the targets of the action, all of which can be viewed as objects.

Subprograms normally are called to provide a function too large or complex to include in the main program. They are also used to implement general purpose functions required by many other programs; in which case, they embody reusable code. In many cases, the subprogram alters the values of some parameters based on the values of other parameters. In other cases, the subprogram accesses files/databases or prints reports. In this case, the parameter list can be viewed as a message to trigger some action on a file or database object.

Wrapping a Procedure-Oriented Program

Wrapping is a technique to integrate existing procedure-oriented code with new object-oriented code. Two of the definitions for *wrap* are:

1. To enclose as if with a protective covering.
2. To conceal as if by enveloping.

Wrappers are objects that provide an interface between object-oriented code and procedure-oriented code. They enclose the procedure-oriented code in a package, concealing its true nature and making it seem like object-oriented code.

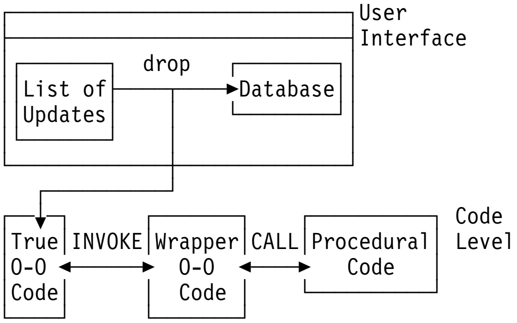
Wrappers are useful in two situations:

- Glass-top coordination
- Boundary interface coordination

Glass-top Coordination

This type of wrapper integrates the old and new code at the user interface or "glass-top" level. As user interfaces move toward a object-oriented approach, you find direct manipulation used more. This implies actions such as "dragging" and "dropping" objects into or onto other objects. When this occurs, the objects involved must work together to take the appropriate action. If one of the objects is really procedure-oriented code, the wrapper is an interface to the true objects.

For example, you have a stable set of procedural code for updating a database. However, you would like to include the database as part of a graphical user interface. The goal is to drop a list object representing an update to the database on the database object and have the update performed. You need to write a wrapper class to accept messages from the list object; i. e. the list object invokes methods in the wrapper. The methods in the wrapper class interpret the information from the list object and use the CALL statement to call the appropriate subprogram in the old procedural code.

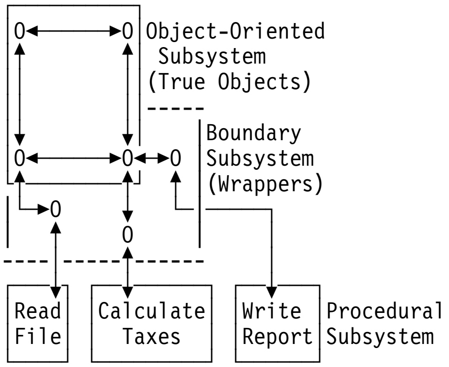


Boundary Interface Coordination

Boundary interface wrappers create objects for procedural code outside the boundaries of the new object-oriented subsystem. These wrappers allow the object-oriented part of the system to deal with the procedure-oriented part of the system as if it were object-oriented. Thus, you can phase in new object-oriented code and continue to use your existing procedure-oriented code.

You can write a wrapper for each subprogram in the procedural code. Or if several subprograms are working with the same object, processing the same file or producing the same report, you can write a single wrapper for all the related subprograms. The appropriate method in the wrapper is invoked from a true object and the method in turn calls the appropriate subprogram.

Converting



Required Change to Procedural Code

If you decide to use wrappers, there is one change you must make to your procedural code. Since methods are always recursive, it is possible to INVOKE method A which CALLs program B. While program B is executing, method A could be invoked again causing a second call to program B. This second call is considered a recursive call. Therefore, any procedural code invoked from a method should have the RECURSIVE clause on the PROGRAM-ID statement.

For example:

```
Identification Division.
Program-Id.  ProgB recursive.
Environment Division.
   .
   .
```

Coexistence

The object-oriented and procedure-oriented parts of your system can exist quite well together. Certainly, you want to reuse your existing code as long as it continues to meet your needs. However, you can add new function to your system using object-oriented implementations. Only if your existing code no longer meets your needs or its maintenance cost is too high should you consider converting the entire procedure-oriented system to an object-oriented system.

Converting a Procedure-Oriented Program

Taking as input a typical COBOL batch or online program, the goal is to produce a formal specification of the program in object-oriented form. The conversion involves four steps:

- Identify objects
- Analyze data flow and usage
- Reallocate code to objects
- Write the object-oriented code

Step Three

Identify Objects

First, partition the DATA DIVISION into potential objects by identifying every record as an object and every field in the record as its instance data. You can start by taking record structures that define files and making them objects called `FffffFile`, where `Fffff` is a name of your choosing. Record structures that define database views become objects called `VvvvvView`. Map or panel record structures become objects called `UuuuuInterface`. Other record structures in the WORKING-STORAGE SECTION not related to files, databases, maps, or panels become objects called `WwwwWork`. Finally, record structures in the LINKAGE SECTION become objects called `PppppParameter`.

Now you have a lot of potential objects, some of which are redundant. Study the potential objects and decide if two or more are slight variations of the same object. Maybe you have two detail lines as potential objects that differ in only one or two of their fields. If possible use REDEFINES or some other technique to combine the two detail lines into one and, thus, combine the two potential objects into one object.

The result of this step is an object list with the name of each object and its instance data.

Analyze Data Flow and Usage

Analyze the file and database accesses to collect the access operations, such as SELECT, UPDATE, INSERT, DELETE, READ, and WRITE, for each object. The purpose is to find the relationship between objects via access sequence. For example, if one record read from the input file results in one detail line written to a report then a relationship exists between the file and the report objects. The relationships are tied to the source object and the target object.

Trace the data flow between objects to identify those objects which use instance data from another object.

If the two objects share a superclass - subclass (parent - child) relationship, then the subclass inherits methods from the superclass and shares instance data via 'get' and 'set' methods. (The `get` and `set` methods are written in the superclass definition.)

If the two objects are separate and distinct, then they are collaborators. Collaborators do not inherit anything from each other. Instance data that needs to be shared between two collaborators is typically passed as parameters on an INVOKE statement.

The result of this step is an object relationship table listing all the inheritance and collaboration relationships between objects.

Reallocate Code to Objects

For each object you identified, collect all references to it from the PROCEDURE DIVISION. Look for procedural code that changes the state of the object's instance data. If an instruction affects several data items in different objects, it must be changed or duplicated and the proper form associated with the correct object.

Step Four

For example:

```
Move 0 To input-z output-z.
```

must be changed to

```
Move 0 To input-z.
Move 0 To output-z.
```

The first MOVE is associated with the appropriate input object and the second with the appropriated output object.

Now couple the procedural instructions from the PROCEDURE DIVISION with the objects to form methods. Take the code you pulled from the program and organize it into task-oriented methods.

Refer to the object relationship table from step two and determine if any new methods must be written to facilitate passing data between two objects. See “Writing a Method Definition” on page 276 for information about writing methods.

The result of this step is completed method definitions.

Write the Object-Oriented Code

Write a class definition using the object list from step one and the methods from step three. See “Writing a Class Definition” on page 272 for information about writing classes.

Also, write the client program to create instances of the classes and invoke methods. See “Writing a Client Definition” on page 285 for information about writing client programs.

Your client program may be a modification of your original procedure-oriented program with invokes and manipulation of object references added where needed in the procedural code. This is the case when all the procedure-oriented code was not placed into methods. However, if all the procedure-oriented code was placed into methods, then your client program is a new program you write from scratch.

Part 4. Advanced Topics

This part of the book covers advanced programming topics. Basic programming topics are covered in Part 1, “Coding Your Program” on page 1.

Chapter 18. Porting Applications between Platforms

Your personal computer (PC) has a different hardware and operating system architecture than IBM mainframes or AIX workstations. Because of fundamental platform differences, some problems can arise as you move COBOL programs between the PC, workstation, and mainframe environments.

The following information describes some of the differences between development platforms and provides instructions to help you minimize portability problems.

Getting Mainframe Applications to Compile on the PC

As you move programs to the PC from the mainframe, one of your first goals is to get the applications you have already been using to compile in the new environment without errors.

Choosing the Right Compiler Options

Some mainframe COBOL compiler options are not applicable on the PC, and are treated as comments. For a full list of differences between host COBOL and VisualAge COBOL (including compiler options), see Appendix A, “Summary of Differences with Host COBOL” on page 540.

Two compiler options might yield unpredictable results and are flagged by the compiler with W-level messages:

CMPR2

This compiler option impacts the interpretation of language elements. The PC compiler does not support VS COBOL II Release 2 language elements that are in conflict with the ANSI 85 COBOL Standard. A program depending on the CMPR2 option is not portable.

NOADV

Programs that require the use of NOADV are sensitive to device control characters and almost certainly are not portable. If your program relies on NOADV, revise it such that language specification does not assume a printer control character as the first character of the `01` record for the file.

Differences from Mainframe COBOL Language Features

The following section describes some language features that are valid under mainframe COBOL but can create errors or unpredictable results in your PC compilation. Where possible, a solution to the potential problem is provided.

ACCEPT and DISPLAY statements

On the PC, the targets of DISPLAY or ACCEPT statements are determined by checking COBOL environment variables (see “Run-Time Environment Variables” on page 137). If your mainframe program assumes the use of host DDNAMEs as the targets of ACCEPT or DISPLAY statements, ensure that these targets are defined by equivalent environment variables with values set to appropriate PC filenames.

Getting Mainframe Applications to Compile on the PC

ASSIGN clause
There is a different syntax and mapping to the system filename based on ASSIGNMENT name (see the *IBM COBOL Language Reference*).

CALL statement
A filename as a CALL argument is not supported.

CLOSE statement
The phrases FOR REMOVAL, WITH NO REWIND, and UNIT/REEL are treated as comments. Avoid their use in portable programs.

LABEL RECORD clause
LABEL RECORD IS *data-name*, USE...AFTER...LABEL PROCEDURE, and GO TO MORE-LABELS are treated as errors. Programs depending on user label processing supported through OS/390 QSAM are not portable.

POINTER and PROCEDURE-POINTER data items
Under mainframe COBOL, a POINTER data item is defined as 4 bytes, and a PROCEDURE-POINTER data item is defined as 8 bytes. On the PC, the size of these data items are consistent with the native pointer definition of the platform (for example, 4 bytes for a 32-bit machine).

RERUN clause
Treated as a comment.

SHIFT-IN, SHIFT-OUT special registers
Not applicable on the PC; results in an E-level message unless the CHAR(EBCDIC) compiler option is in effect.

SORT-CONTROL special register
Because it identifies a system filename, this register is sensitive to the filename conventions of the platform. Be aware of the differences in naming conventions between the PC and the mainframe.

WRITE statement
If you specify WRITE...ADVANCING with the environment names C01-C12 or S01-S05, the ADVANCING phrase is ignored.

Using the COPY Statement to Help Port Programs

In many cases, potential portability problems can be avoided by using the COPY statement to isolate platform-specific code. For example, you can include platform-specific code in a compilation for a given platform and exclude it from compilation for a different platform. You can also use the COPY REPLACING phrase to globally change non-portable source code elements, such as filenames.

For information about the COPY statement, see the *IBM COBOL Language Reference*.

Getting Mainframe Applications to Run on the PC

Once you have downloaded your source program from the mainframe and compiled it on the PC without errors, the next step is to run the program. In many cases, you can get the same results from the PC run as from the mainframe COBOL run without serious source-language modifications. In order to assess whether or not you should make source-language modifications to you program, you need to know about elements and behavior of the COBOL language that vary due to the underlying hardware or software architectures.

Data Representations Causing Run-Time Differences

COBOL stores USAGE PACKED-DECIMAL data in the same manner on both the mainframe and on the PC, but all other computational data is, by default, represented differently. Most programs act the same on the PC as they do on the mainframe regardless of the data representation. To ensure that this is true for your programs, you should try to understand the differences described in the following sections.

ASCII vs EBCDIC

The PC uses the ASCII-based character set while the mainframe uses the EBCDIC character set. This means that most characters have a different hexadecimal value. For example, the hexadecimal value for a blank is `X'20'` in the ASCII character set and `X'40'` in the EBCDIC character set.

Code which is dependent on the EBCDIC hexadecimal values of character data probably fails when run using ASCII. For example, code that tests whether or not a character is a blank by comparing it with `X'40'` fails when run using the ASCII collating sequence.

In the ASCII character set, characters '0' through '9' have the hexadecimal values `X'30'` through `X'39'`. The ASCII lowercase letter `'a'` has the hexadecimal value `X'61'`, and the uppercase letter `'A'` has the hexadecimal value `X'41'`. In the EBCDIC character set, characters '0' through '9' have the hexadecimal values `X'F0'` through `X'F9'`. In EBCDIC, the lowercase letter `'a'` has the hexadecimal value `X'81'`, and the uppercase letter `'A'` has the hexadecimal value `X'C1'`. These differences have some important consequences:

While `'a'` < `'A'` is true for EBCDIC, it is false for ASCII.

While `'A'` < `'1'` is true for EBCDIC, it is false for ASCII.

While x >= `'0'` almost always means that x is a digit in EBCDIC, this is not true for ASCII.

Because of the differences described, the results of sorting character strings are different under EBCDIC and ASCII. For many programs, this has no effect, but you should be aware of potential logic errors if your program depends on the exact sequence in which some character strings are sorted. If your program is dependent on the EBCDIC collating sequence and you are porting it to the PC, you can obtain the EBCDIC collating sequence using PROGRAM COLLATING SEQUENCE IS EBCDIC or the COLLSEQ(EBCDIC) compiler option.

To avoid problems with the different data representation between ASCII and EBCDIC characters, use the CHAR(EBCDIC) compiler option. For more information, see “CHAR” on page 165.

NATIVE vs NONNATIVE

The PC holds integers in a form that is byte-reversed when compared to the form in which they are held on the mainframe.

The mainframe representation is known as “Big Endian,” as in “big-end-in.” In other words, the most significant digit of the number is stored first. The PC representation is known, conversely, as “Little Endian,” as in “little-end-in.” On the PC, the least significant digit of the number is stored first.

For most programs this difference should create no problems. However, if your program depends on the hexadecimal value that an integer has, you should be aware of potential logic errors.

For programs that use mainframe binary data and rely on the internal representation of integer values, you should compile the entire program with the BINARY(S390) compiler option. For such programs, you should avoid the USAGE COMP-5 type, which is treated as the native binary data format regardless of whether or not the BINARY(S390) option is specified.

For more information about the BINARY compiler option, see “BINARY” on page 163.

IEEE vs HEXADEC

The PC represents floating-point data using the IEEE format while the mainframe uses the hexadecimal format.

Figure 74 summarizes the differences between normalized floating-point IEEE and hexadecimal for USAGE COMP-1 data:

Figure 74. Normalized IEEE vs. Normalized Hexadecimal for COMP-1 Data

Specification	IEEE	Hexadecimal
Range	±1.17E-38 to ±3.37E+38	±5.4E-79 to ±7.2E+75
Exponent representation	8 bits	7 bits
Mantissa representation	23 bits	24 bits
Accuracy	6 digits	6 digits

Figure 75 summarizes the differences between normalized floating-point IEEE and hexadecimal for USAGE COMP-2 data:

Figure 75 (Page 1 of 2). Normalized IEEE vs. Normalized Hexadecimal for COMP-2 Data

Specification	IEEE	Hexadecimal
Range	±2.23E-308 to ±1.67E+308	±5.4E-79 to ±7.2E+75
Exponent representation	11 bits	7 bits

Figure 75 (Page 2 of 2). Normalized IEEE vs. Normalized Hexadecimal for COMP-2 Data

Mantissa representation	52 bits	56 bits
Digits of accuracy	15 digits	16 digits

For most programs these differences should create no problems. However, use caution in porting if your program depends on hexadecimal representation of data.

To avoid most problems with the different representation between IEEE and hexadecimal floating-point data, use the FLOAT(S390) compiler option. For more information, see "FLOAT" on page 180.

EBCDIC DBCS vs ASCII multi-byte strings

Mainframe double-byte character strings (DBCS) are enclosed in shift codes, while PC multi-byte character strings (including DBCS) are not enclosed in shift codes. The hexadecimal values used to represent the same characters are also different.

For most programs this should not make porting difficult. However, if your program depends on the hexadecimal value of a graphic string or on a character string containing mixed character and graphic data, use caution in your coding practices.

Note: On the PC, DBCS data can contain single-byte characters as well as double-byte characters.

Environment Differences Affecting Portability

There are some differences, other than data representation, between the PC and mainframe platforms that can also affect the portability of your programs. This section describes some of these differences.

File names

File naming conventions on the PC are very different from those on the mainframe. The following file name, for example, is valid on the PC but not on the mainframe:

```
d:\programs\data\myfile.dat
```

This can have an effect on portability if you use file names in your COBOL source programs.

Control codes

Some characters that have no particular meaning on the mainframe are interpreted as control characters on the PC. This can lead to incorrect processing of ASCII test files. Files should not contain any of the following characters:

X'0A' ("LF - line feed")
X'0D' ("CR - carriage return")
X'1A' ("EOF - end of file")

Device-dependent control codes

Use of device-dependent (platform-specific) control codes in your programs or files can cause problems when trying to port them to other platforms that do not necessarily support the control codes.

As with all other very platform-specific code, it is best to isolate such code as much as possible so that it can be replaced easily when you move the application to another platform.

Language Elements Causing Run-Time Differences

In general, you can expect your portable COBOL programs to behave the same way on the PC that they do on the mainframe. However, be aware of the differences in FILE STATUS values use for I/O processing.

If your program is written to respond to status key data items, you should be concerned with two issues, depending on whether the program is written to respond to the first status key or the second status key:

1. If your program is written to respond to the first file status data item *(data-name-1)*, be aware that values returned in the `9X` range are platform-dependent. If your program depends on the interpretation of a particular `9X` value (for example, `97`), do not expect the value to have the same meaning on the PC that it does on the mainframe. Instead, revise your program so that it responds to **any** `9X` value as a generic I/O failure.
2. If your program is written to respond to the second file status data item *(data-name-8)*, be aware that the values returned are both platform and file system dependent. For example, VSAM returns values with a different record structure on the mainframe than it does on the PC. Also, different file systems (for example, Btrieve rather than IBM's VSAM) return different values. If your program relies on the interpretation of the second file status data item, it is probably not portable.

For more information about the FILE STATUS clause, see the *IBM COBOL Language Reference*.

Writing PC Code to Run on the Mainframe

You can use VisualAge COBOL to write new applications, taking advantage of the productivity gains and increased flexibility of using your PC. The purpose of this section is to make you aware of how to avoid using VisualAge COBOL features not supported by mainframe COBOL.

Language Features Supported Only by the PC Compiler

VisualAge COBOL supports several language features not supported by the mainframe COBOL compiler. As you write code on your PC that is intended to run on the mainframe, avoid using these features:

- ORGANIZATION LINE SEQUENTIAL
- Format 5 SET statement extension that allows setting of pointers or ADDRESS OF special register to an address of level 01, 02-49, or 77 in the LINKAGE SECTION or the WORKING-STORAGE SECTION
- LOCK MODE IS AUTOMATIC
- ASSIGN USING *data-name*

- READ statement using PREVIOUS phrase
- START statement using < or <= in the KEY PHRASE
- USAGE COMP-5 data items

Compiler Options Supported Only on the PC

A number of compile-time options are available with VisualAge COBOL. Do not use any of the following options in your source code if you intend to port this code to the mainframe COBOL compiler:

- BINARY
- CALLINT (also a compiler directive)
- CHAR
- ENTRYINT
- FLOAT
- PROBE
- PROFILE
- THREAD

Names Supported Only on the PC

Be aware of the difference in naming conventions supported on the PC and mainframe file systems. Try to avoid hard-coding the names of files in your source programs. Instead, use mnemonic names (in turn, mapped to mainframe DDNAMEs or PC environment variables) which can be defined on each platform, allowing you to compile your program without source code changes to accommodate the file name changes.

Specifically, consider how you refer to files in the following language elements:

- ACCEPT and DISPLAY target names
- ASSIGN clause
- COPY statement (*text-name* and *library-name*)

Writing Applications That Are Portable between the PC and AIX

The PC and AIX COBOL environments are similar, and their language support is practically identical. However, there are two key differences between these platforms that you should keep in mind when developing applications that are portable between the PC and the AIX workstation:

1. As you can expect when porting programs between the PC and the mainframe, hard-coded filenames in your source programs can lead to problems. See “Names Supported Only on the PC” for a description of how and where to avoid using literal file names in your source programs.
2. As noted in the discussion of NATIVE vs NONNATIVE on 367, the PC represents integers in “Little Endian” format. Like the mainframe, AIX workstations maintain integers in “Big Endian” format. Therefore, if your PC COBOL program depends on the internal representation of an integer, the program is probably not portable to AIX; avoid writing programs that rely on such internal representation. If your

program *requires* manipulating the internal representation of PC-format integers, use the BINARY(S390) compiler option and avoid the USAGE COMP-5 type.

Chapter 19. Subprograms

Often, an application will consist of several, separately compiled programs linked together.

When a run unit consists of several, separately compiled programs that call each other, the programs must be able to communicate with each other. They need to transfer control and usually need to have access to common data. The following sections describe the methods that allow separately compiled programs to communicate with one another.

COBOL programs that are nested within each other can also communicate. All the required subprograms for an application can be contained in one program and thereby require only a single compilation. This method is explained in “Structure of Nested Programs” on page 373.

Transferring Control to Another Program

In the PROCEDURE DIVISION, a program can call another program (generally called a subprogram in COBOL terms), and this called program can itself call other programs. The program that calls another program is referred to as the *calling* program, and the program it calls is referred to as the *called* program. When the called program processing is completed, the program can either transfer control back to the calling program or end the run unit.

The called COBOL program starts running at the top of the PROCEDURE DIVISION.

Not Recomended: You can specify another entry point where the program will begin running by using the ENTRY label in the called program. However, this is not recommended in a structured program.

Recursive Calls

If a called program directly or indirectly executes its caller (such as program X calling program Y; program Y calling program Z; and program Z then calling program X), this is called a *recursive* call. Recursive calls are allowed if you code the RECURSIVE attribute on the PROGRAM-ID paragraph of the recursively invoked program and/or if you specify the THREAD compiler option. If you try to recursively call a COBOL program that does not have the RECURSIVE attribute coded on its PROGRAM-ID paragraph, the run unit will end abnormally.

For considerations in using the LINKAGE SECTION with recursive calls, see “With Recursion or Multithreading” on page 22.

Main Programs and Subprograms

No specific source code statements or options identify a COBOL program to be a main program or a subprogram. If a COBOL program is the first program in the run unit, that

COBOL program is the *main program*. Otherwise, it and all other COBOL programs in the run unit are subprograms.

Whether a COBOL program is a main program or a subprogram can be significant for either of two reasons:

- If execution ends in the main program, the main program must use a STOP RUN or GOBACK statement. STOP RUN terminates the run unit, and closes all files opened by the main program and its called subprograms. Control is returned to the caller of the main program, which is often the operating system. GOBACK has the same effect in the main program. An EXIT PROGRAM performed in a main program has no effect.

  A subprogram can end with an EXIT PROGRAM, a GOBACK, or a STOP RUN statement. If the subprogram ends with an EXIT PROGRAM or a GOBACK statement, control returns to its immediate caller without ending the run unit. An implicit EXIT PROGRAM statement is generated if there is no next executable statement in a called program. If the subprogram ends with a STOP RUN statement, the effect is the same as it is in a main program: all COBOL programs in the run unit are terminated, and control returns to the caller of the main program.

- A subprogram is usually left in its *last-used state* when it terminates with EXIT PROGRAM or GOBACK. The next time it is called in the run unit, its internal values will be as they were left, except that return values for PERFORM statements will be reset to their first values. In contrast, a main program is initialized each time it is called. There are three exceptions:

  1. A subprogram that is called and then cancelled will be in the initial state the next time it is called.
  2. A program with the INITIAL attribute will be in the initial state each time it is called.
  3. Data defined in the LOCAL-STORAGE section is in initial state each time it is called.

Making Calls between COBOL Programs

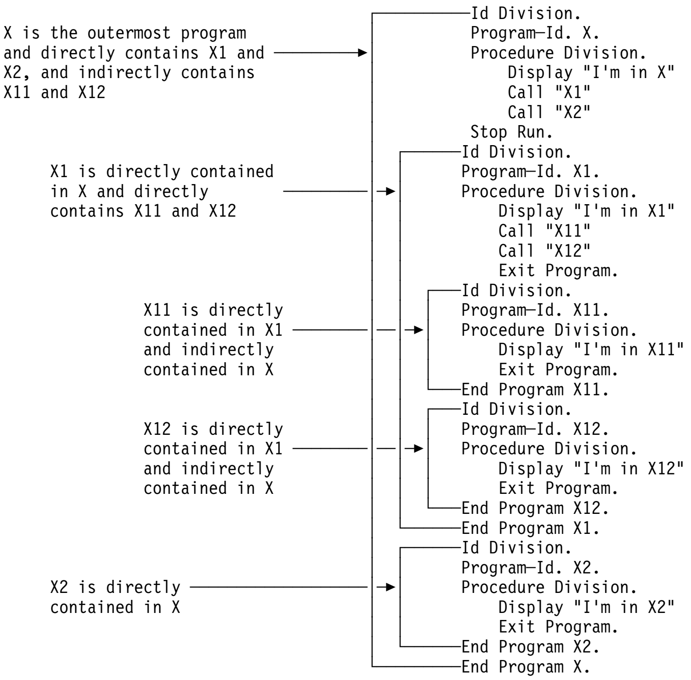
You can transfer control to nested and/or non-nested COBOL programs.

Calls to nested programs allow you to create applications using structured programming techniques. They can also be used in place of PERFORM procedures to prevent unintentional modification of data items. Calls to nested programs can be made using either the CALL *literal* or CALL *identifier* statement. For more information on nested programs, see “Structure of Nested Programs.”

Structure of Nested Programs

A COBOL program can contain or “nest” other COBOL programs. The nested programs can themselves nest yet other programs. A nested program can be directly or indirectly nested in a program.

Figure 76 describes a nested program structure with directly and indirectly nested programs.



X is the outermost program and directly contains X1 and X2, and indirectly contains X11 and X12

X1 is directly contained in X and directly contains X11 and X12

X11 is directly contained in X1 and indirectly contained in X

X12 is directly contained in X1 and indirectly contained in X

X2 is directly contained in X

```
Id Division.
Program-Id. X.
Procedure Division.
    Display "I'm in X"
    Call "X1"
    Call "X2"
Stop Run.
Id Division.
Program-Id. X1.
Procedure Division.
    Display "I'm in X1"
    Call "X11"
    Call "X12"
    Exit Program.
Id Division.
Program-Id. X11.
Procedure Division.
    Display "I'm in X11"
    Exit Program.
End Program X11.
Id Division.
Program-Id. X12.
Procedure Division.
    Display "I'm in X12"
    Exit Program.
End Program X12.
End Program X1.
Id Division.
Program-Id. X2.
Procedure Division.
    Display "I'm in X2"
    Exit Program.
End Program X2.
End Program X.
```

Figure 76. Nested Program Structure with Directly and Indirectly Contained Programs

Conventions for Using Nested Program Structure

Follow these conventions when using nested program structures:

- The IDENTIFICATION DIVISION is required in each program. All other divisions are optional.
- Nested program names must be unique.
- Nested program names can be any valid COBOL word or a non-numeric literal.
- Nested programs cannot have a CONFIGURATION SECTION. The outermost program must set any CONFIGURATION SECTION options that might be required.
- Each nested program is included in the nesting program immediately before its End Program header (see Figure 76).
- Nested and nesting programs must be terminated by an End Program header.

Calling nested Programs

A nested program can be called only by its directly nesting program, unless the nested program is identified as COMMON in its PROGRAM-ID clause. In that case, the *common program* can also be called by any program that is nested (directly or indirectly) in the same program as the common program. Only nested programs can be identified as COMMON. Recursive calls are not allowed.

Figure 77 shows the outline of a nested structure with some nested programs identified as COMMON.

```
┌───Program-Id. A.
│
│  ┌───Program-Id. A1.
│  │
│  │  ┌───Program-Id. A11.
│  │  │
│  │  │  ┌───Program-Id. A111.
│  │  │  │
│  │  │  └───End Program A111.
│  │  └───End Program A11.
│  │  ┌───Program-Id. A12 is Common.
│  │  │
│  │  └───End Program A12.
│  └───End Program A1.
│  ┌───Program-Id. A2 is Common.
│  │
│  └───End Program A2.
│  ┌───Program-Id. A3 is Common.
│  │
│  └───End Program A3.
└───End Program A.
```

Figure 77. A Nested Structure with COMMON Programs

The following table describes the calling hierarchy for the structure that is shown in Figure 77. Programs A12, A2, and A3 are identified as COMMON, and t he calls associated with them differ.

Figure 78. Calling Hierarchy for Nested Structures with COMMON programs

This Program	Can call these programs	And can be called by these programs
A	A1, A2, A3	None
A1	A11, A12, A2, A3	A
A11	A111, A12, A2, A3	A1
A111	A12, A2, A3	A11
A12	A2, A3	A1, A11, A111
A2	A3	A, A1, A11, A111, A12, A3
A3	A2	A, A1, A11, A111, A12, A2

Note that:

- A2 cannot call A1 because A1 is not common and is not contained in A2.
- A1 can call A2 because A2 is common.

Linker and Run-Time Resolution

Scope of Names

Names in nested programs are divided into two classes: local and global. The class determines whether a name is known beyond the scope of the program that declares it. A specific search sequence locates the declaration of a name after it is referenced in a program.

Local Names: Names are local unless declared to be otherwise (except the program name). Local names are not visible or accessible to any program outside of the one in which they were declared; this includes both contained and containing programs.

Global Names: A name that is global (indicated using the GLOBAL clause) is visible and accessible to the program in which it is declared, and to all the programs that are directly and indirectly contained in that program. This allows the contained programs to share common data and files from the containing program, simply by referencing the name of the item.

Any item that is subordinate to a global item (including condition names and indexes) is automatically global.

The same name can be declared with the GLOBAL clause more than one time, providing that each declaration occurs in a different program. Be aware that masking, or hiding, a name in a nested program is possible by having the same name occur in different programs of the same containing structure. This could possibly cause some problems when a search for a name declaration is taking place.

Searching for Name Declarations: When a name is referenced in a program, a search is made to locate the declaration for that name. The search begins in the program that contains the reference and continues outward to containing programs until a match is found. The search follows this process:

1. Declarations in the program are searched first.
2. If no match is found, only global declarations are searched in successive outer containing programs.
3. The search ends when the first matching name is found; otherwise, an error exists if no match is found.

Note that the search is for a global name, not for a particular type of object associated with the name, such as a data item or file connector. The search stops when any match is found, regardless of the type of object. If the object declared is of a different type than that expected, an error condition exists.

Static, Dynamic, and Run-time Linking

COBOL calls can be made to a subprogram that is either linked into the same executable module as the caller (static linking) or a subprogram that is provided in a DLL (dynamic linking). IBM VisualAge COBOL also provides for run-time resolution of the target subprogram from a DLL. If it is linked statically, it is part of the caller's executable module and is loaded with the caller. If it is linked dynamically or resolved at run time, it is provided in a library and is loaded either when the caller is loaded or when it is called.

Static linking and dynamic linking are done for COBOL CALL *literal* subprograms only. See “Static Linking Overview” on page 439 for a discussion on static linking. See “Dynamic Linking Overview” on page 439 for a discussion of dynamic linking.

Run-time resolution is always done for COBOL CALL *identifier* and is done for CALL *literal* when the DYNAM option is in effect.

CALL identifier

The COBOL CALL *identifier*, where *identifier* is a data item that contains the name of a non-nested subprogram when the program is run, always results in the target subprogram being loaded when it is called. Also, the name of the DLL must match the name of the target entry point. See “Terminology Notes” on page 440 for a discussion of COBOL terminology.

CALL literal

The COBOL CALL *literal*, where *literal* is the explicit name of a non-nested subprogram being called, can be resolved statically, dynamically or at run time. If the NODYNAM compile-time option is in effect, either static or dynamic linking can be done. If DYNAM is in effect, the CALL *literal* is resolved the same as CALL *identifier*: the target subprogram is loaded when it is called, and the name of the DLL must match the name of the target entry point.

These call definitions apply only in the case of a COBOL program calling a non-nested program. When a COBOL program calls a nested program, the CALL is resolved by the compiler without any system intervention.

For a detailed description of the CALL statement, see the *IBM COBOL Language Reference*. For more information on calling subprograms in DLLs see Chapter 24, “Building Dynamic Link Libraries” on page 439.

Making Calls between COBOL and C/C++ Programs

You can call functions written in C or C++ from your COBOL programs and vice versa. This chapter describes how to perform such interlanguage calls from your COBOL and C or C++ programs.

Rules and Guidelines for ILC Applications

The following are rules and guidelines for COBOL—C/C++ ILC applications:

- Run Unit/Process Termination and Stack Frame Collapsing

  Functions invoked in one language which result in collapsing of program stack frame(s) of other language(s) should be avoided. This includes:

  - Collapsing some active stack frames from one language with active stack frames written in another language in the to-be-collapsed stack frames (C `longjmp()`).
  - Terminating run unit/process from one language while stack frames written in another language are active: For example, COBOL STOP RUN and C `exit()` or `_exit()`.

    Instead, the application should be structured in such a way that an invoked program terminates by returning to its invoker.

  These function can be used in an ILC environment as long as the use of such a function does not result in collapsing of active stack frame(s) of a language other than the one initiating such a function.

  This is a guideline and will not be enforced. Potential adverse effects for the languages not initiating the collapsing/termination are:

  - Normal clean-up/exit functions of the language might not be performed.

    An example is the closing of files by COBOL on a run unit termination, or the clean-up of dynamically acquired resources by the involuntarily terminated language.
  - User specified exits/functions for the exit/termination might not be invoked. Examples are `destructors` and the C `atexit()` function.

- Exception Handling

  Exceptions incurred during the execution of a stack frame written in one language might not be processed according to the rules of other languages active in the invocation stack.

  In general, such an exception will be handled according to the rules of the language incurring the exception.

  This COBOL product will save the exception environment on entry to the COBOL run time environment and restores it on termination of the COBOL environment. COBOL expects interfacing languages and tools to follow the same convention.

  Since the COBOL implementation does not depend on the interception of exceptions through system services for the support of ANSI COBOL language semantics, the user can specify the run-time option, TRAP(OFF), to enable the exception handling semantics of the non-COBOL language.

- COBOL “main”

  When C programs are invoked in an application where the main program is a COBOL program, the C initialization routine, `_CRT_init`, should be invoked either by the COBOL program before the first C function is called, or by the first C function called by COBOL.

  Likewise, the C termination routine, `_CRT_term`, should be invoked when the C environment is no longer required.

Matching Data and Linkages

Some COBOL data types have C/C++ equivalents, but others do not. When you pass data between COBOL and C/C++ functions, be sure to restrict data exchange to appropriate data types. For a detailed description of how COBOL programs can share data with other programs, see Chapter 20, "Data Sharing" on page 387.

The following table shows the correspondence between the data types available in COBOL and C/C++.

C/C++ Data Types	COBOL Data Types
wchar_t	DISPLAY-1 (PICTURE N, G) wchar_t is the processing code whereas DISPLAY-1 is the file code.
char	PIC X
signed char	No appropriate COBOL equivalent.
unsigned char	No appropriate COBOL equivalent.
short signed int	PIC S9-S9(4) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
short unsigned int	PIC 9-9(4) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
long int	PIC 9(5)-9(9) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
long long int	PIC 9(10)-9(18) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
float	COMP-1
double	COMP-2
enumeration	Equivalent to level 88, but not identical.
char(n)	PICTURE X(n)
array pointer (*) to type	No appropriate COBOL equivalent.
pointer(*) to function	PROCEDURE-POINTER

Figure 79. Correspondence between COBOL and C/C++ Data Types

Example - Calling C/C++ Functions from a COBOL Program

The following COBOL program illustrates several concepts:

- C/C++ programs are called using the COBOL CALL statement. The CALL statement does not indicate if the called program is written in COBOL or C/C++.
- COBOL supports calling programs with mixed-case names.
- Arguments can be passed to C/C++ programs in different ways (for example, CALL BY REFERENCE, CALL BY VALUE, etc.).
- You must declare a function return value on the CALL statement that calls a C/C++ function.
- You must map COBOL data types to appropriate C/C++ data types.

These concepts are illustrated in the following COBOL source program, COBCALLC.CBL:

```
 CBL PGMNAME(MIXED) CALLINT(OPTLINK)
* These compiler options allow for
* case-sensitive names for called programs
* and set the call interface/linking
* convention to that of the IBM C/C++ default.
*
*
*
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "COBCALLC".
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 01 N4               PIC 9(4)  COMP-5.
 01 NS4              PIC S9(4) COMP-5.
 01 N9               PIC 9(9)  COMP-5.
 01 NS9              PIC S9(9) COMP-5.
 01 NS18             USAGE COMP-2.
 01 D1               USAGE COMP-2.
 01 D2               USAGE COMP-2.
 01 R1.
     02 NR1               PIC 9(8) COMP-5.
     02 NR2               PIC 9(8) COMP-5.
     02 NR3               PIC 9(8) COMP-5.
 PROCEDURE DIVISION.
*
* Initialize C environment
*
     CALL "initC".
*
     MOVE 123 TO N4
     MOVE -567 TO NS4
     MOVE 98765432 TO N9
     MOVE -13579456 TO NS9
     MOVE 222.22 TO NS18
     DISPLAY "Call MyFun with n4=" N4 " ns4=" NS4 " N9=" n9
     DISPLAY "            ns9="  NS9 " ns18="  NS18
*
* The following CALL illustrates several ways to pass
* arguments.
*
     CALL "MyFun" USING N4 BY VALUE NS4 BY REFERENCE N9 NS9 NS18
     MOVE 1024 TO N4
```

Figure 80 (Part 1 of 2). COBCALLC.CBL - A COBOL Program That Calls a C Program

```
*
* The following CALL returns the C function return value.
*
     CALL "MyFunR" USING BY VALUE N4 RETURNING NS9
     DISPLAY "n4=" N4 " and ns9= n4 times n4= " NS9
     MOVE -357925680.25 TO D1
     CALL "MyFunD" USING BY VALUE D1 RETURNING D2
     DISPLAY "d1=" D1 " and d2= 2.0 times d2= " D2
     MOVE 11111 TO NR1
     MOVE 22222 TO NR2
     MOVE 33333 TO NR3
     CALL "MyFunV" USING R1
*
* Terminate C environment
*
     CALL "termC".
     STOP RUN.
```

Figure 80 (Part 2 of 2). COBCALLC.CBL - A COBOL Program That Calls a C Program

The COBOL default is that arguments are passed BY REFERENCE. If an argument is passed BY REFERENCE, C gets a pointer to the argument. If you pass an argument BY VALUE in the CALL statement, COBOL passes the actual argument. BY VALUE can be used only for the following data types:

- Alphanumeric DISPLAY items
- BINARY
- COMP
- COMP-1
- COMP-2
- COMP-4
- COMP-5
- OBJECT REFERENCE
- POINTER
- PROCEDURE-POINTER

For a detailed description of the CALL statement, see the *IBM COBOL Language Reference*.

Linkage Conventions for CALLing C Programs from COBOL

There are important linkage considerations for COBOL programs that CALL C/C++ functions. Because C/C++ and COBOL use different default linkage conventions, you may need to use the >>CALLINT compiler directive or CALLINT compiler option for your COBOL programs that call C/C++ functions.

For example, if your COBOL program calls a program which used the default C Set++ linkage convention, compile your program with the CALLINT(OPTLINK) option or use the >>CALLINT OPTLINK compiler directive. Neither is the default.

Selective Change: The compiler directive format is useful where you want to change the linkage convention for a particular call rather than the whole program.

For details about CALLINT, see “CALLINT” on page 164.

Calling C/C++ with a Variable Parameter List

C/C++ allows calls to a given program to be made with a varying number of parameters. You need to know which of the following methods the called C/C++ program uses to determine how many parameters have been passed to it.

OS/2 ▶

The called C/C++ program may use the `_parmdwords` function to get the number of parameters passed from the AL register (See the *C Library Reference* for details.). Using SYSTEM linkage (that is, the CALLINT(SYSTEM) option) will cause COBOL to set the AL register to the number of parameters passed.

◀ OS/2

Alternately, the called C/C++ program may use the `va_start`, `va_arg` and `va_end` macros to manage the variable aspect of the parameter list. If so, you need to know how the called program determines the end of the parameter list. For example, some programs look for a null pointer to signify the end of the parameter list. COBOL does not terminate the parameter list with a null; you can supply it yourself by passing `BY VALUE 0.` as the last argument.

Example - C Programs That are Called by and Call COBOL Programs

The following C program illustrates that a CALLed C function receives arguments in the order in which they were passed in the COBOL CALL statement. It also shows how a C program can call a COBOL program (in this case, a program named `TPROG1`, see Figure 82 on page 385).

The file `MyFun.c` contains the following C source code:

```
#include <stdio.h>

extern int _CRT_init(void);
extern void _CRT_term(void);
extern void TPROG1(double *);

void
initC(void)
{
   int rc;

   rc = _CRT_init();
   setbuf(stdout, NULL);
   if (rc)
     printf("Error occurred during C initialization\n");
}

void
termC(void)
{
   _CRT_term();
}

void
MyFun(short *ps1, short s2, long *k1, long *k2, double *m)
{
    double x;

    x = 2.0*(*m);
    printf("MyFun got s1=%d s2=%d k1=%d k2=%d x=%f\n",
             *ps1, s2, *k1, *k2, x);
}

long
MyFunR(short s1)
{
    return(s1 * s1);
}
```

Figure 81 (Part 1 of 2). MyFun.C - A C Program That Calls and is Called by COBOL Programs

```
double
MyFunD(double d1)
{
    double z;

       /* example of C calling COBOL */
       z = 1122.3344;
       (void) TPROG1(&z);
       /* example of C returning a value to COBOL */
       return(2.0 * d1);
}

void
MyFunV(long *pn)
{
  printf("MyFunV got %d %d %d\n", *pn, *(pn+1), *(pn+2));
}
```

Figure 81 (Part 2 of 2). MyFun.C - A C Program That Calls and is Called by COBOL Programs

`MyFun.c` consists of the following functions:

MyFun Illustrates passing a variety of arguments.

MyFunR Illustrates how to pass and return a long variable.

MyFunD Illustrates C calling a COBOL program and it also illustrates how to pass and return a double variable.

MyFunV Illustrates passing a pointer to a record and accessing the items of the record in a C program.

Example - COBOL Program Called by a C Program

In Figure 80 on page 380, you find a COBOL program that calls C functions. In Figure 81 on page 383, you find a C program that is called by a COBOL program and calls a COBOL program. The following example illustrates how to write COBOL programs that are called by C programs.

The file `TPROG1.CBL` is called by the C function `MYFUND` in the C program `MyFun.c` (see Figure 81 on page 383). The called COBOL program contains the following source code:

```
*
* Sets the calling convention to that of IBM C/C++
*
 CBL ENTRYINT(OPTLINK)
*
 IDENTIFICATION DIVISION.
 PROGRAM-ID. TPROG1.
*
 DATA DIVISION.
 LINKAGE SECTION.
*
 01 X               USAGE COMP-2.
*
 PROCEDURE DIVISION USING X.
     DISPLAY "TPROG1 got x= " X
     GOBACK.
```

Figure 82. TPROG1.CBL - A COBOL Program Called by a C Program

Linkage Conventions for Called COBOL Programs

There are important linkage considerations for COBOL programs called by C functions. Because C and COBOL use different default linkage conventions, you may need to specify the ENTRYINT compiler option for your COBOL programs called by C/C++ programs.

For example, if your COBOL program is called by a C Set++ program, you should compile your program with the ENTRYINT(OPTLINK) option. This option (not the default) sets the linking convention to that of C Set++.

For details about ENTRYINT, see “ENTRYINT” on page 171.

Pre-Initializing the COBOL Environment

If your main program is written in C and makes multiple calls to a COBOL program, you should pre-initialize the COBOL environment in your C program. For example, if your C program repeatedly calls a COBOL program to carry out I/O tasks, you will probably want the COBOL program to remain in its last-used state.

For additional information, see Chapter 28, “Pre-initializing the COBOL Run-Time Environment” on page 489.

Results of Running COBCALLC

Compile and link the COBOL programs `COBCALLC.CBL` and `TPROG.CBL` and the C program `MyFun.c` and run `COBCALLC` using the following commands:

1. `icc -c MyFun.c`
2. `cob2 cobcallc MyFun.out tprog1.cbl -o cobcallc`

   Run the program by entering `COBCALLC`.

The results are as follows:

```
call MyFun with n4=00123 ns4=-00567 n9=0013579456
            ns9=0098765432 ns18=000012345678902468
MyFun got s1=123 s2=-567 k1=13579456 k2=98765432 x=12345678902468
n4=01024 and ns9= n4 times n4= 0001048576
TPROG1 got x=  .11223344000000000E+04
d1=-.35792568025000000E+09 and d2= 2.0 times d2= -.71585136050000000E+09
MyFunV got 11111 22222 33333
```

Figure 83. Result Compiling and Running the Examples in This Chapter

Chapter 20. Data Sharing

When a run unit consists of several, separately-compiled programs that call each other, the programs must be able to communicate with each other. They also usually need to have access to common data.

This chapter will describe how to write programs that can share data with other programs. For the purposes of this discussion, a "subprogram" is any program called by another program.

Passing Data

Data can be passed between programs in three ways:

BY REFERENCE The subprogram refers to and processes the data items in storage of the calling program rather than working on a copy of the data.

BY CONTENT The calling program passes only the contents of the *literal*, or *identifier*. With a CALL . . . BY CONTENT, the called program cannot change the value of the *literal* or *identifier* in the calling program, even if it modifies the variable in which it received the *literal* or *identifier*.

BY VALUE The calling program or method is passing the value of the *literal*, or *identifier*, not a reference to the sending data item.

Whether you pass data items BY REFERENCE, BY CONTENT, or BY VALUE depends on what you want your program to do with the data:

- If you want the definition of the argument of the CALL statement in the calling program and the definition of the parameter in the called program to share the same memory, use:

  CALL . . . BY REFERENCE *identifier*.

  Any changes made by the subprogram to the parameter affects the argument in the calling program.

- If you want to pass the address of a record area to a called program, use:

  CALL . . . BY CONTENT ADDRESS OF *record-name*.

  The subprogram receives the ADDRESS special register for the record-name you specify.

  You must define the record-name as a level-01 or level-77 item in the LINKAGE SECTION of the called and calling programs. A separate ADDRESS special register is provided for each record in the LINKAGE SECTION.

- If you do not want the definition of the argument of the CALL statement in the calling program and the definition of the parameter in the called subprogram to share the same memory, use:

CALL . . . BY CONTENT *identifier*.

- If you want to pass a literal value to a called program, use:

  CALL . . . BY CONTENT *literal*.

  The called program cannot change the value of the literal.

- If you want to pass the length of a data item, use:

  CALL . . . BY CONTENT LENGTH OF *identifier*.

  The calling program passes the length of the *identifier* from its LENGTH special register.

- If you want to pass both a data item and its length to a subprogram, use a combination of BY REFERENCE and BY CONTENT, for example:

  ```
  CALL 'ERRPROC' USING BY REFERENCE A
       BY CONTENT LENGTH OF A
  ```

- If you want to pass data to C programs that expect the value of the arugment, use:

  CALL . . . BY VALUE

  that expect a reference (pointer) to the argument, use:

  ```
     CALL . . . BY REFERENCE
   or
     CALL . . . BY CONTENT
  ```

  Use BY REFERENCE if you want the C program to be able to modify the value of the argument. Use BY CONTENT if you do *not* want the C program to modify the value of the argument.

  Parameters must be of certain data types to be passed BY VALUE. See the *IBM COBOL Language Reference*.

- If you want to return a value, use:

  CALL . . . RETURNING

Describing Arguments in the Calling Program

In the calling program, the arguments are described in the DATA DIVISION in the same manner as other data items in the DATA DIVISION. Data items in a calling program can be described in the LINKAGE SECTION of all the programs it calls directly or indirectly.

Here, storage for these items is allocated in the highest outermost program. That is, program A calls program B, which calls program C. Data items in program A can be described in the LINKAGE sections of programs B and C, and the one set of data can be made available to all three programs.

If you reference data in a file, the file must be open when the data is referenced.

Code the USING clause of the CALL statement to pass the arguments.

Describing Parameters in the Called Program

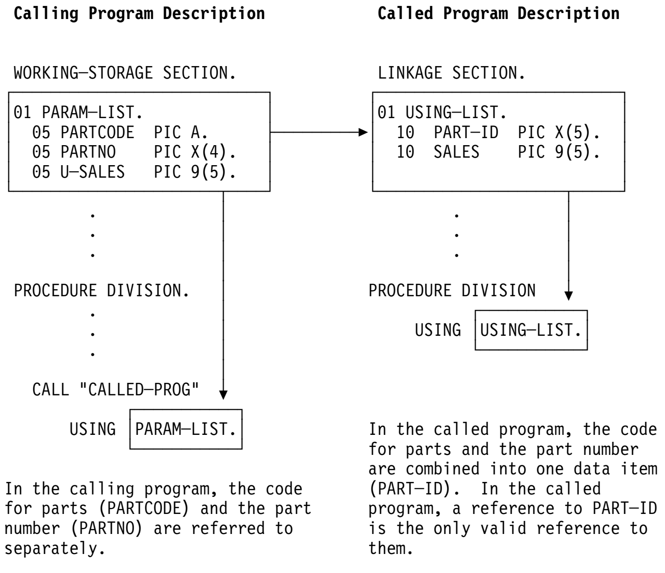
You must know what is being passed from the calling program and describe it in the LINKAGE SECTION of the called program.

Code the USING clause after the PROCEDURE-DIVISION header to receive the parameters.

LINKAGE SECTION

The number of *data-names* in the *identifier* list of a called program must not be greater than the number of *data-names* in the *identifier* list of the calling program. There is a one-to-one positional correspondence; that is, the first *identifier* of the calling program is passed to the first *identifier* of the called program, and so forth. The compiler does not try to match arguments and parameters.

See Figure 84 for an example.



Calling Program Description

```
WORKING—STORAGE SECTION.

01 PARAM—LIST.
  05 PARTCODE  PIC A.
  05 PARTNO    PIC X(4).
  05 U—SALES   PIC 9(5).
        .
        .
        .
PROCEDURE DIVISION.
        .
        .
        .
  CALL "CALLED—PROG"
      USING PARAM—LIST.
```

In the calling program, the code for parts (PARTCODE) and the part number (PARTNO) are referred to separately.

Called Program Description

```
LINKAGE SECTION.

01 USING—LIST.
  10  PART—ID  PIC X(5).
  10  SALES    PIC 9(5).
        .
        .
        .
PROCEDURE DIVISION
     USING USING—LIST.
```

In the called program, the code for parts and the part number are combined into one data item (PART—ID). In the called program, a reference to PART—ID is the only valid reference to them.

Figure 84. Common Data Items in Subprogram Linkage

PROCEDURE DIVISION

If an argument was passed BY VALUE, the PROCEDURE DIVISION header of the subprogram must indicate that:

```
PROCEDURE DIVISION USING BY VALUE.
```

If an argument was passed BY REFERENCE or BY CONTENT, the PROCEDURE DIVISION header does not need to indicate how the argument was passed.

PROCEDURE DIVISION in Subprograms

The header can either be:

```
PROCEDURE DIVISION USING
```

or:

```
PROCEDURE DIVISION USING BY REFERENCE
```

Grouping Data to Be Passed

Think about grouping all the data items you want to pass between programs and putting them under one level-01 item. If you do this, you can pass a single level-01 record between programs. For an example of this method, see Figure 84 on page 389.

To make the possibility of mismatched records even smaller, put the level-01 record in a COPY library, and copy it in both programs. (That is, copy it in the WORKING-STORAGE SECTION of the calling program and in the LINKAGE SECTION of the called program.)

Null-Terminated Strings

Null-terminated strings are supported using syntax shown in the *IBM COBOL Language Reference*. You can manipulate null-terminated strings passed from a C program, for example, by using string handling mechanisms such as those found below:

```
 01 L pic X(20) value z'ab'.
 01 M pic X(20) value z'cd'.
 01 N pic X(20).
 01 N-Length pic 99 value zero.
 01 Y pic X(13) value 'Hello, World!'.

* Display null-terminated string
     Inspect N tallying N-length
      for characters before initial x'00'
     Display 'N: ' N(1:N-length) ' Length: ' N-length

* Move null-terminated string to alphanumeric, strip null
     Unstring N delimited by X'00' into X

* Create null-terminated string
     String Y     delimited by size
            X'00' delimited by size
            into N.

* Concatenate two null-terminated strings
     String L     delimited by x'00'
            M     delimited by x'00'
            X'00' delimited by size
            into N.
```

Using Pointers to Process a Chained List

You can manipulate pointer data items, which are a special type of data item to hold addresses, when you want to pass and receive addresses of record areas. Pointer data items are data items that are either defined with the USAGE IS POINTER clause, or are ADDRESS special registers. A typical application for using pointer data items is in processing a chained list (a series of records where each one points to the next).

For this example, picture a chained list of data that is composed of individual salary records. Figure 85 shows one way to visualize how these records are linked in storage:

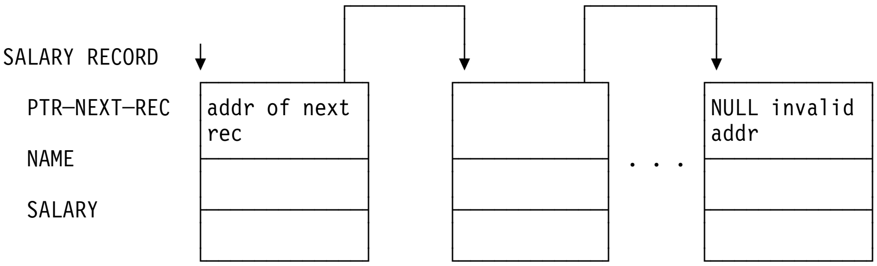


Figure 85. Representation of a Chained List Ending with NULL

The first item in each record points to the next record, except for the last record. The first item in the last record contains a null value instead of an address, to indicate that it is the last record.

The high-level logic of an application that processes these records might look something like this:

```
OBTAIN ADDRESS OF FIRST RECORD IN CHAINED LIST FROM ROUTINE
CHECK FOR END OF THE CHAINED LIST
DO UNTIL END OF THE CHAINED LIST
   PROCESS RECORD
   GO ON TO THE NEXT RECORD
END
```

Figure 86 on page 392 contains an outline of the processing program, LISTS, used in this example of processing a chained list.

PROCEDURE DIVISION in Subprograms

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. LISTS.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
******
 WORKING-STORAGE SECTION.
 77  PTR-FIRST         POINTER  VALUE IS NULL.
 77  DEPT-TOTAL        PIC 9(4) VALUE IS 0.
******
 LINKAGE SECTION.
 01  SALARY-REC.
   02  PTR-NEXT-REC    POINTER.
   02  NAME            PIC X(20).
   02  DEPT            PIC 9(4).
   02  SALARY          PIC 9(6).
 01  DEPT-X            PIC 9(4).
******
 PROCEDURE DIVISION USING DEPT-X.
******
* FOR EVERYONE IN THE DEPARTMENT RECEIVED AS DEPT-X,
* GO THROUGH ALL THE RECORDS IN THE CHAINED LIST BASED ON THE
* ADDRESS OBTAINED FROM THE PROGRAM CHAIN-ANCH
* AND CUMULATE THE SALARIES.
* IN EACH RECORD, PTR-NEXT-REC IS A POINTER TO THE NEXT RECORD
* IN THE LIST; IN THE LAST RECORD, PTR-NEXT-REC IS NULL.
* DISPLAY THE TOTAL.
******
     CALL "CHAIN-ANCH" USING PTR-FIRST
     SET ADDRESS OF SALARY-REC TO PTR-FIRST
******
     PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL
      IF DEPT = DEPT-X
        THEN ADD SALARY TO DEPT-TOTAL
        ELSE CONTINUE
      END-IF
      SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC
     END-PERFORM
******
     DISPLAY DEPT-TOTAL
     GOBACK.
```

Figure 86. Program for Processing a Chained List

Passing Addresses between Programs

To obtain the address of the first SALARY-REC record area, the LISTS program calls the program CHAIN-ANCH:

```
CALL "CHAIN-ANCH" USING PTR-FIRST
```

PTR-FIRST is defined in WORKING-STORAGE in the calling program (LISTS) as a pointer data item:

```
WORKING-STORAGE SECTION.
01  PTR-FIRST         POINTER  VALUE IS NULL.
```

On return from the call to CHAIN-ANCH, PTR-FIRST contains the address of the first record in the chained list.

PTR-FIRST is initially defined as having a null value as a logic check. If something goes amiss with the call, and PTR-FIRST never receives the value of the address of the first record in the chain, a null value remains in PTR-FIRST and, according to the logic of the program, the records will not be processed.

NULL is a figurative constant used to assign the value of an invalid address (nonnumeric 0) to pointer items. It can be used in the VALUE IS NULL clause, in the SET statement, and as one of the operands of a relation condition with a pointer data item.

The LINKAGE SECTION of the calling program contains the description of the records in the chained list. It also contains the description of the department code that is passed, using the USING clause of the CALL statement.

```
LINKAGE SECTION.
01  SALARY-REC.
    02  PTR-NEXT-REC    POINTER.
    02  NAME            PIC X(20).
    02  DEPT            PIC 9(4).
    02  SALARY          PIC 9(6).
01  DEPT-X            PIC 9(4).
```

To base the record description SALARY-REC on the address contained in PTR-FIRST, use a SET statement:

```
CALL "CHAIN-ANCH" USING PTR-FIRST
SET ADDRESS OF SALARY-REC TO PTR-FIRST
```

Checking for the End of the Chained List

The chained list in this example is set up so the last record contains an invalid address. To do this, the pointer data item in the last record would be assigned the value NULL.

A pointer data item can be assigned the value NULL in two ways:

- A pointer data item can be defined with a VALUE IS NULL clause in its data definition.
- NULL can be the sending field in a SET statement.

In the case of a chained list in which the pointer data item in the last record contains a null value, the code to check for the end of the list would be:

```
IF PTR-NEXT-REC = NULL
.
.
.(logic for end of chain)
```

If you haven't reached the end of the list, process the record and move on to the next record.

In the program LISTS, this check for the end of the chained list is accomplished with a DO WHILE structure:

```
PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL
  IF DEPT = DEPT-X
    THEN ADD SALARY TO DEPT-TOTAL
    ELSE CONTINUE
  END-IF
  SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC
END-PERFORM
```

Continuing Processing the Next Record

To move on to the next record, set the address of the record in the LINKAGE-SECTION to be equal to the address of the next record. This is accomplished through the pointer data item sent as the first field in SALARY-REC:

```
SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC
```

Then repeat the record-processing routine, which will process the next record in the chained list.

A Variation: Incrementing Addresses Received from Another Program

The data passed from a calling program might contain header information that you want to ignore (for example, in data received from a CICS application that is not migrated to the command level).

Because pointer data items are not numeric, you cannot directly perform arithmetic on them. However, you can use the SET verb to increment the passed address in order to bypass header information.

You could set up the LINKAGE SECTION like this:

```
   LINKAGE SECTION.
   01  RECORD-A.
     02  HEADER          PIC X(12).
     02  REAL-SALARY-REC PIC X(30).
   .
   .
   .
   01  SALARY-REC.
     02  PTR-NEXT-REC    POINTER.
     02  NAME            PIC X(20).
     02  DEPT            PIC 9(4).
     02  SALARY          PIC 9(6).
```

Within the Procedure Division, “base” the address of SALARY-REC on the address of REAL-SALARY-REC:

```
   SET ADDRESS OF SALARY-REC TO ADDRESS OF REAL-SALARY-REC
```

SALARY-REC is now based on the address of RECORD-A + 12.

Using Procedure Pointers

Procedure pointers are data items defined with the USAGE IS PROCEDURE-POINTER clause. You can set procedure-pointer data items to contain entry addresses of (or pointers to) these entry points:

- Another COBOL program that is not nested.
- An alternate entry point in another COBOL program (as defined in an ENTRY statement).
- A program written in another language. For example, to receive the entry address of a C function, call the function with the CALL RETURNING format of the CALL statement. It will return a pointer that you can convert to a procedure-pointer using a form of the SET statement.

Rules for Using Procedure Pointers

A procedure-pointer data item can be set only using the SET statement. For example:

```
CALL 'MyCFunc' RETURNING ptr.
SET proc-ptr TO ptr.
CALL proc-ptr USING dataname.
```

Therefore, if you set your procedure-pointer item to an entry address in a load module called using CALL-*identifier* and your program subsequently CANCELs that called module, then your procedure-pointer item becomes undefined, and reference to it thereafter is not reliable.

For a complete definition of the USAGE IS PROCEDURE-POINTER clause and the SET statement, refer to the *IBM COBOL Language Reference*.

Using Procedure Pointers

Windows Restriction

Windows

In general, SYSTEM (STDCALL) linkage cannot be used for programs that are called via a procedure pointer if they have any arguments. This is due to the associated convention for forming names (also known as “name decoration”).

With STDCALL linkage, the name is formed by suffixing to the entry name the number of bytes in the parameter list. For example, a program named `abc` that passes an argument by reference and a 4-byte integer by value would have 8 bytes in the parameter list; the resulting name would be `_abc@8`. This creates a restriction on setting a procedure pointer to the address of an entry point: because there is no syntactical way to specify the arguments that are to be passed to the entry point on the SET statement, the generated name will have '0' as the number of bytes in the parameter list. This will cause the link to fail due to unresolved external references when the entry point has arguments.

You can use the CALLINT compiler directive to ensure that calls to programs with arguments, made using a procedure pointer, use the OPTLINK convention.

For example:

```
      CBL
      IDENTIFICATION DIVISION.
      PROGRAM-ID. XC.
      DATA DIVISION.
      WORKING-STORAGE SECTION.
      01   PP1 PROCEDURE-POINTER.
      01   HW  PIC X(12).
      PROCEDURE DIVISION USING XA.
   * Use OPTLINK linkage:
         >>CALLINT OPTLINK
         SET PP1 TO ENTRY "X".
```

Figure 87 (Part 1 of 2). Using CALLINIT to Resolve References

```
* Restore default linkage:
       >>CALLINT
       MOVE "Hello World." to HW
       DISPLAY "Calling X."
* Use OPTLINK linkage:
       >>CALLINT OPTLINK
       CALL PP1 USING HW.
* Restore default linkage:
       >>CALLINT
       GOBACK.
    END PROGRAM X.

* Use OPTLINK linkage:
    CBL ENTRYINT(OPTLINK)
    IDENTIFICATION DIVISION.
    PROGRAM-ID. X.
    DATA DIVISION.
    LINKAGE SECTION.
    01   XA    PIC 9(9).
    PROCEDURE DIVISION USING XA.
       DISPLAY XA.
       GOBACK.
    END PROGRAM X.
```

Figure 87 (Part 2 of 2). Using CALLINIT to Resolve References

Without the CALLINT compiler directives and the CALLINT compiler option, an unresolved reference to `_X@0` would be found when the link was done.

Non-COBOL: If the program being called is C or PL/I and uses the STDCALL interface, use the `pragma` statement in the called program to form the name without STDCALL name decoration.

◀Windows

Multiple Entry Points on Windows

Windows▶

The call interface convention SYSTEM (STDCALL) cannot always be used for calling programs with multiple entry points (PROCEDURE DIVISION USING ... AND ENTRY XXX USING ...). If the number of parameters in each entry point is not the same or if the caller does not pass the number of arguments expected by the called entry point, using the STDCALL convention causes unpredictable results due to corruption of the stack.

The STDCALL convention requires the called program to clean up the stack, where the calling program placed the arguments. Because it has no way of determining how many arguments were actually pass to it, it uses the expected number of arguments.

When this does not accurately reflect the number passed, it is impossible to do the clean up correctly.

Because a common exit point can be used for programs with multiple entry points, the different entry points having a different number arguments also makes it impossible to determine correctly how to clean up the stack.

Data Type: Because STDCALL linkage uses four bytes on the stack for each argument, differences in data type are immaterial.

◀Windows

Passing Return Code Information

You can use the RETURN-CODE special register to pass and receive return codes between programs. Methods do not return information in the RETURN-CODE special register, but they can check the register after a CALL to a program.

You can also use the RETURNING phrase on the PROCEDURE DIVISION header in your method to return information to an invoking program or method. If you use PROCEDURE DIVISION. . .RETURNING with CALL. . .RETURNING, the RETURN-CODE register will not be set.

RETURN-CODE Special Register

When a COBOL program returns to its caller, the contents of the RETURN-CODE special register are set according to the value of the RETURN-CODE of the program returning to the caller.

Setting of the RETURN-CODE by the called program is limited to calls between COBOL programs. For example, if your COBOL program calls a C program, you can't expect the calling program's RETURN-CODE to be set.

For equivalent functionality between COBOL and C programs, have your COBOL program call the C program with the RETURNING option. If the C program (function) correctly declares a function value, the RETURNING value of the calling COBOL program will be set.

INVOKE Note: The RETURN-CODE special register is not set by use of the INVOKE statement.

PROCEDURE DIVISION RETURNING . . .

You can use the RETURNING phrase on the PROCEDURE DIVISION header of you program to return information to the calling program:

```
PROCEDURE DIVISION RETURNING dataname2
```

Upon successful return from the called program to its caller, the value in *data-name-2* is stored into the identifier specified in the CALL-RETURNING phrase:

```
CALL . . . RETURNING dataname2
```

CALL . . . RETURNING

The RETURNING phrase on the CALL statement can be specified for calls to functions written in C/C++ or subroutines written in COBOL.

It has the following format:

```
CALL . . . RETURNING dataname2
```

The return value of the called program is stored into *dataname2*.

dataname2 must be defined in the DATA DIVISION of the calling COBOL program. The data type of the return value declared in the target function must be identical to the data type of *dataname2*.

Sharing Data Using the EXTERNAL Clause

Separately compiled programs and methods (including programs in a batch sequence) can share data items by using EXTERNAL clause.

EXTERNAL is coded on the 01-level data description in the WORKING-STORAGE SECTION of a program or method, and the following rules apply:

- Items subordinate to an EXTERNAL group item are themselves EXTERNAL.
- The name used for the data item cannot be used on another EXTERNAL item in the same program.
- The VALUE clause cannot be coded for any group item, or subordinate item, that is EXTERNAL.

Any COBOL program or method in the run unit, having the same data description for the item as the program containing the item, can access and process the data item. For example, if program A had the following data description:

```
   01 EXT-ITEM1     EXTERNAL     PIC 99.
```

program B could access that data item by having the identical data description in its WORKING-STORAGE SECTION.

Remember, any program that has access to an EXTERNAL data item can change its value. Do not use this clause for data items you need to protect.

Sharing Files between Programs (EXTERNAL Files)

Using the EXTERNAL clause for files allows separately compiled programs or methods in the run unit to have access to common files.

The rules for using EXTERNAL files are described in the *IBM COBOL Language Reference*. In addition, it is recommended that:

- The data-name in the FILE STATUS clause of all the programs that will check the file status code must match.

- For all programs that want to check the same file status field, the EXTERNAL clause should be coded on the level-01 data definition for the file status field in each program.

Advantages of EXTERNAL Files

The example on page 401, shows some of the advantages of using EXTERNAL files:

- The main program can reference the record area of the file, although the main program does not contain any I/O statements.
- Each subprogram can control a single I/O function, such as OPEN or READ.
- Each program has access to the file.

Example Using EXTERNAL Files

The following table gives the names and describes the function of the main program and subprograms used in the example shown in Figure 89 on page 401.

Figure 88. Program Names for Input-Output Using EXTERNAL Files

Name	Function
ef1	This is the main program. It calls all the subprograms, and then verifies the contents of a record area.
ef1openo	This program opens the external file for output and checks the File Status Code.
ef1write	This program writes a record to the external file and checks the File Status Code.
ef1openi	This program opens the external file for input and checks the File Status Code.
ef1read	This program reads a record from the external file and checks the File Status Code.
ef1close	This program closes the external file and checks the File Status Code.

Additionally, COPY statements ensure that each subprogram contains an identical description of the file.

Each program in the example declares a data item with the EXTERNAL clause in its WORKING-STORAGE SECTION. This item is used for checking file status codes, and is also placed using the COPY statement.

Each program uses three copy library members:

- The first is named `efselect` and is placed in the File-Control paragraph.

```
Select ef1
   Assign To ef1
   File Status Is efs1
   Organization Is Sequential.
```

- The second is named `effile` and is placed in the FILE SECTION.

```
Fd ef1 Is External
          Record Contains 80 Characters
          Recording Mode F.
01  ef-record-1.
    02 ef-item-1     Pic X(80).
```

- The third is named `efwrkstg` and is placed in the WORKING-STORAGE SECTION.

```
01  efs1             Pic 99 External.
```

```
 Identification Division.
 Program-Id.
     ef1.
*
* This is the main program that controls the external file
* processing.
*
 Environment Division.
 Input-Output Section.
 File-Control.
     Copy efselect.
 Data Division.
 File Section.
     Copy effile.
 Working-Storage Section.
     Copy efwrkstg.
 Procedure Division.
     Call "ef1openo"
     Call "ef1write"
     Call "ef1close"
     Call "ef1openi"
     Call "ef1read"
     If ef-record-1 = "First record" Then
       Display "First record correct"
     Else
       Display "First record incorrect"
       Display "Expected: " "First record"
       Display "Found   : " ef-record-1
     End-If
     Call "ef1close"
     Goback.
 End Program ef1.
```

Figure 89 (Part 1 of 4). Input-Output Using EXTERNAL Files

```
       Identification Division.
       Program-Id.
           ef1openo.
      *
      * This program opens the external file for output.
      *
       Environment Division.
       Input-Output Section.
       File-Control.
           Copy efselect.
       Data Division.
       File Section.
           Copy effile.
       Working-Storage Section.
           Copy efwrkstg.
       Procedure Division.
           Open Output ef1
           If efs1 Not = 0
             Display "file status " efs1 " on open output"
             Stop Run
           End-If
           Goback.
       End Program ef1openo.
       Identification Division.
       Program-Id.
           ef1write.
      *
      * This program writes a record to the external file.
      *
       Environment Division.
       Input-Output Section.
       File-Control.
           Copy efselect.
       Data Division.
       File Section.
           Copy effile.
       Working-Storage Section.
           Copy efwrkstg.
       Procedure Division.
           Move "First record" to ef-record-1
           Write ef-record-1
           If efs1 Not = 0
             Display "file status " efs1 " on write"
             Stop Run
           End-If
           Goback.
       End Program ef1write.
```

Figure 89 (Part 2 of 4). Input-Output Using EXTERNAL Files

```
 Identification Division.
 Program-Id.
     ef1openi.
*
* This program opens the external file for input.
*
 Environment Division.
 Input-Output Section.
 File-Control.
     Copy efselect.
 Data Division.
 File Section.
     Copy effile.
 Working-Storage Section.
     Copy efwrkstg.
 Procedure Division.
     Open Input ef1
     If efs1 Not = 0
       Display "file status " efs1 " on open input"
       Stop Run
     End-If
     Goback.
 End Program ef1openi.
 Identification Division.
 Program-Id.
     ef1read.
*
* This program reads a record from the external file.
*
 Environment Division.
 Input-Output Section.
 File-Control.
     Copy efselect.
 Data Division.
 File Section.
     Copy effile.
 Working-Storage Section.
     Copy efwrkstg.
 Procedure Division.
     Read ef1
     If efs1 Not = 0
       Display "file status " efs1 " on read"
       Stop Run
     End-If
     Goback.
 End Program ef1read.
```

Figure 89 (Part 3 of 4). Input-Output Using EXTERNAL Files

```
       Identification Division.
       Program-Id.
           ef1close.
      *
      * This program closes the external file.
      *
       Environment Division.
       Input-Output Section.
       File-Control.
           Copy efselect.
       Data Division.
       File Section.
           Copy effile.
       Working-Storage Section.
           Copy efwrkstg.
       Procedure Division.
           Close ef1
           If efs1 Not = 0
             Display "file status " efs1 " on close"
             Stop Run
           End-If
           Goback.
       End Program ef1close.
```

Figure 89 (Part 4 of 4). Input-Output Using EXTERNAL Files

Command Line Arguments

OS/2 and Windows call all main programs with a string that gives the command line arguments. If the `-host` compiler option was specified this string will be in EBCDIC and the length will be in “big endian” format. Figure 90 on page 405 shows how to read the command line arguments:

```
⋮
       IDENTIFICATION DIVISION.
       PROGRAM-ID. "testarg".
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
      *
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      *
       linkage section.
       01  os-parm.
           05 parm-len         pic s999 comp.
           05 parm-string.
               10 parm-char             pic x occurs 0 to 100 times
                               depending on parm-len.
      *
       PROCEDURE DIVISION using os-parm.
           display "parm-len=" parm-len
           display "parm-string='" parm-string "'"
           evaluate parm-string
             when "01" display "case  one"
             when "02" display "case two"
             when "95" display  "case ninety five"
             when other display "case unknown"
           end-evaluate
           GOBACK.
```

Figure 90. testarg — An Example of Command Line Arguments

The result of compiling and running the program:

```
cob2 testarg.cbl
testarg 95
```

is:

```
parm-len=002
parm-string='95'
case ninety five
```

Chapter 21. Programming for a DB2 Environment

In general, the coding for your COBOL program will be the same whether or not you want it to access a DB2 database. However, to retrieve, update, insert, and delete DB2 data and use other DB2 services, you must use SQL statements.

To communicate with DB2, you need to do the following:

- Delimit SQL statements with EXEC SQL and END-EXEC STATEMENTS
- Declare a communications area (SQLCA) in the WORKING-STORAGE SECTION
- Declare all host variables used in SQL statements in the WORKING-STORAGE OR LINKAGE sections
- Code any SQL statements you need
- Start DB2 if it is not already started
- Compile with the SQL compiler option
- Check the return code from DB2 in the SQLCA to handle exceptional conditions that are indicated

These basics of coding SQL in a COBOL program are described in detail in the *DB2 Application Programming Guide* and the *DB2 SQL Reference*.

Compiling with the DB2 Co-Processor

Your source program containing embedded SQL statements is handled by the compiler without your having to use a separate pre-processor. When the compiler encounters SQL statements and at significant points in the source program, it interfaces with the DB2 co-processor, which processes the SQL statements by taking appropriate actions and indicating to the compiler what native COBOL statements to generate at that point.

Because the compiler is working in conjunction with the DB2 co-processor, DB2 must be started before you compile your program. To be connected to the target database for the compile, you can connect before you start the compile or have the compiler make the connection for you by specifying the database either using the DATABASE suboption in the SQL option or by naming it in the DB2DBDFT environment variable.

Options for the DB2 Co-Processor

The option string that you provide on the SQL compiler option is made available to the DB2 co-processor. The content of the string is viewed solely by the DB2 co-processor and not by the compiler. The following cob2 command will pass the database name “SAMPLE” and the DB2 option “BLOCKING ALL” to the co-processor:

```
  cob2 -q"sql('database sample blocking all')" mysql.cbl...
```

The SQL options that you include in the suboption string are cumulative. See the DB2 Command Reference for information on these options.

How Options Are Accumulated

The options specified from multiple sources are concatenated in the order of the specifications.

For example, the command

```
cob2 mypgm.cbl -q"SQL('string')"
```

and the mypgm.cbl source file with

```
 cbl ... SQL("string2") ...
 cbl ... SQL("string3") ...
```

will result in the SQL option string passed to the DB2 co-processor to be

"string1 string2 string3"

Note that the concatenated strings are delimited with single spaces. When multiple instances of the same SQL suboptions are found, the last specification of that suboption in the concatenated string will be in effect.

This concatenation of multiple SQL option specifications allows you to separate SQL suboptions which may not fit into a single CBL statement into multiple CBL statements.

The compiler limits the length of the concatenated DB2 option string to 4K bytes.

Package and Bind File Names

Two of the suboptions that you can specify with the SQL option are `package name` and `bind file name`. If you do not specify these options, default names are constructed for them based on the source file name for a non-batch compilation and on the first program for a batch compilation. For subsequent, non-nested, programs of a batch compilation, the names are based on the PROGRAM ID of each program.

Package Name

The base name (the source file name or the PROGRAM ID) is modified as follows:

- Names longer than eight characters are truncated to eight characters
- Letters are folded to upper case
- Any character other than A-Z, 0-9, or _ (under score) is changed to 0
- If the first character is not alphabetic, it is changed to A

Thus, if the base name is *9123aB-cd*, the package name would be *A123AB0C*.

Bind File Name

The extension .BND is added to the base name.

Unless explicitly specified, the file name is relative to the current directory.

SQL and COBOL

Ignored Options

The following options, which were meaningful to and used by the pre-processor, are ignored by the co-processor:

MESSAGES
NOLINEMACRO
OPTLEVEL
OUTPUT
SQLCA
TARGET
WCHARTYPE

SQL INCLUDE Statement

An SQL INCLUDE statement is treated identically to a native COBOL COPY statement, including the path search and the file extensions used.

For example,

```
EXEC SQL INCLUDE name
```

is treated identically to

```
COPY name.
```

The *name* on an SQL INCLUDE statement follows the same rules as those for the copy text-name and is processed identically to a COPY statement with that text-name without a REPLACING clause. See “Compiler Environment Variables” on page 136 and the discussion on the COPY statement in the “Compiler-Directing Statements” on page 202 section for details.

COBOL does not use the DB2 environment variable DB2INCLUDE for SQL INCLUDE processing. However, if you are using the standard DB2 copy files, there are no other settings that you have to make. If the search rules call for using SYSLIB as the library name, the compiler will find the copy files by using the DB2 environment variable DB2PATH, which is set during DB2 install, to extend the setting of SYSLIB to include the DB2 include directory. The SYSLIB string that is used is essentially `%SYSLIB%;%DB2PATH%\INCLUDE\COBOL_A`.

COBOL Language Usage with SQL

Some restrictions on the use of COBOL language that applied when the pre-processor was used are lifted with the use of the co-processor.

Specifically the following are permitted:

- `EXEC SQL BEGIN DECLARE SECTION` and `EXEC SQL END DECLARE SECTION` statements are no longer required to identify host variables used on SQL statements
- Batch compilation is supported: the source file may contain multiple non-nested COBOL programs

- The source program may contain nested programs
- The source program may contain object-oriented COBOL language extensions

It is recommended that binary data items that are specified in an SQL statement be:

- Declared as USAGE COMP-5 *or*
- Used with the TRUNC(BIN) option and the BINARY(NATIVE) option if USAGE BINARY, COMP or COMP-4 item is specified

If you specify a BINARY, COMP, or COMP-4 item with option TRUNC(OPT) or TRUNC(STD) in effect, it will be accepted by the compiler but may invalidate the data due to the application of the decimal truncation rule. It is your responsibility to insure that truncation does not affect the validity of the data.

For information about using SQL with System/390 host data types, see Appendix B, "System/390 Host Data Type Considerations" on page 543.

Level of SQL Support

SQL statements are supported at the level of DB2 for OS/2 Single-User, Version 2 on OS/2 and DB2 for Windows 95 and Windows NT Version 2.1.1 on Windows, including:

- Large Objects (BLOB, CLOB, ...)
- Compound SQL

Limit: The size of large objects is currently limited to 16M bytes for a group or elementary data item.

Testing the Return Code

In general, when DB2 finishes processing an SQL statement, DB2 sends back a return code in the SQLCODE OF SQLCA and SQLSTATE OF SQLCA fields. Your program should test the SQLCODE and/or SQLSTATE fields and take any necessary action depending on whether the operation succeeded or failed.

Chapter 22. Programming for a CICS Environment

When you develop programs to run under the Customer Information Control System (CICS), be aware of the steps you must take as well as certain COBOL language restrictions. A discussion of these coding considerations follows. For additional information about developing COBOL programs to run under CICS, consult the CICS *Application Programming Guide*.

An Overview of COBOL in a CICS Environment

CICS applications can be written in COBOL and run on the PC using any of the following CICS environments:

- VisualAge CICS Enterprise Application Development
- CICS for OS/2
- CICS for Windows NT

VisualAge CICS Enterprise Application Development is available for both OS/2 and Windows NT, and is primarily intended for developing CICS applications that will ultimately run in a mainframe environment. (For convenience, this will be referred to as VisualAge CICS for the remainder of this chapter.)

CICS for OS/2 and CICS for Windows NT are primarily intended for developing and running workstation-based applications, although CICS for OS/2 supports the host data type options, so it can also be used for developing mainframe-based applications.

Where there are differences between the implementation and use of these CICS products, the differences are indicated throughout this chapter.

Installing and Running CICS Programs

After having installed and configured CICS, there are a number of steps involved in preparing COBOL applications to run under CICS. In general, the process is the same regardless of which CICS system you use, but there are some differences in some of the detail. The steps are outlined here, and where there are differences between the CICS systems, the process for each CICS system is shown.

1. Initialize the environment.

   VisualAge CICS, CICS for OS/2

   Initialize the CICS environment using the CICSENV command file. You can edit CICSENV.CMD to configure your CICS and COBOL environment variables (see CICS *Customization* for details on CICSENV). For example, make sure that your programs, files, and copybooks are accessible by the operating system.

   Note that you do not normally need to run CICSENV before starting a CICS application, because it will be run automatically on the first invocation of a CICS command. However, if you have both VisualAge CICS and CICS for Windows NT installed on your system, you should run CICSENV before building your program (that is, before the CICSTCL step) if you want VisualAge CICS to be used instead of CICS for Windows NT.

CICS for Windows NT
: Initialize the CICS for Windows NT environment. Note that the environment variables for CICS for Windows NT are set when the CICS region is started. To set environment variables that are unique to a particular region, they must be set in `\var\cics_regions\`*xxx*`\environment` (where *xxx* is the name of the region). To set environment variables that are in effect for all CICS regions, they must be set in the system environment variables section of Windows NT. Environment variables set in the user environment variables section of Windows NT will not be seen by any CICS for Windows NT region.

2. Create the application.

   Create your COBOL application program. For CICS-specific COBOL programming considerations, see "Preparing COBOL Applications to Run under CICS" on page 412. Specifically, use an editor to do the following:

   - Code your program using COBOL statements and CICS commands
   - Create COBOL copybooks
   - Create CICS screen maps used by your program

   For a detailed description of COBOL programming under CICS, see the CICS *Application Programming Guide*.

3. Process the maps.

   Process the CICS screen maps using the CICSMAP command. Consult the CICS *Application Programming Guide* for details on CICSMAP.

4. Compile the program.

   Use the CICS command CICSTCL to:

   Translate CICS commands will be translated into valid COBOL statements.

   Compile The COBOL compiler will be invoked and the program will be compiled.

   Link The compiled COBOL program will be linked using standard linkage routines.

5. Start CICS.

   VisualAge CICS, CICS for OS/2
   : Start CICS using CICSRUN.

   CICS for Windows NT
   : This step is deferred until later in the process.

6. Define the resources.

   Define your application's resources, such as transactions, application programs, and files.

VisualAge CICS, CICS for OS/2
: This is done using the CEDA transaction.

CICS for Windows NT
: This is done through the CICS Administration Utility.

7. Activate the resources.

   VisualAge CICS, CICS for OS/2
   Activate CICS resources (such as PCT and FCT entries). You can do this in either of two ways:

   - By shutting down CICS using the CQIT transaction, then restart it using CICSRUN.
   - By using the Install action of the CEDA transaction.

   CICS for Windows NT
   The resources will be activated when the CICS region is started (in the next step).

8. Start CICS.

   VisualAge CICS, CICS for OS/2
   Your CICS system should have already been started.

   CICS for Windows NT
   Start your CICS region using the CICS Administration Utility.

9. Run the application.

   At your CICS terminal, run the application by entering the 4-character transaction-id associated with the application.

If you want to execute code using CICS ECI (External Call Interface), you need to have CICS Client installed. Otherwise, you will encounter an error due to a missing DLL. You also need to start CICS Client before executing.

Preparing COBOL Applications to Run under CICS

In general, the COBOL language is supported in a CICS environment. However, there are certain restrictions and considerations you should be aware of when preparing COBOL applications to run on CICS.

Additional Language Restrictions

The following guidelines should be followed when coding COBOL programs that run under CICS:

- Do not use EXEC, CICS, DLI, or END-EXEC for variable names.
- It is recommended that you do not use the FILE-CONTROL entry in the ENVIRONMENT DIVISION.
- It is recommended that you do not use the FILE SECTION of the DATA DIVISION.
- Do not use user-specified parameters to the main program.

- It is recommended that you do not use USE declaratives (except USE FOR DEBUGGING).
- The following COBOL language statements are not recommended for use in a CICS environment:
  - ACCEPT (Format 1 or 2— see “System Date under CICS” on page 414)
  - CLOSE
  - DELETE
  - DISPLAY
  - MERGE
  - OPEN
  - READ
  - REWRITE
  - SORT
  - START
  - STOP *literal*
  - WRITE

  Attention: Apart from some forms of the ACCEPT statement, mainframe CICS does not support any of the COBOL language elements in the preceding list. If you use any of these COBOL language elements, be aware that:

  - The application is not completely portable to the mainframe CICS environment.
  - In the case of a CICS failure, a backout (restoring the resources associated with the failed task) will not be possible.
- When coding nested (contained) programs, pass DFHEIBLK and DFHCOMMAREA as parameters to any nested programs that contain EXEC commands and/or references to the EIB. The same parameters must also be passed to any program that forms part of the control hierarchy between such a program and its top level program.

Selecting Compiler Options

TRUNC(BIN) is the recommended option under CICS. However, if you are certain that the non-truncated values of BINARY, COMP, or COMP-4 data items conform to PICTURE specifications, using TRUNC(OPT) may improve program performance.

COMP-5 can be used instead of BINARY, COMP, or COMP-4 as EXEC CICS command arguments. COMP-5 is treated with the behavior of BINARY, COMP, or COMP-4 as if BINARY(NATIVE) and TRUNC(BIN) were in effect, regardless of explicitly setting those options.

The PGMNAME(MIXED) option must be used for applications that use CICS Client.

The following options should be avoided when compiling programs to run in a CICS environment:

NOLIB
TRUNC(STD) or TRUNC(OPT)

All other COBOL compiler options are supported. For detailed information on individual compiler options, see Chapter 10, "Compiler Options" on page 160.

For additional information about CICS access and System/390 host data types, see Appendix B, "System/390 Host Data Type Considerations" on page 543.

EBCDIC-Enabled COBOL Programs on CICS

VisualAge CICS, CICS for OS/2

CICS provides support for running COBOL programs as EBCDIC-enabled programs. In order to prepare your IBM COBOL program to run on CICS as an EBCDIC-enabled program, you must do the following:

- Translate the COBOL program using the BINARY(S370) and the EBCDIC translator options.
- Compile the program using the CHAR(EBCDIC), COLLSEQ(EBCDIC), and the BINARY(S390) compiler options.

For more information about EBCDIC-enabled programs on CICS, see the CICS *Application Programming Guide*.

CICS for Windows NT

CICS for Windows NT does not support the host data type options, so EBCDIC enablement is not supported.

Selecting Run-Time Options

Use the FILESYS run-time option to specify the file system used for files when no specific file selection has been made on the ASSIGNment name. For a detailed description of FILESYS, see Chapter 12, "Run-Time Options" on page 240.

Planning for ASCII-EBCDIC Differences

If your CICS program is running on an ASCII platform (such as OS/2 or AIX), and you access EBCDIC data, be aware that the neither CICS nor the COBOL run time will automatically convert the data to the ASCII collating sequence.

Some data access methods (such as VSAM) can carry out such conversions automatically, but you should not assume that the data will be converted. If your program is designed to access mainframe data and you do not build with the host data type options, you might want to add logic to your program to test whether or not the data is EBCDIC and, if necessary, carry out an explicit collating sequence conversion.

System Date under CICS

You should not use a Format 1 ACCEPT statement in a CICS program. Format 2 ACCEPT is supported with the four-digit year options; that is:

```
ACCEPT identifier FROM DATE YYYYMMDD
ACCEPT identifier FROM DAY YYYYDDD
```

The recommended ways of retrieving the system date in a CICS program are these forms of the ACCEPT statement, and the CURRENT-DATE intrinsic function. These methods work in both CICS and non-CICS environments.

Note: The following forms of the ACCEPT statement to receive 2-digit year dates are not supported under CICS:

```
        ACCEPT identifier FROM DATE
        ACCEPT identifier FROM DAY
```

Dynamic Calls under CICS

Dynamic calls work in the CICS environment, however you have to be careful to set the COBPATH environment correctly. Consider the following example.

The program contains

```
WORKING-STORAGE SECTION.
  01  WS-COMMAREA     PIC 9 VALUE ZERO.
  77   SUBPNAME       PIC X(8) VALUE SPACES
 ...
PROCEDURE DIVISION.
  MOVE 'alpha' TO SUBPNAME.
  CALL SUBPNAME USING DFHEIBLK, DFHCOMMAREA, WS-COMMAREA.
 ...
```

Notice that since `alpha` is a COBOL program containing CICS statements, CICS control blocks DFHEIBLK and DFHCOMMAREA must be passed to `alpha`. The source for `alpha` is in the file `alpha.ccp`. The CICS command CICSTCL is used to translate, compile and link `alpha.ccp`.

VisualAge CICS, CICS for OS/2
: CICSTCL creates a DLL called `alpha.dll`. The directory where `alpha.dll` resides must be included in the COBPATH environment variable. CICS documentation describes how to set environment variables for CICS.

CICS for Windows NT
: CICSTCL creates a DLL called `alpha.ibmcob`. The directory where `alpha.ibmcob` resides must be included in the COBPATH environment variable.

Notes:

1. The `-lIBMCOB` option must be used with CICSTCL.
2. The COBPATH environment variable must be set in the system environment variables section because the CICS for Windows NT region does not recognize environment variable settings from the user environment variables section.

   Alternatively you could set COBPATH in the `\var\cics_regions\`*xxx*`\environment` file in which case it would be effective only for the *xxx* region.

DLL Considerations

The same DLL containing COBOL program(s) should not be used in more than one run unit within the same CICS transaction, or the results will be unpredictable. For example, Figure 91 on page 416 shows a CICS transaction where the same subprogram is called from two different run units.

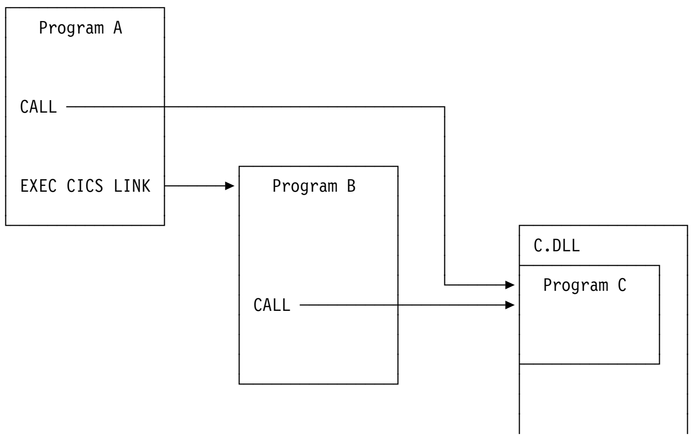


Figure 91. CICS Transaction with Two Run Units

In this example:

- Program A CALLs Program C (in C.DLL)
- Program A LINKs to Program B using an EXEC CICS LINK command. This becomes a new run unit within the same transaction.
- Program B CALLs Program C (in C.DLL)

Programs A and B are sharing the same copy of Program C, and any changes to its state will affect both. In the CICS environment, programs in a DLL are initialized (both the WSCLEAR compiler option and VALUE clause initialization) only on the first call within a run unit. If a COBOL subprogram is called more than once, from either the same or different main programs, the subprogram will be initialized only on the first call.

If you need the subprogram initialized on the first call from each main program, you should statically link a separate copy of the subprogram with each calling program.

If you need the subprogram initialized on every call, you should use one of the following methods:

- Put data to be reinitialized in the subprogram's Local-Storage Section, rather than Working-Storage. Note that this affects initialization by the VALUE clause only, not by the WSCLEAR compiler option.
- CANCEL the subprogram after each use, so the next call will be to the program in its initial state.
- Add the INITIAL attribute to the subprogram.

Accessing Btrieve Data

OS/2

Your non-CICS programs are able to access Btrieve files (the default file system used by both VisualAge CICS on OS/2 and CICS for OS/2). For more information, see "Accessing Files" on page 97.

OS/2

Calls between COBOL and C++ under CICS

Be aware of three rules governing calls between COBOL and C/C++ programs under CICS:

1. COBOL programs which contain CICS commands can call C/C++ programs as long as the called C/C++ programs do not contain any CICS commands.
2. C/C++ Programs which contain CICS commands can call COBOL programs as long as the called COBOL programs do not contain any CICS commands.
3. COBOL programs can EXEC CICS LINK or EXEC CICS XCTL to a C/C++ program regardless of whether or not the C/C++ program contains CICS commands.

Therefore, if your COBOL program invokes a C/C++ program that contains CICS commands (or vice versa), use EXEC CICS LINK or EXEC CICS XCTL rather than the COBOL CALL statement.

Debugging CICS Programs

Before you debug your CICS programs, you need to translate them into COBOL. Then you debug CICS programs the same way you would debug any other COBOL program. For an overview of COBOL language-based debugging techniques, see Chapter 13, "Debugging Techniques" on page 244.

Alternatively, you can debug CICS programs using the graphical debugger shipped with the product. See your CICS *Application Programming Guide* for instructions about how to invoke the graphical debugger under CICS. Be sure to instruct the compiler to produce symbolic information used by the graphical debugger (see "Compiling and Linking Programs" on page 142).

Chapter 23. Open Database Connectivity

This chapter contains information to help you use the Open Database Connectivity (ODBC) interface in your COBOL applications. With ODBC, not only can you access data from a variety of databases and file systems that support the ODBC interface, but you can do so dynamically.

Your COBOL applications that use embedded SQL for database access must be processed by a preprocessor or coprocessor for a particular database and have to be recompiled if the target database changes. Because ODBC is a call interface, there is no compile-time designation of the target database as there is with embedded SQL. Not only can you avoid having multiple versions of your application for multiple databases, but your application can dynamically determine which database to target.

Introducing ODBC

ODBC is a specification for an application program interface (API) that enables applications to access multiple database management systems using Structured Query Language (SQL).

ODBC permits maximum interoperability: a single application can access many different database management systems. This enables you to develop, compile, and ship an application without targeting a specific type of data source. Users can then add the database drivers, which link the application to the database management systems of their choice.

Background

The X/Open Company and the SQL Access Group jointly developed a specification for a callable SQL interface, referred to as the *X/Open Call Level Interface*. The goal of this interface is to increase portability of applications by enabling them to become independent of any one database vendor's programming interface.

ODBC was originally developed by Microsoft for Microsoft operating systems based on a preliminary draft of X/Open CLI. Since this time, other vendors have provided ODBC drivers that run on other platforms, such as OS/2 and UNIX systems. The IBM VisualAge COBOL package includes the DataDirect** ODBC drivers from INTERSOLV**, Inc.

| The descriptions and examples in this chapter apply to ODBC Version 3.0. However, Version 2.x support is also provided. If you are developing your application for Version 2.x ODBC, you will need to use the Version 2.x copybooks instead of the Version 3.0 copybooks listed here. For details, see "Using the Supplied Copybooks" on page 423.

ODBC Driver Manager

When you use the ODBC interface, your application makes calls through a Driver Manager. The Driver Manager dynamically loads the necessary driver for the database server to which the application connects. The driver, in turn, accepts the call, sends the SQL to the specified data source (database), and returns any result.

Choosing Embedded SQL or ODBC

Embedded SQL and ODBC have advantages particular to them. Some of the advantages of embedded SQL are:

- Static SQL usually provides better performance than dynamic SQL. It does not have to be prepared at run time, thus reducing both processing and network traffic.
- With static SQL, database administrators have to grant users access to a package only rather than access to each table or view that will be used.

Some of the advantages of ODBC are:

- It provides a consistent interface regardless of what kind of database server is used.
- You can have more than one concurrent connection.
- Applications do not have to be bound to each database on which they will execute. Although IBM VisualAge COBOL does this bind for you automatically, it binds automatically to only one database. If you want to choose which database to connect to dynamically at run time, you must take extra steps to bind to a different database.

Using the ODBC Drivers

IBM VisualAge COBOL provides an ODBC Driver Manager and a set of ODBC database drivers under an agreement with INTERSOLV, Inc.

To enable ODBC for data access in IBM VisualAge COBOL, you must:

1. Install the ODBC Driver Manager and drivers by selecting the “ODBC Drivers” component during IBM VisualAge COBOL installation.
2. Install the RDBMS client (for example, Oracle 7 SQL*NET, DB2 CAE, etc.).

Important: During the installation process, a license file for the ODBC driver is installed on your system.

File `ivib.lic` is installed in *x*:\cobdir\ODBC, where *x* and *cobdir* is the drive and directory respectively, where COBOL is installed. (*cobdir* defaults to `IBMCOBOL` for OS/2 and `IBMCOBW` for Windows.)

You must keep this file installed in the install directory, because it will be used when you run your application to verify that you are licensed to use the ODBC driver. In “Setting Licensing Information for ODBC Driver Manager/Driver” on page 430 you learn how to use a function call to trigger the verification.

On-line Help

On-line help is available for the ODBC drivers, both as a reference book and as context-sensitive help. The specific file names and so on may differ; you should note the names given in this section for the file names for IBM VisualAge COBOL.

Environment-Specific Information

The ODBC drivers are 32-bit drivers. The required network software supplied by your database system vendors must be 32-bit compliant.

OS/2

OS/2 ▶

The drivers shipped for OS/2 are at the ODBC 2.1 level.

ODBC.INI: OBDC.INI is an operating system binary file located in the directory specified by the USER_INI environment variable. Since this file is binary, you cannot edit it with a text editor.

Configuring Data Sources: A **data source** consists of a DBMS and any remote operating system and network necessary to access it. After the drivers have been installed, the data source must be configured using the ODBC Administrator program. Start the ODBC Administrator by double-clicking on the ODBC Administrator icon in the Tools folder.

Driver Names: The file names for the drivers that come with IBM VisualAge COBOL start with IB and have a file extension of .DLL. (They are dynamic link libraries.) The number in the name corresponds to the version level of the driver. When you install the ODBC drivers, your CONFIG.SYS file is modified to add the correct path to the environment variable LIBPATH.

◀ OS/2

Windows

Windows ▶

The drivers shipped for Windows are at the ODBC 3.0 level.

ODBC.INI: ODBC.INI is a subkey of the HKEY_CURRENT_USER\\SOFTWARE\\ODBC key in the Windows NT and Windows 95 registry. The ODBC.INI subkey is maintained by the ODBC Administrator, which is located in the main COBOL program group. Since Windows can support multiple users, the ODBC.INI subkey is stored under unique user keys in the registry.

Configuring Data Sources: A **data source** consists of a DBMS and any remote operating system and network necessary to access it. After the drivers have been installed, the data source must be configured using the ODBC Administrator program, which is located in the main COBOL program group. Because Windows 95 and Windows NT can host multiple users, each user must configure their own data sources.

For detailed configuration information for the specific driver you wish to configure, refer to the appropriate section of the on-line help.

Driver Names: The file names for the drivers that come with IBM VisualAge COBOL start with IB and have a file extension of .DLL. (They are dynamic link libraries.) The number in the name corresponds to the version level of the driver. When you install the ODBC drivers, your Registry is modified to add the correct path to your environment.

◀ Windows

Connecting to a Data Source

Your ODBC application will need to connect to the data source either using a logon dialog box or a connection string, depending on the data source.

Using a Logon Dialog Box

Some ODBC applications display a logon dialog box when you are connecting to a data source. In these cases, the data source name has already been specified.

In the logon dialog box, do the following:

1. Type the name of the remote database or select the name of the remote database from the Database Name drop-down list.

   You must have cataloged any database you want to access from the client.

2. If required, type your user name (authorization ID).

3. If required, type your password.

   If you leave your user name and password blank, the ODBC application assumes you have already logged on using SQLLOGN2 (under DOS) or using User Profile Management (under OS/2). If you have not, the application returns an error. You must either type your user name and password in the dialog box or log on using SQLLOGN2 and STARTDRQ (under DOS) or using User Profile Management (under OS/2).

4. Click OK to complete the logon and to update the values in ODBC.INI.

Using a Connection String

If your application requires a connection string to connect to a data source, you must specify the data source name that tells the driver which ODBC.INI section to use for the default connection information. Optionally, you may specify `attribute=value` pairs in the connection string to override the default values stored in ODBC.INI. These values are not written to ODBC.INI.

You can specify either long or short names in the connection string. The connection string has the form:

```
DSN=data_source_name[;attribute=value[;attribute=value]...]
```

An example of a connection string for INFORMIX 5 is

```
DSN=INFORMIX TABLES;DB=PAYROLL
```

Supported ODBC Functions

| For a list of ODBC functions that are supported by the supplied drivers, see the *ODBC Conformance Level* topic of the on-line help for each individual driver.

Error Messages

Error messages may come from

- An ODBC driver
- The database system
- The Driver Manager

Message Format

An error reported on an ODBC driver has the following format:

```
[vendor] [ODBC_component] message
```

ODBC_component is the component in which the error occurred. For example, an error message from INTERSOLV's SQL Server driver would look like this:

```
[INTERSOLV] [ODBC SQL Server driver] Login incorrect.
```

If you get this type of error, check the last ODBC call your application made for possible problems or contact your ODBC application vendor.

An error that occurs in the data source includes the data source name, in the following format:

```
[vendor] [ODBC_component] [data_source] message
```

With this type of message, ODBC_component is the component that received the error from the data source indicated. For example, you may get the following message from an Oracle data source:

```
[INTERSOLV] [ODBC Oracle driver] [Oracle] ORA-0919: specified length too long for CHAR column
```

If you get this type of error, you did something incorrectly with the database system. Check your database system documentation for more information or consult your database administrator. In this example, you would check your Oracle documentation.

The Driver Manager is a DLL that establishes connections with drivers, submits requests to drivers, and returns results to applications. An error that occurs in the Driver Manager has the following format:

```
[vendor] [ODBC DLL] message
```

vendor can be Microsoft or INTERSOLV. For example, an error from the Microsoft Driver Manager might look like this:

```
[Microsoft] [ODBC DLL] Driver does not support this
function
```

ODBC APIs from COBOL

Included with IBM VisualAge COBOL are copybooks that make it easier for you to access data bases with ODBC drivers using ODBC calls from your COBOL programs. This section describes the supplied copybooks, how ODBC API argument types map to COBOL data descriptions, and additional COBOL functions and considerations applicable to ODBC APIs.

For details on the ODBC APIs, see the on-line help.

For specific information related to an ODBC driver, such as the ODBC level or extensions supported by that driver, please refer to the specifications available with that driver.

The following illustrate how to access ODBC from COBOL programs:

"CALL Interface Convention"

"Using the Supplied Copybooks"

"Mapping of ODBC C Types" on page 425

"Passing a Pointer as an Argument" on page 426

"Accessing Function Return Values" on page 428

"Testing Bits with a Bit Mask" on page 429

"Setting Licensing Information for ODBC Driver Manager/Driver" on page 430

LIB Files:

OS/2 ▶ When you link your ODBC applications, you must include the import library ODBC.LIB, which is supplied with IBM VisualAge COBOL. ◀ OS/2

Windows ▶ When you link your ODBC applications, you must include the import library ODBC32.LIB, which is included in the ODBC SDK (from Microsoft). ◀ Windows

CALL Interface Convention

Programs making ODBC calls must be compiled with the CALLINT(SYSTEM) option or use the >>CALLINT SYSTEM directive for ODBC calls.

Programs making ODBC calls must be compiled with the PGMNAME(MIXED) compiler option.

Using the Supplied Copybooks

The copybooks described and listed here are for ODBC Version 3.0. However, Version 2.x copybooks are also supplied, and you can substitute them for the Version 3.0 copybooks if you need to develop applications for ODBC Version 2.x. The names of the copybooks are as listed in Figure 92.

Figure 92. Supplied copybooks for ODBC

Version 3.0	Version 2.x	Description
ODBC3.CPY	ODBC2.CPY	Symbols and constants
ODBC3D.CPY	ODBC2D.CPY	Data Division definitions
ODBC3P.CPY	ODBC2P.CPY	Procedure Division statements
ODBC3EG.CBL	ODBC2EG.CBL	Sample program

The supplied copybook, `ODBC3.CPY`, defines the symbols for constant values described for ODBC APIs, mapping constants used in calls to ODBC APIs to symbols specified in ODBC guides so that argument (input and output) and function return values can be specified and tested.

Some COBOL-specific adaptations have been made:

- Underscores, "_", are replaced with hyphens, "-" in the copybook. For example, SQL_SUCCESS is specified as SQL-SUCCESS.
- Names longer than 30 characters are truncated or abbreviated to 30 characters. Figure 93 shows the names that are longer than 30 characters, and their corresponding COBOL names.

Figure 93 (Page 1 of 2). ODBC Names Truncated or Abbreviated for COBOL

ODBC C #define symbol > 30 characters long	Corresponding COBOL name
SQL_AD_ADD_CONSTRAINT_DEFERRABLE	SQL-AD-ADD-CONSTRAINT-DEFER
SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED	SQL-AD-ADD-CONSTRAINT-INIT-DEF
SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE	SQL-AD-ADD-CONSTRAINT-INIT-IMM
SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE	SQL-AD-ADD-CONSTRAINT-NON-DEFE
SQL_AD_CONSTRAINT_NAME_DEFINITION	SQL-AD-CONSTRAINT-NAME-DEFINIT
SQL_AT_CONSTRAINT_INITIALLY_DEFERRED	SQL-AT-CONSTRAINT-INITIALLY-DE
SQL_AT_CONSTRAINT_INITIALLY_IMMEDIATE	SQL-AT-CONSTRAINT-INITIALLY-IM
SQL_AT_CONSTRAINT_NAME_DEFINITION	SQL-AT-CONSTRAINT-NAME-DEFINIT
SQL_AT_CONSTRAINT_NON_DEFERRABLE	SQL-AT-CONSTRAINT-NON-DEFERRAB
SQL_AT_DROP_TABLE_CONSTRAINT_CASCADE	SQL-AT-DROP-TABLE-CONSTRAINT-C
SQL_AT_DROP_TABLE_CONSTRAINT_RESTRICT	SQL-AT-DROP-TABLE-CONSTRAINT-R
SQL_C_INTERVAL_MINUTE_TO_SECOND	SQL-C-INTERVAL-MINUTE-TO-SECON
SQL_CA_CONSTRAINT_INITIALLY_DEFERRED	SQL-CA-CONSTRAINT-INIT-DEFER
SQL_CA_CONSTRAINT_INITIALLY_IMMEDIATE	SQL-CA-CONSTRAINT-INIT-IMMED
SQL_CA_CONSTRAINT_NON_DEFERRABLE	SQL-CA-CONSTRAINT-NON-DEFERRAB
SQL_CA1_BULK_DELETE_BY_BOOKMARK	SQL-CA1-BULK-DELETE-BY-BOOKMAR
SQL_CA1_BULK_UPDATE_BY_BOOKMARK	SQL-CA1-BULK-UPDATE-BY-BOOKMAR
SQL_CDO_CONSTRAINT_NAME_DEFINITION	SQL-CDO-CONSTRAINT-NAME-DEFINI
SQL_CDO_CONSTRAINT_INITIALLY_DEFERRED	SQL-CDO-CONSTRAINT-INITIALLY-D
SQL_CDO_CONSTRAINT_INITIALLY_IMMEDIATE	SQL-CDO-CONSTRAINT-INITIALLY-I
SQL_CDO_CONSTRAINT_NON_DEFERRABLE	SQL-CDO-CONSTRAINT-NON-DEFERRA
SQL_CONVERT_INTERVAL_YEAR_MONTH	SQL-CONVERT-INTERVAL-YEAR-MONT
SQL_CT_CONSTRAINT_INITIALLY_DEFERRED	SQL-CT-CONSTRAINT-INITIALLY-DE

Figure 93 (Page 2 of 2). ODBC Names Truncated or Abbreviated for COBOL

ODBC C #define symbol > 30 characters long	Corresponding COBOL name
SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE	SQL-CT-CONSTRAINT-INITIALLY-IM
SQL_CT_CONSTRAINT_NON_DEFERRABLE	SQL-CT-CONSTRAINT-NON-DEFERRAB
SQL_CT_CONSTRAINT_NAME_DEFINITION	SQL-CT-CONSTRAINT-NAME-DEFINIT
SQL_DESC_DATETIME_INTERVAL_CODE	SQL-DESC-DATETIME-INTERVAL-COD
SQL_DESC_DATETIME_INTERVAL_PRECISION	SQL-DESC-DATETIME-INTERVAL-PRE
SQL_DL_SQL92_INTERVAL_DAY_TO_HOUR	SQL-DL-SQL92-INTERVAL-DAY-TO-H
SQL_DL_SQL92_INTERVAL_DAY_TO_MINUTE	SQL-DL-SQL92-INTERVAL-DAY-TO-M
SQL_DL_SQL92_INTERVAL_DAY_TO_SECOND	SQL-DL-SQL92-INTERVAL-DAY-TO-S
SQL_DL_SQL92_INTERVAL_HOUR_TO_MINUTE	SQL-DL-SQL92-INTERVAL-HR-TO-M
SQL_DL_SQL92_INTERVAL_HOUR_TO_SECOND	SQL-DL-SQL92-INTERVAL-HR-TO-S
SQL_DL_SQL92_INTERVAL_MINUTE_TO_SECOND	SQL-DL-SQL92-INTERVAL-MIN-TO-S
SQL_DL_SQL92_INTERVAL_YEAR_TO_MONTH	SQL-DL-SQL92-INTERVAL-YR-TO-MO
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1	SQL-FORWARD-ONLY-CURSOR-ATTR1
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2	SQL-FORWARD-ONLY-CURSOR-ATTR2
SQL_GB_GROUP_BY_CONTAINS_SELECT	SQL-GB-GROUP-BY-CONTAINS-SELEC
SQL_ISV_CONSTRAINT_COLUMN_USAGE	SQL-ISV-CONSTRAINT-COLUMN-USAG
SQL_ISV_REFERENTIAL_CONSTRAINTS	SQL-ISV-REFERENTIAL-CONSTRAINT
SQL_MAXIMUM_CATALOG_NAME_LENGTH	SQL-MAXIMUM-CATALOG-NAME-LENGT
SQL_MAXIMUM_COLUMN_IN_GROUP_BY	SQL-MAXIMUM-COLUMN-IN-GROUP-B
SQL_MAXIMUM_COLUMN_IN_ORDER_BY	SQL-MAXIMUM-COLUMN-IN-ORDER-B
SQL_MAXIMUM_CONCURRENT_ACTIVITIES	SQL-MAXIMUM-CONCURRENT-ACTIVIT
SQL_MAXIMUM_CONCURRENT_STATEMENTS	SQL-MAXIMUM-CONCURRENT-STAT
SQL_SQL92_FOREIGN_KEY_DELETE_RULE	SQL-SQL92-FOREIGN-KEY-DELETE-R
SQL_SQL92_FOREIGN_KEY_UPDATE_RULE	SQL-SQL92-FOREIGN-KEY-UPDATE-R
SQL_SQL92_NUMERIC_VALUE_FUNCTIONS	SQL-SQL92-NUMERIC-VALUE-FUNCTI
SQL_SQL92_RELATIONAL_JOIN_OPERATORS	SQL-SQL92-RELATIONAL-JOIN-OPER
SQL_SQL92_ROW_VALUE_CONSTRUCTOR	SQL-SQL92-ROW-VALUE-CONSTRUCTO
SQL_TRANSACTION_ISOLATION_OPTION	SQL-TRANSACTION-ISOLATION-OPTI

A COPY statement to include this copybook should be specified in the DATA DIVISION as follows:

- For a program, the COPY statement should be specified in the WORKING-STORAGE SECTION, in the outer-most program if programs are nested.
- For a method, the COPY statement should be specified in the WORKING-STORAGE SECTION of the method (not the WORKING-STORAGE SECTION of the CLASS definition). This must be done for each method that makes ODBC calls.

Mapping of ODBC C Types

The data types specified in ODBC APIs are defined in terms of ODBC C types in the API definitions. The following table shows corresponding COBOL declarations for the indicated ODBC C types of the arguments.

ODBC APIs from COBOL

Figure 94. Mapping of ODBC C Type to COBOL Data Declarations

ODBC C type	COBOL form	Description
SQLSMALLINT	COMP-5 PIC S9(4)	Signed short integer (2 byte binary)
SQLUSMALLINT	COMP-5 PIC 9(4)	Unsigned short integer (2 byte binary)
SQLINTEGER	COMP-5 PIC S9(9)	Signed long integer (4 byte binary)
SQLUINTEGER	COMP-5 PIC 9(9)	Unsigned long integer (4 byte binary)
SQLREAL	COMP-1	Floating point (4 bytes)
SQLFLOAT	COMP-2	Floating point (8 bytes)
SQLDOUBLE	COMP-2	Floating point (8 bytes)
SQLCHAR *	POINTER (See Note)	Pointer to unsigned character.
SQLHDBC	POINTER	Connection handle
SQLHENV	POINTER	Environment handle
SQLHSTMT	POINTER	Statement handle
SQLHWND	POINTER	Window handle

Note: This is a pointer to a null-terminated string. The target (of the pointer) item can be defined with PIC X(*n*), where *n* is large enough to represent the null terminated field. See "Manipulating Null-Terminated Strings" on page 79 for additional considerations on handling null terminated strings in COBOL.

Passing a Pointer as an Argument

If an argument is specified as a pointer to one of the above data types, then you need to do one of the following:

- Pass the target item of the pointer BY REFERENCE, *or*
- Define a pointer data item that will point to the target item and pass that BY VALUE, *or*
- Pass the ADDRESS OF the target item BY VALUE.

To illustrate, assume the function is defined as

```
RETCODE SQLSomeFunction(PSomeArgument)
```

where `PSomeArgument` is defined as an argument pointing to `SomeArgument`.

The argument may be passed to `SQLSomeFunction` in one of the following ways:

1. Pass `SomeArgument` BY REFERENCE:

   ```
   CALL "SQLSomeFunction" USING BY REFERENCE SomeArgument
   ```

   `USING BY CONTENT SomeArgument`, may be used instead if `SomeArgument` is an input argument.

2. Define a pointer data item `PSomeArgument` to point to `SomeArgument`:

   ```
   SET PSomeArgument TO ADDRESSS OF SomeArgument
   CALL "SQLSomeFunction" USING BY VALUE PSomeArgument
   ```

3. Pass ADDRESS OF `SomeArgument` BY VALUE:

   ```
   CALL "SQLSomeFunction" USING BY VALUE ADDRESS OF SomeArgument
   ```

Note that the last approach can be used only if the target argument, `SomeArgument`, is a level 01 item in the LINKAGE SECTION. If `SomeArgument` is a level 01 item in the LINKAGE SECTION, the addressibility to `SomeArgument` can be set in one of the following ways:

1. Explicitly via

   ```
   SET ADDRESS OF SomeArgument TO a-pointer-data-item
   ```

   or

   ```
   SET ADDRESS OF SomeArgument to ADDRESS OF an-identifier
   ```

   or

2. Implicitly by having `SomeArgument` passed in as an argument to the program from which the ODBC function call is being made.

The following shows a fragment of a sample program invoking the `SQLAllocHandle` function:

```
  ...
WORKING-STORAGE SECTION.
      COPY ODBC3.
   ...
01  SQL-RC     COMP-5    PIC S9(4).
01  Henv       POINTER.
   ...
PROCEDURE DIVISION.
   ...
      CALL "SQLAllocHandle"
              USING
                By VALUE     sql-handle-env
                             sql-null-handle
                By REFERENCE Henv
              RETURNING      SQL-RC
      IF SQL-RC NOT = (SQL-SUCCESS or SQL-SUCCESS-WITH-INFO)
         THEN
           DISPLAY "SQLAllocHandle failed."
            ...
         ELSE
            ...
```

The above is further illustrated using `SQLConnect` function. Any one of the following examples can be used for calling the `SQLConnect` function:

ODBC APIs from COBOL

Example 1:

```
  ...
CALL "SQLConnect" USING BY VALUE     ConnectionHandle
                        BY REFERENCE ServerName
                        BY VALUE     SQL-NTS
                        BY REFERENCE UserIdentifier
                        BY VALUE     SQL-NTS
                        BY REFERENCE AuthentificationString
                        BY VALUE     SQL-NTS
                  RETURNING          SQL-RC
       ...
```

Example 2:

```
  ...
SET Ptr-to-ServerName             TO ADDRESS OF ServerName
SET Ptr-to-UserIdentifier         TO ADDRESS OF UserIdentifier
SET Ptr-to-AuthentificationString TO ADDRESS OF AuthentificationString
CALL "SQLConnect" USING BY VALUE     ConnectionHandle
                                     Ptr-to-ServerName
                                     SQL-NTS
                                     Ptr-to-UserIdentifier
                                     SQL-NTS
                                     Ptr-to-AuthentificationString
                                     SQL-NTS
                  RETURNING          SQL-RC
       ...
```

Example 3:

```
  ...
CALL "SQLConnect" USING BY VALUE     ConnectionHandle
                                     ADDRESS OF ServerName
                                     SQL-NTS
                                     ADDRESS OF UserIdentifier
                                     SQL-NTS
                                     ADDRESS OF AuthentificationString
                                     SQL-NTS
                  RETURNING          SQL-RC
       ...
```

In **Example 3**, `Servername`, `UserIdentifier`, and `AuthentificationString` must be defined as level 01 items in the LINKAGE SECTION.

Note that the BY REFERENCE or BY VALUE phrase applies to all arguments until overridden by another BY REFERENCE, BY VALUE, or BY CONTENT phrase.

Accessing Function Return Values

The function return values for an ODBC call should be specified using the RETURNING phrase on the CALL statement:

```
CALL "SQLAllocEnv" USING BY VALUE Phenv RETURNING SQL-RC
IF SQL-RC NOT = SQL-SUCCESS
   THEN
      DISPLAY "SQLAllocEnv failed."
        ...
   ELSE
        ...
 END-IF
        ...
```

Testing Bits with a Bit Mask

Some of the ODBC APIs require you to set bit masks and to query bits. A callable library routine, `iwzODBCTestBits`, is supplied for your use for querying bits[15].

This routine may be called as follows:

```
   CALL "iwzODBCTestBits" USING identifier-1, identifier-2
                     RETURNING identifier-3
```

identifier-1
This is the field being tested. It must be a 2-or 4-byte binary number field, that is, USAGE COMP-5 PIC 9(4) or PIC 9(9).

identifier-2
This is the bit mask field to select the bits to be tested. It must be defined with the same USAGE and PICTURE as *identifier-1*.

identifier-3
This is the return value for the test and has the following return values:

0 None of the bits indicated by *identifier-2* is ON in *identifier-1*.

1 All the bits selected by *identifier-2* are ON in *identifier-1*.

-1 One or more bits are ON and one or more bits are OFF among the bits selected by *identifier-2* for *identifier-1*.

-100 Invalid input argument found (such as an 8-byte binary number field is used as *identifier-1*).

It must be defined with USAGE COMP-5 with PIC S9(4).

Multiple bits can be set in a field using COBOL arithmetic expressions with the bit masks defined in the ODBCCOB copybook. For example, the bits for SQL-CVT-CHAR, SQL-CVT-NUMERIC, and SQL-CVT-DECIMAL can be set in the `InfoValue` field by:

```
COMPUTE InfoValue = SQL-CVT-CHAR + SQL-CVT-NUMERIC + SQL-CVT-DECIMAL
```

[15] The IWZODBC.LIB import library must be linked with any application that calls `iwzODBCTestBits`.

Sample Program

After setting InfoValue, it can be passed to the iwzTestBits function as the second argument.

Note that the operands of the arithmetic expression above represent disjoint bits from each other as defined for each of such bit mask symbols in ODBCCOB copybook. You should be careful not to repeat the same bit in an arithmetic expression for this purpose (since the operations are arithmetic additions not logical ORs). For example,

```
COMPUTE InfoValue = SQL-CVT-CHAR + SQL-CVT-NUMERIC + SQL-CVT-DECIMAL
                    + SQL-CVT-CHAR
```

will result in InfoValue not having the SQL-CVT-CHAR bit on.

The call interface convention in effect at the time of the call must be CALLINT SYSTEM DESCRIPTOR.

Null-Terminated Character Strings

Some ODBC APIs require you to pass null-terminated character strings as arguments. For information on how to construct and manipulate null-terminated strings in COBOL, see "Manipulating Null-Terminated Strings" on page 79.

Setting Licensing Information for ODBC Driver Manager/Driver

If you are using the ODBC Driver Manager/drivers shipped with IBM VisualAge COBOL, you need to call iwzODBCLicInfo immediately following a call to the SQLConnect, SQLDriverConnect, or SQLBrowseConnect function[16]. You need to pass the argument hdbc to iwzODBCLicInfo:

```
CALL "iwzODBCLicInfo" USING BY VALUE Hdbc
```

See "Sample Program Using Supplied Copybooks" for the call.

Sample Program Using Supplied Copybooks

Two other copybooks are supplied for your optional use, if you want to use prepared COBOL statements for commonly used functions for ODBC initialization, error handling, and clean-up (SQLAllocEnv, SQLAllocConnect, iwzODBCLicInfo, SQLAllocStmt, SQLFreeStmt, SQLDisconnect, SQLFreeConnect, and SQLFreeEnv). These copybooks may be used with or without modifications.

In addition to the ODBC3.CPY copybook, the two other copybooks are ODBC3D.CPY and ODBC3P.CPY. Also a skeleton sample program, ODBC3EG.CBL, illustrating the use of these copybooks is included.

ODBC3P.CPY includes COBOL procedure statements that can be performed for initialization, termination, and error processing.

[16] The IWZODBC.LIB import library must be linked with any application that calls iwzODBCLicInfo.

ODBC3D.CPY contains data declarations used by ODBC3P.CPY in the WORKING-STORAGE SECTION (or LOCAL-STORAGE SECTION).

Example Notes:

1. The PGMNAME(MIXED) compiler option should be used; the ODBC entry points are case sensitive.
2. System/390 host data type options should not be used; ODBC APIs expect their parameters to be in native format.
3. The sample code (ODBC3EG.CBL, ODBC3P.CPY, and ODBC3D.CPY) is located in the SAMPLES\ODBC subdirectory under the main COBOL install directory.
4. ODBC3.CPY is in the INCLUDE subdirectory under the main COBOL install directory.
5. Including these two subdirectories in your SYSLIB environment variable will ensure that the copybooks are available to the compiler.

The following illustrates the use of the copybooks:

```
      cbl  pgmname(mixed)
      ******************************************************************
      * ODBC3EG.CBL                                                    *
      *----------------------------------------------------------------*
      * Sample program using ODBC3, ODBC3D and ODBC3P COPY books       *
      ******************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. "ODBC3EG".
       DATA DIVISION.

       WORKING-STORAGE SECTION.
      *  copy ODBC API constant definitions
           COPY "odbc3.cpy" SUPPRESS.

      *  copy additional definitions used by ODBC3P procedures
           COPY "odbc3d.cpy".

      *  arguments used for SQLConnect
       01  ServerName                    PIC X(10) VALUE Z"Oracle7".
       01  ServerNameLength       COMP-5 PIC S9(4) VALUE 10.
       01  UserId                        PIC X(10) VALUE Z"TEST123".
       01  UserIdLength           COMP-5 PIC S9(4) VALUE 10.
       01  Authentification              PIC X(10) VALUE Z"TEST123".
       01  AuthentificationLength COMP-5 PIC S9(4) VALUE 10.
```

Figure 95 (Part 1 of 2). Example of Using the ODBC Copybooks

```
       PROCEDURE DIVISION.
       Do-ODBC SECTION.
        Start-ODBC.
           DISPLAY "Sample ODBC 3.0 program starts"

      *  allocate henv & hdbc
           PERFORM ODBC-Initialization

      *  connect to data source
           CALL "SQLConnect" USING BY VALUE     Hdbc
                                   BY REFERENCE ServerName
                                   BY VALUE     ServerNameLength
                                   BY REFERENCE UserId
                                   BY VALUE     UserIdLength
                                   BY REFERENCE Authentification
                                   BY VALUE     AuthentificationLength
                             RETURNING SQL-RC

           IF SQL-RC NOT = SQL-SUCCESS
             MOVE "SQLConnect" to SQL-stmt
             MOVE SQL-HANDLE-DBC to DiagHandleType
             SET DiagHandle to Hdbc
             PERFORM SQLDiag-Function
           END-IF

      *  set licensing informationm
           PERFORM SQL-SetLicInfo-Function

      *  allocate hstmt
           PERFORM Allocate-Statement-Handle

      *****************************************
      *  add application specific logic here  *
      *****************************************

      *  clean-up environment
           PERFORM ODBC-Clean-Up.

      *  End of sample program execution
           DISPLAY "Sample COBOL ODBC program ended"
           GOBACK.

      *  copy predefined COBOL ODBC calls which are performed
           COPY "odbc3p.cpy".
      *******************************************************
      * End of ODBC3EG.CBL:  Sample program for ODBC 3.0 *
      *******************************************************
```

Figure 95 (Part 2 of 2). Example of Using the ODBC Copybooks

The following shows the copybook `ODBC3D.CPY`:

```
*******************************************************************
* ODBC3D.CPY                 (ODBC Ver 3.0)                       *
*-----------------------------------------------------------------*
* Data definitions to be used with sample ODBC function calls     *
*  and included in WORKING-STORAGE or LOCAL-STORAGE SECTION       *
*******************************************************************
* ODBC Handles
 01  Henv                       POINTER            VALUE NULL.
 01  Hdbc                       POINTER            VALUE NULL.
 01  Hstmt                      POINTER            VALUE NULL.

* Arguments used for GetDiagRec calls
 01  DiagHandleType             COMP-5  PIC 9(4).
 01  DiagHandle                 POINTER.
 01  DiagRecNumber              COMP-5  PIC 9(4).
 01  DiagRecNumber-Index        COMP-5  PIC 9(4).
 01  DiagSQLState.
     02 DiagSQLState-Chars              PIC X(5).
     02 DiagSQLState-Null               PIC X.
 01  DiagNativeError            COMP-5  PIC S9(9).
 01  DiagMessageText                    PIC X(511) VALUE SPACES.
 01  DiagMessageBufferLength    COMP-5  PIC S9(4)  VALUE 511.
 01  DiagMessageTextLength      COMP-5  PIC S9(4).

* Misc declarations used in sample function calls
 01  SQL-RC                     COMP-5  PIC S9(4)  VALUE 0.
 01  Saved-SQL-RC               COMP-5  PIC S9(4)  VALUE 0.
 01  SQL-stmt                           PIC X(30).

*************************
* End of ODBC3D.CPY     *
*************************
```

Figure 96. Supplied Copybook ODBC3D.CPY

The following shows the copybook `ODBC3P.CPY`:

```
******************************************************************
* ODBC3P.CPY                                                     *
*----------------------------------------------------------------*
*  Sample ODBC initialization, clean-up and error handling       *
*    procedures    (ODBC Ver 3.0)                                *
******************************************************************
*** Initialization functions SECTION ****************************
 ODBC-Initialization SECTION.
*
  Allocate-Environment-Handle.
     CALL "SQLAllocHandle" USING
                             BY VALUE     SQL-HANDLE-ENV
                             BY VALUE     SQL-NULL-HANDLE
                             BY REFERENCE Henv
                           RETURNING      SQL-RC

     IF SQL-RC NOT = SQL-SUCCESS
       MOVE "SQLAllocHandle for Env" TO SQL-stmt
       MOVE SQL-HANDLE-ENV to DiagHandleType
       SET DiagHandle to Henv
       PERFORM SQLDiag-Function
     END-IF.
*
  Set-Env-Attr-to-Ver30-Behavior.
     CALL "SQLSetEnvAttr" USING
                            BY VALUE      Henv
                            BY VALUE      SQL-ATTR-ODBC-VERSION
                            BY VALUE      SQL-OV-ODBC3
*                                      or SQL-OV-ODBC2         *
*                                         for Ver 2.x behavior *
                            BY VALUE      SQL-IS-UINTEGER
                         RETURNING        SQL-RC

     IF SQL-RC NOT = SQL-SUCCESS
       MOVE "SQLSetEnvAttr" TO SQL-stmt
       MOVE SQL-HANDLE-ENV to DiagHandleType
       SET DiagHandle to Henv
       PERFORM SQLDiag-Function
     END-IF.
*
  Allocate-Connection-Handle.
     CALL "SQLAllocHandle" USING
                             By VALUE     SQL-HANDLE-DBC
                             BY VALUE     Henv
                             BY REFERENCE Hdbc
                           RETURNING      SQL-RC
```

Figure 97 (Part 1 of 5). Supplied Copybook ODBC3P.CPY:

```
     IF SQL-RC NOT = SQL-SUCCESS
         MOVE "SQLAllocHandle for Connection" to SQL-stmt
         MOVE SQL-HANDLE-ENV to DiagHandleType
         SET DiagHandle to Henv
         PERFORM SQLDiag-Function
     END-IF.

*** SQL-SetLicInfo SECTION ***********************************
 SQL-SetLicInfo-Function SECTION.
  SQL-SetLicInfo.
     CALL "iwzODBCLicInfo" USING BY VALUE Hdbc.

*** SQLAllocHandle for statement function SECTION ***************
 Allocate-Statement-Handle SECTION.
  Allocate-Stmt-Handle.
     CALL "SQLAllocHandle" USING
                             By VALUE     SQL-HANDLE-STMT
                             BY VALUE     Hdbc
                             BY REFERENCE Hstmt
                           RETURNING      SQL-RC

     IF SQL-RC NOT = SQL-SUCCESS
         MOVE "SQLAllocHandle for Stmt" TO SQL-stmt
         MOVE SQL-HANDLE-DBC to DiagHandleType
         SET DiagHandle to Hdbc
         PERFORM SQLDiag-Function
     END-IF.

*** Cleanup Functions SECTION ***********************************
 ODBC-Clean-Up SECTION.
*
  Free-Statement-Handle.
     CALL "SQLFreeHandle" USING
                            BY VALUE SQL-HANDLE-STMT
                            BY VALUE Hstmt
                          RETURNING  SQL-RC

     IF SQL-RC NOT = SQL-SUCCESS
         MOVE "SQLFreeHandle for Stmt" TO SQL-stmt
         MOVE SQL-HANDLE-STMT to DiagHandleType
         SET DiagHandle to Hstmt
         PERFORM SQLDiag-Function
     END-IF.
*
```

Figure 97 (Part 2 of 5). Supplied Copybook ODBC3P.CPY:

```
  SQLDisconnect-Function.
     CALL "SQLDisconnect" USING
                            BY VALUE Hdbc
                          RETURNING  SQL-RC

     IF SQL-RC NOT = SQL-SUCCESS
         MOVE "SQLDisconnect" TO SQL-stmt
         MOVE SQL-HANDLE-DBC to DiagHandleType
         SET DiagHandle to Hdbc
         PERFORM SQLDiag-Function
     END-IF.
*
  Free-Connection-Handle.
     CALL "SQLFreeHandle" USING
                            BY VALUE SQL-HANDLE-DBC
                            BY VALUE Hdbc
                          RETURNING  SQL-RC
     IF SQL-RC NOT = SQL-SUCCESS
         MOVE "SQLFreeHandle for DBC" TO SQL-stmt
         MOVE SQL-HANDLE-DBC to DiagHandleType
         SET DiagHandle to Hdbc
         PERFORM SQLDiag-Function
     END-IF.
*
  Free-Environment-Handle.
     CALL "SQLFreeHandle" USING
                            BY VALUE SQL-HANDLE-ENV
                            BY VALUE Henv
                          RETURNING  SQL-RC

     IF SQL-RC NOT = SQL-SUCCESS
         MOVE "SQLFreeHandle for Env" TO SQL-stmt
         MOVE SQL-HANDLE-ENV to DiagHandleType
         SET DiagHandle to Henv
         PERFORM SQLDiag-Function
     END-IF.

*** SQLDiag function SECTION ************************************
 SQLDiag-Function SECTION.
  SQLDiag.
     MOVE SQL-RC TO SAVED-SQL-RC
     DISPLAY "Return Value = " SQL-RC

       IF SQL-RC = SQL-SUCCESS-WITH-INFO
         THEN
            DISPLAY SQL-stmt " successful with information"
         ELSE
            DISPLAY SQL-stmt " failed"
       END-IF
```

Figure 97 (Part 3 of 5). Supplied Copybook ODBC3P.CPY:

```
*    - get number of diagnostic records - *
     CALL "SQLGetDiagField"
             USING
               BY VALUE     DiagHandleType
                            DiagHandle
                            0
                            SQL-DIAG-NUMBER
               BY REFERENCE DiagRecNumber
               BY VALUE     SQL-IS-SMALLINT
               BY REFERENCE OMITTED
             RETURNING      SQL-RC

     IF SQL-RC = SQL-SUCCESS or SQL-SUCCESS-WITH-INFO
       THEN

*        - get each diagnostic record - *
         PERFORM WITH TEST AFTER
           VARYING DiagRecNumber-Index FROM 1 BY 1
             UNTIL DiagRecNumber-Index > DiagRecNumber
              or   SQL-RC NOT =
                     (SQL-SUCCESS or SQL-SUCCESS-WITH-INFO)

*          - get a diagnostic record - *
           CALL "SQLGetDiagRec"
                   USING
                     BY VALUE     DiagHandleType
                                  DiagHandle
                                  DiagRecNumber-Index
                     BY REFERENCE DiagSQLState
                                  DiagNativeError
                                  DiagMessageText
                     BY VALUE     DiagMessageBufferLength
                     BY REFERENCE DiagMessageTextLength
                   RETURNING          SQL-RC
```

Figure 97 (Part 4 of 5). Supplied Copybook ODBC3P.CPY:

```
           IF SQL-RC = SQL-SUCCESS OR SQL-SUCCESS-WITH-INFO
             THEN
               DISPLAY "Information from diagnostic record number"
                       " " DiagRecNumber-Index " for "
                       SQL-stmt ":"
               DISPLAY "  SQL-State = " DiagSQLState-Chars
               DISPLAY "  Native error code = " DiagNativeError
               DISPLAY "  Diagnostic message = "
                        DiagMessageText (1:DiagMessageTextLength)
             ELSE
               DISPLAY "SQLGetDiagRec request for " SQL-stmt
                       " failed with return code of: " SQL-RC
                       " from SQLError"
               PERFORM Termination
           END-IF
         END-PERFORM

       ELSE
*        - indicate SQLGetDiagField failed - *
         DISPLAY "SQLGetDiagField failed with return code of: "
                 SQL-RC
     END-IF

     MOVE Saved-SQL-RC to SQL-RC
     IF Saved-SQL-RC NOT = SQL-SUCCESS-WITH-INFO
       PERFORM Termination
     END-IF.

*** Termination Section******************************************
 Termination Section.
  Termination-Function.
     DISPLAY "Application being terminated with rollback"
     CALL "SQLTransact" USING BY VALUE henv
                                       hdbc
                                       SQL-ROLLBACK
                        RETURNING      SQL-RC

     IF SQL-RC = SQL-SUCCESS
       THEN
         DISPLAY "Rollback successful"
       ELSE
         DISPLAY "Rollback failed with return code of: "
                 SQL-RC
     END-IF
     STOP RUN.

*************************
* End of ODBC3P.CPY     *
*************************
```

Figure 97 (Part 5 of 5). Supplied Copybook ODBC3P.CPY:

Chapter 24. Building Dynamic Link Libraries

By using linking you can have a program call another program which is not contained in the calling program's source code. Before or during execution, the calling program's object module is linked with the called program's object module.

Before you begin creating COBOL dynamic link libraries, you should understand the differences between static and dynamic linking.

Static Linking Overview

Static linking involves a calling program being linked to a called program module, resulting in a single executable module. The result of statically linking programs is an .EXE file or DLL subprogram that contains the executable code for multiple programs. This may include both the calling program and the called program. When the program is loaded, the operating system attempts to place a single file containing the executable code and data into memory.

The primary advantage of static linking is that you can create self-contained, independent programs. In other words, the executable program consists of one part (the .EXE file) that you need to keep track of. The disadvantages of static linking are as follows:

- Linked external programs are built into the executable files, making these files larger.
- The behavior of executable files cannot be changed without re-linking.
- External called programs cannot be shared, requiring duplicate copies of programs to be loaded in memory if more than one calling program needs to access them.

To overcome these disadvantages, use dynamic linking.

Dynamic Linking Overview

Dynamic linking allows several programs to use a single copy of an executable module. The executable module is completely separate from the programs that use it. Several subprograms can be built into a dynamic link library (DLL), and calling programs can use these subprograms as if they were part of the calling program's own executable code. You can change the dynamically-linked subprograms without recompiling or relinking the calling program.

DLLs are typically used to provide common functions that can be used by a number of programs. For example, DLLs can be used to implement subprogram packages, subsystems, and interfaces to other programs. DLLs are also used to create object-oriented class libraries (see Chapter 14, “Writing Object-Oriented Programs” on page 270).

DLL Reference Resolution

You can dynamically link with the supplied run-time DLLs, as well as with your own COBOL DLLs.

OS/2 For an in-depth discussion of static versus dynamic linking, refer to the *OS/2 Application Design Guide*. OS/2

Terminology Notes

If you are new to the PC, you might find the terminology used to discuss DLLs confusing.

Keep in mind that a dynamic link library (DLL) is, above all, a *library* of functions. Even if there is only one function provided by the DLL (as in the example provided in this chapter), the purpose of a DLL is to serve as a repository of frequently-used functions.

In COBOL terms, a DLL is a collection of outermost programs. While these outermost programs may contain nested programs, only the outermost programs (known as entry points) are callable by programs external to the DLL. Just as you can compile and link several COBOL programs together as a single executable (.EXE) you link one or more compiled outermost COBOL programs together to create a DLL.

Because outermost programs in the DLL are part of a library of programs, each program is referred to as a *subprogram* in the DLL. Even if a DLL provides only one program, that program is considered a subprogram of the DLL.

How the Linker Resolves References to DLLs

When you compile a program, the compiler generates an object module for the code in the program. If you use any subprograms (“functions” as described in C, “subroutines” in other languages) that are in an external object module, the compiler adds an external program reference to your program object module.

The linker resolves these external references. If it finds a reference to external subprograms in an import library (see “Creating an Import Library” on page 444) or in a DLL's module-definition file (see “Module Definition Files” on page 442), the code for the external subprogram is in a DLL. To resolve external reference to DLLs, the linker adds information to the executable file that tells the loader where to find the DLL code when the executable file is loaded.

The DLLs that you reference can be created to load when the executable that calls them is loaded (pre-load) or to load when they are first referenced (load on call). However, not all references to DLLs by COBOL CALLs are resolved by the linker: CALL *identifier* and CALL *literal* with the DYNAM option are resolved by COBOL when the CALL is executed.

Creating a DLL

A DLL is built using compiled source code and a module-definition (.DEF) file (OS/2) or export (.EXP) file (Windows).

OS/2

Every DLL must have an accompanying .DEF file. The .DEF file contains information that includes:

- The name of the DLL (limited to eight characters)
- When to load the DLL (pre-load or load on call)
- How to manage memory for the DLL
- When to initialize the DLL
- The names of subprograms included in the DLL

OS/2

In every case, the first step is to construct your DLL. You write the source code for DLL subprograms the way you write any other COBOL source programs.

Example of a DLL Source File

The following COBOL source code is a simple example of a DLL source subprogram. When compiled, a DLL can contain numerous outermost programs, each of which is considered a subprogram within the DLL.

In the case of the following example, the DLL contains only a single subprogram. When the subprogram named MYDLL contained in the DLL named MYDLL.DLL, is called by another program and executed, it will DISPLAY the text MYDLL Entered on the computer screen.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MYDLL.
PROCEDURE DIVISION.
    DISPLAY "MYDLL Entered".
    EXIT PROGRAM.
```

Figure 98. MYDLL.CBL— Example of a Simple COBOL DLL Source Program

The source code for the subprograms contained in the DLL is compiled and linked using cob2 (for more details on how to compile and link DLLs, see Chapter 9, “Compiling, Linking, and Running Programs” on page 134). However, on OS/2, before you are ready to build your DLL, you need to create its module definition file. This is discussed in the following section.

Module Definition Files

You must use a module definition file when you create any DLL. A module definition (.DEF) file is a text file that describes the names, attributes, exports, imports, and other characteristics of a program or dynamic link library.

When you create a DLL, its .DEF file must contain a list of all the subprograms in the DLL that can be called by a program (or by another DLL). You specify these external subprograms by using an EXPORTS statement in the file.

Example of a Module Definition File

The .DEF file for the compiled MYDLL.CBL source program (see Figure 98 on page 441) is shown in the following figure. It illustrates the statements most frequently used in module definition files that build DLLs.

```
;***********************************************
;* MYDLL.DEF                                   *
;* Description: Provides the module definition *
;* for MYDLL.DLL, a simple COBOL DLL           *
;***********************************************
LIBRARY MYDLL INITINSTANCE TERMINSTANCE
PROTMODE
DATA MULTIPLE READWRITE LOADONCALL NONSHARED
CODE LOADONCALL EXECUTEREAD
EXPORTS
   MYDLL
```

Figure 99. MYDLL.DEF— DLL Module Definition File for MYDLL.DLL

For a complete description of module definition files and their statement syntax, see Chapter 25, “Creating Module Definition Files” on page 448.

Export Files (Windows Only)

Windows ▶

You must use an export file when you create any DLL. An export (.EXP) file is a binary file that tells the linker what parts are exported by the DLL. It is built by the cob2 command when you compile and link the DLL.

◀ Windows

Coding for CALL identifier

Your COBOL program can make a CALL to a user-defined identifier rather than to a literal DLL subprogram name when the name of the target subprogram is not known until execution time. For more information about the CALL statement, see the *IBM COBOL Language Reference*.

For example, the following COBOL source program will call `MYDLL` in `MYDLL.DLL` (see Figure 98 on page 441):

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. RTDLLRES.
      *
      *    THIS PROGRAM USES CALL identifier to call a subprogram
      *          NAMED MYDLL in a DLL. IT REQUIRES A DLL
      *          NAMED MYDLL.DLL.
      *
      *
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77  CALLNAM PIC IS X(8).
       PROCEDURE DIVISION.
           DISPLAY "Start sample program RTDLLRES".
           MOVE "MYDLL" TO CALLNAM.
           CALL CALLNAM.
           DISPLAY "RTDLLRES successful".
           STOP RUN.
```

Figure 100. DYNAM.CBL— Run-Time DLL Reference Resolution

In the preceding example, the statement:

```
MOVE "MYDLL" TO CALLNAM.
```

is a reference to the single subprogram `MYDLL` of the DLL `MYDLL.DLL`.

Note: This DLL must be in a directory defined by the COBPATH environment variable (see “Run-Time Environment Variables” on page 137).

Coding for CALL literal

Rather than using calls that are resolved at run time, you can use COBOL CALL *literal*. By default, these calls are resolved by the linker.

DYNAM Setting: Whether these calls are resolved by the linker depends on the setting of the DYNAM compile-time option:

- If the setting is DYNAM, the call is treated the same as CALL *identifier* and is resolved at run time.
- If the setting is NODYNAM, CALL *literal* is resolved by the linker. With call resolution by the linker, calls to subprograms in a DLL using CALL *literal*, will not cause the DLL's object code to be included in the executable module for your main program.

  With CALL *literal* and NODYNAM, you can also statically link.

CALL literal

To make such a call to an entry point in a DLL, use an import library. The import library tells the linker where to find the DLL subprograms used by your program.[17]

Unlike calls that result in run-time resolution, with link-time resolution you can have multiple entry points (outermost programs) in the DLL called.

Creating an Import Library

Windows ▶ On Windows the import library is created by the `cob2` command when you compile and link your DLL. ◀ Windows

OS/2 ▶

For call resolution to a DLL by the linker you need to explicitly create an import library. To create an import library, use `IMPLIB.EXE`, the import library manager included with VisualAge COBOL.

To create the import library for your DLL, first create its `.DEF` file (see "Module Definition Files" on page 442). Then use `IMPLIB` to create the import library. For the sample `MYDLL` previously discussed, we could create a library named `MYDLL.LIB` with the following syntax:

```
     implib mydll.lib mydll.def
```

For `IMPLIB` syntax, enter the command at an OS/2 command prompt with no arguments.

Once you have created the import library, convert the library to the new library format:

```
  ILIB /CONV /NOE /NOBR mydll.lib
```

This improves linking performance.

An import library does not contain any object modules. Instead, the import library contains information that tells the linker which DLLs are used by your program and which individual subprograms are used within each DLL. When you link an executable module, the linker uses this import library to resolve external references to DLLs.

You use import libraries every time you compile and link a program that uses the system APIs. For example, all the OS/2 functions are implemented in DLLs, and `OS2386.LIB` is an import library that tells the linker where to find each OS/2 function.

If you invoke the linker directly, give the name of the import library where you normally specify library names.

[17] Alternatively, you can use an IMPORTS statement in a `.DEF` file for your program.

OS/2 ▶ (Using `.DEF` files for stand-alone programs is not discussed here; see the *OS/2 Application Design Guide*).
◀ OS/2

OS/2

Sample Program Using Call Resolution by the Linker

Figure 100 on page 443 shows a COBOL main program that makes a call to an entry point in a DLL that is resolved at *run time.*

OS/2 If you have created an import library using IMPLIB, you can call the same DLL, but have the entry resolved by the linker. OS/2

The following program is identical to RTDLLRES.CBL, except that the CALL is made directly to a symbol exported in a .DEF file rather than to a user-defined word:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. LTDLLRES.
*
*    THIS PROGRAM CALLS A SUBPROGRAM CALLED MYDLL WHICH
*          RESOLVES AT LINK-TIME.  THE DLL IN WHICH "MYDLL"
*          IS FOUND IS NOT REQUIRED TO BE NAMED MYDLL.DLL
*
 PROCEDURE DIVISION.
     DISPLAY "Start sample program LTDLLRES".
     CALL "MYDLL"
     DISPLAY "LTDLLRES successful".
     STOP RUN.
```

Figure 101. LTDLLRES.CBL— Link-Time DLL Reference Resolution

In this example, the CALL reference to MYDLL is not a direct reference to MYDLL.DLL (although, in this case, the DLL happens to have the same name). The CALL is to the symbolic name MYDLL, which was exported in the MYDLL.DEF file (see Figure 99 on page 442). MYDLL, in turn, is the name of the only COBOL subprogram in MYDLL.DLL. When LTDLLRES is built using cob2, the CALL to MYDLL in MYDLL.DLL will be resolved by the linker.

Compiling and Linking Your DLL

Use cob2 to compile your source files and create a DLL (see Chapter 9, “Compiling, Linking, and Running Programs” on page 134).

When you use cob2 to compile and link your DLL, specify the names of all the DLL source files followed by the name of the module definition file (OS/2 only) on the command line. The name of the first source file is used as the name of the DLL unless you specify a filename with filetype .DLL, as in TEST.DLL on OS/2 or use the -dll option, as in -dll:TEST on Windows.

For example, use cob2 to compile and link the DLL and main executable programs described in this chapter. The DLL and programs that call it must be compiled in separate steps.

- The following cob2 command will build the DLL named MYDLL.DLL (see Figure 98 on page 441):

  OS/2 ▶

  ```
  cob2 mydll.cbl mydll.def
  ```

  ◀ OS/2

  Windows ▶

  ```
  cob2 mydll.cbl -dll:mydll
  ```

  ◀ Windows

- The following cob2 command will build the program RTDLLRES.EXE, which calls the DLL subprogram MYDLL with run-time resolution (see Figure 100 on page 443):

  ```
  cob2 rtdllres.cbl
  ```

- The following cob2 command will build the program LTDLLRES.EXE, which calls the DLL subprogram MYDLL with link-time resolution (see Figure 101 on page 445):

  ```
  cob2 ltdllres.cbl mydll.lib
  ```

Creating Object-Oriented DLLs

COBOL supports the use of class programs as DLLs (for information about creating class programs, see Chapter 14, “Writing Object-Oriented Programs” on page 270).

The procedure for creating class program DLLs is the same as that for creating procedural program DLLs, with one exception. Class program DLLs require a different set of entries in the EXPORTS section of the module definition file for the DLL

To build the definition file for a class:

1. Compile the COBOL class with the IDLGEN compiler option
2. Compile the IDL with the SOM compiler's def emitter

For example

```
cob2 -qidlgen ClassA.cbl
sc   -sdef ClassA.idl
```

See the *SOMobjects Developer's Toolkit User's Guide* (available online) for details. (For more information about module definition files, see “Module Definition Files” on page 442.)

Your class program is identified as a SOM subclass in the REPOSITORY paragraph of the ENVIRONMENT DIVISION. In the .DEF file for the DLL, export the name of the class program exactly as it is identified in the REPOSITORY paragraph, with 3 required strings appended to the end of the class name. These strings are:

1. *class-name*NewCLass
2. *class-name*ClassData
3. *class-name*CClassData

Mixed Case: The export specification is case-sensitive: specify the class name and required strings in correct mixed case.

For example, suppose you have two class programs, `CLASSA.CBL` and `CLASSB.CBL`, specified as `ClassA` and `ClassB` in their respective REPOSITORY paragraphs. If you want to compile and link-edit these two class programs as a single DLL, the EXPORTS section of the DLL's module definition file needs to specify the following:

```
EXPORTS
     ClassANewClass
     ClassAClassData
     ClassACClassData
     ClassBNewClass
     ClassBClassData
     ClassBCClassData
```

Note that a variation on each class name is exported verbatim, with three different strings appended onto the class name. In the case of `ClassA`, the resulting export statement includes:

*ClassA*NewCLass
*ClassA*ClassData
*ClassA*CClassData

This book does not address the details of creating SOM DLLs. If you would like a detailed explanation of why these three entries are required in the EXPORTS paragraph, see the *SOMobjects Developer's Toolkit User's Guide* (available online).

Chapter 25. Creating Module Definition Files

A module definition file contains one or more module statements. These statements:

- Define various attributes of your executable output file
- Define attributes of code and data segments in the file
- Identify data and functions that are imported into or exported from your file

If a module definition file is supplied, it is used. If not, the .DEF file is created by `cob2` as follows:

OS/2 ▶

The .DEF file is created by cob2 for OS/2 if the NOSEPOBJ compiler option is in effect.

◀ OS/2

Windows ▶

For Windows it is created if a .LIB file is supplied.

◀ Windows

In other cases (i.e. not the case where `cob2` creates a .DEF file), if linking a DLL, the .DEF file must be supplied to `cob2`. `cob2` will generate an .IMP file (OS/2, Windows) and an .EXP file (Windows).

You can use module definition files when:

- You are creating a DLL.
- You are linking with a DLL, and are not using an import library to do so. You can use the `IMPORTS` module statement to define imports, instead of linking to an import library to resolve references to a DLL.
- You need to define attributes of the executable output file more precisely than you can with options alone (for example, you want to define library initialization and termination behavior, with the `LIBRARY` statement).
- You need to define segment attributes more precisely than you can with options alone

When creating a module definition file, follow these rules:

- Use a NAME or LIBRARY statement to define the type of executable output you want. You can only use one of these statements, and it must precede all other statements in the module definition file.
- Begin comments with a semicolon (;). The linker ignores any line in the file that begins with a semicolon, and any portion of a line that follows a semicolon.

- Enter all module definition keywords (for example, NAME, LIBRARY, and IOPL) in uppercase letters.
- Do not use module definition keywords, or reserved words, as a text parameter to a statement (for example, you cannot use the LIBRARY statement to name a library `SHARED`, because SHARED is a keyword). See “Reserved Words” for a list of keywords and reserved words.

Reserved Words

The following words cannot be used as text parameters to a module statement. For example, you cannot use these words as the names of functions defined with the `EXPORTS` statement, or to name a stub file with the `STUB` statement.

The words are either module definition keywords, or reserved by the linker.

Note: Although module definition keywords should always be entered in uppercase letters, and only the uppercase forms are shown below, the mixed- and lower-case forms of these words are also reserved. For example, `CONTIGUOUS`, `ContiGuous`, and `contiguous` are all reserved.

ALIAS
BASE
CODE
CONFORMING
CONTIGUOUS
DATA
DESCRIPTION
DEV386
DISCARDABLE
DOS4
DYNAMIC
EXECUTEONLY
EXECUTEREAD
EXETYPE
EXPANDDOWN
EXPORTS
FIXED
HEAPSIZE
HUGE
IOPL
IMPORTS
IMPURE
INCLUDE
INITGLOBAL
INITINSTANCE
INVALID
LIBRARY
LOADONCALL
LONGNAMES
MAXVAL
MIXED1632
MOVABLE
MOVEABLE
MULTIPLES
NAME
NEWFILES
NODATA
NOEXPANDDOWN
NOIOPL
NONAME
NONCONFORMING
NONDISCARDABLE
NONE
NONPERMANENT
NONSHARED
NOTWINDOWCOMPAT
OBJECTS
OLD
ORDER
OS2
PERMANENT
PRELOAD
PRIVATE
PROTECT
PURE
READONLY
READWRITE
REALMODE
RESIDENT
RESIDENTNAME
ROBASE
SEGMENTS
SHARED
SINGLE
STACKSIZE
STUB
SWAPPABLE
SYSBASE
TERMGLOBAL
TERMINSTANCE
UNKNOWN
WINDOWAPI
WINDOWCOMPAT
WINDOWS

Summary of Module Statements

Figure 102. Linker Module Statements Summary. Default parameters are underlined. The defaults for NONE|SINGLE|MULTIPLE, SHARED|NONSHARED, INITGLOBAL|INITINSTANCE, and TERMGLOBAL|TERMINSTANCE are described in the detailed description of the option.

Statement	Description	Parameters
BASE	Set preferred loading address.	Loading address
CODE	Give default attributes for code segments.	CONFORMING|NONCONFORMING EXECUTEONLY|EXECUTEREAD IOPL|NOIOPL PRELOAD|LOADONCALL
DATA	Give default attributes for data segments. See detailed description for defaults of NONE|SINGLE|MULTIPLE, SHARED|NONSHARED.	IOPL|NOIOPL NONE|SINGLE|MULTIPLES PRELOAD|LOADONCALL READONLY|READWRITE SHARED|NONSHARED
DESCRIPTION	Describe the executable.	Descriptive text
EXETYPE	Identify operating system.	OS2|WINDOWS|UNKNOWN
EXPORTS	Define exported functions and data.	Entry name Internal name Ordinal position RESIDENTNAME|NONAME Parameter size
HEAPSIZE	Specify local heap size.	*bytes*|MAXVAL
IMPORTS	Define imported functions.	Internal name Name of exporting module Entry name or ordinal value
LIBRARY	Identify output as dynamic link library (DLL). See detailed description for defaults of parameters.	Library name INITGLOBAL|INITINSTANCE TERMGLOBAL|TERMINSTANCE
NAME	Identify output as executable (EXE).	Application name WINDOWAPI|WINDOWCOMPAT |NOTWINDOWCOMPAT
OLD	Preserve ordinal values from old DLL.	Name of old DLL
SEGMENTS	Give attributes for specific segments. See detailed description for default of SHARED|NONSHARED.	Segment name Class of the segment ALIAS CONFORMING|NONCOMFORMING EXECUTEONLY|EXECUTEREAD IOPL|NOIOPL MIXED1632 PRELOAD|LOADONCALL READONLY|READWRITE SHARED|NONSHARED
STACKSIZE	Specify local stack size.	Stack size (in bytes)
STUB	Add DOS executable file to module.	File name to add

Linker Module Statements

The following linker module statements can be used to create module definition files.

BASE

```
►►──BASE──=──address─────────────────────────────────────────►◄
```

Use the BASE statement to specify the preferred load address for the first load segment of the module. The number you give for the option is rounded up to the nearest multiple of 64K. The second load segment is then loaded at the next available multiple of 64K, and so on.

If the module's load segments cannot be loaded beginning at this preferred address, then the preferred address is ignored and the load segments are loaded according to the internal relocation records retained in the file data.

For .EXE files, accept the default base address of 64K (BASE=0x00010000). Any other address will result in a warning, and 64K will be used anyway.

This statement has the same effect as the `/BASE` linker option. If you specify both the statement and the option, the statement value overrides the option value.

CODE

```
►►──CODE──┬──────────────┬──┬─────────────┬──┬────────┬──────────►
          ├─CONFORMING───┤  ├─EXECUTEONLY─┤  ├─IOPL───┤
          └─NONCONFORMING┘  └─EXECUTEREAD─┘  └─NOIOPL─┘

►──┬────────────┬─────────────────────────────────────────────►◄
   ├─PRELOAD────┤
   └─LOADONCALL─┘
```

Defaults are: NONCONFORMING
EXECUTEREAD
NOIOPL
LOADONCALL

Use the CODE statement to define default attributes for code segments within the executable you are creating. You can override these default attributes for specific code segments by using the SEGMENTS statement (see “SEGMENTS” on page 462), or the `/SECTION` linker option.

Attribute Rules

- You can only specify one attribute from each pair. If you specify neither attribute, ILINK uses the default. See the description of the parameter for its default.
- Attributes can appear in any order.

CONFORMING|NONCONFORMING

Use these attributes to specify whether a code segment is a 286-conforming segment. These attributes are relevant for device drivers, or system-level code. They apply to code segments only.

CONFORMING specifies that the segment is conforming, and uses a range of instructions that can be executed by a 286 (16-bit) processor. A CONFORMING segment can be called from either Ring 2 or Ring 3, and executes at the privilege level of the caller.

- NONCONFORMING specifies that the segment is nonconforming, and uses instructions that require a 386 processor or higher. The segment is not guaranteed to be executable by a 286 processor.

The default is NONCONFORMING.

EXECUTEONLY|EXECUTEREAD

Use these attributes to specify whether a code segment can be read as well as executed. These attributes apply to code segments only.

- EXECUTEONLY specifies that the segment can only be executed.
- EXECUTEREAD specifies that the segment can be both executed and read.

The default is EXECUTEREAD.

IOPL|NOIOPL

Use these attributes to determine whether a segment has I/O privilege, that is, whether it can access the hardware directly.

- IOPL specifies that the segment has I/O privilege.
- NOIOPL specifies that the segment does not have I/O privilege.

Note: 32-bit segments must be NOIOPL. You cannot specify a 32-bit segment as IOPL.

PRELOAD|LOADONCALL

Use these attributes to specify when the segment is loaded.

Note: These attributes are ignored on OS/2 Version 2.0 or later.

- PRELOAD specifies that the segment will be loaded automatically when the program starts.
- LOADONCALL specifies that the segment will not be loaded until accessed.

The default is LOADONCALL.

Example

Given the following line in a .DEF file,

```
CODE LOADONCALL IOPL
```

CODE segments are not loaded until accessed (LOADONCALL), and have I/O hardware privilege (IOPL). In addition, the linker assumes the following defaults:

- EXECUTEREAD (can be read as well as executed)

- NONCONFORMING (not guaranteed to run on a machine based on the 80286 microprocessor)

These attributes apply to all CODE segments, except when you override them with the SEGMENTS statement

DATA

```
►►──DATA──┬──────────┬──┬────────────┬──┬──────────────┬──┬─────────────┬──────►
          ├─IOPL─────┤  ├─NONE───────┤  ├─PRELOAD──────┤  ├─READONLY────┤
          └─NOIOPL───┘  ├─SINGLE─────┤  └─LOADONCALL───┘  └─READWRITE───┘
                        └─MULTIPLE───┘

►──┬─────────────┬──────────────────────────────────────────────────────────►◄
   ├─SHARED──────┤
   └─NONSHARED───┘
```

Defaults are: NOIOPL
SINGLE for .DLL files, MULTIPLE for .EXE files.
LOADONCALL
READWRITE
SHARED for .DLL files, NONSHARED for .EXE files

Use the DATA statement to define default attributes for data segments within the executable you are creating. You can override these default attributes for specific data segments by using the SEGMENTS statement (described on page 462), or the /SECTION linker option.

Attribute Rules

- You can only specify one attribute from each group. If you specify none of the attributes in a group, the linker uses the default. See the description of the parameter for its default.
- Attributes can appear in any order.

IOPL|NOIOPL

Use these attributes to determine whether a segment has I/O privilege, that is, whether it can access the hardware directly.

- IOPL specifies that the segment has I/O privilege.
- NOIOPL specifies that the segment does not have I/O privilege.

Note: 32-bit segments must be NOIOPL. You cannot specify a 32-bit segment as IOPL.

NONE|SINGLE|MULTIPLE

Use these attributes to specify how the automatic data segment can be shared. The automatic data segment is the physical segment represented by the group name DGROUP. This segment group makes up the physical segment that contains the local stack and heap of the application.

- NONE specifies that no automatic data segment is created.
- SINGLE specifies that a single automatic data segment is shared by all instances of the module. In this case, the module is said to have solo data. SINGLE is the default for .DLL files.
- MULTIPLE specifies that the automatic data segment is copied for each instance of the module. In this case, the module is said to have instance data. MULTIPLE is the default for .EXE files.

PRELOAD|LOADONCALL

Use these attributes to specify when the segment is loaded.

Note: These attributes are ignored on OS/2 Version 2.0 or later.

- PRELOAD specifies that the segment will be loaded automatically whenthe program starts.
- LOADONCALL specifies that the segment will not be loaded until accessed.

The dafault is LOADONCALL

READONLY|READWRITE

Use these attributes to set the access rights to a data segment. These attributes apply to data segments only.

- READONLY specifies that the segment can only be read.
- READWRITE specifies that the segment can both be read and written to.

The default is READWRITE.

SHARED|NONSHARED

Use these attributes to specify whether the segment can be shared by other processes. These attributes apply to data segments only.

- SHARED specifies that one copy of the shared segment is loaded and shared among all processes accessing the module. SHARED is the default for dynamic link library (.DLL) files.
- NONSHARED specifies that the segment cannot be shared, and must be loaded separately for each process. NONSHARED is the default for executable program (.EXE) files.

Example

Given the following line in a .DEF file,

```
DATA LOADONCALL NONSHARED
```

DATA segments are not loaded until they are accessed (LOADONCALL), and cannot be shared between multiple copies of the program (NONSHARED). In addition, the linker assumes the following defaults:

- READWRITE (DATA segments can be read and written to)
- MULTIPLE (the automatic data segment is copied for each instance of the module)
- NOIO (DATA segments have no I/O hardware privilege)

These attributes apply to all DATA segments, except for the CONST32_RO segment, and any segments whose attributes you explicitly set with the SEGMENTS statement.

DESCRIPTION

```
►►──DESCRIPTION──'text'──────────────────────────────────────────►◄
```

Use the DESCRIPTION statement to insert the specified text into the .EXE or .DLL file you are creating. The DESCRIPTION statement is useful for embedding source control or copyright information into your program or DLL.

The inserted text must be a one-line string enclosed in single quotation marks.

Example

Given the following line in a .DEF file,

```
DESCRIPTION 'Template Program'
```

the linker inserts the text `Template Program` into the .EXE or .DLL file.

EXETYPE

```
►►──EXETYPE──┬─OS2─────┬─────────────────────────────────────────►◄
             ├─WINDOWS─┤
             └─UNKNOWN─┘
```

Default is: OS2

Use the EXETYPE statement to specify the operating system the .EXE or .DLL will run under. This statement is optional, and can provide an additional degree of protection against the program being run in an incorrect operating system.

Specify one of the following operating systems:

OS2 OS/2 .EXE and .DLL files (this is the default)

WINDOWS Microsoft Windows applications

UNKNOWN Other applications

When you use EXETYPE, the linker sets bits in the header that identify operating-system type. Operating-system loaders can then check these bits before running the application.

EXPORTS

```
              ┌───────────────────────────────────────────────────┐
►►──EXPORTS───▼─enm──=──inm──┬─────────────────────────┬──┬─────────┬─┴───►◄
                             └─@ord──┬───────────────┬─┘  └─pwrds─┘
                                     ├─RESIDENTNAME─┤
                                     └─NONAME───────┘
```

Use the EXPORT statement when you are creating a dynamic link library (DLL) to define the names and attributes of data and functions exported from the DLL, and of functions that run with I/O hardware privilege.

Note: Exported data and functions are those available to other .EXE or .DLL files. Data and functions that are **not** exported can only be accessed within your DLL, and cannot be accessed by other .EXE or .DLL files.

Give export definitions for functions and data in your DLL that you want to make available to other .EXE or .DLL files.

The EXPORTS keyword marks the beginning of the export definitions. Enter each definition on a separate line. You can provide the following information for each export:

enm The entry name of the data construct or function , which is the name other files use to access it. Always provide an entry name for each export.

inm The internal name of the data construct or function, which is its actual name as it appears **within** the DLL. If you do not specify an internal name, the linker assumes it is the same as *enm*.

ord The data construct or function's ordinal position in the module definition table. If you provide the ordinal position, the data construct or function can be referenced either by its entry name or by the ordinal. It is faster to access by ordinal positions, and may save space.

You can specify one of two values:

RESIDENTNAME Indicates that you want the data construct or function's name kept resident in memory at all times. You only need to specify RESIDENTNAME if you gave an ordinal position in *ord*. When a data construct or function does not have an ordinal position defined, OS/2 keeps the name of the exported data construct or function resident in memory by default.

NONAME Indicates that you want the data construct or function to always be referenced by its ordinal number. If you specify NONAME, the data construct or function cannot be referenced by name: it can only be referenced by ordinal number.

You cannot specify both values.

pwrds The total size of the function's parameters, as measured in words (bytes divided by two). This field is required only if the function executes with I/O privilege. When a function with I/O privilege is called, the operating system consults *pwrds* to determine how many words to copy from the caller's stack to the stack of the I/O-privileged function.

Example

The following example defines three exported functions:

- SampleRead
- StringIn
- CharTest

```
EXPORTS
  SampleRead = read2bin @8
  StringIn = str1       @4 RESIDENTNAME
  CharTest  6
```

The first two functions can be accessed either by their exported names or by an ordinal number. Note that in the module's own source code, these functions are actually defined as read2bin and str1, respectively. The last function runs with I/O privilege, and so has *pwrds* (the total size of the parameters) defined for it: six words.

HEAPSIZE

```
►►──HEAPSIZE──┬─size───┬──────────────────────────────────────────────►◄
              └─MAXVAL─┘
```

Use the HEAPSIZE statement to define the size of the application's local heap in bytes. This value affects the size of the automatic data segment (DGROUP), which contains the local stack and heap of the application.

You can enter any positive integer for the heap size.

Instead of entering the number of bytes, you can enter the keyword MAXVAL. This increases the size of DGROUP to 64K, if it is smaller than 64K. MAXVAL is useful in bound applications, when you want to force a 64K requirement for DGROUP. MAXVAL is not generally useful for 32-bit programs.

Example

Given the following line in a .DEF file,

```
HEAPSIZE 4000
```

the linker sets the local heap to 4000 bytes.

IMPORTS

```
              ┌────────────────────────────┐
►►──IMPORTS───▼─┬─────────────┬─dllname.entry─┴──────────────────────►◄
                └─intname──=──┘
```

Use the IMPORT statement to define the names of the functions imported from a DLL for your .EXE or .DLL file to use.

If your file references functions that are defined in a DLL, you must import the functions from the DLL before your file can use them. You must define them in the module definition file, to give the name of the DLL the functions are defined in.

Note: Instead of using the IMPORTS statement, you can use an import library (created by the IMPLIB utility) to resolve external references to DLL symbols.

The IMPORTS keyword marks the beginning of the import definitions. Enter each definition on a separate line. Each import definition corresponds to a particular function. The only limit on the number of import definitions is that the total amount of space required for their names must be less than 64K.

You can provide the following information for each import definition:

intname
The internal name of the function, that is used within your module to call the function. This is the name used by the importing module, although the function can have a different name in the module where it is defined (the exporting module). If *entry* contains a name, then by default, *intname* uses the same name.

dllname
The name of the DLL that contains the function. You must provide this information for each import you define.

entry
The function to be imported, identified either by entry name or by ordinal value.

You can only use an ordinal value if one is defined for the function in its export definition (see “EXPORTS” on page 456). If you use the ordinal value, then you must also define an *intname* for your module to use.

The entry name for the function is always defined in its export definition.

By default, the exporting module and importing module both call the function by its entry name. However, each module can provide its own internal name for the function. It is possible for the function to have up to three distinct names:

- The exporting module's internal name for the function (associated with the entry name by the export statement)
- The function's entry name (and an optional ordinal value)

- The importing module's internal name for the function (associated with either the entry name or the ordinal value by the import statement)

Example

The following example defines three functions to be imported:

- SampleRead
- SampleWrite
- A function that has been assigned an ordinal value of 1

```
IMPORTS
  Sample.SampleRead
  SampleA.SampleWrite
  ReadChar = Read.1
```

The functions are found in the modules Sample, SampleA, and Read, respectively. The SampleRead and SampleWrite functions are called by their entry names. The function from the Read module is called by the internal name ReadChar, which maps to the ordinal value 1. Its actual entry name is not shown, because it is called by ordinal value instead of by its entry name.

LIBRARY

```
►►──LIBRARY──┬─────────┬──┬──────────────┬──┬──────────────┬──────────►◄
             └─libname─┘  ├─INITGLOBAL───┤  ├─TERMGLOBAL───┤
                          └─INITINSTANCE─┘  └─TERMINSTANCE─┘
```

Use the LIBRARY statement to identify the output file as a dynamic link library (DLL), and optionally define the name, library module initialization, and library module termination.

You can also identify the output file as a DLL with the `/DLL` option.

The following table shows defaults for the fields, depending on whether the DLL has 16-bit entry points, or 32-bit entry points:

Figure 103. LIBRARY Default Values

Attribute	Default for DLLs with 16-bit Entry Points	Default for DLLs with 32-bit Entry Points
libname	Name of output file with .DLL extension removed	Name of output file with .DLL extension removed
Initialization routine	INITGLOBAL	Matches *term*, if termination given. Otherwise INITGLOBAL.
Termination routine	None (applies only to DLLs with 32-bit entry points)	Matches initialization routine, if initialization given. Otherwise TERMGLOBAL.

If you use the LIBRARY statement in your module definition (.DEF) file, it must be the first statement in the .DEF file, and you cannot use the NAME statement.

If you provide a name in *libname*, it becomes the name of the library as it is known by OS/2. The name can be any valid file name.

Specify one of the following library initialization routines to use:

INITGLOBAL The library initialization routine is called only when the library module is initially loaded into memory.

INITINSTANCE The library initialization routine is called each time a new process gains access to the library.

If you are generating a DLL with 32-bit entry points, you can set the type of library termination you want:

TERMGLOBAL The library termination routine is called only when the library module is unloaded from memory.

TERMINSTANCE The library termination routine is called each time a process gives up access to the library.

Example

The following example assigns the name `calendar` to the dynamic link library (DLL), and specifies that library initialization be performed each time a new process gains access. If `calendar` has 32-bit entry points, the linker will assume TERMINSTANCE.

```
LIBRARY calendar INITINSTANCE
```

NAME

```
►►──NAME──┬─────────┬──┬─────────────────┬──────────────────────────►◄
          └─appname─┘  ├─WINDOWAPI───────┤
                       ├─WINDOWCOMPAT────┤
                       └─NOTWINDOWCOMPAT─┘
```

Use the NAME statement to identify the output file as an executable program (.EXE file), and optionally define the name and type of the .EXE file.

You can also identify the output file as an .EXE file with the `/EXEC` option.

If you use the NAME statement in your module definition (.DEF) file, it must be the first statement in the .DEF file, and you cannot use the LIBRARY statement.

If you specify *appname*, it becomes the name of the .EXE as it is known by OS/2. The name can be any valid file name. If you do not provide a name, the name of the executable program is the same as the name of the output file, with the .EXE extension removed.

The NAME statement also allows you to define the type of the program:

Figure 104. NAME Statement Parameters

Type	Description	/PMTYPE option equivalent
WINDOWAPI	Presentation Manager application. The application uses the API provided by the Presentation Manager, and must run in the Presentation Manager environment.	PM
WINDOWCOMPAT	Application compatible with Presentation Manager. The application can run inside the Presentation Manager, or it can run in a separate screen group. An application can be of this type if it uses the proper subset of OS/2 video, keyboard, and mouse functions supported in the Presentation Manager applications.	VIO
NOTWINDOWCOMPAT	Application that is not compatible with the Presentation Manager and must run in a separate screen group from the Presentation Manager.	NOVIO

You can also use the `/PMTYPE` option to set the type. If conflicting types are defined by the option and in the NAME statement, the type defined by the NAME statement overrides the option value.

Example

The following example assigns the name `calendar` to the executable program, and specifies it as compatible with PM.

```
NAME calendar WINDOWCOMPAT
```

OLD

```
►►──OLD──'─┬─────┬─name'──────────────────────────────────────►◄
           └─dir─┘
```

Use the OLD statement when you create a dynamic link library (DLL) to preserve compatibility with an older version of the DLL. When you provide the *name* of the old DLL, specify the directory it is in as well, unless it is in the current working directory.

The linker compares exported data constructs or functions in the old DLL with exported data constructs or functions in the current DLL. If the old data construct or function has an ordinal value assigned to it, the linker assigns the ordinal value to the equivalent data construct or function in the new DLL.

If another run-time module called functions or referenced data from the old DLL by ordinal value, it can continue calling functions and referencing data from the new DLL using the same ordinal values.

The linker will only assign the old ordinal value to a data construct or function when:

- The data construct or function name in the old DLL matches the data construct or function name in the new DLL exactly
- The old data construct or function has an ordinal value assigned to it
- The new data construct or function does not already have an ordinal value assigned to it.

See “EXPORTS” on page 456 for more information on assigning ordinal values.

SEGMENTS

```
►►──SEGMENTS───────────────────────────────────────────────────────────►

   ┌─────────────────────────────────────────────────────────────┐
►──▼─┬───┬─segname─┬───┬──┬─────────────────────┬─┤ Attribs ├─┴──────►◄
     └─'─┘         └─'─┘  └─CLASS──'class'───────┘

Attribs:
├──┬───────┬──┬─CONFORMING────┬──┬─EXECUTEONLY─┬──┬─IOPL───┬────────────►
   └─ALIAS─┘  └─NONCONFORMING─┘  └─EXECUTEREAD─┘  └─NOIOPL─┘

►──┬────────────┬──┬─PRELOAD────┬──┬─READONLY──┬──┬─SHARED────┬──────────┤
   └─MIXED1632──┘  └─LOADONCALL─┘  └─READWRITE─┘  └─NONSHARED─┘
```

Defaults are: NONCONFORMING
EXECUTEREAD
NOIOPL
LOADONCALL
READWRITE
SHARED for .DLL files, NONSHARED for .EXE files.

Use the SEGMENTS statement to define the attributes of one or more segments in the .EXE or .DLL file on a segment-by-segment basis. The attributes you specify in this statement override any defaults set in the CODE and DATA statements.

You can also set some segment attributes with the /SECTION option.

The SEGMENTS keyword marks the beginning of the segment definitions. Enter each definition on a separate line. You can enter up to 256 separate definitions.

Each segment definition begins with its name (*segname*). If *segname* is the same as a module statement or keyword (such as DATA or IOPL), then you must enclose *segname* in single quotation marks (').

You can specify the class of the segment with the CLASS keyword, followed by the class of the segment, enclosed in single quotation marks ('). If you do not specify a class for the segment, the linker assumes the segment is of class CODE.

After the name and class, you can set attributes for the segment. If you do not specify attributes for a segment, the linker assumes a default set of attributes, as underlined in the parameters list above, or as set by the CODE and DATA statements. The default for SHARED|NONSHARED varies, depending on the type of output file.

Attribute Rules

- You can only specify one attribute from each pair. If you specify neither attribute, ILINK uses the default. See the description of the parameter for its default.
- Attributes can appear in any order.

ALIAS

Use ALIAS to allow the segment to be addressed using both the 16-bit segmented method (_far16), and the 32-bit linear method. When you specify ALIAS, the loader prepares an additional segment selector for the segment that allows for 16-bit addressing of the segment. The segment can then be called using 16-bit far calls and 32-bit near calls.

By default, segments are addressable only by the 32-bit linear method.

CONFORMING|NONCONFORMING

Use these attributes to specify whether a code segment is a 286-conforming segment. These attributes are relevant for device drivers, or system-level code. They apply to code segments only.

CONFORMING specifies that the segment is conforming, and uses a range of instructions that can be executed by a 286 (16-bit) processor. A CONFORMING segment can be called from either Ring 2 or Ring 3, and executes at the privilege level of the caller.

- NONCONFORMING specifies that the segment is nonconforming, and uses instructions that require a 386 processor or higher. The segment is not guaranteed to be executable by a 286 processor.

The default is NONCONFORMING.

EXECUTEONLY|EXECUTEREAD

Use these attributes to specify whether a code segment can be read as well as executed. These attributes apply to code segments only.

- EXECUTEONLY specifies that the segment can only be executed.
- EXECUTEREAD specifies that the segment can be both executed and read.

The default is EXECUTEREAD.

IOPL|NOIOPL

Use these attributes to determine whether a segment has I/O privilege, that is, whether it can access the hardware directly.

- IOPL specifies that the segment has I/O privilege.

- NOIOPL specifies that the segment does not have I/O privilege.

Note: 32-bit segments must be NOIOPL. You cannot specify a 32-bit segment as IOPL.

MIXED1632

Use MIXED1632 to specify that the segment is part of a group that allows a mix of 16-bit and 32-bit code. If you create a group that allows such mixing, you must declare each segment in the group as MIXED1632.

PRELOAD|LOADONCALL

Use these attributes to specify when the segment is loaded.

Note: These attributes are ignored on OS/2 Version 2.0 or later.

- PRELOAD specifies that the segment will be loaded automatically when the program starts.
- LOADONCALL specifies that the segment will not be loaded until accessed.

The default is LOADONCALL.

READONLY|READWRITE

Use these attributes to set the access rights to a data segment. These attributes apply to data segments only.

- READONLY specifies that the segment can only be read.
- READWRITE specifies that the segment can both be read and written to.

The default is READWRITE.

SHARED|NONSHARED

Use these attributes to specify whether the segment can be shared by other processes. These attributes apply to data segments only.

- SHARED specifies that one copy of the shared segment is loaded and shared among all processes accessing the module. SHARED is the default for dynamic link library (.DLL) files.
- NONSHARED specifies that the segment cannot be shared, and must be loaded separately for each process. NONSHARED is the default for executable program (.EXE) files.

Example

The following example specifies segments named cseg1, cseg2, and dseg. The first segment is assigned class `mycode`, and the third segment is assigned class `data`. The second segment is assigned class `CODE` by default. Each segment is given different attributes.

```
SEGMENTS
  cseg1 CLASS 'mycode' IOPL
  cseg2 EXECUTEONLY PRELOAD CONFORMING
  dseg CLASS 'data' LOADONCALL READONLY
```

STACKSIZE

```
►►──STACKSIZE──size──────────────────────────────────────────────►◄
```

Use STACKSIZE to set the stack size (in bytes) of your program. The size must be an even number, from 0 to 0xFfffFffe. If you specify an odd number, it is rounded up to the next even number.

You cannot specify a stack size in which the second most significant byte is either 02 or 04 (in hex), because of a restriction in OS/2 2.0. The linker issues a warning, and adds 64k to the stack size to avoid the restriction.

For example, if you specify `STACKSIZE 0x00020000` the linker adds 64k, which results in `STACKSIZE 0x00030000`

Similarly, if you specify `STACKSIZE 0x11041111` the linker adds 64k, which results in `STACKSIZE 0x11051111`

If your program generates a stack-overflow message, use the STACKSIZE statement to increase the size of the stack.

If your program uses the stack very little, you can save some space by decreasing the stack size.

Note: Once the output file is produced, you can still change its stack size, using the EXEHDR utility in the Warp toolkit.

The STACKSIZE statement is equivalent to the `/STACK` linker option. If you specify both the statement and the option, the statement value overrides the option value.

Example

The following example allocates 4 K of local-stack space:

```
STACKSIZE 4096
```

STUB

```
►►──STUB──'filename'─────────────────────────────────────────────►◄
```

Use the STUB statement to add a DOS .EXE file to the beginning of your .EXE or .DLL file. The stub function is then invoked whenever your .EXE or .DLL file is run under DOS. Typically, the stub displays the message that the program cannot run in DOS mode, and ends the program.

STUB Example

If you do not use the STUB statement, the linker adds its own standard stub for this purpose.

The linker searches for the file name you specify as the stub as follows:

1. In the directory you specify, or in the current directory if you did not give a path
2. In the directories listed in the PATH environment variable

Example

The following example adds the DOS .EXE file STOPIT.EXE to the beginning of the file you are creating. STOPIT.EXE runs whenever your file is run under DOS.

```
STUB 'STOPIT.EXE'
```

Chapter 26. Preparing COBOL Programs for Multithreading

In the PC environment, programs may run within the threads of processes. COBOL supports multithreaded execution by means of the THREAD compiler option (see "THREAD" on page 194). In order to understand this chapter's discussion of COBOL support for multithreading, you need to be familiar with the following terms:

Process
The operating system and multithreading applications can handle execution flow within a process. Multiple processes can run concurrently, and programs run within a process can share resources. Processes can be manipulated (for example, they can be given a high or low priority in terms of the amount of time the system devotes to running the process).

Thread
Within a process, an application can initiate one or more threads. Within the thread, control is transferred between executing programs.

Run-unit
In a multithreaded environment, a COBOL run-unit is defined as the portion of the process including threads with actively executing COBOL programs. The COBOL run-unit continues until no COBOL program is active in the execution stack for any of the threads (for example, a called COBOL program contains a GOBACK statement and returns control to a C program). Within the run-unit, COBOL programs can call non-COBOL programs, and vice versa.

Program Invocation Instance
Within a thread, control is transferred between separate COBOL and non-COBOL programs. For example, a COBOL program can CALL another COBOL program or a C program. Each separately invoked (as in, CALLed) program is a program invocation instance. Program invocation instances of a particular program might exist in multiple threads within a given process.

The following illustration shows the relationships between processes, threads, run-units, and program invocation instances.

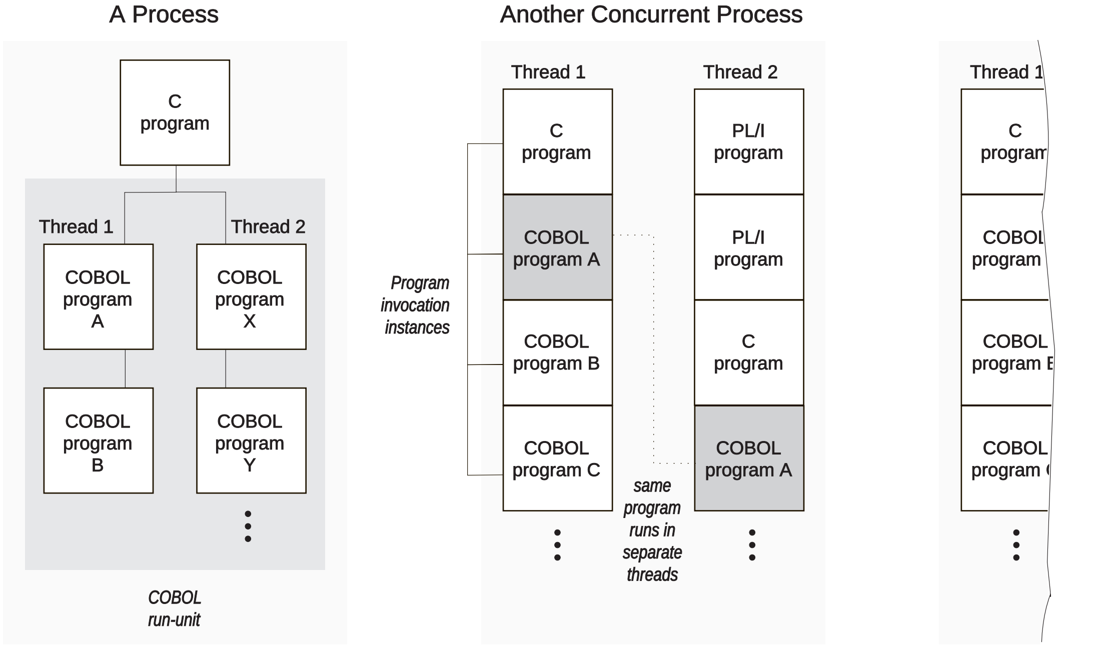


Figure 105. Schematic Illustration of Multithreading Concepts

COBOL does not directly support initiating or managing program threads. However, COBOL programs can run as threads in multithreaded environments. In other words, COBOL programs can be invoked by other applications such that they are running in multiple threads within a process or as multiple program invocation instances within a thread. This enables COBOL programs to run in multithreading environments like Presentation Manager and MQSeries Three Tier applications.

This remainder of this chapter contains information that will help you prepare your COBOL programs for multithreaded environments.

Caution: Do not confuse multiprocessing or multithreading with “multitasking,” which is generally used to describe the external behavior of applications. That is, the operating system appears to be running more than one application simultaneously. Multitasking has no relevance in this discussion.

How Language Elements Are Interpreted in a Multithreaded Environment

Because your COBOL programs can be run as separate threads within a process, be aware that language elements might be interpreted in two ways:

Run-unit scope

The language element persists for the duration of the COBOL run-unit execution and is available to other programs within the thread.

Program invocation instance scope
The language element persists only within a particular program invocation instance.

These two types of scope are important in two contexts:

Reference
Describes where an item can be referenced from. For example, if a data item has run-unit reference scope, any program invocation instance in the run-unit can reference the data item.

State
Describes how long an item persists in storage. For example, if a data item has program invocation instance state scope, it will remain in storage only while the program invocation instance is running.

The following table summarizes the reference and state scope of various COBOL language elements.

Figure 106. Reference and State Scope for Language Elements in a Multithreading Environment

Language Element	Reference Scope	State Scope
ADDRESS-OF special register	Same as associated record	Program invocation instance
Files	Run-unit	Run-unit
Index data	Program	Program invocation instance
LENGTH of special register	Same as associated identifier	Same as associated identifier
LINAGE-COUNTER special register	Run-unit	Run-unit
LINKAGE-SECTION data	Run-unit	Based on scope of underlying data.
LOCAL-STORAGE data	Within the thread	Program invocation instance
RETURN-CODE	Run-unit	Program invocation instance
SORT-CONTROL, SORT-CORE-SIZE, SORT-RETURN, TALLY special registers	Run-unit	Program invocation instance
WHEN-COMPILED special register	Run-unit	Run-unit
WORKING-STORAGE data	Run-unit	Run-unit

Working with Run-Unit Scoped Elements

If you have resources with run-unit scope (such as GLOBAL data declared in the WORKING-STORAGE section), it is your responsibility to synchronize access to that data from multiple threads using logic in the application. You can do one or both of the following:

- Structure the application such that run-unit scoped resources are not accessed simultaneously from multiple threads.
- If you are going to access resources simultaneously from separate threads, synchronize access using facilities provided by C or by platform functions.

If you have resources with run-unit scope, and you want those resources to be isolated within an individual program invocation instance (for example, programs with individual copies of data), define the data in the local storage section so that it will have program invocation instance scope.

Working with Program Invocation Instance Scoped Elements

With these language elements, storage is allocated for each individual program invocation instance. Therefore, even if a program is invoked multiple times among multiple threads, each time it is invoked it will be allocated separate storage. For example, if program X is invoked in two or more threads, each program invocation instance of X gets its own set of resources, such as storage.

Because the storage associated with these language elements is program invocation instance scoped, data is protected from access across threads and you do not have to concern yourself with access synchronization. However, this data cannot be shared between invocations of programs unless it is explicitly passed.

Choosing THREAD for Multithreading Support

Select the THREAD compiler option for multithreading support. Choose THREAD only if you think your program will be invoked more than once in a single process by an application (such as a MQSeries Three Tier application.) Compiling with THREAD prepares the COBOL run-time environment for threading support. However, compiling with THREAD may reduce program performance. You must compile all of the programs in the run unit with THREAD; you cannot mix programs compiled with THREAD and those compiled with NOTHREAD in one run unit.

The default option is NOTHREAD. For more information about the THREAD compiler option, see “THREAD” on page 194.

Language Restrictions under THREAD

When THREAD is in effect, the following language elements are not supported and are flagged by the compiler with error-level messages:

- ALTER statement
- DEBUG-ITEM special register
- GO TO statement without a procedure name

- INITIAL PROGRAM
- RERUN
- Segmentation module
- STOP literal statement
- STOP RUN

Recursion with Threading

If a program is compiled with the THREAD compiler option, the program may be invoked recursively in a threading or non-threading environment. This applies whether or not the RECURSIVE phrase is specified in the PROGRAM-ID paragraph.

For considerations in using the LINKAGE SECTION with recursive calls, see “With Recursion or Multithreading” on page 22.

Control Transfer within a Multithreaded Environment

Be aware of the following control transfer issues when writing COBOL programs for a multithreaded environment:

CALL and CANCEL
As is the case in single-threaded environments, a program invoked is in its initial state the first time it is called within a run unit and the first time it is called after a CANCEL to the CALLED program.

EXIT PROGRAM
EXIT PROGRAM from the first program of a thread terminates that thread. EXIT PROGRAM returns to the caller of the program without terminating the thread in all other cases. EXIT PROGRAM from a main program is treated as a comment.

GOBACK
Same as EXIT PROGRAM, except that GOBACK from a main program terminates the thread.

If it can be determined that there are no other COBOL programs active in the run unit, the COBOL run unit termination process (including closing all open COBOL files) is performed on the GOBACK from the first program of this thread. This determination can be made if all COBOL programs invoked within the run unit have returned to their invokers via GOBACK or EXIT PROGRAM.

Examples on when this determination cannot be made are:

- A thread with one or more active COBOL programs was terminated (for example, because of an exception or via pthread_exit).
- A `longjmp()` was executed which resulted in collapsing active COBOL programs in the invocation stack.

In general, it is recommended that the programs initiating and managing multiple threads use the COBOL pre-initialization interface.

Pre-initialization
If your program initiates multiple COBOL threads (for example, your C program calls COBOL programs to carry out I/O), do not assume the COBOL programs

will “clean up” their environment. Particularly, do not assume that files will be automatically closed. You should pre-initialize the COBOL environment so that your application can control the COBOL “clean-up” (see Chapter 28, “Pre-initializing the COBOL Run-Time Environment” on page 489).

STOP RUN: There is no COBOL function that effectively does a STOP RUN in a threaded environment. If you need this behavior, consider invoking the C `exit()` function from your COBOL program and using `_iwzCOBOLTerm` after the run-unit termination exit.

Limitations on COBOL in a Multithreaded Environment

Some COBOL applications depend on subsystems or other applications. In a multi-threaded environment, these dependencies result in some limitations on COBOL programs:

DB2
The DB2 application may be run in multiple threads. However, any necessary synchronization for DB2 data access is the responsibility of the application.

SORT/MERGE
SORT and MERGE should only be active in one thread at a time. However, this is not enforced by the COBOL run-time environment— it must be controlled by the application.

VSAM file I/O
I/O for VSAM files should be active from only one thread at a time. However, this is not enforced by the COBOL run-time environment— it must be controlled by the application.

In general, synchronizing access to resources visible to an application within a run unit is the responsibility of the application. The exceptions to this are DISPLAY and ACCEPT, which can be executed from multiple threads without any synchronization by applications.

Example of Using COBOL in a Multithreaded Environment

This example consists of a C main program that creates two COBOL threads, waits for the COBOL threads to finish, then exits.

Sample Code for the Multithreading Example

The example has three code samples:

thrcob.c A C main program that creates the COBOL threads, waits for them to finish, then exits.

subd.cbl A COBOL program that is run by the thread created by `thrcob.c`.

sube.cbl A second COBOL program that is run by the thread created by `thrcob.c`.

The sample code for `thrcob.c` is shown in Figure 107.

```
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#define LINKAGE _System
#define STACKSIZE 4096
#include <os2.h>
#include   <stdio.h>
#include   <setjmp.h>
#pragma handler(SUBD)
#pragma handler(SUBE)

long done;
jmp_buf Jmpbuf;

extern void _iwzCOBOLInit(int fcode, PFN StopFun, int *err_code, void *StopArg);
extern void _iwzCOBOLTerm(int fcode, int *err_code);
extern void LINKAGE SUBD(void);
extern void LINKAGE SUBE(void);

int LINKAGE
StopFun(long *stoparg)
{
   printf("inside StopFun. Got stoparg = %d\n", *stoparg);
   *stoparg = 123;
   longjmp(Jmpbuf,1);
}


long StopArg = 0;

void LINKAGE
testrc(int rc, const char *s)
{
   if (rc != 0){
           printf("%s: Fatal error rc=%d\n",s,rc);
           exit(-1);
        }
}
```

Figure 107 (Part 1 of 2). Source Code for thrcob.c

```
void LINKAGE
pgmy()
{
   TID  t1, t2;
   int rc;
   int parm1, parm2;

   parm1 = 20;
   parm2 = 10;
   _iwzCOBOLInit(1, StopFun, &rc, &StopArg);
   printf( "_iwzCOBOLinit got %d\n",rc);
   rc = DosCreateThread(&t1, (PFNTHREAD)SUBD,(ULONG)&parm1, 0, STACKSIZE);
   testrc(rc,"create 1");
   rc = DosCreateThread(&t2, (PFNTHREAD)SUBE,(ULONG)&parm2, 0, STACKSIZE);
   testrc(rc,"create 2");
   printf("threads are %x and %x\n",t1, t2);

   DosWaitThread(&t1, DCWW_WAIT);
   DosWaitThread(&t2, DCWW_WAIT);

   printf("test gets done = %d \n",done );
   _iwzCOBOLTerm(1, &rc);
   printf( "_iwzCOBOLTerm got expect rc=0, got rc=%d\n",rc);
}

LINKAGE
main()
{
   if (setjmp(Jmpbuf) ==0) {
           pgmy();
        }
}
```

Figure 107 (Part 2 of 2). Source Code for thrcob.c

The sample code for `subd.cbl` is shown in Figure 108 on page 475.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. "subd".
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
      *
       DATA DIVISION.
       LOCAL-STORAGE SECTION.
       01 N2               PIC 9(8) comp-5 value 0.
      *
       LINKAGE SECTION.
       01 N1               PIC 9(8) comp-5.
      *
       PROCEDURE DIVISION using by reference n1.
           perform n1 times
                DISPLAY "subd gets " n1 " " n2
                compute n2 = n2 + 1
           end-perform
           DISPLAY "subd goback with " n1 " " n2
           GOBACK.
```

Figure 108. Source Code for subd.cbl

The sample code for `sube.cbl` is shown in Figure 109.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. "sube".
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
      *
       DATA DIVISION.
       LOCAL-STORAGE SECTION.
       01 N2               PIC 9(8) comp-5 value 0.
      *
       LINKAGE SECTION.
       01 N1               PIC 9(8) comp-5.
      *
       PROCEDURE DIVISION using by reference n1.
           perform n1 times
                DISPLAY "sube gets " n1 " " n2
                compute n2 = n2 + 1
           end-perform
           DISPLAY "sube goback with " n1 " " n2
           GOBACK.
```

Figure 109. Source Code for sube.cbl

Compiling, Linking, and Running the Multithreading Example

To create and run the multithreading example, follow these steps:

1. To compile `thrcob.c`, enter `icc /Gm+ /C+ thrcob.c` at a command prompt
2. To compile `subd.cbl`, enter `cob2 -qthread -c subd.cbl`

3. To compile `sube.cbl`, enter `cob2 -qthread -c sube.cbl`
4. To generate the executable `thrcob.exe`, enter `cob2 thrcob.obj subd.obj sube.obj`
5. To run the program `thrcob`, enter `thrcob`

Chapter 27. National Language Support Considerations

This chapter provides an overview of things you need to know about National Language Support (NLS) when using IBM VisualAge COBOL.

The NLS support of this product includes the support for multiple code pages. The characters represented in a supported code page can be used in COBOL names, data definitions, literals, and in common entries.

The following list summarizes the double-byte character set (DBCS) code page support:

- **User-defined names**

  DBCS names are supported.

- **Data type**

  DBCS data type (with PICTURE N or G) is supported.

- **Literal**

  DBCS literals are supported.

- **Comments**

  DBCS comments are supported.

- **Collating sequence**

  Collating sequences for single-byte character set (ASCII and DBCS data are locale sensitive: that is, based on the collating sequence indicated by the locale, except for ASCII compares with a non-NATIVE collating sequence in effect).

  This collating sequence rule applies to the single-byte characters whether the single-byte characters are from an ASCII or DBCS code page.

  Since the ANSI COBOL language elements dictate the knowledge of the collating sequence at compile time, it is expected that the locale setting in effect at the compile time and at the application execution time are consistent.

Locale: A *locale* is a collection of data that encodes information about a cultural environment. *Localization* is an action that establishes a cultural environment for an application by selecting the active locale. Only one locale can be active at one time. The active locale affects the behavior on the locale-sensitive interfaces for the entire program. This is called the *global locale model*.

The current locale is set using the LANG environment variable.

Locales and Code Sets Supported

OS/2

Figure 110 shows the locales supported by VisualAge COBOL in an OS/2 environment.

Figure 110 (Page 1 of 2). Locales Supplied with VisualAge COBOL (OS/2)

Locale Name	Language	Country/Area	ASCII Code Sets	EBCDIC Code Sets
ar_AA	Arabic	Arabic Area	IBM-864	IBM-420
bg_BG	Bulgarian	Bulgaria	IBM-855	IBM-880[1], IBM-1025
cz_CZ	Czech	Czech Republic	IBM-852	IBM-870
da_DK	Danish	Denmark	IBM-850, IBM-437	IBM-277
de_CH	German	Switzerland	IBM-850, IBM-437	IBM-500
de_DE	German	Germany	IBM-850, IBM-437	IBM-273
el_GR	Greek	Greece	IBM-869	IBM-875
en_GB	English	United Kingdom	IBM-850, IBM-437	IBM-285
en_US	English	United States	IBM-850, IBM-437	IBM-037
es_ES	Spanish	Spain	IBM-850, IBM-437	IBM-284
fi_FI	Finnish	Finland	IBM-850, IBM-437	IBM-278
fr_BE	French	Belgium	IBM-850, IBM-437	IBM-500
fr_CA	French	Canada	IBM-850, IBM-437	IBM-037
fr_CH	French	Switzerland	IBM-850, IBM-437	IBM-500
fr_FR	French	France	IBM-850, IBM-437	IBM-297
hr_HR	Croatian	Croatia	IBM-852	IBM-870
hu_HU	Hungarian	Hungary	IBM-852	IBM-870
it_IT	Italian	Italy	IBM-850, IBM-437	IBM-280
iw_IL	Hebrew	Israel	IBM-856	IBM-424[1], IBM-803
ja_JP	Japanese	Japan	IBM-932, IBM-942, IBM-943	IBM-930[1], IBM-939
ko_KR	Korean	Korea	IBM-949	IBM-933

Figure 110 (Page 2 of 2). Locales Supplied with VisualAge COBOL (OS/2)

Locale Name	Language	Country/Area	ASCII Code Sets	EBCDIC Code Sets
mk_MK	Macedonian	Macedonia, former Yugoslav Republic of	IBM-855	IBM-880[1], IBM-1025
nl_BE	Flemish	Belgium	IBM-850, IBM-437	IBM-500
nl_NL	Dutch	Netherlands	IBM-850, IBM-437	IBM-037
no_NO	Norwegian	Norway	IBM-850, IBM-437	IBM-277
pl_PL	Polish	Poland	IBM-852	IBM-870
pt_PT	Portuguese	Portugal	IBM-850, IBM-437	IBM-037
ru_RU	Russian	Russia	IBM-855	IBM-880[1], IBM-1025
sh_SP	Serbo-Croation	Serbia	IBM-852	IBM-870
sk_SK	Slovak	Slovakia	IBM-852	IBM-870
sl_SI	Slovene	Slovenia	IBM-852	IBM-870
sr_SP	Serbian	Serbia	IBM-855	IBM-880[1], IBM-1025
sv_SE	Swedish	Sweden	IBM-850, IBM-437	IBM-278
tr_TR	Turkish	Turkey	IBM-857	IBM-1026
zh_CN	Simplified Chinese	China	IBM-1381, IBM-1386	IBM-935
zh_TW	Traditional Chinese	Taiwan	IBM-938, IBM-948, IBM-950	IBM-937

Note:

1. Indicates default EBCDIC code set

OS/2

Windows

Figure 111 shows the locales supported by VisualAge COBOL in a Windows environment.

Figure 111 (Page 1 of 2). Locales Supplied with VisualAge COBOL (Windows)

Locale Name	Language	Country/Area	ASCII Code Sets	EBCDIC Code Sets
ar_AA	Arabic	Arabic Area	IBM-1046	IBM-420
bg_BG	Bulgarian	Bulgaria	IBM-1251	IBM-880[1], IBM-1025
cz_CZ	Czech	Czech Republic	IBM-1250	IBM-870
da_DK	Danish	Denmark	IBM-1252	IBM-277
de_CH	German	Switzerland	IBM-1252	IBM-500
de_DE	German	Germany	IBM-1252	IBM-273
el_GR	Greek	Greece		IBM-875
en_GB	English	United Kingdom	IBM-1252	IBM-285
en_US	English	United States	IBM-1252	IBM-037
es_ES	Spanish	Spain	IBM-1252	IBM-284
fi_FI	Finnish	Finland	IBM-1252	IBM-278
fr_BE	French	Belgium	IBM-1252	IBM-500
fr_CA	French	Canada	IBM-1252	IBM-037
fr_CH	French	Switzerland	IBM-1252	IBM-500
fr_FR	French	France	IBM-1252	IBM-297
hr_HR	Croatian	Croatia	IBM-1250	IBM-870
hu_HU	Hungarian	Hungary	IBM-1250	IBM-870
it_IT	Italian	Italy	IBM-1252	IBM-280
iw_IL	Hebrew	Israel	IBM-1255	IBM-424[1], IBM-803
ja_JP	Japanese	Japan	IBM-943	IBM-930[1], IBM-939
ko_KR	Korean	Korea	IBM-1363	IBM-933
mk_MK	Macedonian	Macedonia, former Yugoslav Republic of	IBM-1251	IBM-880[1], IBM-1025
nl_BE	Flemish	Belgium	IBM-1252	IBM-500
nl_NL	Dutch	Netherlands	IBM-1252	IBM-037
no_NO	Norwegian	Norway	IBM-1252	IBM-277
pl_PL	Polish	Poland	IBM-1250	IBM-870
pt_PT	Portuguese	Portugal	IBM-1252	IBM-037
ru_RU	Russian	Russia	IBM-1251	IBM-880[1], IBM-1025

Figure 111 (Page 2 of 2). Locales Supplied with VisualAge COBOL (Windows)

Locale Name	Language	Country/Area	ASCII Code Sets	EBCDIC Code Sets
sh_SP	Serbo-Croation	Serbia	IBM-1250	IBM-870
sk_SK	Slovak	Slovakia	IBM-1250	IBM-870
sl_SI	Slovene	Slovenia	IBM-1250	IBM-870
sr_SP	Serbian	Serbia	IBM-1251	IBM-880[1], IBM-1025
sv_SE	Swedish	Sweden	IBM-1252	IBM-278
tr_TR	Turkish	Turkey	IBM-1254	IBM-1026
zh_CN	Simplified Chinese	China	IBM-1386	IBM-935
zh_TW	Traditional Chinese	Taiwan	IBM-950	IBM-937

Note:

1. Indicates default EBCDIC code set

◀Windows

Figure 112 shows the code set translations supported by VisualAge COBOL.

Figure 112. Supported VisualAge COBOL ASCII to EBCDIC Code Set Translations

Language Group	From ASCII Code Set	To EBCDIC Code Set
Arabic	IBM-864	IBM-420, IBM-8612
Cyrillic	IBM-866	IBM-880, IBM-1025, IBM-1123
Latin-1	IBM-437, IBM-850, IBM-860, IBM-861, IBM-863, IBM-865	IBM-037, IBM-273, IBM-277, IBM-278, IBM-280, IBM-284, IBM-285, IBM-297, IBM-500, IBM-871
Latin-2	IBM-852, IBM-4948	IBM-870
Thai	IBM-874	IBM-838, IBM-9030
Turkey	IBM-857	IBM-905, IBM-1026

Note: Other code set translations are possible but might result in substitution characters (`X'3F'`) being used for characters which are incompatible.

DBCS User-Defined Word Support

You can form user-defined words using double-byte characters.

National Language Support Considerations

Usage Notes

A user-defined word containing double-byte characters must not contain more than 15 characters.

A DBCS user-defined word can contain both multi-byte and single-byte characters. When a character exists in both single-byte and multi-byte forms, its single-byte and multi-byte representations will not be regarded as equivalent. For example, “A” represented in double-bytes will not be considered to match “A” represented in a single byte.

Alphabet-names, class-names, condition-names, data-names, file-names, mnemonic-names, record-names, and symbolic characters must contain at least one single-byte alphabetic character or one multi-byte character.

A user-defined word containing multi-byte characters may not be continued.

Restrictions on Specific User-Defined Words

The IBM COBOL compiler supports the *level-number* user-defined word only when represented in SBCS digits.

Support for the *library-name*, *program-name*, and *text-name* user-defined words with DBCS depends on the DBCS name support of the platform. IBM COBOL will allow double-byte or EUC characters in these names.

DBCS Literal Support

There are two literal types to represent double-byte character constants: N'dbcs characters' and G'dbcs characters'.

Additionally, you can specify any character in one of the supported code pages using the alphanumeric literal syntax. However, such a literal is treated as alphanumeric in COBOL language semantics (that is, semantics appropriated for single-byte characters).

The literal delimiters can be apostrophes or quotes depending on the APOST or QUOTE compiler option setting.

A non-numeric literal containing double-byte characters cannot be continued. The maximum length of a N or G literal is 28 double-byte characters. The maximum length of a N or G literal is limited only by the available space in Area B on a single source line.

Figurative constant [ALL]SPACE and [ALL]SPACES represent space characters in SBCS or DBCS.

The ALL literal represents all or part of the string generated by successive concatenations of the single-byte characters or double-byte characters comprising the literal. The literal must be a non-numeric literal or a DBCS literal. The literal must not be a figurative constant.

DBCS Data Type Support

The DBCS class and category are defined as shown in the following table.

Level of Item	Class	Category
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Numeric edited Alphanumeric edited Alphanumeric
	DBCS	DBCS
Nonelementary (Group)	Alphanumeric	Alphabetic Numeric Numeric edited Alphanumeric edited Alphanumeric DBCS

If a data item is declared with PICTURE N or G, the selected locale must indicate a DBCS code page. In all other cases, the PICTURE characters N and G and USAGE DISPLAY-1 are flagged as errors.

Declaring DBCS Data

DBCS data is declared as follows:

PICTURE clause

A double-byte character position is represented by picture symbols N, G, or B.

A DBCS data item has the PICTURE character string with PICTURE symbols G, G and B, or N. Each PICTURE symbol represents a DBCS character position. The number of bytes occupied by each double-byte character is assumed to be two. That is, single-byte characters of a DBCS code page should not be included in a DBCS data item.

Operations on DBCS strings not conforming to this rule might produce unpredictable results, such as the truncation of a string at a byte position in the middle of a double-byte character.

This rule will not be enforced at run time. For a code page with characters represented in double bytes, the following padding and truncation rules apply where COBOL language semantics specify truncation or padding with spaces:

- Padding

  For operations involving DBCS data items, the padding is done using the double-byte space characters until the data area is filled. This is based on the number of byte positions allocated for the data area.

Where the padding may not be in the multiple of the code page width (for example, a group item moved to a DBCS data item), the padding is done with single-byte space characters.

- Truncation

  The truncation is done based on the size of the target data area on the byte boundary of the end of that data area. It is the application program's responsibility to ensure that such a truncation does not result in truncation of bytes representing a partial double-type character.

USAGE clause
A DBCS data item is specified with USAGE DISPLAY-1. When you use PICTURE symbol G, you must specify USAGE DISPLAY-1. When you use PICTURE symbol N, USAGE DISPLAY-1 is implied and the USAGE clause may be omitted.

VALUE clause
The associated VALUE clause (if specified) must specify a DBCS literal or the figurative constants SPACE OR SPACES.

Reference modification
For the purpose of handling *reference modifications*, each character in a DBCS data item is considered to occupy the number of bytes corresponding to the code page width (that is, 2).

DBCS Class Test

Kanji and DBCS class test are defined to be consistent with their IBM System/390 definitions. Both class tests are internally performed by converting the double-byte characters to the double-byte characters defined for IBM System/390. The converted double-byte characters are tested for DBCS and Japanese graphic characters.

Kanji class test results in testing for valid Japanese graphic characters. This includes Katakana, Hiragana, Roman, and Kanji character sets.

The Kanji class test is done by checking the converted characters for X'41' - X'7E' for the first byte and X'41' - X'FE' for the second byte plus the space character, X'4040'.

DBCS class test results in testing for valid graphic characters for the code page.

The DBCS class test is done by checking the converted characters for X'41' - X'FE' for both the first and second byte of each character plus the space character, X'4040'.

Collating Sequence

Considerations for DBCS and ASCII collating sequence are as described below. Any comparison involving a group item will be handled based on the comparison of the byte for byte positions in hex.

The following clauses in the SPECIAL-NAMES paragraph may be specified only if the code page in effect is an ASCII code page:

- ALPHABET clause
- SYMBOLIC CHARACTER clause
- CLASS clause

These clauses, if specified with a DBCS code page in effect, will be diagnosed and treated as comments.

DBCS Collating Sequence

Data items and literals of class DBCS can be used in a relation condition with any relational operator. Each operand must be either of class DBCS, alphabetic, or alphanumeric (elementary or group). Note that this allows, for example, a comparison of a DBCS item with an alphanumeric item. No conversion or editing is done. No distinction is made between items of category DBCS and items of category DBCS edited.

DBCS comparisons are performed based on the rules for non-numeric comparisons. The comparison is based on the *locale* setting for the collating sequence if the operands are elementary DBCS or alphanumeric data items.

The PROGRAM COLLATING SEQUENCE clause has no effect on comparisons involving data items of class DBCS or DBCS literals.

ASCII Collating Sequence

The ANSI COBOL rules on the PROGRAM COLLATING SEQUENCE clause and the COLLATING SEQUENCE clause on SORT and MERGE apply.

If the collating sequence in effect is NATIVE (which is default if neither the COLLATING SEQUENCE clause nor the PROGRAM COLLATING SEQUENCE clause is specified), the collating sequence is based on the *locale* setting. This applies to SORT or MERGE statements as well as to the program collating sequence.

Note that the collating sequence impacts the processing of the alphabetic clause (for example, *literal-1* THRU *literal-2*), SYMBOLIC CHARACTERS specifications, and VALUE range specifications for Level 88 items as well as relation conditions and SORT and MERGE statements.

Since the rules of the COBOL user-defined alphabet name and symbolic characters assume a character-by-character collating sequence (not a collating sequence which may depend on a sequence of multiple characters), the locale-sensitive collating is that aspect that can be expressed by assigning a weight on each character in the code set.

Intrinsic Functions with Collating Sequence Sensitivity

The following intrinsic functions are dependent on the ordinal positions of characters. These intrinsic functions are not supported for the DBCS data type (for example, supported for single-byte characters, alphabetic or numeric). For an ASCII code page, these intrinsic functions are supported based on the collating sequence in effect. For a DBCS code page, the ordinal positions of single-byte characters are assumed to correspond to the hex representations of the single-byte characters. For example, the ordinal positions for 'A' is 66 (X'41' + 1) and the ordinal position for '*' is 43 (X'2A' + 1).

- CHAR

  Returns the character of the ordinal position given.

- MAX

  Returns the contents of the argument that contains the maximum value. Note that the arguments may be alphabetic or alphanumeric.

- ORD

  Returns the ordinal position of the given character.

- ORD-MAX

  Returns the highest ordinal position of the characters given. ORD-MAX with numeric arguments are supported independent of the code page in effect.

- ORD-MIN

  Returns the lowest ordinal position of the characters given. ORD-MIN with numeric arguments are supported independent of the code page in effect.

Any comparisons involving a group item will be handled based on the comparison of the byte-for-byte positions in hex.

Comments

Character strings that form comments may contain any characters, including DBCS characters. A single DBCS character may not be split (and continued) across multiple source lines.

Messages Enabled for NLS

The following messages are NLS enabled and appropriate message text and formats are used based on the locale setting:

- Compiler messages.
- Run-time messages.
- Compiler listing headers. This includes locale-sensitive date and time formats.
- Debugger user interface.

Cross-Reference Output Sequence

The cross-reference output is ordered in the collating sequence indicated by the locale setting.

Locale Sensitivity

This product is sensitive to the locale setting for the following features:

- Code page selection

The locale in effect determines the code set for both compilation of source programs, including non-numeric literal values, and their execution. That is, the code set used for compilation is based on the locale setting at compile time, and the code set used for application program execution is based on the locale setting at run time.

The EBCDIC code set is based on the current locale setting. Figure 110 on page 477 shows the locales and code sets supported with VisualAge COBOL under OS/2. Figure 111 on page 479 shows the locales and code sets supported with VisualAge COBOL under Windows.

If more than one EBCDIC code set is applicable for the current locale, and you want to use other than the default, then:

- Set the CHAR compiler option to EBCDIC; "CHAR" on page 165 discusses this option.
- Set the EBCDIC_CODEPAGE to establish the EBCDIC code set applicable; see page 138.

- Messages

  This applies to the message text, the date and time format, and order for XREF for the compiler listing output and to run-time message text. The compile-time locale is used for compiler output, the run-time setting for run-time output.

- Collating sequence

  Locale sensitivity for the collating sequence applies only when the collating sequence is NATIVE; the locale has no impact on the collating sequence if COLLSEQ(BIN) or COLLSEQ(EBCDIC) is in effect.

  The collating sequence for single-byte alphanumeric characters for the program collating sequence is based on the compile-time or run-time locale. If the PROGRAMMING COLLATING SEQUENCE clause is specified in the source program, the collating sequence is set at compile-time and is used regardless of the run-time locale. If the collating sequence is not set using this clause, but is set using the COLLSEQ compiler option, the run-time locale takes precedence.

  The collating sequence for SORT or MERGE statements is always based on the run-time locale.

  The run-time locale-based collating sequence is always applicable to DBCS data, independent of the COBOL source-level collating sequence specification (which applies to single-byte alphanumeric data), except for comparisons of literals. Comparisons of DBCS literals are based on the compile-time locale. Thus, DBCS literals should not be used in the source program within a statement with an implied relational condition between two DBCS literals (such as VALUE G *literal1* THRU G *literal2*) unless the intended run-time locale is the same as the compile-time locale.

  The compile-time and run-time locale settings are assumed to be the same for other uses of the collating sequence.

National Language Support Considerations

The following are *not* affected by the locale setting, as the ANSI COBOL Standard defines specific COBOL language elements for controlling these items:

- Decimal point and numeric separator
- Currency sign

Chapter 28. Pre-initializing the COBOL Run-Time Environment

Pre-initialization allows an application to initialize the COBOL run-time environment once, perform multiple executions using the environment, and then explicitly terminate the environment. Pre-initialization is used to invoke COBOL programs multiple times from a non-COBOL (for example, C or C++) environment.

Note: Pre-initialization is not supported under CICS.

The pre-initialization has two primary benefits:

- Keeps the COBOL environment ready for program calls

  Since the COBOL run unit is not terminated on return from first COBOL program in the run unit, the COBOL programs invoked from a non-COBOL environment can be invoked in its last-used state.

- Performance

  Creating and taking down the COBOL run-time environment repeatedly uses a great deal of overhead and can slow down your application's performance.

Use pre-initialization services for multilanguage applications where non-COBOL programs need to use a non-COBOL program in its last-used state. For example, a file may be opened on a first call to a COBOL program, and the invoking program expects subsequent calls to that program to find the file left open.

Use the interfaces described below to initialize and terminate a persistent COBOL run-time environment. Any DLL that contains a COBOL program used in a pre-initialized environment cannot be deleted until the pre-initialized environment is terminated.

If you plan to run your program in an OS/390 or VM environment, use the pre-initialization interfaces described in *IBM Language Environment for OS/390 & VM Programming Guide*.

Initialize Persistent COBOL Environment

Syntax

```
►►──call──Init_routine──(──function_code──,──routine──,──error_code───►
►──,──token──)───────────────────────────────────────────────────────►◄
```

call
  Invocation of *Init_routine*, using language elements appropriate to the language from which the call is being made.

Init_routine
  The name of the initialization routine: `_iwzCOBOLInit` or `IWZCOBOLINIT` for OS/2 and Windows (using OPTLINK linkage convention); `_IwzCOBOLInit` for Windows (using STDCALL linkage convention).

Terminate COBOL Environment

function_code (input) — A 4-byte binary number, passed by value
function_code can be:

1 The first COBOL program invoked following this function invocation is treated as a subprogram.

routine (input)
Address of the routine to be invoked if the run unit terminates. The token argument passed to this function will be passed on to the run unit termination exit routine. This routine, when invoked on the run unit termination, must not return to the invoker of the routine but rather do a `longjmp()` or `exit()`. This routine will be invoked with the SYSTEM linkage convention.

error_code (output) — A 4-byte binary number
error_code can be:

0 pre-initialization was successful.

1 pre-initialization failed.

token (input)
4 byte token to be passed on to the exit routine specified in the earlier argument when that routine is invoked on the run unit termination.

Terminate Pre-initialized COBOL Environment

Syntax

```
►►──call──Term_routine──(──function_code──,──error_code──)──────────►◄
```

call
Invocation of *Term_routine*, using language elements appropriate to the language from which the call is being made.

Term_routine
The name of the initialization routine: `_iwzCOBOLTerm` or `IWZCOBOLTERM` for OS/2 or Windows (using OPTLINK linkage convention); `_IwzCOBOLTerm` for Windows (using STDCALL linkage convention)

function_code (input), a 4-byte binary number, passed by value
function_code can be:

1 Clean up the pre-initialized COBOL run-time environment as if a COBOL STOP RUN statement had been performed; for example, all COBOL files are closed. However, the control returns to the caller of this service, not to the invoker of the COBOL main program; the routine named in the call to the pre-initialization routine is not invoked. (See Figure 114 on page 492. StopFun does not get called.)

error_code (output), a 4-byte binary number
error_code can be:

0 termination was successful.

1 termination failed.

The first COBOL program called following the invocation of the pre-inialization routine is treated as a subprogram. Thus, a GOBACK from this (initial) program *does not* trigger run-unit termination semantics (such as the closing of files). Note that the run unit termination (such as with STOP RUN) *does* free the pre-initialized COBOL environment prior to the invocation of the run unit exit routine.

If Not Active: If your program invokes the termination routine and the COBOL environment is not already active, the termination routine invocation has no effect on the execution and the control is returned to the invoker with an error code of 0.

Example of Pre-initializing the COBOL Environment

Figure 113 illustrates how the pre-initialized COBOL environment works. The example shows a C program initializing the COBOL environment, calling COBOL programs, then terminating the COBOL environment.

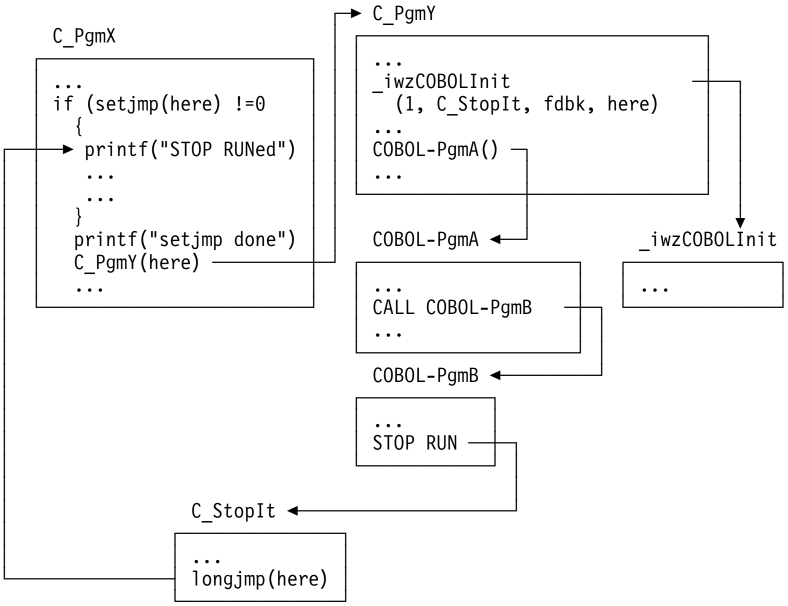


Figure 113. Illustration of Pre-Initialized COBOL Environment

The following example shows the use of COBOL pre-initialization. A C main program calls the COBOL program XIO several times. The first call to XIO opens the file, the second call writes one record, and so on. The final call closes the file. The C program then uses C-stream I/O to open and read the file. It assumes the use of VisualAge for C++.

Example of Pre-initializing

To test and run the program, enter the following commands from a command window:

```
cob2 -c xio.cbl
icc testinit.c xio.obj
testinit
```

The result is:

```
_iwzCOBOLinit got 0
xio entered with x=0000000000
xio entered with x=0000000001
xio entered with x=0000000002
xio entered with x=0000000003
xio entered with x=0000000004
xio entered with x=0000000099
StopArg=0
_iwzCOBOLTerm expects rc=0 and got rc=0
FILE1 contains ----
11111
22222
33333
---- end of FILE1
```

Note that in this example, the run unit was not terminated by a COBOL STOP RUN; it was terminated when the main program called `_iwzCOBOLTerm`.

The following C program is in the file `testinit.c`:

```
#ifdef _AIX
typedef int (*PFN)();
#define LINKAGE
#else
#include <os2.h>
#define LINKAGE _System
#endif

#include   <stdio.h>
#include   <setjmp.h>
```

Figure 114 (Part 1 of 2). Source Code for testinit.c

```
extern void _iwzCOBOLInit(int fcode, PFN StopFun, int *err_code, void *StopArg);
extern void _iwzCOBOLTerm(int fcode, int *err_code);
extern void LINKAGE XIO(long *k);

jmp_buf Jmpbuf;
long StopArg = 0;

int LINKAGE
StopFun(long *stoparg)
{
        printf("inside StopFun\n");
        *stoparg = 123;
        longjmp(Jmpbuf,1);
}

main()
{
        int rc;
        long k;
        FILE *s;
        int c;

        if (setjmp(Jmpbuf) ==0) {
                _iwzCOBOLInit(1, StopFun, &rc, &StopArg);
                printf( "_iwzCOBOLinit got %d\n",rc);
                for (k=0; k <= 4; k++) XIO(&k);
                k = 99; XIO(&k);
        }
        else printf("return after STOP RUN\n");
        printf("StopArg=%d\n", StopArg);
        _iwzCOBOLTerm(1, &rc);
        printf("_iwzCOBOLTerm expects rc=0 and got rc=%d\n",rc);
        printf("FILE1 contains ---- \n");
        s = fopen("FILE1", "r");
        if (s) {
                while (  (c = fgetc(s) ) != EOF ) putchar(c);
        }
        printf("---- end of FILE1\n");
}
```

Figure 114 (Part 2 of 2). Source Code for testinit.c. A C program that shows the use of pre-initialization.

Example of Pre-initializing

The following COBOL program is in the file `xio.cbl`:

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.     xio.


      ******************************************************************

       ENVIRONMENT    DIVISION.
       CONFIGURATION   SECTION.
       INPUT-OUTPUT    SECTION.
       FILE-CONTROL.
           SELECT file1 ASSIGN TO FILE1
             ORGANIZATION IS LINE SEQUENTIAL
             FILE STATUS IS file1-status.

       DATA           DIVISION.
       FILE SECTION.
       FD FILE1.
       01 file1-id pic x(5).

       WORKING-STORAGE SECTION.
       01 file1-status  pic xx    value is zero.

       LINKAGE SECTION.
      *
       01 x               PIC S9(8) COMP-5.

       PROCEDURE DIVISION using x.

           display "xio entered with x=" x
           if x = 0 then
              OPEN output FILE1
           end-if
```

Figure 115 (Part 1 of 2). Source Code for xio.cbl

```
           if x = 1  then
              MOVE ALL "1" to file1-id
              WRITE file1-id
           end-if
           if x = 2 then
              MOVE ALL "2" to file1-id
              WRITE file1-id
           end-if
           if x = 3 then
              MOVE ALL "3" to file1-id
              WRITE file1-id
           end-if
           if x = 99 then
              CLOSE file1
           end-if
           GOBACK.
```

Figure 115 (Part 2 of 2). Source Code for xio.cbl. A COBOL program that shows the use of pre-initialization.

Chapter 29. Productivity and Tuning Techniques

This chapter provides techniques to improve programmer productivity using built-in functions and services, and contains guidelines on performance optimization to help you make the most of your COBOL applications.

Simplifying Complex Coding and Other Programming Tasks

By using COBOL intrinsic functions and Language Environment callable services, you can avoid having to code a lot of arithmetic or other complex tasks.

Intrinsic Functions

COBOL provides various string- and number-manipulation capabilities that can help you simplify your coding. For more information, see “Numeric Intrinsic Functions” on page 40.

Date and Time Callable Services

With the date and time callable services, you can get the current local time and date in several formats, as well as perform date and time conversions. Two callable services, CEEQCEN and CEESCEN, provide a predictable way to handle 2-digit years, such as 91 for 1991 or 02 for 2002.

The following lists the date and time callable services available:

CEECBLDY Converts character date value to COBOL integer date format. Day one is 01 January 1601 and the value is incremented by one for each subsequent day. This service is similar to CEEDAYS, except that it provides an answer in COBOL integer date format, so that it is compatible with ANSI COBOL intrinsic functions. The returned value from this service should not be used with other date and time callable services.

CEEDATE Converts dates in the Lilian format back to character values.

CEEDATM Convert number of seconds to character timestamp.

CEEDAYS Convert character date values to the Lilian format. Day one is 15 October 1582 and the value is incremented by one for each subsequent day.

CEEDYWK Provides day of week calculation.

CEEGMT Gets current Greenwich Mean Time (date and time).

CEEGMTO Gets difference between Greenwich Mean Time and local time.

CEEISEC Converts binary year, month, day, hour, second, and millisecond to a number representing the number of seconds since 00:00:00 15 October 1582.

CEELOCT Gets current date and time.

CEEQCEN Queries the callable services century window.

CEESCEN Sets the callable services century window.

CEESECI Converts a number representing the number of seconds since 00:00:00 15 October 1582 to seven separate binary integers representing year, month, day, hour, minute, second, and millisecond.

CEESECS Converts character timestamps (a date and time) to the number of seconds since 00:00:00 15 October 1582.

CEEUTC Same as CEEGMT.

For details on individual date and time callable services, see Appendix E, “Date and Time Callable Services Reference” on page 558.

All of the above date and time callable services allow source code compatibility with COBOL for OS/390 & VM and COBOL for MVS & VM. There are, however, significant differences in the way conditions are handled. See “Condition Handling” on page 498 for details.

How to Invoke Date and Time Callable Services

To invoke a date and time callable service, use a CALL statement with the correct parameters for that particular service, for example:

```
CALL "CEEDATE" using argument, format, result, feedback-code.
```

You define the variables in the CALL statement in the DATA DIVISION of your program with the data definitions required by the particular function you are calling:

```
77  argument            pic  s9(9)  comp.
01  format.
    05 format-length    pic  s9(4)  comp.
    05 format-string    pic  x(80).
77  result              pic  x(80).
77  feedback-code       pic  x(12)  display.
```

In this example, the date and time callable service CEEDATE converts a number representing a Lilian date in the variable `argument` to a date in character format which is written to the variable `result`. The format of the conversion is controlled by the picture string contained in the variable `format`. Information about the success or failure of the call is returned in the variable `feedback-code`.

All of the date and time callable services allow the specification of a feedback code parameter which is optional. Specify OMITTED for this parameter if you do not want the date and time callable service to return information about the success or failure of the call. However, if omitting this parameter, the program will abend if the callable service does not complete successfully. See “Condition Handling” on page 498 for additional information.

When calling a date and time callable service and you specify OMITTED for the feedback code, the RETURN-CODE special register is set to 0 if the service is successful but is not altered if the service is unsuccessful. If the feedback code is not OMITTED, the RETURN-CODE special register is always set to 0 regardless of whether the service completed successfully.

For a description of the OMITTED phrase, see *IBM COBOL Language Reference*.

The date and time callable services must be invoked using the standard system linkage convention. This can be achieved by either compiling the program using the CALLINT(SYSTEM) compiler option (this is the default), or by using the >>CALLINTERFACE SYSTEM compiler-directing statement.

Note: The CALL statements used to invoke the date and time callable services must use a literal for the program name as opposed to an identifier. See *IBM COBOL Language Reference* for details on the CALL statement.

Condition Handling

There is a significant difference in the condition handling between VisualAge COBOL and IBM Language Environment on the host. VisualAge COBOL adheres to the native COBOL condition handling scheme and does not provide the level of support in Language Environment. If a feedback token is passed as an argument, it will simply be returned after the appropriate information has been filled in. The caller can then examine the contents and perform any actions, if necessary. The condition will not be signaled. If a date and time callable service is called with the OMITTED phrase as a substitute for the feedback code, the program will abend if the service does not complete successfully.

A feedback token contains feedback information in the form of a condition token. The condition token set by the service will be returned to the calling routine, indicating whether the service was completed successfully or not. VisualAge COBOL uses the same feedback token as Language Environment which is defined as follows:

```
01  FC.
    02  Condition-Token-Value.
    COPY  CEEIGZCT.
        03  Case-1-Condition-ID.
            04  Severity    PIC S9(4) COMP.
            04  Msg-No      PIC S9(4) COMP.
        03  Case-2-Condition-ID
                  REDEFINES Case-1-Condition-ID.
            04  Class-Code  PIC S9(4) COMP.
            04  Cause-Code  PIC S9(4) COMP.
        03  Case-Sev-Ctl    PIC X.
        03  Facility-ID     PIC XXX.
    02  I-S-Info            PIC S9(9) COMP.
```

The following describes what each field will contain and identifies any differences with IBM Language Environment on the host:

Severity This is the severity number with the following possible values:

- **0** Information only (or, if the entire token is zero, no information).
- **1** Warning - service completed, probably correctly.
- **2** Error detected - correction attempted; service completed, perhaps incorrectly.
- **3** Severe error - service not completed.
- **4** Critical error - service not completed.

Msg-No This is the associated message number. See Appendix F, “Run-Time Messages” on page 606 for additional information.

Case-Sev-Ctl This field will always contain the value 1.

Facility-ID This field will always contain the characters CEE.

I-S-Info This field will always contain the value 0.

Sample COPY files are provided which define the condition tokens. The file CEEIGZCT.CPY contains the definitions of the condition tokens when you are using native data types in your program. The file CEEIGZCT.EBC contains the definitions of the condition tokens when you are using host data types. In order to use the COPY file with the host data type support, you need to rename the file to CEEIGZCT.CPY. These files can be found in the SAMPLES\CEE directory.

The condition tokens contained in the files are equivalent to those provided with Language Environment, except that for native data types, the character representations are in ASCII instead of EBCDIC, and the bytes within binary fields are reversed.

The descriptions of the individual callable services include a listing of the possible symbolic feedback codes that might be returned in the feedback code output field specified on invocation of the service. In addition to these, the symbolic feedback code CEE0PD might be returned for any callable service. See message IWZ0813S for details.

All date and time callable services are based on the Gregorian calendar. Date variables associated with this calendar have architectural limits. These limits are:

Starting Lilian date The beginning of the Lilian date range is Friday 15 October 1582, the date of adoption of the Gregorian calendar. Lilian dates preceding this date are undefined. Therefore:

- Day zero is 00:00:00 14 October 1582
- Day one is 00:00:00 15 October 1582

All valid input dates must be after 00:00:00 15 October 1582.

End Lilian date The end Lilian date range is set to 31 December 9999. Lilian dates following this date are undefined. The reason for this limit is a 4-digit year.

Picture Character Terms and Strings

Picture character terms and strings are used to define the format of a date and/or time field used by several of the date and time callable services. A picture string is a template that indicates the format of the input of the data or the desired format of the output. Figure 116 and Figure 117 on page 501 define the supported picture character terms and string values.

Figure 116 (Page 1 of 2). Picture Character Terms Used in Picture Strings for Date and Time Services

Picture Terms	Explanations	Valid Values	Notes
`Y`	1-digit year	0-9	`Y` valid for output only. `YY` assumes range set by CEESCEN. `YYY`/`ZYY` used with `<JJJJ>`, `<CCCC>` and `<CCCCCCCC>`.
`YY`	2-digit year	00-99	
`YYY`	3-digit year	000-999	
`ZYY`	3-digit year within era	1-999	
`YYYY`	4-digit year	1582-9999	
`<JJJJ>`	Japanese era name in DBCS characters	*Heisei* (`X'95BD90AC'`) *Showa* (`X'8FBA9861'`) *Taisho* (`X'91E590B3'`) *Meiji* (`X'96BE8EA1'`)	Affects `YY` field: if `<JJJJ>` specified, `YY` means the year within Japanese era, for example, 1988 equals Showa 63. See example in Figure 117 on page 501.

`<CCCC>`
`<CCCCCCCC>`
	Republic of China (ROC) era name in DBCS characters	*MinKow*
		(`X'8D8196CD'`)
		ChuHwaMinKow
		(`X'8C839ADC8D8196CD'`)
			Affects `YY` field: if `<CCCC>` specified, `YY` means the year within ROC era, for example, 1988 equals Minkow 77. See example in Figure 117 on page 501.

`MM`	2-digit month	01-12	
`ZM`	1- or 2-digit month	1-12	For output, leading zero suppressed. For input, `ZM` treated as `MM`.
`RRRR` `RRRZ`	Roman numeral month	Iƀƀƀ-XIIƀ (Left justified)	For input, source string is folded to uppercase. For output, uppercase only. I=Jan, II=Feb, ..., XII=Dec.
`MMM`	3-char month, uppercase	JAN-DEC	For input, source string always folded to uppercase. For output, `M` generates uppercase and `m` generates lowercase. Output is padded with blanks (ƀ) (unless `Z` specified) or truncated to match the number of `M`s, up to 20.
`Mmm`	3-char month, mixed case	Jan-Dec	
`MMMM...M`	3-20 char mo., uppercase	JANUARYƀƀ-DECEMBERƀ	
`Mmmm...m`	3-20 char mo., mixed case	Januaryƀƀ-Decemberƀ	
`MMMMMMMMMZ`	trailing blanks suppressed	JANUARY-DECEMBER	
`Mmmmmmmmmz`	trailing blanks suppressed	January-December	
`DD`	2-digit day of month	01-31	For output, leading zero is always suppressed. For input, `ZD` treated as `DD`.
`ZD`	1- or 2-digit day of mo.	1-31	
`DDD`	Day of year (Julian day)	001-366	
`HH`	2-digit hour	00-23	
`ZH`	1- or 2-digit hour	0-23	For output, leading zero suppressed. For input, `ZH` treated as `HH`. If `AP` specified, valid values are 01-12.
`MI`	Minute	00-59	
`SS`	Second	00-59	

Figure 116 (Page 2 of 2). Picture Character Terms Used in Picture Strings for Date and Time Services

Picture Terms	Explanations	Valid Values	Notes

`9`
`99`
`999`
	Tenths of a second
	Hundredths of a second
	Thousandths of a second
		0-9
		00-99
		000-999
			No rounding.

`AP`
`ap`
`A.P.`
`a.p.`
	AM/PM indicator	AM or PM
		am or pm
		A.M. or P.M.
		a.m. or p.m.
			`AP` affects `HH`/`ZH` field. For input, source string always folded to uppercase. For output, `AP` generates uppercase and `ap` generates lowercase.

`W`
`WWW`
`Www`
`WWW...W`
`Www...w`
`WWWWWWWWWZ`
`Wwwwwwwwwz`
	1-char day-of-week
	3-char day, uppercase
	3-char day, mixed case
	3-20 char day, uppercase
	3-20 char day, mixed case
	Trailing blanks suppressed
	Trailing blanks suppressed
		S, M, T, W, T, F, S
		SUN-SAT
		Sun-Sat
		SUNDAYƀƀƀ-SATURDAYƀ
		Sundayƀƀƀ-Saturdayƀ
		SUNDAY-SATURDAY
		Sunday-Saturday
			For input, `W`s are ignored. For output, `W` generates uppercase and `w` generates lowercase. Output padded with blanks (unless `Z` specified) or truncated to match the number of `W`s, up to 20.

All others	Delimiters	`X'01'`-`X'FF'` (`X'00'` is reserved for “internal” use by the date and time callable services)	For input, treated as delimiters between the month, day, year, hour, minute, second, and fraction of a second. For output, copied exactly as is to the target string.

Note:

Blank characters are indicated by the symbol ƀ.

Figure 117 (Page 1 of 2). Examples of Picture Strings Recognized by Date and Time Services

Picture Strings	Examples	Notes
`YYMMDD`	880516	
`YYYYMMDD`	19880516	
`YYYY-MM-DD`	1988-05-16	1988-5-16 would also be valid input.
`<JJJJ> YY.MM.DD`	*Showa* 63.05.16	*Showa* is a Japanese Era name. *Showa* 63 equals 1988.
`<CCCC> YY.MM.DD`	*MinKow* 77.05.16	*MinKow* is an ROC Era name. *MinKow* 77 equals 1988.
`MMDDYY`	050688	
`MM/DD/YY`	05/06/88	
`ZM/ZD/YY`	5/6/88	
`MM/DD/YYYY`	05/06/1988	
`MM/DD/Y`	05/06/8	1-digit year format (`Y`) valid for output only
`DD.MM.YY`	09.06.88	
`DD-RRRR-YY`	09-VI -88	
`DD MMM YY`	09 JUN 88	
`DD Mmmmmmmmmm YY`	09 June 88	
`ZD Mmmmmmmmmz YY`	9 June 88	`Z` suppresses zeros/blanks
`Mmmmmmmmmz ZD, YYYY`	June 9, 1988	
`ZDMMMMMMMMzYY`	9JUNE88	

Figure 117 (Page 2 of 2). Examples of Picture Strings Recognized by Date and Time Services

Picture Strings	Examples	Notes
YY.DDD YYDDD YYYY/DDD	88.137 88137 1988/137	Julian date
YYMMDDHHMISS YYYYMMDDHHMISS YYYY-MM-DD HH:MI:SS.999 WWW, ZM/ZD/YY HH:MI AP Wwwwwwwwwz, DD Mmm YYYY, ZH:MI AP	880516204229 19880516204229 1988-05-16 20:42:29.046 MON, 5/16/88 08:42 PM Monday, 16 May 1988, 8:42 PM	Timestamp—valid only for CEESECS and CEEDATM. If used with CEEDATE, time positions are filled with zeros. If used with CEEDAYS, HH, MI, SS, and 999 fields are ignored.

Note: Lowercase characters must be used only for alphabetic picture terms.

Figure 118. Japanese Eras Used by Date/Time Services when <JJJJ> Specified

First Date of Japanese Era	Era Name	Era Name in Japanese DBCS Code	Valid Year Values
1868-09-08	Meiji	X'96BE8EA1'	01-45
1912-07-30	Taisho	X'91E590B3'	01-15
1926-12-25	Showa	X'8FBA9861'	01-64
1989-01-08	Heisei	X'95BD90AC'	01-999 (01 = 1989)

Figure 119. Republic of China Eras Used by Date/Time Services when <CCCC> or <CCCCCCCC> Specified

First Date of ROC Era	Era Name	Era Name in Chinese DBCS Code	Valid Year Values
1912-01-01	MinKow	X'96BE8EA1'	01-999 (77 = 1988)
	ChuHwaMinKow	X'8C839ADC8D8196CD'	

Performing Calculations

The date and time callable services store dates as fullword binary integers and timestamps as doubleword floating-point values, formats that permit you to perform arithmetic calculations on date and time values in a simple and efficient manner. This eliminates the need to write special subroutines that use services outside of the language library for your application in order to perform these calculations. The following is a generic example of how you can use date and time callable services to convert a date to a different format and perform a simple calculation on the formatted date:

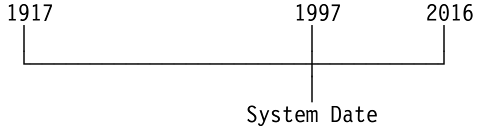
```
CALL CEEDAYS USING dateof_hire, 'YYMMDD', doh_lilian, fc.
CALL CEELOCT USING todayLilian, today_seconds, today_Gregorian, fc.
COMPUTE servicedays = today_Lilian - doh_Lilian.
COMPUTE serviceyears = service_days / 365.25.
```

In the example, you want to calculate the number of years of service for an employee in your organization, and are using the original date of hire in the format YYMMDD to make the calculations. Use the CEEDAYS (Convert Date to Lilian Format) service to convert these dates to a Lilian format.

The CEELOCT (Get Current Local Time) service is called next to get the current local time. `doh_Lilian` is then subtracted from `today_Lilian` (the number of days from the beginning of the Gregorian calendar to the current local time) to calculate the employee's total number of days of employment. The final calculation divides that number by 365.25 to get the number of service years.

The Century Window Concept

To process 2-digit years in the year 2000 and beyond, the date and time callable services employ a sliding scheme by which all 2-digit years are assumed to lie within a 100-year interval starting 80 years before the current system date:



One hundred years, in 1997 spanning from 1917 to 2016, is the default century window for the date and time callable services. For example, in 1997 years 17 through 99 are recognized as 1917-1999, and years 00 through 16 are recognized as 2000-2016. In 1998, years 18 through 99 are recognized as 1918-1999, and years 00 through 17 are recognized as 2000-2017.

By year 2080, all 2-digit years would be recognized as 20xx. In 2081, 00 would be recognized as year 2100.

Some applications may need to set up a different 100-year interval. For example, banks often deal with 30-year bonds, which could be due 01/31/20. The CEESCEN callable service (see Appendix E, “Date and Time Callable Services Reference” on page 558) allows you to change the century window.

A companion service, CEEQCEN, queries the current century window. This allows a subroutine, for example, to use a different interval for date processing than the parent routine. Before returning, the subroutine would reset the interval back to the way it was previously, as shown in Figure 120 on page 504.

```
.
.
.
       WORKING-STORAGE SECTION
       77 OLDCEN PIC S9(9) COMP.
       77 TEMPCEN PIC S9(9) COMP.
       77 QCENFC PIC X(12).

       77 SCENFC1 PIC X(12).
       77 SCENFC2 PIC X(12).

       PROCEDURE DIVISION.

      ** Call CEEQCEN to retrieve and save current century window
       CALL "CEEQCEN" USING OLDCEN , QCENFC.

      ** Call CEESCEN to temporarily change century window to 30
       MOVE 30 TO TEMPCEN.
       CALL "CEESCEN" USING TEMPCEN , SCENFC1.

      ** Perform date processing with 2-digit years...
.
.
.
      ** Call CEESCEN again to reset century window
       CALL "CEESCEN" USING OLDCEN , SCENFC2.

       GOBACK.
```

Figure 120. Example of Querying and Changing the Century Window

Example Using Date and Time Callable Services

Many callable services offer the COBOL programmer entirely new function that would require extensive coding using previous versions of COBOL. Two such services are CEEDAYS and CEEDATE, which you can use effectively when you want to format dates for output.

Figure 121 on page 505 shows a sample COBOL program that uses date and time callable services to format and display a date from the results of a COBOL ACCEPT statement.

```
CBL QUOTE
       ID DIVISION.
       PROGRAM-ID. HOHOHO.
      ************************************************************
      * FUNCTION:  DISPLAY TODAY'S DATE IN THE FOLLOWING FORMAT: *
      *           WWWWWWWWW, MMMMMMMM DD, YYYY                  *
      *                                                          *
      *           For example:  MONDAY, MARCH 10, 1997          *
      *                                                          *
      ************************************************************
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.

       01   CHRDATE.
           05 CHRDATE-LENGTH     PIC S9(4) COMP VALUE 10.
           05 CHRDATE-STRING     PIC X(10).
       01   PICSTR.
           05 PICSTR-LENGTH      PIC  S9(4) COMP.
           05 PICSTR-STRING      PIC  X(80).

       77   LILIAN PIC           S9(9) COMP.
       77   FORMATTED-DATE       PIC X(80).

       PROCEDURE DIVISION.
      ****************************************************************
      *    USE  DATE/TIME CALLABLE SERVICES TO PRINT OUT             *
      *    TODAY'S DATE FROM COBOL ACCEPT STATEMENT.                 *
      ****************************************************************
           ACCEPT CHRDATE-STRING FROM DATE.

           MOVE "YYMMDD" TO PICSTR-STRING.
           MOVE 6 TO PICSTR-LENGTH.
           CALL "CEEDAYS" USING CHRDATE , PICSTR , LILIAN , OMITTED.

           MOVE " WWWWWWWWWZ, MMMMMMMMMZ DD, YYYY " TO PICSTR-STRING.
           MOVE 50 TO PICSTR-LENGTH.
           CALL "CEEDATE" USING LILIAN , PICSTR , FORMATTED-DATE ,
                 OMITTED.

           DISPLAY "******************************".
           DISPLAY FORMATTED-DATE.
           DISPLAY "******************************".

           STOP RUN.
```

Figure 121. Example with Date and Time Callable Services

Optimization

Optimization

The OPTIMIZE Compiler Option

This section discusses the benefits of the OPTIMIZE compiler option as well as other compiler features affecting optimization.

The COBOL optimizer is activated when you use the OPTIMIZE compiler option. The purpose of the OPTIMIZE compiler option is to do the following:

- Eliminate unnecessary transfers of control or simplify inefficient branches, including those generated by the compiler that are not evident from looking at the source program.
- Simplify the compiled code for a CALL statement to a contained (nested) program. Where possible, the optimizer places the statements inline, eliminating the need for linkage code. This optimization, known as *procedure integration*, is further discussed in “Contained Program Procedure Integration.” If procedure integration cannot be done, the optimizer uses the simplest linkage possible (perhaps as few as two instructions) to get to and from the called program.
- Eliminate duplicate computations (such as subscript computations and repeated statements) that have no effect on the results of the program.
- Eliminate constant computations by performing them when the program is compiled.
- Eliminate constant conditional expressions.
- Aggregate moves of contiguous items (such as those that often occur with the use of MOVE CORRESPONDING) into a single move. Both the source and target must be contiguous for the moves to be aggregated.
- The FULL suboption requests that the compiler discard any unreferenced data items from the DATA DIVISION, and suppress generation of code to initialize these data items to their VALUE clauses.

To see how the optimizer works on your program, compile it with and without the OPTIMIZE option and then compare the generated code. (Use the LIST compiler option to request the assembler language listing of the generated code.)

For unit testing your programs, you might find it easier to debug code that has not been optimized. But when the program is ready for final test, specify OPTIMIZE, so that the tested code and the production code are identical. You might also want to use the option during development, if a program is used frequently without recompilation. However, the overhead for OPTIMIZE might outweigh its benefits if you re-compile frequently, unless you are using the assembler language expansion (LIST option) to fine tune your program.

Contained Program Procedure Integration

Contained program procedure integration is the process where a CALL to a contained program is replaced by the program code. The advantage here is that the resulting program runs faster without the overhead of CALL linkage and with more linear control flow.

Program Size: If the contained programs are invoked by several CALL statements and replace each such CALL statement, the program may become larger. The optimizer limits this increase to no more than 50 percent, after which it no longer uses procedure integration. The optimizer then chooses the next best optimization for the CALL statement; the linkage overhead can be as few as two instructions.

Unreachable Code Elimination: As a result of procedure integration, one contained program might be repeated several times. As further optimization proceeds on each copy of the program, portions might be found to be unreachable, depending on the context into which the code was copied.

Other Compiler Features that Affect Optimization

Another compiler feature that might have a significant influence on the effects of the optimizer option is the USE FOR DEBUGGING ON ALL PROCEDURES statement.

The ON ALL PROCEDURES option of the USE FOR DEBUGGING statement generates extra code at each transfer to every procedure name. It can be very useful for debugging, but can make the program significantly larger as well as substantially inhibit optimization.

Additionally, compiler options can also have an effect (see “Compiler Options” for details).

Compiler Options

You might have a customized system that requires certain options for optimum performance. Check with your systems programmer to ensure that installed options are not required before changing defaults. You can see what your system defaults are by issuing ERRMSG. For instructions on issuing ERRMSG, see “Generating a List of All Compiler Error Messages” on page 151.

The tuning methods and performance information discussed here are intended to help you select from various COBOL options for compiling your programs.

Important

Make sure that COBOL serves your needs. Please confer with system programmers on how you should tune your COBOL programs. Doing so will ensure that the options you choose are appropriate for programs being developed at your site.

A brief description of each item is followed by performance advantages and disadvantages, reference information, and usage notes where applicable. Refer to specified pages for additional information.

DYNAM The DYNAM compiler option dynamically loads subprograms invoked through the CALL statement at run time.

Performance advantages

Using DYNAM means easier subprogram maintenance because the application will not have to be link-edited again if the subprogram is changed.

When using the DYNAM option, you can free virtual storage that is no longer needed by issuing the CANCEL statement.

Performance disadvantages

You pay a slight performance penalty using DYNAM because the call must go through a Language Environment routine.

Reference information

For a description of the DYNAM option, see “DYNAM” on page 171.

OPTIMIZE Use the OPTIMIZE compiler option to ensure your code is optimized for better performance.

Performance advantages

Generally results in more efficient run-time code.

Performance disadvantages

OPTIMIZE requires more processing time for compiles than NOOPTIMIZE.

Reference information

For further description of OPTIMIZE, see “The OPTIMIZE Compiler Option” on page 506. See “OPTIMIZE” on page 185 for the OPTIMIZE syntax.

Usage notes

NOOPTIMIZE is generally used during program development when frequent compiles are necessary, and it also allows for easier debugging. For production runs, however, using OPTIMIZE is recommended.

SSRANGE The SSRANGE option verifies that all table references and reference modification expressions are in proper bounds.

Performance advantages

No performance advantages.

Performance disadvantages

SSRANGE generates additional code for verifying table references.

Reference information

For SSRANGE syntax, see “SSRANGE” on page 193.

Usage notes

In general, if you only need to verify the table references a few times in the application instead of at every reference, coding your own checks may be faster than using the SSRANGE compiler option. SSRANGE can be turned off at run time with the CHECK(OFF) run-time option. For performance-sensitive applications, using the NOSSRANGE compiler option is recommended.

TEST The TEST option produces object code that can take full advantage of the Interactive Debugger.

Performance advantages
No performance advantages.

Performance disadvantages
The TEST option causes a significant increase in the size of the object file because debugging information is added to the object file. When linking the program, the linker can be directed to exclude the debugging information, resulting in approximately the same size executable as would be created if the modules were compiled with NOTEST. However, if the debugging information is included in the executable, a slight performance degradation might occur because the increased size of the executable will take longer to load and might cause increased paging activity.

Reference information
For TEST syntax, see “TEST” on page 194.

Usage notes
TEST forces the NOOPTIMIZE compiler option into effect. For production runs, using NOTEST is recommended.

TRUNC This compiler option creates code that will truncate the receiving fields of arithmetic operations.

Performance advantages
TRUNC(OPT) does not generate extra code and generally improves performance.

Performance disadvantages
Both TRUNC(BIN) and TRUNC(STD) generate extra code whenever a BINARY data item is changed. TRUNC(BIN) is usually the slowest of these options.

Reference information
For syntax and a more detailed explanation of the TRUNC option, see “TRUNC” on page 195.

Usage notes
TRUNC(STD) conforms to the COBOL 85 Standard, whereas TRUNC(BIN) and TRUNC(OPT) do not. When using TRUNC(OPT), the compiler assumes that the data conforms to the PICTURE and USAGE specifications. TRUNC(OPT) is recommended where possible.

Chapter 30. The "Year 2000" Problem

This chapter provides some information on date processing problems associated with the year 2000, and recommends some solutions that you can adopt to help resolve them.

Date Processing Problems

Many applications use two digits rather than four to represent the year in date fields, and assume that these values represent years from 1900 to 1999. This compact date format works well for the 1900s, but it does not work for the year 2000 and beyond because these applications interpret “00” as 1900 rather than 2000, producing incorrect results.

This chapter outlines a number of approaches you can adopt to resolve problems of this nature, and points to facilities available in the COBOL compiler and in the date and time callable services that can assist you.

For more information on the new features of the COBOL language that can help resolve date-related problems, see Chapter 31, “Using the Millennium Language Extensions” on page 520.

For more information about Year 2000 issues, and IBM software products that can help you identify and resolve their related problems, visit the website at: http://www.software.ibm.com/year2000.

Year 2000 Solutions

There are several solutions to the Year 2000 problem. Many of these solutions refer to a “century window”. A century window is a 100-year interval, such as 1950–2049, within which any 2-digit year is unique. For example, with a century window of 1930–2029, 2-digit years would be interpreted as follows:

Year values from 00 through 29 are interpreted as years 2000–2029
Year values from 30 through 99 are interpreted as years 1930–1999

The solutions outlined in this chapter are:

- The Full Field Expansion Solution (the long-term approach)
- The Internal Bridging Solution
- The Century Window Solution
- The Mixed Field Expansion and Century Window Solution
- The Century Encoding/Compression Solution
- The Integer Format Date Solution

Each of these is discussed in more detail below.

The Full Field Expansion Solution

To take your programs through to the year 9999, you *must* eventually rewrite applications and rebuild databases and files to use 4-digit year fields rather than 2-digit year fields.

The field expansion method is a long-term solution and is the recommended approach to addressing the Year 2000 problem. To achieve this field expansion, you need to develop a program to read in the old data, convert it, and write it back into a copy of the original file or data base that has been expanded to hold the 4-digit year data. All new data would then go into the new file or database. All of your application programs that use those files and databases need to be changed to act on the new 4-digit year date fields instead of the 2-digit year fields.

Your conversion program needs to use a century window when expanding 2-digit years to 4 digits, to ensure that the output dates are correct.

There are several ways to use VisualAge COBOL to help convert your databases or files from 2-digit year dates to 4-digit year dates, with a century window being taken into account:

DATEPROC processing

Use the DATEPROC compiler option and the DATE FORMAT clause to define date fields, and use MOVE statements to expand the dates based on the century window specified by the YEARWINDOW compiler option. For example:

```
05  Date-Short      Pic x(6) Date Format yyxxxx.
⋮
05  Date-Long       Pic x(8) Date Format yyyyxxxx.
⋮
Move Date-Short to Date-Long.
```

For more information, see Chapter 31, “Using the Millennium Language Extensions” on page 520.

COBOL coding

You can move a 2-digit year date field to an expanded receiving field, and “hard code” a century component as part of the move. For example:

```
05  Date-Short      Pic x(6) Date Format yyxxxx.
⋮
05  Date-Long       Pic x(8) Date Format yyyyxxxx.
⋮
String "19" Date-Short Delimited by Size
       Into Date-Long.
```

The hard-coded century component assumes a century window of 1900–1999 in this example, but you can add code to recognize different date ranges and assign a different century based on the value of Date-Short. For example, the following code expands the date based on a century window of 1930–2029:

```
        05  Date-Short      Pic x(6) Date Format yyxxxx.
        ⋮
        05  Date-Long       Pic x(8) Date Format yyyyxxxx.
        ⋮
    77  Century             Pic x(2).
        ⋮
        If Date-Short Less than "300000" Then
           Move "20" to Century
        Else
           Move "19" to Century
        End-If.
        String Century Date-Short Delimited by Size
               Into Date-Long.
```

Intrinsic functions

Three intrinsic functions are available to expand 2-digit year dates into 4-digit year dates, with the window being specified as an argument to the function. The functions are:

DATE-TO-YYYYMMDD
Expand a Gregorian date with a 2-digit year to the same date with a 4-digit year.

DAY-TO-YYYYDDD
Expand a Julian date with a 2-digit year to the same date with a 4-digit year.

YEAR-TO-YYYY
Expand a 2-digit year to a 4-digit year.

With these functions, you specify the century window to be used when expanding the year. For full details and syntax of these functions, see *IBM COBOL Language Reference*.

Callable services

The date and time callable services can help you manipulate and convert dates. Some of these services can accept a date with a 2-digit year as input, and in this case, the callable services will apply their own century window. The following services either affect or can be affected by this century window:

CEECBLDY	Convert a date to a COBOL integer number of days.
CEEDAYS	Convert a date to a Lilian integer number of days.
CEEQCEN	Query the callable services century window.
CEESCEN	Change the callable services century window.
CEESECS	Convert a date and time stamp into a number of Lilian seconds.

For full details on these and other callable services, see Appendix E, “Date and Time Callable Services Reference” on page 558.

For additional information about the century window feature of the date and time callable services, see “The Century Window Concept” on page 503.

Advantages:

- The code changes are straightforward.
- Minimum testing is required and possibly no need for simulation of future dates on dedicated machines.
- Faster resulting code.
- The issue is addressed once and for all.
- Maintenance will become cheaper.

Disadvantages:

- Databases and files must be changed.

The Internal Bridging Solution

This solution involves keeping the dates in your files and databases as 2-digit year dates, and expanding them into other data items in your program.

In your application progams, you need to add some data items to hold the 4-digit year dates, and some processing logic to expand and contract the date fields. The resultant program will be structured like this:

1. Read the input files with 2-digit year dates.
2. Declare “shadow” data items that contain 4-digit year dates, and expand the 2-digit year fields into these work fields.
3. Use the 4-digit year dates for all date processing in the program.
4. Copy (window) the 4-digit year date fields back to 2-digit format for the output process.
5. Write the 2-digit year dates to the output files.

There are several ways to use VisualAge COBOL to achieve the field expansion and windowing needed for this solution.

For date field expansion:

- Use the DATEPROC compiler option and the DATE FORMAT clause to define the dates in the input records as windowed date fields, and the work fields as expanded date fields. Perform expanded MOVEs or stores using MOVE or COMPUTE statements.
- Use the intrinsic functions DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD, and YEAR-TO-YYYY to copy and expand date fields from the input records to work fields.
- Use the date and time callable services CEEDAYS and CEEDATE.

For date windowing:

- Simply MOVE the last 2 digits of the year back to the 2-digit year date fields. You should also add some code to check that the date is still within the century window, and take some error action if it is not. For example, if the 4-digit year field contains 2010 and the century window is 1905–2004, the date is outside the century window, and to simply store the last 2 digits would be incorrect.

- With the DATEPROC compiler option and the DATE FORMAT clause, copy the expanded date fields back to windowed date fields. If you use a COMPUTE statement to do this, you can use the ON SIZE ERROR phrase to ensure that the date remains within the century window, or to take some action if it doesn't. For details, see "ON SIZE ERROR Phrase" on page 530.

Advantages:

- Databases and files need not be changed.
- The code changes are straightforward.
- Good interim step towards a full field expansion solution.
- Faster resulting code.

Disadvantages:

- Some risk of data corruption.
- Short- to medium-term solution only.

The Century Window Solution

The century window solution allows 2-digit years to be interpreted in a 100-year window (because each 2-digit number can only occur once in any 100-year period).

There are several ways to use VisualAge COBOL to help you achieve this:

- Use the DATEPROC compiler option and the DATE FORMAT clause to define date fields. This provides an automated windowing capability using the century window defined by the YEARWINDOW compiler option.

  For more information, see Chapter 31, "Using the Millennium Language Extensions" on page 520.

- Use the intrinsic functions DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD, and YEAR-TO-YYYY to interpret date fields based on a century window. The century window is specified as an argument to the intrinsic function. For example:

  ```
  If Function YEAR-TO-YYYY (Current-Year, 48) Greater Than
     Function YEAR-TO-YYYY (Due-Year, 48) Then
     Display "Due date has passed."
  End-If.
  ```

  In this example, the century window begins 48 years prior to the year at the time the program is being run. If the program is running in 1998, then the century window is 1950–2049. This would allow a Current-Year value of 00 to be "greater" than a Due-Year value of 99.

- Insert IF statements around the references to date fields in your program, to determine how to apply a century component. For example, the following code implements a century window of 1940–2039:

```
If YY-1 less than "40" Then
   Move "20" to CC-1
Else
   Move "19" to CC-1
End-If.
```

- Use the date and time callable services to manipulate date fields using a century window defined by the CEESCEN service.

You cannot use the century window forever because a 2-digit year can only be unique in a given 100-year period. Over time you will need more than 100 years for your data window—in fact, many companies need more than 100 years now. For example, the century window cannot solve the problem of trying to figure out how old a customer is if the customer is older than 100 years and you only have 2-digit year dates. Another example is sorting. All of the records that you want to sort by date must have 4-digit year dates. For these issues and others you need to adopt The Full Field Expansion Solution.

Advantages:

- No database or file changes.

Disadvantages: The following disadvantages apply to the Century Window solution regardless of which method you use to implement it:

- Performance will be slower due to increased logic.
- More testing is required to validate changes, and simulation of future dates on dedicated machines is essential.
- Very difficult to manage across applications.
- The problem is not permanently solved and it will become necessary to change date programs and databases to use 4-digit years eventually.

In addition, if you do not use the DATEPROC and DATE FORMAT method, the following disadvantages apply:

- Risk of performing incorrect translations.
- Code changes are more error-prone and require more expertise.
- Increased maintenance costs.

The Mixed Field Expansion and Century Window Solution

You don't have to convert all of your files and databases at one time. Where a data base is shared by many applications, it might be more convenient to keep any dates that it contains in 2-digit year form. But where a file is used by a limited number of programs, it is best to eliminate the 2-digit year constraint as soon as possible.

For those dates that are still in 2-digit year form, you can use internal bridging or century windowing, both of which are described in detail in “The Internal Bridging Solution” on page 513, and “The Century Window Solution” on page 514, respectively.

You change the data descriptions for dates that you have expanded to 4-digit year form, and then use any of the techniques described in “The Full Field Expansion Solution” for processing them.

The DATEPROC compiler option is a particularly convenient way of implementing this solution, since it directly supports the use of both expanded and windowed date fields within a single statement.

The mixed solution has the advantages and disadvantages of the individual techniques that are discussed in these sections. In addition, the mixed solution has the advantage that files and databases can be changed as convenient, and kept unmodified otherwise.

The Century Encoding/Compression Solution

The century encoding/compression solution involves encoding/compressing numbers greater than 99 into existing 2-byte date fields. (For example, using hexadecimal rather than decimal digits.) This means rewriting applications to correctly interpret encoded/compressed values in the database.

This solution is the least desirable way to address the Year 2000 problem.

Advantages:

- Uses existing 2-byte date fields.

Disadvantages:

- Performance will be slower due to increased logic.
- More testing is required to validate changes and simulation of future dates on dedicated machines is essential.
- Very difficult to manage across applications.
- Code changes are more error-prone and require more expertise.
- Increased maintenance costs.
- The problem is not permanently solved and it will become necessary to change date programs and databases to use 4-digit years eventually.
- Cannot be read in dumps or listings.
- Must be translated whenever externalized.
- Risk of performing incorrect translations.

The Integer Format Date Solution

Integer dates specify a number of days from some point in the past.

Integer dates are provided by COBOL intrinsic functions and by the date and time callable services.

This solution is *not* the recommended way to address the Year 2000 problem. Instead, use the The Full Field Expansion Solution described on page 511.

Advantages:

- Uses only 4 bytes to store a date.

Disadvantages:

- Performance will be slower due to increased logic.
- More testing is required to validate changes and simulation of future dates on dedicated machines is essential.
- Very difficult to manage across applications.
- Code changes are more error-prone and require more expertise.
- Increased maintenance costs.
- The problem is not permanently solved and it will become necessary to change date programs and databases to use 4-digit years eventually.
- Cannot be read in dumps or listings.
- Must be translated whenever externalized.
- Risk of performing incorrect translations.
- There are too many different integer format starting dates:

  – CICS, OS/390, and DB2 start with January 1, 1900
  – PL/I does not support integer date values
  – C starts with January 1, 1970
  – COBOL starts with January 1, 1601
  – Date and time callable services start with October 15, 1582 (Lilian integer dates)

  There will be no problems with integer dates if conversion to and from integer is done using the same method in the same program. There will only be problems if the integer values are stored or passed between programs. These problems could still be avoided by:

  – Not using the value returned by CEECBLDY as input to other date and time callable services; CEECBLDY returns an ANSI COBOL integer date that can be used with COBOL intrinsic functions.

  – Only using date and time callable services, or only COBOL intrinsic functions, for getting and manipulating 4-digit year dates.

  Note: A program can be compiled on the host using the INTDATE(LILIAN) compiler option to provide compatibility between the Lilian date returned by CEEDAYS and COBOL intrinsic functions. However, VisualAge COBOL does not support the INTDATE compiler option and such a program will therefore not produce correct results on the workstation or PC.

Performance Considerations

Any implementation of a solution to the year 2000 problem will have some impact on the performance of your application. This section discusses some of the performance aspects that you should consider.

Performance Comparison

The following implementation methods are listed in order of least performance impact to most performance impact.

Full field expansion
The best performance can be obtained by expanding all of the dates in your files as a one-time task, and thereafter using the 4-digit year date fields in all processing.

Mixed field expansion and DATEPROC windowing
If the dates in your files have not yet been expanded, the best performance can be obtained by expanding the date fields as they are read from the files, and using these expanded dates in the main processing body of the program. In this way, the expansion process is only performed once for each date field.

Mixed field expansion and manual windowing
You can expand your input date fields manually, using combinations of COBOL coding, intrinsic functions, and callable services to apply the century window. This has more performance impact than DATEPROC windowing, even though the expansion process is still only performed once for each date field.

DATEPROC windowing
The millennium language extensions provide a windowing method that is designed to be efficient. However, the action of viewing a windowed date field for a COBOL IF or MOVE statement still imposes some processor overhead.

Manual windowing
Date windowing using COBOL IF statements adds a level of complexity to the program, and adds some processor overhead because of the extra COBOL statements. Typically the overhead of an IF statement of this form is more than the overhead of the automatic DATEPROC windowing process.

How to Get 4-digit Year Dates

Many COBOL programs need to obtain the date at the time of execution, to use as "Date-Of-..." fields in output files or reports, or to compare against other dates read from input files. COBOL provides a number of methods of obtaining the current date with a 4-digit year. The simplest of these are:

The intrinsic function CURRENT-DATE
Retrieves the date in Gregorian form, and can also retrieve the current time and the offset from Greenwich Mean Time.

ACCEPT *identifier* FROM DATE YYYYMMDD
Retrieves the date in Gregorian form.

How to Get 4-digit Year Dates

| ACCEPT *identifier* FROM DAY YYYYDDD
| Retrieves the date in Julian form.

| The CEELOCT callable service
| Retrieves the date in three different forms, including Gregorian with a 4-digit
| year.

Chapter 31. Using the Millennium Language Extensions

This chapter provides information on the millennium language extensions that have been incorporated into the IBM COBOL language to assist with Year 2000 processing.

Description

The term “Millennium Language Extensions” refers collectively to the features of VisualAge COBOL that are activated by the DATEPROC compiler option to help with Year 2000 date logic problems.

The DATEPROC compiler option enables special date-oriented processing of identified date fields, and the YEARWINDOW compiler option specifies the 100-year window (the century window) to be used for the interpretation of 2-digit windowed years. For a description of the DATEPROC compiler option, see “DATEPROC” on page 170. For a description of the YEARWINDOW compiler option, see “YEARWINDOW” on page 201.

The extensions, when enabled, include:

- The DATE FORMAT clause. This is added to items in the Data Division to identify date fields, and to specify the location of the year component within the date.
- The reinterpretation of the function return value as a date field, for the following intrinsic functions:

  DATE-OF-INTEGER
  DATE-TO-YYYYMMDD
  DAY-OF-INTEGER
  DAY-TO-YYYYDDD
  YEAR-TO-YYYY
- The reinterpretation as a date field of the conceptual data items DATE, DATE YYYYMMDD, DAY, and DAY YYYYDDD in the following forms of the ACCEPT statement:

  ACCEPT *identifier* FROM DATE
  ACCEPT *identifier* FROM DATE YYYYMMDD
  ACCEPT *identifier* FROM DAY
  ACCEPT *identifier* FROM DAY YYYYDDD
- The intrinsic functions UNDATE and DATEVAL, used for selective reinterpretation of date fields and non-dates.
- The intrinsic function YEARWINDOW, which retrieves the starting year of the century window set by the YEARWINDOW compiler option.

This chapter describes how you can use these new facilities to help solve date logic problems in your COBOL programs.

Getting Started

With the millennium language extensions, you can make simple changes to your COBOL programs to define date fields, and the compiler recognizes and acts on those dates using a century window to ensure consistency.

A century window is a 100-year interval, such as 1950–2049, within which any 2-digit year is unique. For windowed date fields, the century window start date is specified by the YEARWINDOW compiler option. When the DATEPROC option is in effect, the compiler applies this window to 2-digit year, or windowed, date fields in the program. For example, with a century window of 1930–2029, COBOL interprets 2-digit years as:

> Year values from 00 through 29 are interpreted as years 2000–2029
> Year values from 30 through 99 are interpreted as years 1930–1999

To implement date windowing using COBOL date processing, you define the century window with the YEARWINDOW compiler option, and identify the date fields in your program with DATE FORMAT clauses. The compiler then automatically applies the century window to operations on those dates. It is often possible to implement a solution in which the windowing process is fully automatic; that is, you simply identify the fields that contain windowed dates, and you do not need any extra program logic to implement the windowing.

Implementing Date Processing

Following is a list of simple steps that you need to follow in order to implement automatic date recognition in a COBOL program:

- Use the DATEPROC compiler option to enable the process. You specify this as either DATEPROC(FLAG) to get some helpful diagnostic messages, or DATEPROC(NOFLAG). For full information, see “DATEPROC” on page 170.
- Use the YEARWINDOW compiler option to set the century window. There are two ways of doing this:
  - For a *fixed window*, specify a 4-digit year between 1900 and 1999 as the YEARWINDOW option value. For example, YEARWINDOW(1950) defines a fixed window of 1950–2049.
  - For a *sliding window*, specify a negative integer from -1 through -99 as the YEARWINDOW option value. For example, YEARWINDOW(-48) defines a sliding window that starts 48 years before the year that the program is running. So if the program is running in 1998, the century window is 1950–2049, and in 1999 it automatically becomes 1951–2050, and so on. For a full description and syntax, see “YEARWINDOW” on page 201.
- Add the DATE FORMAT clause to the data description entries of those data items in the program that contain dates that you want the compiler to recognize as windowed or expanded dates. For a full description of the DATE FORMAT clause, see *IBM COBOL Language Reference*.
- To expand dates, use MOVE or COMPUTE statements to copy the contents of windowed date fields to expanded date fields.

Basic Remediation

- If necessary, use the DATEVAL and UNDATE intrinsic functions, to convert between date fields and non-dates. For a full description of these functions, see *IBM COBOL Language Reference*.
- Compile the program with the DATEPROC(FLAG) option, and review the diagnostic messages to see if date processing has produced any unexpected side effects (see "Eliminating Warning-Level Messages" on page 534). When the compilation has only Information-level diagnostics, you can recompile with the DATEPROC(NOFLAG) option to produce a "clean" listing.

This provides a simple implementation of date windowing and expansion capabilities in a COBOL program.

Resolving Date-Related Logic Problems

This section discusses three approaches that you can adopt to assist with date-related processing problems, and shows how you can use the millennium language extensions with each approach to achieve a solution.

These and other approaches are discussed in conceptual terms in "Year 2000 Solutions" on page 510. The description here concentrates on the application of COBOL date processing capabilities as a tool to implement the solutions.

The approaches outlined here are:

- Basic Remediation (the century window solution)
- Internal Bridging
- Full Field Expansion

Basic Remediation

The simplest method of ensuring that your programs will continue to function through the year 2000 is to implement a century window solution.

With this method, you define a century window, and specify the fields that contain windowed dates. The compiler then interprets the 2-digit years in those date fields according to the century window.

The following sample code shows how a program can be modified to use this automatic date windowing capability. The program checks whether a video tape was returned on time:

```
CBL  LIB,QUOTE,NOOPT,DATEPROC(FLAG),YEARWINDOW(-60)
        ⋮
    01  Loan-Record.
        05  Member-Number   Pic X(8).
        05  Tape-ID         Pic X(8).
        05  Date-Due-Back   Pic X(6) Date Format yyxxxx.
        05  Date-Returned   Pic X(6) Date Format yyxxxx.
        ⋮
        If Date-Returned Greater than Date-Due-Back Then
           Perform Fine-Member.
```

In this example, there are no changes to the Procedure Division from the program's previous version. The addition of the DATE FORMAT clause on the two date fields means that the compiler recognizes them as windowed date fields, and therefore applies the century window when processing the IF statement. For example, if Date-Due-Back contains “000102” (January 2, 2000) and Date-Returned contains “991231” (December 31, 1999), Date-Returned is less than (earlier than) Date-Due-Back, so the program does not perform the Fine-Member paragraph.

Advantages:

- Fast and easy to implement.
- No change to the program's logic, therefore less testing required.
- This solution will allow your programs to function into and beyond the year 2000.

Disadvantages:

- This should be regarded as a short-term solution, not as a permanent fix.
- There may be some performance degradation introduced by the date windowing functions.
- Implementation of this solution is application-dependent. It will not suit all applications.

Internal Bridging

If your files and databases have not yet been converted to 4-digit year dates, you can use an internal bridge technique to process the dates as 4-digit years. Your program will be structured as follows:

1. Read the input files with 2-digit year dates.
2. Declare these 2-digit dates as windowed date fields and move them to expanded date fields, so that the compiler automatically expands them to 4-digit year dates.
3. In the main body of the program, use the 4-digit year dates for all date processing.
4. Window the dates back to 2-digit years.
5. Write the 2-digit year dates to the output files.

Internal Bridging

This process provides a convenient migration path to a full expanded-date solution, and also may have performance advantages over using windowed dates. For more information, see “Performance Considerations” on page 518.

Using this technique, you do not change any of the logic in the main body of the program. You simply use the 4-digit year date fields in Working-Storage instead of the 2-digit year fields in the records.

Because you are converting the dates back to 2-digit years for output, you should allow for the possibility of the year being outside the century window. For example, if a date field contains the year 2005, but the century window is 1905–2004, then the date is outside the window, and simply storing it as a 2-digit year would be incorrect. To protect against this, you can use a COMPUTE statement to store the date, with the ON SIZE ERROR phrase to detect whether or not the date is within the century window. For more details, see “ON SIZE ERROR Phrase” on page 530.

The following example shows how a program can be changed to implement an internal bridge method:

```
CBL  LIB,QUOTE,NOOPT,DATEPROC(FLAG),YEARWINDOW(-60)
        ⋮
    File Section.
    FD  Customer-File.
    01  Cust-Record.
        05  Cust-Number     Pic 9(9) Binary.
        ⋮
        05  Cust-Date       Pic 9(6) Date Format yyxxxx.
    Working-Storage Section.
    77  Exp-Cust-Date       Pic 9(8) Date Format yyyyxxxx.
        ⋮
    Procedure Division.
        Open I-O Customer-File.
        Read Customer-File.
        Move Cust-Date to Exp-Cust-Date.
        ⋮
   *=====================================================*
   * Use expanded date in the rest of the program logic  *
   *=====================================================*
        ⋮
        Compute Cust-Date = Exp-Cust-Date
                On Size Error Display "Exp-Cust-Date outside
                                       century window"
                End-Compute
        Rewrite Cust-Record.
```

Advantages:

- Little change to the program logic, therefore testing is easy.
- This solution will allow your programs to function into and beyond the year 2000.
- This is a good incremental step towards a full expanded-year solution.

- Good performance.

Disadvantages:

- Some risk of data corruption.

Full Field Expansion

The full field expansion solution involves explicitly expanding 2-digit year date fields to contain full 4-digit years in your files and databases, and then using those fields in expanded form in your programs. This is the only method by which you can be assured of reliable date processing for all applications.

The millennium language extensions allow you to progressively move towards a full date field expansion solution, using the following steps:

1. Apply the short-term (basic remediation) solution, and use this until you have the resources to implement a more permanent solution.
2. Apply the internal bridging scheme. This allows you to use expanded dates in your programs while your files continue to hold dates in 2-digit year form. This in turn will allow you to progress more easily to a full field expansion solution, because there will be no changes to the logic in the main body of the programs.
3. Change the file layouts and database definitions to use 4-digit year dates.
4. Change your COBOL copybooks to reflect these 4-digit year date fields.
5. Run a utility program (or special-purpose COBOL program) to copy from the old format files to the new format. For a sample program, see Figure 122 on page 526.
6. Recompile your programs and perform regression testing and date testing.

After you have completed the first two steps, the remaining steps in the sequence can be repeated any number of times. You do not need to change every date field in every file at the same time. Using this method, you can select files for progressive conversion based on criteria such as business needs or interfaces with other applications.

When you use this method, you will need to write special-purpose programs to convert your files to expanded-date form. Figure 122 on page 526 shows a simple program that copies from one file to another while expanding the date fields. Note that the record length of the output file is larger than that of the input file because the dates are expanded.

Full Field Expansion

```
CBL  LIB,QUOTE,NOOPT,DATEPROC(FLAG),YEARWINDOW(-80)
      ************************************************
      ** CONVERT - Read a file, convert the date    **
      **          fields to expanded form, write   **
      **          the expanded records to a new    **
      **          file.                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  CONVERT.

       ENVIRONMENT DIVISION.

       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT INPUT-FILE
                  ASSIGN TO INFILE
                  FILE STATUS IS INPUT-FILE-STATUS.

           SELECT OUTPUT-FILE
                  ASSIGN TO OUTFILE
                  FILE STATUS IS OUTPUT-FILE-STATUS.

       DATA DIVISION.
       FILE SECTION.
       FD  INPUT-FILE
           RECORDING MODE IS F.
       01  INPUT-RECORD.
           03  CUST-NAME.
               05  FIRST-NAME  PIC X(10).
               05  LAST-NAME   PIC X(15).
           03  ACCOUNT-NUM     PIC 9(8).
           03  DUE-DATE        PIC X(6) DATE FORMAT YYXXXX.         1
           03  REMINDER-DATE   PIC X(6) DATE FORMAT YYXXXX.
           03  DUE-AMOUNT      PIC S9(5)V99 COMP-3.

       FD  OUTPUT-FILE
           RECORDING MODE IS F.
       01  OUTPUT-RECORD.
           03  CUST-NAME.
               05  FIRST-NAME  PIC X(10).
               05  LAST-NAME   PIC X(15).
           03  ACCOUNT-NUM     PIC 9(8).
           03  DUE-DATE        PIC X(8) DATE FORMAT YYYYXXXX.     2
           03  REMINDER-DATE   PIC X(8) DATE FORMAT YYYYXXXX.
           03  DUE-AMOUNT      PIC S9(5)V99 COMP-3.

       WORKING-STORAGE SECTION.

       01  INPUT-FILE-STATUS   PIC 99.
       01  OUTPUT-FILE-STATUS  PIC 99.

       PROCEDURE DIVISION.

           OPEN INPUT INPUT-FILE.
           OPEN OUTPUT OUTPUT-FILE.
```

Figure 122 (Part 1 of 2). Expanding File Dates

```
       READ-RECORD.
           READ INPUT-FILE
                AT END GO TO CLOSE-FILES.
           MOVE CORRESPONDING INPUT-RECORD TO OUTPUT-RECORD.        3
           WRITE OUTPUT-RECORD.

           GO TO READ-RECORD.

       CLOSE-FILES.
           CLOSE INPUT-FILE.
           CLOSE OUTPUT-FILE.

           EXIT PROGRAM.

       END PROGRAM CONVERT.
```

Figure 122 (Part 2 of 2). Expanding File Dates

The following notes apply to Figure 122 on page 526.

1 The fields DUE-DATE and REMINDER-DATE in the input record are both Gregorian dates with 2-digit year components. They have been defined with a DATE FORMAT clause in this program so that the compiler will recognize them as windowed date fields.

2 The output record contains the same two fields in expanded date format. They have been defined with a DATE FORMAT clause so that the compiler will treat them as 4-digit year date fields.

3 The MOVE CORRESPONDING statement moves each item in INPUT-RECORD individually to its matching item in OUTPUT-RECORD. When the two windowed date fields are moved to the corresponding expanded date fields, the compiler will expand the year values using the current century window.

Advantages:

- This is a permanent solution; no more changes are required. This solution will allow your programs to function into and beyond the year 2000.
- Best performance.
- Maintenance will be easier.

Disadvantages:

- Need to ensure that changes to databases, copybooks, and programs are all synchronized.

Programming Techniques

This section describes the techniques you can use in your programs to take advantage of date processing, and the effects of using date fields on COBOL language elements.

For full details of COBOL syntax and restrictions, see *IBM COBOL Language Reference*.

Date Comparisons

When you compare two date fields, the two dates must be compatible; that is, they must have the same number of non-year characters (see “Compatible Dates” on page 536). The number of digits for the year component need not be the same. For example:

```
77  Todays-Date         Pic X(8) Date Format yyyyxxxx.
01  Loan-Record.
    05  Date-Due-Back   Pic X(6) Date Format yyxxxx.
    ⋮
    If Date-Due-Back Greater than Todays-Date Then...
```

In this example, a windowed date field is compared to an expanded date field, so the century window is applied to Date-Due-Back.

Note that Todays-Date must have a DATE FORMAT clause in this case to define it as an expanded date field. If it did not, it would be treated as a non-date field, and would therefore be considered to have the same number of year digits as Date-Due-Back. The compiler would apply the assumed century window to it, which would create an inconsistent comparison. For more information, see “The Assumed Window” on page 538.

Level 88 Condition-Name

If a windowed date field has an 88-level condition-name associated with it, the literal in the VALUE clause is windowed against the century window for the compilation unit rather than the assumed century window of 1900–1999. For example:

```
05  Date-Due        Pic 9(6) Date Format yyxxxx.
88  Date-Target              Value 051220.
```

If the century window is 1950–2049 and the contents of Date-Due is 051220 (representing December 20, 2005), then the condition

```
If Date-Target
```

would evaluate to TRUE, but the condition

```
If Date-Due = 051220
```

would evaluate to FALSE. This is because the literal 051220 is treated as a non-date, and therefore windowed against the assumed century window of 1900–1999 to represent December 20, 1905. But where the same literal is specified in the VALUE clause of an 88-level condition-name, it becomes part of the data item to which it is attached. Because this data item is a windowed date field, the century window is applied whenever it is referenced.

You can also use the DATEVAL intrinsic function in a comparison expression to convert a literal to a date field, and the output from the intrinsic function will then be treated as either a windowed or expanded date field to ensure a consistent comparison. For example, using the above definitions, both of these conditions

```
If Date-Due = Function DATEVAL (051220 "YYXXXX")
If Date-Due = Function DATEVAL (20051220 "YYYYXXXX")
```

would evaluate to TRUE. For more information on the DATEVAL intrinsic function, see “DATEVAL” on page 532.

Restriction: With a level-88 condition name, you can also specify the THRU option on the VALUE clause, for example:

```
    05  Year-Field      Pic 99 Date Format yy.
    88  In-Range               Value 98 Thru 06.
```

With this form, the windowed value of the second item must be greater than the windowed value of the first item. However, the compiler can only verify this if the YEARWINDOW compiler option specifies a fixed century window (for example, YEARWINDOW(1940) rather than YEARWINDOW(-60)).

For this reason, if the YEARWINDOW compiler option specifies a sliding century window, you cannot use the THRU option on the VALUE clause of a level-88 condition name.

Arithmetic Expressions

You can perform arithmetic operations on numeric date fields in the same manner as any numeric data item, and, where appropriate, the century window will be used in the calculation. However, there are some restrictions on where date fields can be used in arithmetic expressions.

Arithmetic operations that include date fields are restricted to:

- Adding a non-date to a date field
- Subtracting a non-date from a date field
- Subtracting a date field from a compatible date field to give a non-date result

The following arithmetic operations are not allowed:

- Any operation between incompatible date fields
- Adding two date fields
- Subtracting a date field from a non-date
- Unary minus, applied to a date field
- Multiplication, division, or exponentiation of or by a date field

Windowed Date Fields

Where a windowed date field participates in an arithmetic operation, it is processed as if the value of the year component of the field were first incremented by 1900 or 2000, depending on the century window. For example:

```
01  Review-Record.
    03  Last-Review-Year    Pic 99 Date Format yy.
    03  Next-Review-Year    Pic 99 Date Format yy.
    ⋮
    Add 10 to Last-Review-Year Giving Next-Review-Year.
```

If the century window is 1910–2009, and the value of Last-Review-Year is 98, then the computation proceeds as if Last-Review-Year is first incremented by 1900 to give 1998. Then the ADD operation is performed, giving a result of 2008. This is stored in Next-Review-Year as 08.

Order of Evaluation

Because of the restrictions on date fields in arithmetic expressions, you may find that programs that previously compiled successfully now produce diagnostic messages when some of the data items are changed to date fields.

Consider the following example:

```
01  Dates-Record.
    03  Start-Year-1        Pic 99 Date Format yy.
    03  End-Year-1          Pic 99 Date Format yy.
    03  Start-Year-2        Pic 99 Date Format yy.
    03  End-Year-2          Pic 99 Date Format yy.
    ⋮
    Compute End-Year-2 = Start-Year-2 + End-Year-1 - Start-Year-1.
```

In this example, the first arithmetic expression evaluated is:

```
Start-Year-2 + End-Year-1
```

However, this is the addition of two date fields, which is not permitted. To resolve this, you should use parentheses to isolate those parts of the arithmetic expression that are allowed. For example:

```
Compute End-Year-2 = Start-Year-2 + (End-Year-1 - Start-Year-1).
```

In this case, the first arithmetic expression evaluated is:

```
End-Year-1 - Start-Year-1
```

This is the subtraction of one date field from another, which is permitted, and gives a non-date result. This non-date result is then added to the date field End-Year-1, giving a date field result which is stored in End-Year-2.

ON SIZE ERROR Phrase

In the example in “Windowed Date Fields” on page 529, the result of 2008 falls within the century window of 1910–2009, so a value of 08 in Next-Review-Year will be recognized as 2008 by subsequent statements in the program.

However, the statement:

```
Add 20 to Last-Review-Year Giving Next-Review-Year.
```

would give a result of 2018. As this falls outside the range of the century window, if the result is stored in Next-Review-Year it would be incorrect, because later references to Next-Review-Year would interpret it as 1918. In this case, the result of the operation depends on whether the ON SIZE ERROR phrase is specified on the ADD statement, as follows:

- If SIZE ERROR is specified, the receiving field is not changed, and the SIZE ERROR imperative statement is executed.

- If SIZE ERROR is not specified, the result is stored in the receiving field with the left-hand digits truncated.

This is an important consideration when developing an internal bridging solution to resolve a date processing problem (see “Internal Bridging” on page 523). When you contract a 4-digit year date field back to 2 digits to write it to the output file, you need to ensure that the date falls within the century window, and that therefore the 2-digit year date will be represented correctly in the field.

You can achieve this using a COMPUTE statement to do the contraction, with a SIZE ERROR phrase to handle the out-of-window condition. For example:

```
Compute Output-Date-YY = Work-Date-YYYY
        On Size Error Go To Out-of-Window-Error-Proc.
```

Note: SIZE ERROR processing for windowed date receivers recognizes *any* year value that falls outside the century window. That is, a year value less than the starting year of the century window raises the SIZE ERROR condition, as does a year value greater than the ending year of the century window.

Other Date Formats

To be eligible for automatic windowing, a date field must contain a 2-digit year as the first or only part of the field. The remainder of the field, if present, must be between 2 and 4 characters, but its content is not important. For example, it can contain a 3-digit Julian day, or a 2-character identifier of some event specific to the enterprise.

If there are date fields in your application that do not fit these criteria, it is not possible to define them as date fields with the DATE FORMAT clause. Some examples of unsupported date formats are:

- A 3-character field consisting of a 2-digit year and a single character to represent the month (A–L representing 1–12). This is not supported because date fields can have only zero, 2, 3, or 4 non-year characters.
- A Gregorian date of the form DDMMYY. This is not supported because the year component is not the first part of the date.

If you need to use date windowing in cases like these, you will need to add some code to isolate the year portion of the date.

In the following example, the two date fields contain dates of the form DDMMYY:

```
03  Last-Review-Date Pic 9(6).
03  Next-Review-Date Pic 9(6).
⋮
Add 1 to Last-Review-Date Giving Next-Review-Date.
```

In this example, if Last-Review-Date contains 230197 (January 23, 1997), then Next-Review-Date will contain 230198 (January 23, 1998) after the ADD statement is executed. This is a simple method of setting the next date for an annual review. However, if Last-Review-Date contains 230199, then adding 1 gives 230200, which is not the desired result.

Because the year is not the first part of these date fields, the DATE FORMAT clause cannot be applied without some code to isolate the year component. In the next example, the year component of both date fields has been isolated so that COBOL can apply the century window and maintain consistent results:

```
03  Last-Review-Date.
    05  Last-R-DDMM  Pic 9(4).
    05  Last-R-YY    Pic 99 Date Format yy.
03  Next-Review-Date Pic 9(6).
    05  Next-R-DDMM  Pic 9(4).
    05  Next-R-YY    Pic 99 Date Format yy.
⋮
Move Last-R-DDMM to Next-R-DDMM.
Add 1 to Last-R-YY Giving Next-R-YY.
```

Controlling Date Processing Explicitly

There may be times when you want COBOL data items to be treated as date fields only under certain conditions, or only in specific parts of the program. Or your application may contain 2-digit year date fields that cannot be declared as windowed date fields because of some interaction with another software product. For example, if a date field is used in a context where it is only recognized by its true binary contents without further interpretation, the date in this field cannot be windowed. This includes:

- A key on a VSAM file
- A search field in a database system such as DB2
- A key field in a CICS command

Conversely, there may be times when you want a date field to be treated as a non-date in specific parts of the program.

COBOL provides two intrinsic functions to cater for these conditions:

DATEVAL Converts a non-date into a date field.

UNDATE Converts a date field into a non-date.

DATEVAL

You can use the DATEVAL intrinsic function to convert a non-date into a date field, so that COBOL will apply the relevant date processing to the field. The first argument to the function is the non-date to be converted, and the second argument specifies the date format. The second argument is a literal string with a specification similar to that of the date pattern in the DATE FORMAT clause.

As an example, assume that a program contains a field Date-Copied, and that this field is referenced many times in the program, but most of these references simply move it between records or reformat it for printing. Only one reference relies on it containing a date, for comparison with another date.

In this case, it is better to leave the field as a non-date, and use the DATEVAL intrinsic function in the comparison statement. For example:

```
03  Date-Distributed Pic 9(6) Date Format yyxxxx.
03  Date-Copied      Pic 9(6).
⋮
If FUNCTION DATEVAL(Date-Copied "YYXXXX") Less than
                    Date-Distributed ...
```

In this example, the DATEVAL intrinsic function converts Date-Copied into a date field so that the comparison will be meaningful.

In most cases, the compiler makes the correct assumption about the interpretation of a non-date, but accompanies this assumption with a warning-level diagnostic message. This typically happens when a windowed date is compared to a literal:

```
03  When-Made        Pic x(6) Date Format yyxxxx.
⋮
If When-Made = "850701" Perform Warranty-Check.
```

The literal is assumed to be a compatible windowed date but with a century window of 1900–1999, thus representing July 15, 1985. You can use the DATEVAL intrinsic function to make the year of the literal date explicit, and eliminate the warning message:

```
If When-Made = Function Dateval("19850701" "YYYYXXXX")
    Perform Warranty-Check.
```

For a full description and syntax of the DATEVAL intrinsic function, see *IBM COBOL Language Reference.*

UNDATE

The UNDATE intrinsic function converts a date field to a non-date, so that it can be referenced without any date processing.

In the following example, the field Invoice-Date in Invoice-Record is a windowed Julian date. In some records, it contains a value of "00999" to indicate that this is not a “true” invoice record, but a record containing file control information.

Invoice-Date has been given a DATE FORMAT clause because most of its references in the program are date-specific. However, in the instance where it is checked for the existence of a control record, the value of "00" in the year component will lead to some confusion. A year of "00" in Invoice-Date will represent a “true” year of either 1900 or 2000, depending on the century window. This is compared to a non-date (the literal "00999" in the example), which will always be windowed against the assumed century window and will therefore always represent the year 1900.

To ensure a consistent comparison, you should use the UNDATE intrinsic function to convert Invoice-Date to a non-date. This means that the IF statement is not comparing any date fields, so it does not need to apply any windowing. For example:

```
01  Invoice-Record.
    03  Invoice-Date    Pic x(5) Date Format yyxxx.
    ⋮
    If FUNCTION UNDATE(Invoice-Date) Equal "00999" ...
```

For a full description and syntax of the UNDATE intrinsic function, see *IBM COBOL Language Reference*.

Eliminating Warning-Level Messages

When the DATEPROC(FLAG) compiler option is in effect, the compiler produces diagnostic messages for every statement that defines or references a date field. These should normally be information-level messages, but it is possible to get warning-level messages for COBOL code that appears to be correct, and even produces correct results.

You should always eliminate error-level and severe-level messages from your program, and it is good programming practice to eliminate warning-level messages as much as possible. When the program is compiled and tested satisfactorily, you can then use the DATEPROC(NOFLAG) compiler option to produce a listing with as few diagnostic messages as possible.

To reduce or eliminate warning-level diagnostic messages when DATEPROC(FLAG) is in effect, you should follow these simple guidelines:

- The diagnostic messages may indicate some date data items that should have had a DATE FORMAT clause but were missed. You should either add DATE FORMAT clauses to these items, or use the DATEVAL intrinsic function in references to them.
- Don't specify a date field in a context where a date field doesn't make sense, such as a FILE STATUS, PASSWORD, ASSIGN USING, LABEL RECORD, or LINAGE item. If you do, you'll get a warning-level message and the date field will be treated as a non-date.
- Ensure that implicit or explicit aliases for date fields are compatible, such as in a group item that consists solely of a date field, or when using the REDEFINES or RENAMES clauses.
- Ensure that if a date field is defined with a VALUE clause, the value is compatible with the date field definition.
- Use the DATEVAL intrinsic function if you want a non-date treated as a date field, such as when moving a non-date to a date field, or comparing a windowed date field with a non-date and you want a windowed date comparison. If you don't use DATEVAL, the compiler will make an assumption about the use of the non-date, and produce a warning-level diagnostic message. Even if the assumption is correct, you may want to use DATEVAL just to eliminate the message. For more information on the DATEVAL intrinsic function, see “DATEVAL” on page 532.
- Use the UNDATE intrinsic function if you want a date field treated as a non-date, such as moving a date field to a non-date, or comparing a non-date and a windowed date field and you don't want a windowed comparison. For more information on the UNDATE intrinsic function, see “UNDATE” on page 533.
- Don't subtract one date field from another unless the result field is a non-date. For more information, see “Arithmetic Expressions” on page 529.

Principles

To gain the most benefit from the millennium language extensions, it is important to understand the reasons for their introduction into the COBOL language, and the rationale behind their design. In particular, there are some apparent inconsistencies that only make sense with an understanding of what the extensions are, and what they are not.

You should not consider using the extensions in new applications, or in enhancements to existing applications, unless the applications are using old data that cannot be expanded until later.

The extensions do not provide fully-specified or complete date-oriented data types, with semantics that recognize, for example, the month and day parts of Gregorian dates. They do however provide special semantics for the year part of dates.

The millennium language extensions focus on a few key principles:

1. Programs to be re-compiled with date semantics are fully-tested and valuable assets of the enterprise. Their only relevant limitation is that any 2-digit years in the programs are restricted to the range 1900–1999.
2. No special processing is done for the non-year part of dates. That is why the non-year part of the supported date formats is denoted by Xs. To do otherwise might change the meaning of existing programs. The only date-sensitive semantics that are provided involve automatically expanding (and contracting) the 2-digit year part of dates with respect to the century window for the program.
3. Dates with 4-digit year parts are generally only of interest when used in combination with windowed dates. Otherwise there is little difference between 4-digit year dates and non-dates.

Objectives

Based on these principles, the millennium language extensions are designed to meet a number of objectives. You should evaluate the objectives that you need to meet in order to resolve your date processing problems, and compare them against the objectives of the millennium language extensions, to determine how your application can benefit from them.

The objectives of the millennium language extensions are as follows:

1. The primary objective is to extend the useful life of your application programs, as they are currently specified, into the twenty-first century.
2. Source changes to accomplish this must be held to the bare minimum, preferably limited to augmenting the declarations of date fields in the Data Division. To implement basic remediation of date problems, you should not be required to make any changes to the program logic in the Procedure Division.
3. The existing semantics of the programs will not be changed by the addition of date fields. For example, where a date is expressed as a literal, as in:

```
If Expiry-Date Greater Than 980101 ...
```

the literal is considered to be compatible (windowed or expanded) with the date field to which it is compared. Further, because the existing program assumes that 2-digit year dates expressed as literals are in the range 1900–1999, the extensions do not change this assumption (see “The Assumed Window” on page 538).

4. The windowing feature is not intended for long-term use. Its intention is to extend the useful life of applications through the year 2000, as a start towards a long-term solution that can be implemented later.
5. The expanded date field feature *is* intended for long-term use, as an aid for expanding date fields in files and databases.

Concepts

With these principles and objectives in mind, you can better understand some of the concepts of the millennium language extensions, and how they interact with other parts of COBOL. This section describes some of these concepts.

Date Semantics

All arithmetic, whether performed on date fields or not, acts only on the numeric contents of the fields; date semantics for the non-year parts of date fields are not provided. For example, adding 1 to a windowed Gregorian date field that contains the value 980831 gives a result of 980832, not 980901.

However, date semantics *are* provided for the year parts of date fields. For example, if the century window is 1950–2049, and the value of windowed date field TwoDigitYear is 49, then the following ADD statement will result in the SIZE ERROR imperative statement being executed:

```
Add 1 to TwoDigitYear
  on Size Error Perform CenturyWindowOverflow
End-Add
```

Compatible Dates

The meaning of the term *compatible dates* depends on the COBOL division in which the usage occurs, as follows:

- The Data Division usage is concerned with the declaration of date fields, and the rules governing COBOL language elements such as subordinate data items and the REDEFINES clause. In the following example, Review-Date and Review-Year are compatible because Review-Year can be declared as a subordinate data item to Review-Date:

  ```
  01  Review-Record.
      03  Review-Date            Date Format yyxxxx.
          05  Review-Year Pic XX Date Format yy.
          05  Review-M-D  Pic XXXX.
  ```

  For full details, see *IBM COBOL Language Reference*.

- The Procedure Division usage is concerned with how date fields can be used together in operations such as comparisons, moves, and arithmetic expressions. To be considered compatible, date fields must have the same number of non-year characters. For example, a field with DATE FORMAT YYXXXX is compatible with another field that has the same DATE FORMAT, and with a YYYYXXXX field, but not with a YYXXX field.

The remainder of this discussion relates to the Procedure Division usage of compatible dates.

You can perform operations on date fields, or on a combination of date fields and non-dates, provided that the date fields in the operation are compatible. For example, with the following definitions:

```
01  Date-Gregorian-Win  Pic 9(9) Packed Date Format yyxxxx.
01  Date-Julian-Win     Pic 9(9) Packed Date Format yyxxx.
01  Date-Gregorian-Exp  Pic 9(9) Packed Date Format yyyyxxxx.
```

The statement:

```
    If Date-Gregorian-Win Less than Date-Julian-Win ...
```

is inconsistent because the number of non-year digits is different between the two fields. The statement:

```
    If Date-Gregorian-Win Less than Date-Gregorian-Exp ...
```

is accepted because the number of non-year digits is the same for both fields. In this case the century window is applied to the windowed date field (Date-Gregorian-Win) to ensure that the comparison is meaningful.

Where a non-date is used in conjunction with a date field, the non-date is either assumed to be compatible with the date field, or treated as a simple numeric value, as described in the following section.

Treatment of Non-Dates

The simplest kind of non-date is just a literal value. The following items are also non-dates:

- A data item whose data description does not include a DATE FORMAT clause.
- The results (intermediate or final) of some arithmetic expressions. For example, the difference of two date fields is a non-date, wheras the sum of a date field and a non-date is a date field.
- The output from the UNDATE intrinsic function.

When you use a non-date in conjunction with a date field, the compiler interprets the non-date as either a date whose format is compatible with the date field, or a simple numeric value. This interpretation depends on the context in which the date field and non-date are used, as follows:

Comparison

Where a date field is compared to a non-date, the non-date is considered to be compatible with the date field in the number of year and non-year characters. In the following example:

```
01  Date-1             Pic 9(6) Date Format yyxxxx.
    ⋮
    If Date-1 Greater than 971231 ...
```

Because the non-date literal 971231 is being compared to a windowed date field, it is treated as if it had the same DATE FORMAT as Date-1, but with a base year of 1900.

Arithmetic operations

In all supported arithmetic operations, non-date fields are treated as simple numeric values. In the following example:

```
01  Date-2             Pic 9(6) Date Format yyxxxx.
    ⋮
    Add 10000 to Date-2.
```

the numeric value 10000 is added to the Gregorian date in Date-2, effectively adding one year to the date.

MOVE statement

Moving a date field to a non-date is not supported. However, you can use the UNDATE intrinsic function to achieve this. For more information, see “UNDATE” on page 533.

When you move a non-date to a date field, the sending field is assumed to be compatible with the receiving field in the number of year and non-year characters. For example, when you move a non-date to a windowed date field, the non-date field is assumed to contain a compatible date with a 2-digit year.

The Assumed Window

Where the program operates on windowed date fields, the compiler applies the century window for the compilation unit; that is, the one defined by the YEARWINDOW compiler option. Where a windowed date field is used in conjunction with a non-date, and the context demands that the non-date also be treated as a windowed date, the compiler uses an assumed century window to resolve the non-date field.

The assumed century window is 1900–1999, which is typically not the same as the century window for the compilation unit.

In many cases, particularly for literal non-dates, this assumed century window will be the correct choice. For example, in the construct:

```
01  manufacturingRecord.
    03  makersDate Pic X(6) Date Format yyxxxx.
    ⋮
    If makersDate Greater than "720101" ...
```

the literal should retain its original meaning of January 1, 1972, and not change to 2072 if the century window is, for example, 1975–2074. Even if the assumption is correct, it is better to make the year explicit, and also eliminate the warning-level diagnostic message that accompanies application of the assumed century window, by using the DATEVAL intrinsic function:

```
        If makersDate Greater than
                      Function Dateval("19720101" "YYYYXXXX") ...
```

In other cases however, the asumption may not be correct. For example:

```
    01  Project-Controls.
        03  Date-Target     Pic 9(6).
        ⋮
    01  Progress-Record.
        03  Date-Complete   Pic 9(6) Date Format yyxxxx.
        ⋮
        If Date-Complete Less than Date-Target ...
```

For this example, assume that Project-Controls is in a COPY member that is used by other applications that have not yet been upgraded for Year 2000 processing, and therefore Date-Target cannot have a DATE FORMAT clause. In the example, if:

- The century window is 1910–2009,
- Date-Complete is 991202 (Gregorian date: December 2, 1999), and
- Date-Target is 000115 (Gregorian date: January 15, 2000),

then:

- Date-Complete is earlier than (less than) Date-Target.

However, because Date-Target does not have a DATE FORMAT clause, it is a non-date, so the century window applied to it is the assumed century window of 1900–1999, which means that it is processed as January 15, 1900. So Date-Complete will be greater than Date-Target, which is not the desired result.

In this case, you should use the DATEVAL intrinsic function to convert Date-Target to a date field for this comparison. For example:

```
        If Date-Complete Less than
           Function Dateval (Date-Target "YYXXXX") ...
```

For more information on the DATEVAL intrinsic function, see “DATEVAL” on page 532.

Appendix A. Summary of Differences with Host COBOL

This appendix gives an overview of the product function differences between IBM COBOL for OS/390 & VM and IBM VisualAge COBOL.

Figure 123 lists the differences between IBM COBOL for OS/390 & VM and IBM VisualAge COBOL. For information on COBOL language differences between the different platforms, see the *IBM COBOL Language Reference*. For information on developing applications that are portable between the different platforms, see Chapter 18, "Porting Applications between Platforms" on page 364.

Figure 123 (Page 1 of 3). Product Differences Between Mainframe and Workstation IBM COBOL

Product Function	Workstation Implementation
Compiler Options	The following compiler options are treated as comments: ADV, AWO, BUFSIZE, DATA, DECK, DBCS, FASTSRT, FLAGMIG, INTDATE, LANGUAGE, NAME, OUTDD, RENT, and suboptions of TEST. These options are flagged with I-level messages. The following compiler options are treated as comments; however, if specified, the application might yield unpredictable results: NOADV and CMPR2. These options are flagged with W-level messages. LIB is the IBM-supplied default on the PC; NOLIB is the IBM-supplied default on the host.
Data Representation	Binary data types are handled based on the specification of the BINARY compiler option. Sign representation for external decimal data are ASCII-based. Specifying NUMPROC(NOPFD) allows the full range of valid sign values for the numeric class test. EBCDIC vs ASCII: • You can specify the EBCDIC collating sequence using the following language elements: ALPHABET clause, PROGRAM COLLATING SEQUENCE clause, and the COLLATING SEQUENCE phrase of the SORT and MERGE verbs. • You can specify the CHAR(EBCDIC) compiler option to indicate that DISPLAY data items are in the System/390 data representation (EBCDIC). You can use the FLOAT(S390) compiler option to indicate that floating point data items are in the System/390 data representation (hexadecimal) as opposed to the native (IEEE) format. DBCS—Under AIX, OS/2, and Windows, you do not use shift-in or shift-out delimiters for DBCS literals unless the CHAR(EBCDIC) compiler option is in effect. Within an alphanumeric literal, using control characters X'00' through X'1F' can yield unpredictable results.

Figure 123 (Page 2 of 3). Product Differences Between Mainframe and Workstation IBM COBOL

Product Function	Workstation Implementation
Environment Variables	IBM VisualAge COBOL recognizes the following as environment variables: • ASSIGNment name • COBMSGS • COBOPT • COBPATH • COBRTOPT • DB2DBDFT • EBCDIC_CODEPAGE • LANG • LC_COLLATE • LC_MESSAGES • LC_TIME • LIBPATH • library-name specified as a user-defined word • LOCPATH • NLSPATH • SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, and SYSPCH • SYSLIB • TEMP • text-name specified as a user-defined word • TZ
File Specification	All files are treated as single volume files. All other file specifications are treated as comments. This affects the following: REEL, UNIT, MULTIPLE FILE TAPE clause, and CLOSE...UNIT/REEL.
Inter-Language Communication (ILC)	ILC is available with C, C++, and PL/I programs. The following is a list of differences in ILC behavior on the workstation compared to using ILC on the host with Language Environment: • There are differences in termination behavior when a COBOL STOP RUN, a C `exit()`, or a PLI STOP is used. • There is no coordinated condition handling on the workstation. Use of a C `longjmp()` that crosses COBOL programs should be avoided on the workstation. • On the host, the first program which is invoked within the process and which is enabled for Language Environment is considered to be the “main” program. On the workstation, the first COBOL program invoked within the process is considered to be the main program by COBOL. This impacts the language semantics which is sensitive to the definition of the run unit (the execution unit with starts with a main program). For example, a STOP RUN will result in the return of the control to the invoker of the main program, which in a mixed language environment may be different as stated above.

Figure 123 (Page 3 of 3). Product Differences Between Mainframe and Workstation IBM COBOL

Product Function	Workstation Implementation
I-O	I-O support for sequential, relative, and indexed files is provided using STL file system, VSAM (only remote files supported on Windows), and Btrieve. Sizes and values are different for the data-name returned from the file system. IBM VisualAge COBOL does not provide direct support for tape drives or diskette drives. Line Sequential I-O is supported using the native byte stream file support of the platform. The following language elements are treated as comments for Line Sequential files, as well as for Sequential, Relative, and Indexed files: • ADVANCING phrase of WRITE statement • APPLY WRITE ONLY clause • AT END-OF-PAGE phrase of WRITE statement • BLOCK CONTAINS clause • CODE-SET clause • DATA RECORDS clause • FILE STATUS value 39 (fixed file attribute conflict) • LABEL RECORDS clause • LINAGE clause • OPEN I-O option • PADDING CHARACTER clause • RECORD CONTAINS 0 clause • RECORD CONTAINS clause (format 3) • RECORD DELIMITER clause • RECORDING MODE clause • RERUN clause • RESERVE clause • REVERSED phrase of OPEN statement • VALUE OF clause of file description entry
Run-Time Options	The following run-time options are not recognized by IBM VisualAge COBOL and will be treated as invalid options: AIXBLD, ALL31, CBLPSHPOP, CBLQDA, COUNTRY, HEAP, MSGFILE, NATLANG, SIMVRD, and STACK. On the host, the STORAGE run-time option can be used to initialize COBOL working storage. With IBM VisualAge COBOL, this is achieved with the WSCLEAR compiler option.
Source Code Line	A COBOL source line can be less than 72 characters. A line ends on column 72 or where a carriage control character is found.

Appendix B. System/390 Host Data Type Considerations

The following are considerations, restrictions, and limitations which apply to the use of System/390 host data types. The BINARY, CHAR, and FLOAT compiler options determine if System/390 host data types or native data types are used. See Chapter 10, “Compiler Options” on page 160 for details on how to specify these options and for specific information about how each option is used.

CICS Access

CICS allows you to specify various data conversion choices at various places and at various granularities. For example, client CICS translator option specifications on the server for different resources (file, EIBLK, COMMAREA, transient data queue, etc.). Your use of host versus native data depends on such selections. Refer to the appropriate CICS documentation for specific information about how such choices can best be made.

Note: System/390 host data type support is only allowed on the following CICS systems using the EBCDIC enablement support:

- CICS for OS/2
- VisualAge CICS Enterprise Application Development

It will not work for COBOL programs that are translated by the CICS translator and run on CICS for Windows NT or CICS for AIX.

Date and Time Callable Services

The date and time callable services can be used with the System/390 host data types. All of the parameters passed to the callable services must be in System/390 host data type format. You cannot mix native and host data types in the same call to a date and time service.

Floating Point Overflow Exceptions

Due to differences in the limits of floating point data representations on the workstation and the System/390 host platform, it is possible that a floating point overflow exception can occur during conversion between the two formats. For example, you might get the following message on the workstation:

```
IWZ053S An overflow occurred on conversion to floating point
```

when running a program which executes successfully on the System/390 host platform.

To avoid this problem, you must be aware of the maximum floating point values supported on either platform for the respective data types. The limits are shown in Figure 124

Figure 124. Floating Point Value Limits

Data Type	Maximum Workstation Value	Maximum System/390 Host Value
COMP-1	±(2**128-2**4) (Approx. ±3.4028E+38)	±(16**63-16**57) (Approx. ±7.2370E+75)
COMP-2	±(2**1024-2**971) (Approx. ±1.7977E+308)	±(16**63-16**49) (Approx. ±7.2370E+75)

Note: ** indicates "in the power of."

As shown above, the System/390 host can carry a larger COMP-1 value than the workstation and the workstation can carry a larger COMP-2 value than the System/390 host.

DB2

| The System/390 host data type compiler options can be used with DB2 programs.

MQSeries

The System/390 host data type compiler options should not be used with MQSeries programs.

Remote File Access

- If you are accessing remote host files using SMARTdata Utilities, (via COBOL file I/O statements), you do not need to specify A Data Language (ADL) for data conversion. You can access the data in the VSAM host files directly when you compile with the host data options.
- If you are already using ADL for conversion of remote file data, do not use the host data support.
- Note that file records (01 record under FD) implicitly take on the characteristics of the CHAR compiler option.

SORT

| All of the System/390 host data types can be used as sort keys.

Appendix C. Intermediate Results and Arithmetic Precision

The compiler handles arithmetic statements as a succession of operations, performed according to operator precedence, and sets up an intermediate field to contain the results of these operations.

Intermediate results are possible in the following cases:

- In an ADD or SUBTRACT statement containing more than one operand immediately following the verb.
- In a COMPUTE statement specifying a series of arithmetic operations or multiple result fields.
- In arithmetic expressions contained in conditional statements and reference modification specifications.
- In the GIVING option with multiple result fields for the ADD, SUBTRACT, MULTIPLY, or DIVIDE statements.
- In a statement with an intrinsic function used as an operand.

For a discussion on when the compiler uses fixed-point or floating-point arithmetic, refer to “Fixed-Point versus Floating-Point Arithmetic” on page 43.

Calculating Precision of Intermediate Results

The compiler uses algorithms to determine the number of *integer* and decimal places reserved for intermediate results.

In the following discussion of how the compiler determines the number of integer and decimal places reserved for intermediate results, these abbreviations are used:

i The number of integer places carried for an intermediate result.

d The number of decimal places carried for an intermediate result.

ROUNDED
If the ROUNDED option is used, one more integer or decimal might be added for accuracy, if necessary. Only the final results are rounded; the intermediate results are not rounded.

dmax In a particular statement, the largest of:

- The number of decimal places needed for the final result field(s).
- The maximum number of decimal places defined for any operand, except divisors or exponents.
- The *outer-dmax* for any function operand.

inner-dmax
The *inner-dmax* for a function is the largest of:

- The number of decimal places defined for any of its elementary arguments.
- The *dmax* for any of its arithmetic expression arguments.
- The *outer-dmax* for any of its embedded functions.

outer-dmax
The number that determines how a function result contributes to operations outside of its own evaluation (for example, if the function is an operand in an arithmetic expression or an argument to another function).

op1 The first operand in a generated arithmetic statement. For division, *op1* is the divisor.

op2 The second operand in a generated arithmetic statement. For division, *op2* is the dividend.

i1*,*i2 The number of integer places in *op1* and *op2*, respectively.

d1*,*d2 The number of decimal places defined for *op1* and *op2*, respectively.

ir The intermediate result field obtained when a generated arithmetic statement or operation is performed. *ir1*, *ir2*,. . ., represent successive intermediate results. These intermediate results are generated either in registers or in storage locations. Successive intermediate results might have the same location.

The compiler handles each statement as a succession of operations. For example, the following statement:

```
COMPUTE Y = A + B * C - D / E + F ** G
```

is calculated as:

** F	BY G	yielding *ir1*
MULTIPLY B	BY C	yielding *ir2*
DIVIDE E	INTO D	yielding *ir3*
ADD A	TO ir2	yielding *ir4*
SUBTRACT *ir3*	FROM *ir4*	yielding *ir5*
ADD *ir5*	TO *ir1*	yielding Y

Fixed-Point Data and Intermediate Results

The number of integer and decimal places in an *intermediate result* can be determined by using the following guidelines:

Figure 125. Determining the Precision of an Intermediate Result

Operation	Integer Places	Decimal Places
+ or -	(*i1* or *i2*) + 1, whichever is greater	*d1* or *d2*, whichever is greater
*	*i1* + *i2*	*d1* + *d2*
/	*i2* + *d1*	(*d2* - *d1*) or *dmax*, whichever is greater

You must define the operands of any arithmetic statements with enough decimal places to give the desired accuracy in the final result.

Figure 126 indicates the action of the compiler when handling intermediate results for fixed-point numbers.

Figure 126. Determining When the Compiler Might Truncate Intermediate Results

Value of i + d	Value of d	Value of i + dmax	Action Taken
<30 =30	Any value	Any value	*i* integer and *d* decimal places are carried for *ir*.
>30	<*dmax* =*dmax*	Any value	30-*d* integer and *d* decimal places are carried for *ir*.
	>*dmax*	<30 =30	*i* integer and 30-*i* decimal places are carried for *ir*.
		>30	30-*dmax* integer and *dmax* decimal places are carried for *ir*.

Exponentiations Evaluated in Fixed-Point Arithmetic

Exponentiation is represented by the expression *op1* ** *op2*. Based on the characteristics of *op2*, the compiler handles exponentiation of fixed-point numbers in one of three ways:

- When *op2* is expressed with decimals, floating-point rules (see “Floating-Point Data and Intermediate Results” on page 551) are used to calculate the exponentiation.
- When *op2* is an integral literal or constant, the value *d* is computed as:

  ```
  d = d1 * |op2|
  ```

  When *op1* is a data-name or variable, the value *i* is computed as:

  ```
  i = i1 * |op2|
  ```

  When *op1* is a literal or constant, the actual value of *op1* ** |*op2*| is computed and *i* is set equal to the number of integers in that value.

Fixed-Point Data

Having calculated *i* and *d*, the compiler takes the action indicated in the following figure to handle intermediate results:

Figure 127. Determining When the Compiler Might Truncate Intermediate Results for Exponentiation

Value of i + d	Other Conditions	Action Taken
<30	Any	*i* integer and *d* decimal places are carried for *ir*.
=30	*op1* has an odd number of digits	*i* integer and *d* decimal places are carried for *ir*.
	op1 has an even number of digits	The exponentiation is handled the same as it is when *op2* is an integral data-name or a variable, except in the case of a 30-digit integer raised to the power of literal 1, where the computation is done following the rules for *op1* with an odd number of digits.
>30	Any	The exponentiation is handled the same as it is when *op2* is an integral data-name or a variable.

If *op2* is negative, the value of 1 is divided by the result produced by the preliminary computation described above. The values of *i* and *d* that are used are calculated using the rules for division found in Figure 125 on page 547.

- When *op2* is an integral data-name or a variable, *dmax* decimals and `30-dmax` integers are used. Here, *op1* is multiplied by itself (`|op2| - 1`) times. For example, the following statement:

  `COMPUTE Y = A ** B`, where B has a value of 4

  is calculated as:

MULTIPLY A	BY A	yielding *ir1*
MULTIPLY *ir1*	BY A	yielding *ir2*
MULTIPLY *ir2*	BY A	yielding *ir3*
MOVE *ir3*	TO *ir4*	which has *dmax* decimals

  The values of *i* and *d* that are used for the above multiplications are calculated using the rules for multiplication found in Figure 125 on page 547.

  If B is positive, Y = *ir4*.

  If B is negative, however,

  DIVIDE *ir4* INTO 1 yielding *ir5*, which has *dmax* decimals
  Y = *ir5*

  If *op2* is equal to zero, the answer is 1. Division-by-0 and exponentiation SIZE ERROR conditions apply. For specific information on the SIZE ERROR option, see the *IBM COBOL Language Reference*.

Fixed-point exponents with more than 9 significant digits are always truncated to 9 digits. If the exponent is a literal or constant, an E-level compiler diagnostic message is issued; otherwise, an informational message is issued at run time.

Truncated Intermediate Results

Whenever the number of digits in a decimal is greater than 30, the field is truncated to 30 digits. You will get a warning message when you compile the program. If truncation happens at run time, a message is issued and the program continues running.

If you think an intermediate result field might exceed 30 digits, you can use floating-point operands (COMP-1 and COMP-2) to avoid truncation.

Binary Data and Intermediate Results

If an operation involving binary operands requires intermediate results greater than 18 digits, the compiler converts the operands to internal decimal before performing the operation. If the result field is binary, the result will be converted from internal decimal to binary.

Binary items are used most efficiently when the intermediate result is not greater than 9 digits.

Intrinsic Functions Evaluated in Fixed-Point Arithmetic

Integer functions and mixed functions can both return an integer result. The *inner-dmax* and *outer-dmax* values are determined by the characteristics of the function.

Integer Functions

These functions always return an integer, and the *outer-dmax* will always be zero. For those functions whose arguments must be integer, the *inner-dmax* will also always be zero.

The following table summarizes the precision of the function results:

Figure 128 (Page 1 of 2). Precision of Integer Intrinsic Functions

Function	Inner-Dmax	Outer-Dmax	Function Result
DATE-OF-INTEGER	0	0	8-digit integer
DATE-TO-YYYYMMDD	0	0	8-digit integer
DAY-OF-INTEGER	0	0	7-digit integer
DAY-TO-YYYYDDD	0	0	7-digit integer
FACTORIAL	0	0	fixed-point, 30-digit integer
INTEGER-OF-DATE	0	0	7-digit integer
INTEGER-OF-DAY	0	0	7- digit integer
LENGTH	n/a	0	9- digit integer
MOD	0	0	integer with as many digits as min(i1 i2)
ORD	n/a	0	3-digit integer
ORD-MAX		0	9-digit integer
ORD-MIN		0	9-digit integer

Figure 128 (Page 2 of 2). Precision of Integer Intrinsic Functions

Function	Inner-Dmax	Outer-Dmax	Function Result
YEAR-TO-YYYY	0	0	4-digit integer
INTEGER		0	With a fixed-point argument, result will be fixed-point integer with one more integer digit than the argument. With a floating-point argument, result will be fixed-point, 30-digit integer.
INTEGER-PART		0	With a fixed-point argument, result will be fixed-point integer with the same number of integer digits as the argument. With a floating-point argument, result will be fixed-point, 30-digit integer.

Mixed Functions

When the compiler handles a mixed function as fixed-point arithmetic, the result will be either integer or fixed-point with decimals (when any argument is floating-point, the function becomes a floating-point function and will follow floating-point rules). For MAX, MIN, RANGE, REM, and SUM, the *outer-dmax* is always equal to the *inner-dmax*. To determine the precision of the result returned for these functions, apply the rules for fixed-point arithmetic to each step in the algorithm used to calculate the function result.

MAX

1. Assign the first argument to your function result.
2. For each remaining argument:
   a. Compare the algebraic value of your function result with the argument.
   b. Assign the greater of the two to your function result.

MIN

1. Assign the first argument to your function result.
2. For each remaining argument:
   a. Compare the algebraic value of your function result with the argument.
   b. Assign the lesser of the two to your function result.

RANGE

1. Use the steps for MAX to select your maximum argument.
2. Use the steps for MIN to select your minimum argument.
3. Subtract the minimum argument from the maximum.
4. Assign the difference to your function result.

REM

1. Divide argument-1 by argument-2.
2. Remove all non-integer digits from the result of step 1.
3. Multiply the result of step 2 by argument-2.
4. Subtract the result of step 3 from argument-1.

5. Assign the difference to your function result.

SUM

1. Assign the value 0 to your function result.
2. For each argument:
   a. Add the argument to your function result.
   b. Assign the sum to your function result.

Floating-Point Data and Intermediate Results

Floating-point instructions are used to compute an arithmetic expression if any of the following conditions are true:

- A receiver or operand in the expression is COMP-1, COMP-2, external floating-point data, or a floating-point literal.
- An exponent contains decimal places.
- An exponent is an expression that contains an exponentiation or divide operator and *dmax* is greater than zero.
- An intrinsic numeric function is a floating-point function.

If any operation in an arithmetic expression is computed in floating-point, the entire expression is computed as if all operands were converted to floating-point and the operations are evaluated using floating-point instructions.

If an expression is computed in floating-point, double-precision floating-point is used if any receiver or operand in the expression is not COMP-1, or if a multiplication or exponentiation operation appears in the expression. Whenever double-precision floating-point is used for one operation in an arithmetic expression, all operations in the expression are computed as if double-precision floating-point instructions were used.

Alert: If a floating-point operation has an intermediate result field in which exponent overflow occurs, the job will be abnormally terminated.

Exponentiations Evaluated in Floating-Point Arithmetic

Floating-point exponentiations are always evaluated using double-precision floating-point arithmetic.

The value of a negative number raised to a fractional power is undefined. For example, (-2) ** 3 is equal to -8, but (-2) ** (3.000001) is not defined. When an exponentiation is evaluated in floating-point and there is a possibility that the value of the exponentiation will be undefined (as in the example above), then the value of the exponent is evaluated at run time to determine if it is actually an integer.

Intrinsic Functions Evaluated in Floating-Point Arithmetic

The floating-point numeric functions will always return a double precision floating-point value. For a list of the floating-point, fixed-point and mixed functions, see “Numeric Intrinsic Functions” on page 40.

Mixed functions with floating-point arguments will be evaluated using floating-point arithmetic.

Arithmetic Expressions in Non-arithmetic Statements

Arithmetic expressions can appear in contexts other than arithmetic statements. For example, an arithmetic expression can be used with the IF statement. In such statements, the rules for intermediate results, floating point, and double-precision floating-point apply, with the following changes:

- Abbreviated IF statements are handled as though the statements were not abbreviated.
- An explicit relation condition exists when a required relational operator is used to define the comparison between two operands (here referred to as comparands). In an explicit relation condition where one or both of the comparands is an arithmetic expression, the rules for intermediate results are determined taking into consideration the attributes of both comparands. That is to say, *dmax* is defined to be the maximum number of decimal places defined for any operand of either comparand, except divisors and exponents. The rules for floating-point and double-precision floating-point apply if any of the following conditions are true:
  - Any operand in either comparand is COMP-1, COMP-2, external floating- point data, or a floating-point literal.
  - An exponent contains decimal places.
  - An exponent is an expression that contains an exponentiation or divide operator and *dmax* is greater than zero.

  For example, in the statement:

  ```
  IF operand-1 = expression-1 THEN . . .
  ```

  where `operand-1` is a data-name defined to be COMP-2, and `expression-1` contains only fixed-point operands, the rules for floating-point arithmetic apply to `expression-1` because it is being compared to a floating-point operand.
- When the comparison between an arithmetic expression and either a data item or another arithmetic expression is defined without using a relational operator, then no explicit relation condition is said to exist. Here, the arithmetic expression is evaluated without regard to the attributes of the operand with which the comparison is being made. For example, in the statement:

  ```
  EVALUATE expression-1
    WHEN expression-2 THRU expression-3
    WHEN expression-4
    .
    .
  END-EVALUATE
  ```

  each arithmetic expression is evaluated in fixed-point or floating-point arithmetic based on its own characteristics.

Appendix D. Complex OCCURS DEPENDING ON

Complex OCCURS DEPENDING ON (ODO) is supported as an extension to the COBOL 85 Standard.

The basic forms of complex ODO permitted by the compiler are:

- A data item described by an OCCURS clause with the DEPENDING ON option is followed by a non-subordinate element or group (a variably-located item).
- A data item described by an OCCURS clause with the DEPENDING ON option is followed by a non-subordinate data item described by an OCCURS clause with the DEPENDING ON option (variably-located table).
- A data item described by an OCCURS clause with the DEPENDING ON option is nested within another data item described by an OCCURS clause with the DEPENDING ON option (table with variable-length elements).
- Index-name for a table with variable-length elements.

Be Sure to Set Values of ODO Objects

Every ODO object in an 01-level must be set before any reference is made to a complex ODO item in the 01-level. (Note: An ODO object cannot be variably located.) For instance, in the following example, before EMPLOYEE-NUMBER can be referred to, COUNTER-1 and COUNTER-2 must be set, even though EMPLOYEE-NUMBER does not directly depend on either of the ODO objects for its value.

The length of the variable portions of each record is the product of the ODO object and the length of the subject of the OCCURS clause. The length is calculated at the time of a reference to one of the following:

- A data item following, and not subordinate to, a variable-length table in the same level-01 record (variably-located item).

  1 in the following example.

- A group item following, and not subordinate to, a variable-length table in the same level-01 record (variably-located group).

  2 in the following example.

- An index name for a table that has variable-length elements.

  3 in the following example.

- An element of a table that has variable-length elements.

  4 in the following example.

Complex OCCURS DEPENDING ON

Complex ODO Example

Any item that meets one of these four criteria is considered to be a “complex ODO item.” The following example illustrates each of the possible occurrences of a complex ODO item.

```
01 FIELD-A.
   02 COUNTER-1                             PIC S99.
   02 COUNTER-2                             PIC S99.
   02 TABLE-1.
      03 RECORD-1 OCCURS 1 TO 5 TIMES
                  DEPENDING ON COUNTER-1    PIC X(3).
   02 EMPLOYEE-NUMBER                       PIC X(5). 1
   02 TABLE-2 OCCURS 5 TIMES                          2
               INDEXED BY INDX.                       3
      03 TABLE-ITEM                         PIC 99.   4
      03 RECORD-2 OCCURS 1 TO 3 TIMES
               DEPENDING ON COUNTER-2.
         04 DATA-NUM                        PIC S99.
```

How Length Will be Calculated

Whenever a reference is made to one of these four data items, the actual length, if used, is computed as follows:

- The contents of COUNTER-1 are multiplied by 3 to calculate the length of TABLE-1.
- The contents of COUNTER-2 are multiplied by 2 and added to the length of TABLE-ITEM to calculate the length of TABLE-2.
- The length of FIELD-A is calculated by adding the length of COUNTER-1, COUNTER-2, TABLE-1, EMPLOYEE-NUMBER, and TABLE-2 times 5.

Changes in ODO Object Value

If a data item described by an OCCURS clause with the DEPENDING ON option is followed in the same level-01 record by non-subordinate data items, a change in the value of the ODO object, and a subsequent reference to a complex ODO item during the course of program execution, will have the following effects:

- The size of any group containing the related OCCURS clause will reflect the new value of the ODO object.
- Whenever a MOVE to a group containing an ODO object is executed, the MOVE is made based on the current contents of the object of the DEPENDING ON option.

  Caution: The value of the ODO object may change because a MOVE is made to it or to the group in which it is contained. The value of the ODO object may also change because the group in which it is contained is a record area that has been changed by execution of a READ statement.
- The location of any non-subordinate items following the item described with the OCCURS clause will be affected by the new value of the ODO object. If you wish to preserve the contents of these items, the following procedure can be used: Prior to the change in the ODO object, move all non-subordinate items following

the variable item to a work area; after the change in the ODO object, move all the items back.

Changing ODO Object with Complex-ODO Index Names

You must be careful when using complex-ODO index names. If you set an index name (like 'INDX' in the previous example) for a table with variable-length entries ('TABLE-2'), and then change the value of the ODO object ('COUNTER-2'), be aware that the offset in your index is no longer valid for the table, since the table has changed. If, at this point, you were to code statements that used your index name, thinking the index name had a valid value for the table, the statements would yield unexpected results. This would apply to coding:

- A reference (using your index name) to an element of the table
- A format-1 SET statement of the type SET INTEGER-DATA-ITEM TO INDEX-NAME
- A format-2 SET statement of the type SET INDEX-NAME UP/DOWN BY INTEGER.

To avoid making this type of error, you can do the following:

1. Save the value of your index name (in the form of its integer occurrence number) in an integer data-item before changing the ODO object.
2. Immediately after changing the ODO object, restore the value of your index name from the integer data-item.

For example:

```
77   INTEGER-DATA-ITEM-1      PIC 99.

     SET INDX TO 5
*          INDX is valid at this point.
     SET INTEGER-DATA-ITEM-1 TO INDX
     MOVE NEW-VALUE TO COUNTER-2.
*          INDX is not valid at this point.
     SET INDX TO INTEGER-DATA-ITEM-1.
*          INDX Is now valid and can be
*          used with expected results.
```

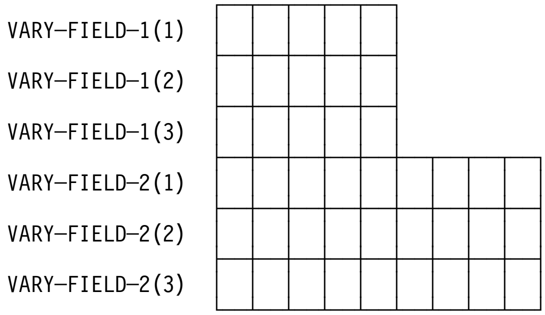
Changing ODO Object with Variable Occurrence Table

The following example applies to changing an ODO object by adding an element to a variable occurrence table with variably-located items following it. The example updates a record containing an OCCURS clause with the DEPENDING ON option and at least one other subsequent entry. In this case, the subsequent entry is another OCCURS clause with the DEPENDING ON option.

Complex OCCURS DEPENDING ON

```
WORKING-STORAGE SECTION.
01  VARIABLE-REC.
    05  FIELD-1                              PIC X(10).
    05  CONTROL-1                            PIC S99.
    05  CONTROL-2                            PIC S99.
    05  VARY-FIELD-1 OCCURS 1 TO 10 TIMES
         DEPENDING ON CONTROL-1              PIC X(5).
    05  GROUP-ITEM-1.
        10  VARY-FIELD-2
             OCCURS 1 TO 10 TIMES
             DEPENDING ON CONTROL-2          PIC X(9).
01  STORE-VARY-FIELD-2.
    05  GROUP-ITEM-2.
        10  VARY-FLD-2
             OCCURS 1 TO 10 TIMES
             DEPENDING ON CONTROL-2          PIC X(9).
```

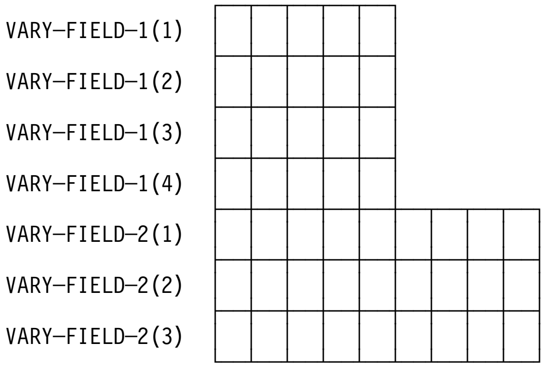
Assume that both CONTROL-1 and CONTROL-2 contain the value 3. In this situation, storage for VARY-FIELD-1 and VARY-FIELD-2 would look like this:



In order to add a fourth field to VARY-FIELD-1, the following steps are required to prevent VARY-FIELD-1 from overlaying the first 5 bytes of VARY-FIELD-2:

```
MOVE GROUP-ITEM-1 TO GROUP-ITEM-2
ADD 1 TO CONTROL-1
MOVE "additional field" TO
  VARY-FIELD-1 (CONTROL-1)
MOVE GROUP-ITEM-2 TO GROUP-ITEM-1
```

The updated storage for VARY-FIELD-1 and VARY-FIELD-2 would now look like this:



The intent of this last example is to emphasize that if you want to preserve the values contained in data items that follow a variable-length item within the same record, you must move them to another field prior to changing the length of the variable-length item, and then move them back after the length indicator has been changed.

Appendix E. Date and Time Callable Services Reference

CEECBLDY—Convert Date to COBOL Integer Format

CEECBLDY converts a string representing a date into a COBOL Integer format, which is the number of days since 31 December 1600. This service is similar to CEEDAYS, except that it provides a string in COBOL Integer format, which is compatible with ANSI intrinsic functions. Use CEECBLDY to access the century window of the date and time callable services and to perform date calculations with ANSI intrinsic functions.

Syntax

```
►►──CALL──"CEECBLDY"──USING──input_char_date──,──picture_string──,─────►
►──output_Integer_date──,──fc──.───────────────────────────────────►◄
```

input_char_date (input)

A halfword length-prefixed character string, representing a date or timestamp, in a format conforming to that specified by *picture_string*.

The character string must contain between 5 and 255 characters, inclusive. *input_char_date* can contain leading or trailing blanks. Parsing for a date begins with the first nonblank character (unless the picture string itself contains leading blanks, in which case CEECBLDY skips exactly that many positions before parsing begins).

After parsing a valid date, as determined by the format of the date specified in *picture_string*, CEECBLDY ignores all remaining characters. Valid dates range between and include 01 January 1601 to 31 December 9999.

See Figure 116 on page 500 for a list of valid picture character terms that can be specified in *input_char_date*.

picture_string (input)

A halfword length-prefixed character string, indicating the format of the date specified in *input_char_date*.

Each character in the *picture_string* corresponds to a character in *input_char_date*. For example, if you specify MMDDYY as the *picture_string*, CEECBLDY reads an *input_char_date* of 060288 as 02 June 1988.

If delimiters such as the slash (/) appear in the picture string, leading zeros can be omitted. For example, the following calls to CEECBLDY:

```
MOVE '6/2/88' TO DATEVAL-STRING.
MOVE 6 TO DATEVAL-LENGTH.
MOVE 'MM/DD/YY' TO PICSTR-STRING.
MOVE 8 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.
```

```
MOVE '06/02/88' TO DATEVAL-STRING.
MOVE 8 TO DATEVAL-LENGTH.
MOVE 'MM/DD/YY' TO PICSTR-STRING.
MOVE 8 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.

MOVE '060288' TO DATEVAL-STRING.
MOVE 6 TO DATEVAL-LENGTH.
MOVE 'MMDDYY' TO PICSTR-STRING.
MOVE 6 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.

MOVE '88154' TO DATEVAL-STRING.
MOVE 5 TO DATEVAL-LENGTH.
MOVE 'YYDDD' TO PICSTR-STRING.
MOVE 5 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.
```

would each assign the same value, 141502 (02 June 1988), to COBINTDTE.

Whenever characters such as colons or slashes are included in the *picture_string* (such as HH:MI:SS YY/MM/DD), they count as placeholders but are otherwise ignored.

See Figure 116 on page 500 for a list of valid picture character terms and Figure 117 on page 501 for examples of valid picture strings.

If *picture_string* includes a Japanese Era symbol <JJJJ>, the YY position in *input_char_date* is replaced by the year number within the Japanese Era. For example, the year 1988 equals the Japanese year 63 in the Showa era. See Figure 117 on page 501 for an additional example. See Figure 118 on page 502 for a list of Japanese Eras supported.

If *picture_string* includes an ROC (Republic of China) Era symbol <CCCC> or <CCCCCCCC>, the YY position in *input_char_date* is replaced by the year number within the ROC Era. For example, the year 1988 equals the ROC year 77 in the MinKow Era. See Figure 117 on page 501 for an additional example. See Figure 119 on page 502 for a list of ROC Eras supported.

output_Integer_date (output)

A 32-bit binary integer representing the COBOL Integer date, the number of days since 31 December 1600. For example, 16 May 1988 is day number 141485.

If *input_char_date* does not contain a valid date, *output_Integer_date* is set to 0 and CEECBLDY terminates with a non-CEE000 symbolic feedback code.

Date calculations are performed easily on the *output_Integer_date*, because *output_Integer_date* is an integer. Leap year and end-of-year anomalies do not affect the calculations.

fc (output)

A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2EB	3	2507	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
CEE2EC	3	2508	The date value passed to CEEDAYS or CEESECS was invalid.
CEE2ED	3	2509	The Japanese or Republic of China Era passed to CEEDAYS or CEESECS was not recognized.
CEE2EH	3	2513	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
CEE2EL	3	2517	The month value in a CEEISEC call was not recognized.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EO	3	2520	CEEDAYS detected non-numeric data in a numeric field, or the date string did not match the picture string.
CEE2EP	3	2521	The Japanese (<JJJJ>) or Chinese (<CCCC>) year-within-Era value passed to CEEDAYS or CEESECS was zero.

Usage Notes

- Call CEECBLDY only from COBOL programs that use the returned value as input to COBOL intrinsic functions. You should not use the returned value with other date and time callable services, nor should you call CEECBLDY from any non-COBOL programs. Unlike CEEDAYS, there is no inverse function of CEECBLDY, because it is only for COBOL users who want to use the date and time century window service together with COBOL intrinsic functions for date calculations. The inverse function of CEECBLDY is provided by the DATE-OF-INTEGER and DAY-OF-INTEGER intrinsic functions.
- To perform calculations on dates earlier than 1 January 1601, add 4000 to the year in each date, convert the dates to COBOL Integer format, then do the calculation. If the result of the calculation is a date, as opposed to a number of days, convert the result to a date string and subtract 4000 from the year.
- By default, 2-digit years lie within the 100-year range starting 80 years prior to the system date. Thus, in 1997, all 2-digit years represent dates between 1917 and 2016, inclusive. You can change this default range by using the CEESCEN callable service.

Example

```
CBL LIB,APOST
      *************************************************
      **                                             **
      ** Function: Invoke CEECBLDY callable service  **
      ** to convert date to COBOL Integer format.    **
      ** This service is used when using the         **
      ** Century Window feature of the date and time **
      ** callable services mixed with COBOL          **
      ** Intrinsic Functions.                        **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLDY.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  CHRDATE.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                               of CHRDATE.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                               of  PICSTR.
       01  INTEGER                 PIC S9(9) BINARY.
       01  NEWDATE                 PIC 9(8).
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                       REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
```

```
 PROCEDURE DIVISION.
 PARA-CBLDAYS.
*************************************************
** Specify input date and length               **
*************************************************
     MOVE 25 TO Vstring-length of CHRDATE.
     MOVE '1 January 00'
         to Vstring-text of CHRDATE.

*************************************************
** Specify a picture string that describes     **
** input date, and set the string's length.    **
*************************************************
     MOVE 23 TO Vstring-length of PICSTR.
     MOVE 'ZD Mmmmmmmmmmmmmmmz YY'
               TO Vstring-text of PICSTR.

*************************************************
** Call CEECBLDY to convert input date to a    **
** COBOL Integer date                          **
*************************************************
     CALL 'CEECBLDY' USING CHRDATE, PICSTR,
                           INTEGER, FC.

*************************************************
** If CEECBLDY runs successfully, then compute **
**     the date of the 90th day after the      **
**     input date using Intrinsic Functions    **
*************************************************
     IF CEE000 of FC  THEN
         COMPUTE INTEGER = INTEGER + 90
         COMPUTE NEWDATE = FUNCTION
             DATE-OF-INTEGER (INTEGER)
         DISPLAY NEWDATE
             ' is Lilian day: ' INTEGER
     ELSE
         DISPLAY 'CEEBLDY failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

CEEDATE—Convert Lilian Date to Character Format

CEEDATE converts a number representing a Lilian date to a date written in character format. The output is a character string, such as 1996/04/23.

Syntax

```
►►──CALL──"CEEDATE"──USING──input_Lilian_date──,──picture_string──,────►
►──output_char_date──,──fc──.──────────────────────────────────────────►◄
```

input_Lilian_date (input)

A 32-bit integer representing the Lilian date. The Lilian date is the number of days since 14 October 1582. For example, 16 May 1988 is Lilian day number 148138. The valid range of Lilian dates is 1 to 3,074,324 (15 October 1582 to 31 December 9999).

picture_string (input)

A halfword length-prefixed character string, representing the desired format of *output_char_date*, for example `MM/DD/YY`. Each character in *picture_string* represents a character in *output_char_date*. If delimiters such as the slash (/) appear in the picture string, they are copied as is to *output_char_date*.

See Figure 116 on page 500 for a list of valid picture characters, and Figure 117 on page 501 for examples of valid picture strings.

If *picture_string* includes a Japanese Era symbol `<JJJJ>`, the `YY` position in *output_char_date* is replaced by the year number within the Japanese Era. For example, the year 1988 equals the Japanese year 63 in the Showa era. See Figure 117 on page 501 for an additional example. See Figure 118 on page 502 for a list of Japanese Eras supported.

If *picture_string* includes an ROC (Republic of China) Era symbol `<CCCC>` or `<CCCCCCCC>`, the `YY` position in *output_char_date* is replaced by the year number within the ROC Era. For example, the year 1988 equals the ROC year 77 in the MinKow Era. See Figure 117 on page 501 for an additional example. See Figure 119 on page 502 for a list of ROC Eras supported.

output_char_date (output)

A fixed-length 80-character string that is the result of converting *input_Lilian_date* to the format specified by *picture_string*. See Figure 129 on page 566 for sample output dates. If *input_Lilian_date* is invalid, *output_char_date* is set to all blanks and CEEDATE terminates with a non-CEE000 symbolic feedback code.

fc (output)

A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2EG	3	2512	The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EQ	3	2522	Japanese (<JJJJ>) or Republic of China (<CCCC> or <CCCCCCCC>) Era was used in a picture string passed to CEEDATE, but the Lilian date value was not within the supported range. The era could not be determined.
CEE2EU	2	2526	The date string returned by CEEDATE was truncated.
CEE2F6	1	2534	Insufficient field width was specified for a month or weekday name in a call to CEEDATE or CEEDATM. Output set to blanks.

Usage Notes

- The inverse of CEEDATE is CEEDAYS, which converts character dates to the Lilian format.

Example

```
CBL LIB,APOST
      ************************************************
      **                                            **
      ** Function: CEEDATE - convert Lilian date to **
      **                     character format       **
      **                                            **
      ** In this example, a call is made to CEEDATE **
      ** to convert a Lilian date (the number of    **
      ** days since 14 October 1582) to a character **
      ** format (such as 6/22/88). The result is    **
      ** displayed.  The Lilian date is obtained    **
      ** via a call to CEEDAYS.                     **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLDATE.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  LILIAN                  PIC S9(9) BINARY.
```

```
 01  CHRDATE                 PIC X(80).
 01  IN-DATE.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char    PIC X
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                         of IN-DATE.
 01  PICSTR.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char    PIC X
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                        of PICSTR.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.

 PROCEDURE DIVISION.
 PARA-CBLDAYS.
*************************************************
** Call CEEDAYS to convert date of 6/2/88 to   **
**     Lilian representation                   **
*************************************************
     MOVE 6 TO Vstring-length of IN-DATE.
     MOVE '6/2/88' TO Vstring-text of IN-DATE(1:6).
     MOVE 8 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY' TO Vstring-text of PICSTR(1:8).
     CALL 'CEEDAYS' USING IN-DATE, PICSTR,
                          LILIAN, FC.

*************************************************
** If CEEDAYS runs successfully, display result**
*************************************************
     IF  CEE000 of FC  THEN
         DISPLAY Vstring-text of IN-DATE
             ' is Lilian day: ' LILIAN
     ELSE
         DISPLAY 'CEEDAYS failed with msg '
```

```
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

*************************************************
** Specify picture string that describes the   **
**  desired format of the output from CEEDATE, **
**  and the picture string's length.           **
*************************************************
     MOVE 23 TO Vstring-length OF PICSTR.
     MOVE 'ZD Mmmmmmmmmmmmmmz YYYY' TO
                  Vstring-text OF PICSTR(1:23).

*************************************************
** Call CEEDATE to convert the Lilian date     **
**     to  a picture string.                   **
*************************************************
     CALL 'CEEDATE' USING LILIAN, PICSTR,
                          CHRDATE, FC.

*************************************************
** If CEEDATE runs successfully, display result**
*************************************************
     IF CEE000 of FC  THEN
         DISPLAY 'Input Lilian date of ' LILIAN
             ' corresponds to:  ' CHRDATE
     ELSE
         DISPLAY 'CEEDATE failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

Figure 129 shows the sample output from CEEDATE.

Figure 129 (Page 1 of 2). Sample Output of CEEDATE

input_Lilian_date	**picture_string**	**output_char_date**
148138	`YY`	88
	`YYMM`	8805
	`YY-MM`	88-05
	`YYMMDD`	880516
	`YYYYMMDD`	19880516
	`YYYY-MM-DD`	1988-05-16
	`YYYY-ZM-ZD`	1988-5-16
	`<JJJJ> YY.MM.DD`	*Showa* 63.05.16 (in a DBCS string)
	`<CCCC> YY.MM.DD`	*MinKow* 77.05.16 (in a DBCS string)

Figure 129 (Page 2 of 2). Sample Output of CEEDATE

input_Lilian_date	picture_string	output_char_date
148139	`MM`	05
	`MMDD`	0517
	`MM/DD`	05/17
	`MMDDYY`	051788
	`MM/DD/YYYY`	05/17/1988
	`ZM/DD/YYYY`	5/17/1988
148140	`DD`	18
	`DDMM`	1805
	`DDMMYY`	180588
	`DD.MM.YY`	18.05.88
	`DD.MM.YYYY`	18.05.1988
	`DD Mmm YYYY`	18 May 1988
148141	`DDD`	140
	`YYDDD`	88140
	`YY.DDD`	88.140
	`YYYY.DDD`	1988.140
148142	`YY/MM/DD HH:MI:SS.99`	88/05/20 00:00:00.00
	`YYYY/ZM/ZD ZH:MI AP`	1988/5/20 0:00 AM
148143	`WWW., MMM DD, YYYY`	SAT., MAY 21, 1988
	`Www., Mmm DD, YYYY`	Sat., May 21, 1988

	`Wwwwwwwwww,`
	`Mmmmmmmmmm DD, YYYY`
		Saturday ,
		May 21, 1988

	`Wwwwwwwwwz,`
	`Mmmmmmmmmz DD, YYYY`
		Saturday, May 21, 1988

CEEDATM—Convert Seconds to Character Timestamp

CEEDATM converts a number representing the number of seconds since 00:00:00 14 October 1582 to a character string format. The format of the output is a character string timestamp, for example, `1988/07/26 20:37:00`.

Syntax

```
►►──CALL──"CEEDATM"──USING──input_seconds──,──picture_string──,──────►
►──output_timestamp──,──fc──.────────────────────────────────────────►◄
```

input_seconds (input)

A 64-bit double floating-point number representing the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). The valid range of *input_seconds* is 86,400 to 265,621,679,999.999 (23:59:59.999 31 December 9999).

picture_string (input)

A halfword length-prefixed character string, representing the desired format of *output_timestamp*, for example, `MM/DD/YY HH:MI AP`.

Each character in the *picture_string* represents a character in *output_timestamp*. If delimiters such as a slash (/) appear in the picture string, they are copied as is to *output_timestamp*.

See Figure 116 on page 500 for a list of valid picture character terms and Figure 117 on page 501 for examples of valid picture strings.

If *picture_string* includes the Japanese Era symbol `<JJJJ>`, the `YY` position in *output_timestamp* represents the year within Japanese Era. See Figure 117 on page 501 for an example. See Figure 118 on page 502 for a list of Japanese Eras supported.

If *picture_string* includes the ROC (Republic of China) Era symbol `<CCCC>` or `<CCCCCCCC>`, the `YY` position in *output_timestamp* represents the year within ROC Era. See Figure 117 on page 501 for an example. See Figure 119 on page 502 for a list of ROC Eras supported.

output_timestamp (output)

A fixed-length 80-character string that is the result of converting *input_seconds* to the format specified by *picture_string*.

If necessary, the output is truncated to the length of *output_timestamp*. See Figure 130 on page 571 for sample output.

If *input_seconds* is invalid, *output_timestamp* is set to all blanks and CEEDATM terminates with a non-CEE000 symbolic feedback code.

fc (output)

A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2E9	3	2505	The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.
CEE2EA	3	2506	Japanese (<JJJJ>) or Republic of China (<CCCC> or <CCCCCCCC>) Era was used in a picture string passed to CEEDATM, but the input number-of-seconds value was not within the supported range. The era could not be determined.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EV	2	2527	The timestamp string returned by CEEDATM was truncated.

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE2F6	1	2534	Insufficient field width was specified for a month or weekday name in a call to CEEDATE or CEEDATM. Output set to blanks.

Usage Notes

- The inverse of CEEDATM is CEESECS, which converts a timestamp to number of seconds.

Example

```
CBL LIB,APOST
      *************************************************
      **                                             **
      ** Function: CEEDATM - convert seconds to      **
      **                     character timestamp     **
      **                                             **
      ** In this example, a call is made to CEEDATM  **
      ** to convert a date represented in Lilian     **
      ** seconds (the number of seconds since        **
      ** 00:00:00 14 October 1582) to a character    **
      ** format (such as 06/02/88 10:23:45). The     **
      ** result is displayed.                        **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLDATM.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  DEST          PIC S9(9) BINARY VALUE 2.
       01  SECONDS                 COMP-2.
       01  IN-DATE.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
                         OCCURS 0 TO 256 TIMES
                         DEPENDING ON Vstring-length
                             of IN-DATE.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
```

```
     02  Vstring-text.
         03  Vstring-char    PIC X
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                        of PICSTR.
 01  TIMESTP                 PIC X(80).
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.

 PROCEDURE DIVISION.
 PARA-CBLDATM.
*************************************************
** Call CEESECS to convert timestamp of 6/2/88 **
**     at 10:23:45 AM to Lilian representation **
*************************************************
     MOVE 20 TO Vstring-length of IN-DATE.
     MOVE '06/02/88 10:23:45 AM'
         TO Vstring-text of IN-DATE.
     MOVE 20 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY HH:MI:SS AP'
         TO Vstring-text of PICSTR.
     CALL 'CEESECS' USING IN-DATE, PICSTR,
                          SECONDS, FC.

*************************************************
** If CEESECS runs successfully, display result**
*************************************************
     IF  CEE000 of FC  THEN
         DISPLAY Vstring-text of IN-DATE
             ' is Lilian second:  ' SECONDS
     ELSE
         DISPLAY 'CEESECS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.


*************************************************
** Specify desired format of the output.       **
```

```
*************************************************
     MOVE 35 TO Vstring-length OF PICSTR.
     MOVE 'ZD Mmmmmmmmmmmmmmz YYYY at HH:MI:SS'
             TO Vstring-text OF PICSTR.

*************************************************
** Call CEEDATM to convert Lilian seconds to   **
**     a character timestamp                   **
*************************************************
     CALL 'CEEDATM' USING SECONDS, PICSTR,
                          TIMESTP, FC.

*************************************************
** If CEEDATM runs successfully, display result**
*************************************************
     IF CEE000 of FC  THEN
         DISPLAY 'Input seconds of ' SECONDS
             ' corresponds to: ' TIMESTP
     ELSE
         DISPLAY 'CEEDATM failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

Figure 130 shows the sample output of CEEDATM.

Figure 130 (Page 1 of 2). Sample Output of CEEDATM

input_seconds	picture_string	output_timestamp
12,799,191,601.000	YYMMDD	880516
	HH:MI:SS	19:00:01
	YY-MM-DD	88-05-16
	YYMMDDHHMISS	880516190001
	YY-MM-DD HH:MI:SS	88-05-16 19:00:01
	YYYY-MM-DD HH:MI:SS AP	1988-05-16 07:00:01 PM
12,799,191,661.986	DD Mmm YY	16 May 88
	DD MMM YY HH:MM	16 MAY 88 19:01
	WWW, MMM DD, YYYY ZH:MI AP	MON, MAY 16, 1988 7:01 PM
	Wwwwwwwwwz, ZM/ZD/YY HH:MI:SS.99	Monday, 5/16/88 19:01:01.98

Figure 130 (Page 2 of 2). Sample Output of CEEDATM

input_seconds	picture_string	output_timestamp
12,799,191,662.009	YYYY	1988
	YY	88
	Y	8
	MM	05
	ZM	5
	RRRR	Vƀƀƀ
	MMM	MAY
	Mmm	May
	Mmmmmmmmmm	Mayƀƀƀƀƀƀ
	Mmmmmmmmmz	May
	DD	16
	ZD	16
	DDD	137
	HH	19
	ZH	19
	MI	01
	SS	02
	99	00
	999	009
	AP	PM
	WWW	MON
	Www	Mon
	Wwwwwwwwww	Mondayƀƀƀƀ
	Wwwwwwwwwz	Monday

CEEDAYS—Convert Date to Lilian Format

CEEDAYS converts a string representing a date into a Lilian format, which represents a date as the number of days from the beginning of the Gregorian calendar. CEEDAYS converts the specified *input_char_date* to a number representing the number of days since day zero in the Lilian format: Friday, 14 October, 1582.

Do not use CEEDAYS in combination with COBOL intrinsic functions. Use CEECBLDY for programs that use intrinsic functions.

Syntax

```
►►──CALL──"CEEDAYS"──USING──input_char_date──,──picture_string──,──────►

►──output_Lilian_date──,──fc──.────────────────────────────────────────►◄
```

input_char_date (input)

A halfword length-prefixed character string, representing a date or timestamp, in a format conforming to that specified by *picture_string*.

The character string must contain between 5 and 255 characters, inclusive. *input_char_date* can contain leading or trailing blanks. Parsing for a date begins with the first nonblank character (unless the picture string itself contains leading blanks, in which case CEEDAYS skips exactly that many positions before parsing begins).

After parsing a valid date, as determined by the format of the date specified in *picture_string*, CEEDAYS ignores all remaining characters. Valid dates range between and include 15 October 1582 to 31 December 9999.

See Figure 116 on page 500 for a list of valid picture character terms that can be specified in *input_char_date*.

picture_string (input)

A halfword length-prefixed character string, indicating the format of the date specified in *input_char_date*.

Each character in the *picture_string* corresponds to a character in *input_char_date*. For example, if you specify MMDDYY as the *picture_string*, CEEDAYS reads an *input_char_date* of 060288 as 02 June 1988.

If delimiters such as a slash (/) appear in the picture string, leading zeros can be omitted. For example, the following calls to CEEDAYS:

```
CALL CEEDAYS USING '6/2/88'  , 'MM/DD/YY', lildate, fc.
CALL CEEDAYS USING '06/02/88', 'MM/DD/YY', lildate, fc.
CALL CEEDAYS USING '060288'  , 'MMDDYY'  , lildate, fc.
CALL CEEDAYS USING '88154'   , 'YYDDD'   , lildate, fc.
```

would each assign the same value, 148155 (02 June 1988), to *lildate*.

Whenever characters such as colons or slashes are included in the *picture_string* (such as `HH:MI:SS YY/MM/DD`), they count as placeholders but are otherwise ignored.

See Figure 116 on page 500 for a list of valid picture character terms, and Figure 117 on page 501 for examples of valid picture strings.

If *picture_string* includes a Japanese Era symbol `<JJJJ>`, the `YY` position in *input_char_date* is replaced by the year number within the Japanese Era. For example, the year 1988 equals the Japanese year 63 in the Showa era. See Figure 117 on page 501 for an additional example. See Figure 118 on page 502 for a list of Japanese Eras supported.

If *picture_string* includes an ROC (Republic of China) Era symbol `<CCCC>` or `<CCCCCCCC>`, the `YY` position in *input_char_date* is replaced by the year number within the ROC Era. For example, the year 1988 equals the ROC year 77 in the MinKow Era. See Figure 117 on page 501 for an additional example. See Figure 119 on page 502 for a list of ROC Eras supported.

output_Lilian_date (output)

A 32-bit binary integer representing the Lilian date, the number of days since 14 October 1582. For example, 16 May 1988 is day number 148138.

If *input_char_date* does not contain a valid date, *output_Lilian_date* is set to 0 and CEEDAYS terminates with a non-CEE000 symbolic feedback code.

Date calculations are performed easily on the *output_Lilian_date*, because it is an integer. Leap year and end-of-year anomalies do not affect the calculations.

fc (output)

A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2EB	3	2507	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
CEE2EC	3	2508	The date value passed to CEEDAYS or CEESECS was invalid.
CEE2ED	3	2509	The Japanese or Republic of China Era passed to CEEDAYS or CEESECS was not recognized.
CEE2EH	3	2513	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
CEE2EL	3	2517	The month value in a CEEISEC call was not recognized.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EO	3	2520	CEEDAYS detected non-numeric data in a numeric field, or the date string did not match the picture string.
CEE2EP	3	2521	The Japanese (<JJJJ>) or Chinese (<CCCC>) year-within-Era value passed to CEEDAYS or CEESECS was zero.

Usage Notes

- The inverse of CEEDAYS is CEEDATE, which converts *output_Lilian_date* from Lilian format to character format.
- To perform calculations on dates earlier than 15 October 1582, add 4000 to the year in each date, convert the dates to Lilian, then do the calculation. If the result of the calculation is a date, as opposed to a number of days, convert the result to a date string and subtract 4000 from the year.
- By default, 2-digit years lie within the 100-year range starting 80 years prior to the system date. Thus, in 1997, all 2-digit years represent dates between 1917 and 2016, inclusive. This default range is changed by using the callable service CEESCEN.
- Date calculations can be performed easily on the *output_Lilian_date*, because it is an integer. Leap year and end-of-year anomalies are avoided.

Example

```
CBL LIB,APOST
      *******************************************
      **                                       **
      ** Function: CEEDAYS - convert date to   **
      **                     Lilian format     **
      **                                       **
      *******************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLDAYS.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  CHRDATE.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                               of CHRDATE.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                               of PICSTR.
       01  LILIAN                  PIC S9(9) BINARY.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.

       PROCEDURE DIVISION.
       PARA-CBLDAYS.
      *************************************************
      ** Specify input date and length               **
```

```
*************************************************
     MOVE 16 TO Vstring-length of CHRDATE.
     MOVE '1 January 2000'
         TO Vstring-text of CHRDATE.

*************************************************
** Specify a picture string that describes     **
** input date, and the picture string's length.**
*************************************************
     MOVE 25 TO Vstring-length of PICSTR.
     MOVE 'ZD Mmmmmmmmmmmmmmz YYYY'
             TO Vstring-text of PICSTR.

*************************************************
** Call CEEDAYS to convert input date to a     **
** Lilian date                                 **
*************************************************
     CALL 'CEEDAYS' USING CHRDATE, PICSTR,
                          LILIAN, FC.

*************************************************
** If CEEDAYS runs successfully, display result**
*************************************************
     IF  CEE000 of FC  THEN
         DISPLAY Vstring-text of CHRDATE
             ' is Lilian day: ' LILIAN
     ELSE
         DISPLAY 'CEEDAYS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

CEEDYWK—Calculate Day of Week from Lilian Date

CEEDYWK calculates the day of the week on which a Lilian date falls. The day of the week is returned to the calling routine as a number between 1 and 7.

The number returned by CEEDYWK is useful for end-of-week calculations.

Syntax

```
►►──CALL──"CEEDYWK"──USING──input_Lilian_date──,──output_day_no──,────►
►──fc──.──────────────────────────────────────────────────────────────►◄
```

input_Lilian_date (input)

A 32-bit binary integer representing the Lilian date, the number of days since 14 October 1582.

For example, 16 May 1988 is day number 148138. The valid range of *input_Lilian_date* is between 1 and 3,074,324 (15 October 1582 and 31 December 9999).

output_day_no (output)

A 32-bit binary integer representing *input_Lilian_date*'s day-of-week: 1 equals Sunday, 2 equals Monday, ..., 7 equals Saturday.

If *input_Lilian_date* is invalid, *output_day_no* is set to 0 and CEEDYWK terminates with a non-CEE000 symbolic feedback code.

fc (output)

A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2EG	3	2512	The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.

Example

```
CBL LIB,APOST
      *************************************************
      **                                             **
      ** Function: Call CEEDYWK to calculate the     **
      **           day of the week from Lilian date **
      **                                             **
      ** In this example, a call is made to CEEDYWK **
      ** to return the day of the week on which a   **
      ** Lilian date falls. (A Lilian date is the   **
      ** number of days since 14 October 1582)      **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLDYWK.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  LILIAN                  PIC S9(9) BINARY.
       01  DAYNUM                  PIC S9(9) BINARY.
       01  IN-DATE.
```

```
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char        PIC X,
                 OCCURS 0 TO 256 TIMES
                 DEPENDING ON Vstring-length
                     of IN-DATE.
 01  PICSTR.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char        PIC X,
                 OCCURS 0 TO 256 TIMES
                 DEPENDING ON Vstring-length
                     of PICSTR.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
               REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.

 PROCEDURE DIVISION.
 PARA-CBLDAYS.
** Call CEEDAYS to convert date of 6/2/88 to
**     Lilian representation
     MOVE 6 TO Vstring-length of IN-DATE.
     MOVE '6/2/88' TO Vstring-text of IN-DATE(1:6).
     MOVE 8 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY' TO Vstring-text of PICSTR(1:8).
     CALL 'CEEDAYS' USING IN-DATE, PICSTR,
         LILIAN, FC.

** If CEEDAYS runs successfully, display result.
     IF  CEE000 of FC  THEN
         DISPLAY Vstring-text of IN-DATE
             ' is Lilian day: ' LILIAN
     ELSE
         DISPLAY 'CEEDAYS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

 PARA-CBLDYWK.
```

```
** Call CEEDYWK to return the day of the week on
** which the Lilian date falls
     CALL 'CEEDYWK' USING LILIAN , DAYNUM , FC.

** If CEEDYWK runs successfully, print results
     IF CEE000 of FC  THEN
         DISPLAY 'Lilian day ' LILIAN
             ' falls on day ' DAYNUM
             ' of the week, which is a:'
** Select DAYNUM to display the name of the day
**     of the week.
         EVALUATE DAYNUM
           WHEN 1
             DISPLAY 'Sunday.'
           WHEN 2
             DISPLAY 'Monday.'
           WHEN 3
             DISPLAY 'Tuesday'
           WHEN 4
             DISPLAY 'Wednesday.'
           WHEN 5
             DISPLAY 'Thursday.'
           WHEN 6
             DISPLAY 'Friday.'
           WHEN 7
             DISPLAY 'Saturday.'
         END-EVALUATE
     ELSE
         DISPLAY 'CEEDYWK failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

CEEGMT—Get Current Greenwich Mean Time

CEEGMT returns the current Greenwich Mean Time (GMT) as both a Lilian date and as the number of seconds since 00:00:00 14 October 1582. GMT is also known as Coordinated Universal Time (UTC). The returned values are compatible with those generated and used by the other date and time callable services.

CEEGMT

Syntax

```
►►──CALL──"CEEGMT"──USING──output_GMT_Lilian──,──output_GMT_seconds────►
►──,──fc──.────────────────────────────────────────────────────────────►◄
```

output_GMT_Lilian (output)

A 32-bit binary integer representing the current date in Greenwich, England, in the Lilian format (the number of days since 14 October 1582).

For example, 16 May 1988 is day number 148138. If GMT is not available from the system, *output_GMT_Lilian* is set to 0 and CEEGMT terminates with a non-CEE000 symbolic feedback code.

output_GMT_seconds (output)

A 64-bit double floating-point number representing the current date and time in Greenwich, England, as the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). 19:00:01.078 on 16 May 1988 is second number 12,799,191,601.078. If GMT is not available from the system, *output_GMT_seconds* is set to 0 and CEEGMT terminates with a non-CEE000 symbolic feedback code.

fc (output)

A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2E6	3	2502	The UTC/GMT was not available from the system.

Usage Notes

- CEEDATE converts *output_GMT_Lilian* to a character date, and CEEDATM converts *output_GMT_seconds* to a character timestamp.
- In order for the results of this service to be meaningful, your system's clock must be set to the local time and the environment variable TZ must be set correctly. See “Setting Environment Variables” on page 134 for details on how to set environment variables and “Run-Time Environment Variables” on page 137 for specific information about the TZ environment variable.
- The values returned by CEEGMT are handy for elapsed time calculations. For example, you can calculate the time elapsed between two calls to CEEGMT by calculating the differences between the returned values.
- CEEUTC is identical to this service.

Example

```
CBL LIB,APOST
      *************************************************
      **                                             **
      ** Function: Call CEEGMT to get current        **
      **           Greenwich Mean Time               **
      **                                             **
      ** In this example, a call is made to CEEGMT   **
      ** to return the current GMT as a Lilian date  **
      ** and as Lilian seconds. The results are      **
      ** displayed.                                  **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. IGZTGMT.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  LILIAN                  PIC S9(9) BINARY.
       01  SECS                    COMP-2.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
       PROCEDURE DIVISION.
       PARA-CBLGMT.
           CALL 'CEEGMT' USING LILIAN , SECS , FC.

           IF CEE000 of FC  THEN
               DISPLAY 'The current GMT is also '
                   'known as Lilian day: ' LILIAN
               DISPLAY 'The current GMT in Lilian '
```

```
        'seconds is: ' SECS
ELSE
    DISPLAY 'CEEGMT failed with msg '
        Msg-No of FC UPON CONSOLE
    STOP RUN
END-IF.

GOBACK.
```

CEEGMTO—Get Offset from Greenwich Mean Time to Local Time

CEEGMTO returns values to the calling routine representing the difference between the local system time and Greenwich Mean Time (GMT).

Syntax

```
►►──CALL──"CEEGMTO"──USING──offset_hours──,──offset_minutes──,────────►
►──offset_seconds──,──fc──.──────────────────────────────────────────►◄
```

offset_hours (output)

A 32-bit binary integer representing the offset from GMT to local time, in hours.

For example, for Pacific Standard Time, *offset_hours* equals -8.

The range of *offset_hours* is -12 to +13 (+13 = Daylight Savings Time in the +12 time zone).

If local time offset is not available, *offset_hours* equals 0 and CEEGMTO terminates with a non-CEE000 symbolic feedback code.

offset_minutes (output)

A 32-bit binary integer representing the number of additional minutes that local time is ahead of or behind GMT.

The range of *offset_minutes* is 0 to 59.

If the local time offset is not available, *offset_minutes* equals 0 and CEEGMTO terminates with a non-CEE000 symbolic feedback code.

offset_seconds (output)

A 64-bit double floating-point number representing the offset from GMT to local time, in seconds.

For example, Pacific Standard Time is eight hours behind GMT. If local time is in the Pacific time zone during standard time, CEEGMTO would return -28,800 (-8 * 60 * 60). The range of *offset_seconds* is -43,200 to +46,800. *offset_seconds* can be used with CEEGMT to calculate local date and time. See "CEEGMT—Get Current Greenwich Mean Time" on page 579 for more information.

If the local time offset is not available from the system, *offset_seconds* is set to 0 and CEEGMTO terminates with a non-CEE000 symbolic feedback code.

fc (output)

A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2E7	3	2503	The offset from UTC/GMT to local time was not available from the system.

Usage Notes

- CEEDATM is used to convert *offset_seconds* to a character timestamp.
- In order for the results of this service to be meaningful, your system's clock must be set to the local time and the environment variable TZ must be set correctly. See "Setting Environment Variables" on page 134 for details on how to set environment variables and "Run-Time Environment Variables" on page 137 for specific information about the TZ environment variable.

Example

```
CBL LIB,APOST
      *************************************************
      **                                             **
      ** Function:  Call CEEGMTO to get offset from  **
      **            Greenwich Mean Time to local     **
      **            time                             **
      **                                             **
      ** In this example, a call is made to CEEGMTO  **
      ** to return the offset from GMT to local time **
      ** as separate binary integers representing    **
      ** offset hours, minutes, and seconds. The     **
      ** results are displayed.                      **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. IGZTGMTO.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
```

```
01  HOURS                  PIC S9(9) BINARY.
01  MINUTES                PIC S9(9) BINARY.
01  SECONDS COMP-2.
01  FC.
    02  Condition-Token-Value.
    COPY  CEEIGZCT.
        03  Case-1-Condition-ID.
            04  Severity    PIC S9(4) COMP.
            04  Msg-No      PIC S9(4) COMP.
        03  Case-2-Condition-ID
                  REDEFINES Case-1-Condition-ID.
            04  Class-Code  PIC S9(4) COMP.
            04  Cause-Code  PIC S9(4) COMP.
        03  Case-Sev-Ctl    PIC X.
        03  Facility-ID     PIC XXX.
    02  I-S-Info           PIC S9(9) COMP.
PROCEDURE DIVISION.
PARA-CBLGMTO.
    CALL 'CEEGMTO' USING HOURS , MINUTES ,
        SECONDS , FC.

    IF CEE000 of FC  THEN
        DISPLAY 'Local time differs from GMT '
            'by: ' HOURS ' hours, '
            MINUTES ' minutes, OR  '
            SECONDS ' seconds. '
    ELSE
        DISPLAY 'CEEGMTO failed with msg '
            Msg-No of FC UPON CONSOLE
        STOP RUN
    END-IF.

    GOBACK.
```

CEEISEC—Convert Integers to Seconds

CEEISEC converts separate binary integers representing year, month, day, hour, minute, second, and millisecond to a number representing the number of seconds since 00:00:00 14 October 1582. Use CEEISEC instead of CEESECS when the input is in numeric format rather than character format.

Syntax

```
►►──CALL──"CEEISEC"──USING──input_year──,──input_months──,──────────────►
►──input_day──,──input_hours──,──input_minutes──,──input_seconds──,─────►
►──input_milliseconds──,──output_seconds──,──fc──.─────────────────────►◄
```

input_year (input)

A 32-bit binary integer representing the year.

The range of valid values for *input_year* is 1582 to 9999, inclusive.

input_month (input)
A 32-bit binary integer representing the month.

The range of valid values for *input_month* is 1 to 12.

input_day (input)
A 32-bit binary integer representing the day.

The range of valid values for *input_day* is 1 to 31.

input_hours (input)
A 32-bit binary integer representing the hours.

The range of valid values for *input_hours* is 0 to 23.

input_minutes (input)
A 32-bit binary integer representing the minutes.

The range of valid values for *input_minutes* is 0 to 59.

input_seconds (input)
A 32-bit binary integer representing the seconds.

The range of valid values for *input_seconds* is 0 to 59.

input_milliseconds (input)
A 32-bit binary integer representing milliseconds.

The range of valid values for *input_milliseconds* is 0 to 999.

output_seconds (output)
A 64-bit double floating-point number representing the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). The valid range of *output_seconds* is 86,400 to 265,621,679,999.999 (23:59:59.999 31 December 9999).

If any input values are invalid, *output_seconds* is set to zero.

To convert *output_seconds* to a Lilian day number, divide *output_seconds* by 86,400 (the number of seconds in a day).

fc (output)
A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2EE	3	2510	The hours value in a call to CEEISEC or CEESECS was not recognized.

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE2EF	3	2511	The day parameter passed in a CEEISEC call was invalid for year and month specified.
CEE2EH	3	2513	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
CEE2EI	3	2514	The year value passed in a CEEISEC call was not within the supported range.
CEE2EJ	3	2515	The milliseconds value in a CEEISEC call was not recognized.
CEE2EK	3	2516	The minutes value in a CEEISEC call was not recognized.
CEE2EL	3	2517	The month value in a CEEISEC call was not recognized.
CEE2EN	3	2519	The seconds value in a CEEISEC call was not recognized.

Usage Notes

- The inverse of CEEISEC is CEESECI, which converts number of seconds to integer year, month, day, hour, minute, second, and millisecond.

Example

```
CBL LIB,APOST
      *************************************************
      **                                             **
      ** Function: Call CEEISEC to convert integers  **
      **          to seconds                         **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLISEC.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  YEAR                    PIC S9(9) BINARY.
       01  MONTH                   PIC S9(9) BINARY.
       01  DAYS                    PIC S9(9) BINARY.
```

```
 01  HOURS                  PIC S9(9) BINARY.
 01  MINUTES                PIC S9(9) BINARY.
 01  SECONDS                PIC S9(9) BINARY.
 01  MILLSEC                PIC S9(9) BINARY.
 01  OUTSECS                COMP-2.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info           PIC S9(9) COMP.
 PROCEDURE DIVISION.
 PARA-CBLISEC.
*************************************************
** Specify seven binary integers representing  **
** the date and time as input to be converted  **
** to Lilian seconds                           **
*************************************************
     MOVE 2000 TO YEAR.
     MOVE 1 TO MONTH.
     MOVE 1 TO DAYS.
     MOVE 0 TO HOURS.
     MOVE 0 TO MINUTES.
     MOVE 0 TO SECONDS.
     MOVE 0 TO MILLSEC.
*************************************************
** Call CEEISEC to convert the integers        **
** to seconds                                  **
*************************************************
     CALL 'CEEISEC' USING YEAR, MONTH, DAYS,
                          HOURS, MINUTES, SECONDS,
                          MILLSEC, OUTSECS , FC.
*************************************************
** If CEEISEC runs successfully, display result**
*************************************************
     IF CEE000 of FC  THEN
         DISPLAY MONTH '/' DAYS '/' YEAR
           ' AT ' HOURS ':' MINUTES ':' SECONDS
```

```
      ' is equivalent to ' OUTSECS ' seconds'
ELSE
    DISPLAY 'CEEISEC failed with msg '
      Msg-No of FC UPON CONSOLE
    STOP RUN
END-IF.

GOBACK.
```

CEELOCT—Get Current Local Date or Time

CEELOCT returns the current local date or time in three formats:

- Lilian date (the number of days since 14 October 1582)
- Lilian seconds (the number of seconds since 00:00:00 14 October 1582)
- Gregorian character string (in the form YYYYMMDDHHMISS999).

These values are compatible with other date and time callable services, and with existing language intrinsic functions.

CEELOCT performs the same function as calling the CEEGMT, CEEGMTO, and CEEDATM date and time services separately. CEELOCT, however, performs the same services with much greater speed.

Syntax

```
►►──CALL──"CEELOCT"──USING──output_Lilian──,──output_seconds──,────────►
►──output_Gregorian──,──fc──.──────────────────────────────────────────►◄
```

output_Lilian (output)
A 32-bit binary integer representing the current local date in the Lilian format, that is, day 1 equals 15 October 1582, day 148,887 equals 4 June 1990.

If the local time is not available from the system, *output_Lilian* is set to 0 and CEELOCT terminates with a non-CEE000 symbolic feedback code.

output_seconds (output)
A 64-bit double-floating point number representing the current local date and time as the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds. For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). 19:00:01.078 on 4 June 1990 is second number 12,863,905,201.078.

If the local time is not available from the system, *output_seconds* is set to 0 and CEELOCT terminates with a non-CEE000 symbolic feedback code.

output_Gregorian (output)
A 17-byte fixed-length character string in the form YYYYMMDDHHMISS999 representing local year, month, day, hour, minute, second, and millisecond.

If the format of *output_Gregorian* does not meet your needs, you can use the CEEDATM callable service to convert *output_seconds* to another format.

fc (output)

A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2F3	3	2531	The local time was not available from the system.

Usage Notes

- You can use the CEEGMT callable service to determine Greenwich Mean Time (GMT).
- You can use the CEEGMTO callable service to obtain the offset from GMT to local time.
- The character value returned by CEELOCT is designed to match that produced by existing intrinsic functions. The numeric values returned can be used to simplify date calculations.

Example

```
CBL LIB,APOST
      ************************************************
      **                                            **
      ** Function: Call CEELOCT to get current      **
      **           local time                       **
      **                                            **
      ** In this example, a call is made to CEELOCT **
      ** to return the current local time in Lilian **
      ** days (the number of days since 14 October  **
      ** 1582), Lilian seconds (the number of       **
      ** seconds since 00:00:00 14 October 1582),   **
      ** and a Gregorian string (in the form        **
      ** YYYMMDDMISS999). The Gregorian character   **
      ** string is then displayed.                  **
   **                                            **
   ************************************************
    IDENTIFICATION DIVISION.
    PROGRAM-ID. CBLLOCT.
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01  LILIAN                  PIC S9(9) BINARY.
    01  SECONDS                 COMP-2.
    01  GREGORN                 PIC X(17).
    01  FC.
        02  Condition-Token-Value.
        COPY  CEEIGZCT.
            03  Case-1-Condition-ID.
                04  Severity    PIC S9(4) COMP.
                04  Msg-No      PIC S9(4) COMP.
            03  Case-2-Condition-ID
                      REDEFINES Case-1-Condition-ID.
                04  Class-Code  PIC S9(4) COMP.
                04  Cause-Code  PIC S9(4) COMP.
            03  Case-Sev-Ctl    PIC X.
            03  Facility-ID     PIC XXX.
        02  I-S-Info            PIC S9(9) COMP.
    PROCEDURE DIVISION.
    PARA-CBLLOCT.
        CALL 'CEELOCT' USING LILIAN, SECONDS,
                             GREGORN, FC.
   ************************************************
      ** If CEELOCT runs successfully, display      **
          **     Gregorian character string             **
          ************************************************
        IF CEE000 of FC  THEN
            DISPLAY 'Local Time is ' GREGORN
        ELSE
            DISPLAY 'CEELOCT failed with msg '
                Msg-No of FC UPON CONSOLE
            STOP RUN
        END-IF.

        GOBACK.
```

CEEQCEN—Query the Century Window

CEEQCEN queries the century which is a 2-digit year value. When you want to change the setting, use CEEQCEN to get the setting and then use CEESCEN to save and restore the current setting.

Syntax

```
►►──CALL──"CEEQCEN"──USING──century_start──,──fc──.──────────────────►◄
```

century_start (output)

An integer between 0 and 100 indicating the year on which the century window is based.

For example, if the date and time callable services default is in effect, all 2-digit years lie within the 100-year window starting 80 years prior to the system date. CEEQCEN then returns the value 80. An 80 value indicates to the date and time callable services that, in 1997, all 2-digit years lie within the 100-year window starting 80 years before the system date (between 1917 and 2016, inclusive).

fc (output)

A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.

Example

```
CBL LIB,APOST
      *************************************************
      **                                             **
      ** Function: Call CEEQCEN to query the         **
      **           date and time callable services   **
      **           century window                    **
      **                                             **
      ** In this example, CEEQCEN is called to query **
      ** the date at which the century window starts **
      ** The century window is the 100-year window   **
      ** within which the date and time callable     **
      ** services assume all two-digit years lie.    **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
```

```
 PROGRAM-ID. CBLQCEN.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  STARTCW                 PIC S9(9) BINARY.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.
 PROCEDURE DIVISION.

 PARA-CBLQCEN.
*************************************************
** Call CEEQCEN to return the start of the     **
**     century window                          **
*************************************************

     CALL 'CEEQCEN' USING STARTCW, FC.
*************************************************
** CEEQCEN has no non-zero feedback codes to   **
**     check, so just display result.          **
*************************************************
     IF CEE000 of FC  THEN
         DISPLAY 'The start of the century '
             'window is: ' STARTCW
     ELSE
         DISPLAY 'CEEQCEN failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

CEESCEN—Set the Century Window

CEESCEN sets the century to a 2-digit year value for use by other date and time callable services. Use it in conjunction with CEEDAYS or CEESECS when:

- You process date values containing 2-digit years (for example, in the YYMMDD format).
- The default century interval does not meet the requirements of a particular application.

To query the century window, use CEEQCEN.

Syntax

```
►►──CALL──"CEESCEN"──USING──century_start──,──fc──.──────────────────►◄
```

century_start

An integer between 0 and 100, setting the century window.

A value of 80, for example, places all two-digit years within the 100-year window starting 80 years before the system date. In 1997, therefore, all two-digit years are assumed to represent dates between 1917 and 2016, inclusive.

fc (output)

A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2E6	3	2502	The UTC/GMT was not available from the system.
CEE2F5	3	2533	The value passed to CEESCEN was not between 0 and 100.

Example

```
CBL LIB,APOST
      *************************************************
      **                                             **
      ** Function: Call CEESCEN to set the           **
      **           date and time callable services   **
      **           century window                    **
      **                                             **
      ** In this example, CEESCEN is called to change **
      ** the start of the century window to 30 years  **
      ** before the system date. CEEQCEN is then      **
      ** called to query that the change made.  A     **
      ** message that this has been done is then      **
      ** displayed.                                   **
      **                                             **
      *************************************************
```

CEESCEN

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CBLSCEN.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  STARTCW                 PIC S9(9) BINARY.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.
 PROCEDURE DIVISION.
 PARA-CBLSCEN.
**************************************************
** Specify 30 as century start, and two-digit
**     years will be assumed to lie in the
**     100-year window starting 30 years before
**     the system date.
**************************************************
     MOVE 30 TO STARTCW.

**************************************************
** Call CEESCEN to change the start of the century
**     window.
**************************************************
     CALL 'CEESCEN' USING STARTCW, FC.
     IF NOT CEE000 of FC  THEN
         DISPLAY 'CEESCEN failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

 PARA-CBLQCEN.
**************************************************
** Call CEEQCEN to return the start of the century
**     window
**************************************************
     CALL 'CEEQCEN' USING STARTCW, FC.

**************************************************
** CEEQCEN has no non-zero feedback codes to
**     check, so just display result.
```

```
**************************************************
         DISPLAY 'The start of the century '
             'window is: ' STARTCW
     GOBACK.
```

CEESECI—Convert Seconds to Integers

CEESECI converts a number representing the number of seconds since 00:00:00 14 October 1582 to seven separate binary integers representing year, month, day, hour, minute, second, and millisecond. Use CEESECI instead of CEEDATM when the output is needed in numeric format rather than in character format.

Syntax

```
►►──CALL──"CEESECI"──USING──input_seconds──,──output_year──,──────────────►
►──output_month──,──output_day──,──output_hours──,──output_minutes───────►
►──,──output_seconds──,──output_milliseconds──,──fc──.──────────────────►◄
```

input_seconds

A 64-bit double floating-point number representing the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). The range of valid values for *input_seconds* is 86,400 to 265,621,679,999.999 (23:59:59.999 31 December 9999).

If *input_seconds* is invalid, all output parameters except the feedback code are set to 0.

output_year (output)

A 32-bit binary integer representing the year.

The range of valid values for *output_year* is 1582 to 9999, inclusive.

output_month (output)

A 32-bit binary integer representing the month.

The range of valid values for *output_month* is 1 to 12.

output_day (output)

A 32-bit binary integer representing the day.

The range of valid values for *output_day* is 1 to 31.

output_hours (output)

A 32-bit binary integer representing the hour.

The range of valid values for *output_hours* is 0 to 23.

output_minutes (output)

A 32-bit binary integer representing the minutes.

The range of valid values for *output_minutes* is 0 to 59.

output_seconds (output)
A 32-bit binary integer representing the seconds.

The range of valid values for *output_seconds* is 0 to 59.

output_milliseconds (output)
A 32-bit binary integer representing milliseconds.

The range of valid values for *output_milliseconds* is 0 to 999.

fc (output)
A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2E9	3	2505	The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.

Usage Notes

- The inverse of CEESECI is CEEISEC, which converts separate binary integers representing year, month, day, hour, second, and millisecond to a number of seconds.
- If the input value is a Lilian date instead of seconds, multiply the Lilian date by 86,400 (number of seconds in a day), and pass the new value to CEESECI.

Example

```
CBL LIB,APOST
      *************************************************
      **                                             **
      ** Function: Call CEESECI to convert seconds   **
      **           to integers                       **
      **                                             **
      ** In this example a call is made to CEESECI   **
      ** to convert a number representing the number **
      ** of seconds since 00:00:00 14 October 1582   **
** to seven binary integers representing year, **
** month, day, hour, minute, second, and       **
** millisecond.  The results are displayed in  **
** this example.                               **
**                                             **
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CBLSECI.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  INSECS                  COMP-2.
 01  YEAR                    PIC S9(9) BINARY.
 01  MONTH                   PIC S9(9) BINARY.
 01  DAYS                    PIC S9(9) BINARY.
 01  HOURS                   PIC S9(9) BINARY.
 01  MINUTES                 PIC S9(9) BINARY.
 01  SECONDS                 PIC S9(9) BINARY.
 01  MILLSEC                 PIC S9(9) BINARY.
 01  IN-DATE.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char        PIC X,
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                         of IN-DATE.
 01  PICSTR.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char        PIC X,
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                        of PICSTR.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.
 PROCEDURE DIVISION.
 PARA-CBLSECS.
*************************************************
** Call CEESECS to convert timestamp of 6/2/88
**     at 10:23:45 AM to Lilian representation
*************************************************
     MOVE 20 TO Vstring-length of IN-DATE.
     MOVE '06/02/88 10:23:45 AM'
             TO Vstring-text of IN-DATE.
     MOVE 20 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY HH:MI:SS AP'
             TO Vstring-text of PICSTR.
     CALL 'CEESECS' USING IN-DATE, PICSTR,
                          INSECS, FC.
     IF NOT CEE000 of FC  THEN
         DISPLAY 'CEESECS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

 PARA-CBLSECI.
*************************************************
** Call CEESECI to convert seconds to integers
*************************************************
     CALL 'CEESECI' USING INSECS, YEAR, MONTH,
                          DAYS, HOURS,  MINUTES,
                          SECONDS, MILLSEC, FC.
*************************************************
** If CEESECI runs successfully, display results
*************************************************
     IF CEE000 of FC  THEN
         DISPLAY 'Input seconds of ' INSECS
             ' represents:'
         DISPLAY '   Year......... ' YEAR
         DISPLAY '   Month........ ' MONTH
         DISPLAY '   Day.......... ' DAYS
         DISPLAY '   Hour......... ' HOURS
         DISPLAY '   Minute....... ' MINUTES
         DISPLAY '   Second....... ' SECONDS
         DISPLAY '   Millisecond.. ' MILLSEC
     ELSE
         DISPLAY 'CEESECI failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

CEESECS—Convert Timestamp to Seconds

CEESECS converts a string representing a timestamp into the number of Lilian seconds (number of seconds since 00:00:00 14 October 1582). This service makes it easier to perform time arithmetic, such as calculating the elapsed time between two timestamps.

Syntax

```
►►──CALL──"CEESECS"──USING──input_timestamp──,──picture_string──,──────►
►──output_seconds──,──fc──.───────────────────────────────────────────►◄
```

input_timestamp (input)

A halfword length-prefixed character string, representing a date or timestamp in a format matching that specified by *picture_string*.

The character string must contain between 5 and 80 picture characters, inclusive. *input_timestamp* can contain leading or trailing blanks. Parsing begins with the first nonblank character (unless the picture string itself contains leading blanks; in this case, CEESECS skips exactly that many positions before parsing begins).

After a valid date is parsed, as determined by the format of the date you specify in *picture_string*, all remaining characters are ignored by CEESECS. Valid dates range between and including the dates 15 October 1582 to 31 December 9999. A full date must be specified. Valid times range from 00:00:00.000 to 23:59:59.999.

If any part or all of the time value is omitted, zeros are substituted for the remaining values. For example:

```
1992-05-17-19:02 is equivalent to 1992-05-17-19:02:00
1992-05-17       is equivalent to 1992-05-17-00:00:00
```

picture_string (input)

A halfword length-prefixed character string, indicating the format of the date or timestamp value specified in *input_timestamp*.

Each character in the *picture_string* represents a character in *input_timestamp*. For example, if you specify MMDDYY HH.MI.SS as the *picture_string*, CEESECS reads an *input_char_date* of 060288 15.35.02 as 3:35:02 PM on 02 June 1988. If delimiters such as the slash (/) appear in the picture string, leading zeros can be omitted. For example, the following calls to CEESECS all assign the same value to variable *secs*:

```
CALL CEESECS USING '92/06/03 15.35.03',
                   'YY/MM/DD HH.MI.SS', secs, fc.
CALL CEESECS USING '92/6/3 15.35.03',
                   'YY/MM/DD HH.MI.SS', secs, fc.
CALL CEESECS USING '92/6/3 3.35.03 PM',
                   'YY/MM/DD HH.MI.SS AP', secs, fc.
CALL CEESECS USING '92.155 3.35.03 pm',
                   'YY.DDD   HH.MI.SS AP', secs, fc.
```

If *picture_string* includes a Japanese era symbol `<JJJJ>`, the `YY` position in *input_timestamp* represents the year number within the Japanese era. For example, the year 1988 equals the Japanese year 63 in the Showa era. See Figure 118 on page 502 for a list of Japanese Eras supported.

If *picture_string* includes a Republic of China (ROC) Era symbol `<CCCC>` or `<CCCCCCCC>`, the `YY` position in *input_timestamp* represents the year number within the ROC Era. For example, the year 1988 equals the ROC year 77 in the MinKow Era.

See Figure 119 on page 502 for a list of ROC Eras supported.

See Figure 116 on page 500 for a list of valid picture characters, and Figure 117 on page 501 for examples of valid picture strings.

output_seconds (output)

A 64-bit double floating-point number representing the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds. For example, 00:00:01 on 15 October 1582 is second 86,401 (24*60*60 + 01) in the Lilian format. 19:00:01.12 on 16 May 1988 is second 12,799,191,601.12.

The largest value represented is 23:59:59.999 on 31 December 9999, which is second 265,621,679,999.999 in the Lilian format.

A 64-bit double floating-point value can accurately represent approximately 16 significant decimal digits without loss of precision. Therefore, accuracy is available to the nearest millisecond (15 decimal digits).

If *input_timestamp* does not contain a valid date or timestamp, *output_seconds* is set to 0 and CEESECS terminates with a non-CEE000 symbolic feedback code.

Elapsed time calculations are performed easily on the *output_seconds*, because it represents elapsed time. Leap year and end-of-year anomalies do not affect the calculations.

fc (output)

A 12-byte feedback code (optional), that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2EB	3	2507	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
CEE2EC	3	2508	The date value passed to CEEDAYS or CEESECS was invalid.
CEE2ED	3	2509	The Japanese or Republic of China Era passed to CEEDAYS or CEESECS was not recognized.

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE2EE	3	2510	The hours value in a call to CEEISEC or CEESECS was not recognized.
CEE2EH	3	2513	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
CEE2EK	3	2516	The minutes value in a CEEISEC call was not recognized.
CEE2EL	3	2517	The month value in a CEEISEC call was not recognized.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EN	3	2519	The seconds value in a CEEISEC call was not recognized.
CEE2EP	3	2521	The Japanese (<JJJJ>) or Chinese (<CCCC>) year-within-Era value passed to CEEDAYS or CEESECS was zero.
CEE2ET	3	2525	CEESECS detected non-numeric data in a numeric field, or the timestamp string did not match the picture string.

Usage Notes

- The inverse of CEESECS is CEEDATM, which converts *output_seconds* to character format.
- By default, 2-digit years lie within the 100 year range starting 80 years prior to the system date. Thus, in 1997, all 2-digit years represent dates between 1917 and 2016, inclusive. You can change this range by using the callable service CEESCEN.

Example

CEESECS

```
CBL LIB,APOST
      ************************************************
      **                                            **
      ** Function: Call CEESECS to convert          **
      **           timestamp to number of seconds   **
      **                                            **
      ** In this example, calls are made to CEESECS **
      ** to convert two timestamps to the number of **
      ** seconds since 00:00:00 14 October 1582.    **
      ** The Lilian seconds for the earlier         **
      ** timestamp are then subtracted from the     **
      ** Lilian seconds for the later timestamp     **
      ** to determine the number of between the     **
      ** two.  This result is displayed.            **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLSECS.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  SECOND1                COMP-2.
       01  SECOND2                COMP-2.
       01  TIMESTP.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X,
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of TIMESTP.
       01  TIMESTP2.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X,
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of TIMESTP2.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X,
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of PICSTR.
```

```
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.
 PROCEDURE DIVISION.

 PARA-SECS1.
************************************************
** Specify first timestamp and a picture string
**     describing the format of the timestamp
**     as input to CEESECS
************************************************
     MOVE 25 TO Vstring-length of TIMESTP.
     MOVE '1969-05-07 12:01:00.000'
             TO Vstring-text of TIMESTP.
     MOVE 25 TO Vstring-length of PICSTR.
     MOVE 'YYYY-MM-DD HH:MI:SS.999'
             TO Vstring-text of PICSTR.

************************************************
** Call CEESECS to convert the first timestamp
** to Lilian seconds
************************************************
     CALL 'CEESECS' USING TIMESTP, PICSTR,
                          SECOND1, FC.
     IF NOT CEE000 of FC  THEN
         DISPLAY 'CEESECS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

 PARA-SECS2.
************************************************
** Specify second timestamp and a picture string
**     describing the format of the timestamp as
**     input to CEESECS.
************************************************
     MOVE 25 TO Vstring-length of TIMESTP2.
     MOVE '2000-01-01 00:00:01.000'
             TO Vstring-text of TIMESTP2.
     MOVE 25 TO Vstring-length of PICSTR.
     MOVE 'YYYY-MM-DD HH:MI:SS.999'
             TO Vstring-text of PICSTR.

************************************************
** Call CEESECS to convert the second timestamp
**     to Lilian seconds
************************************************
     CALL 'CEESECS' USING TIMESTP2, PICSTR,
                          SECOND2, FC.
     IF NOT CEE000 of FC  THEN
         DISPLAY 'CEESECS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

 PARA-SECS2.
************************************************
** Subtract SECOND2 from SECOND1 to determine the
**     number of seconds between the two timestamps
************************************************
     SUBTRACT SECOND1 FROM SECOND2.
     DISPLAY 'The number of seconds between '
         Vstring-text OF TIMESTP ' and '
         Vstring-text OF TIMESTP2 ' is: ' SECOND2.

     GOBACK.
```

CEEUTC—Get Coordinated Universal Time

CEEUTC is identical to CEEGMT. See “CEEGMT—Get Current Greenwich Mean Time” on page 579.

IGZEDT4—Get Current Date

Note: In addition to the previous date and time callable services, VisualAge COBOL supports the VS COBOL II callable service IGZEDT4.

IGZEDT4 returns the current date with a 4-digit year in the form YYYYMMDD.

Syntax

```
►►──CALL──"IGZEDT4"──USING──output_char_date──.──────────────────────────►◄
```

output_char_date (output)
An 8-byte fixed-length character string in the form YYYYMMDD representing current year, month, and day.

Usage Notes

- IGZEDT4 is not supported under CICS.

Example

```
CBL LIB,APOST
      **************************************************
      ** Function: IGZEDT4 - get current date in the  **
      **                     format YYYYMMDD.         **
      **************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLEDT4.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  CHRDATE                 PIC S9(8) USAGE DISPLAY.

       PROCEDURE DIVISION.
       PARA-CBLEDT4.
      **************************************************
      ** Call IGZEDT4.
      **************************************************
           CALL 'IGZEDT4' USING BY REFERENCE CHRDATE.

      **************************************************
      ** IGZEDT4 has no non-zero return code to
      **     check, so just display result.
      **************************************************
               DISPLAY 'The current date is: '
                   CHRDATE

           GOBACK.
```

Appendix F. Run-Time Messages

Messages for VisualAge COBOL contain a message prefix, message number, severity code, and descriptive text. The message prefix is always IWZ, followed by the message number. The severity code will be either I (Information), W (Warning), S (Severe), or C (Critical). The message text provides a brief explanation of the condition.

In the following example message:

```
IWZ2519S  The seconds value in a CEEISEC call was not recognized.
```

- The message prefix is IWZ.
- The message number is 2519.
- The severity code is S.
- The message text is “The seconds value in a CEEISEC call was not recognized.”

The date and time callable services messages also contain a symbolic feedback code, which represents the first 8 bytes of a 12-byte condition token. You can think of the symbolic feedback code as the nickname for a condition. Note that the callable services messages contain a 4-digit message number.

IWZ006S **The reference to table** *table-name* **by verb number** *verb-number* **on line** *line-number* **addressed an area outside the region of the table.**

Explanation: When the SSRANGE option is in effect, this message is issued to indicate that a fixed-length table has been subscripted in a way that exceeds the defined size of the table, or, for variable-length tables, the maximum size of the table.

The range check was performed on the composite of the subscripts and resulted in an address outside the region of the table. For variable-length tables, the address is outside the region of the table defined when all OCCURS DEPENDING ON objects are at their maximum values; the ODO object's current value is not considered. The check was not performed on individual subscripts.

Programmer Response: Ensure that the value of literal subscripts and/or the value of variable subscripts as evaluated at run-time do not exceed the subscripted dimensions for subscripted data in the failing statement.

System Action: The application was terminated.

IWZ007S The reference to variable length group *group-name* **by verb number** *verb-number* **on line** *line-number* **addressed an area outside the maximum defined length of the group.**

Explanation: When the SSRANGE option is in effect, this message is issued to indicate that a variable-length group generated by OCCURS DEPENDING ON has a length that is less than zero, or is greater than the limits defined in the OCCURS DEPENDING ON clauses.

The range check was performed on the composite length of the group, and not on the individual OCCURS DEPENDING ON objects.

Programmer Response: Ensure that OCCURS DEPENDING ON objects as evaluated at run-time do not exceed the maximum number of occurrences of the dimension for tables within the referenced group item.

System Action: The application was terminated.

IWZ012I Invalid run unit termination occurred while sort or merge is running.

Explanation: A sort or merge initiated by a COBOL program was in progress and one of the following was attempted:

1. A STOP RUN was issued.
2. A GOBACK or an EXIT PROGRAM was issued within the input procedure or the output procedure of the COBOL program that initiated the sort or merge. Note that the GOBACK and EXIT PROGRAM statements are allowed in a program called by an input procedure or an output procedure.

Programmer Response: Change the application so that it does not use one of the above methods to end the sort or merge.

System Action: The application was terminated.

IWZ013S Sort or merge requested while sort or merge is running in a different thread.

Explanation: Running sort or merge in two or more threads at the same time is not supported.

Programmer Response: Always run sort or merge in the same thread. Alternatively, include code before each call to the sort or merge that determines if sort or merge is running in another thread. If sort or merge is running in another thread, then wait for that thread to finish. If it isn't, then set a flag to indicate sort or merge is running and call sort or merge.

System Action: The thread is terminated.

IWZ026W The SORT-RETURN special register was never referenced, but the current content indicated the sort or merge operation in program *program-name* **on line number** *line-number* **was unsuccessful. The sort or merge return code was** *return code*

Explanation: The COBOL source does not contain any references to the sort-return register. The compiler generates a test after each sort or merge verb. A nonzero return code has been passed back to the program by Sort/Merge.

Programmer Response: Determine why the Sort/Merge was unsuccessful and fix the problem.

System Action: No system action was taken.

IWZ029S **Argument-1 for function** *function-name* **in program** *program-name* **at line** *line-number* **was less than zero.**

Explanation: An illegal value for argument-1 was used.

Programmer Response: Ensure that argument-1 is greater than or equal to zero.

System Action: The application was terminated.

IWZ030S **Argument-2 for function** *function-name* **in program** *program* **at line** *line-number* **was not a positive integer.**

Explanation: An illegal value for argument-1 was used.

Programmer Response: Ensure that argument-2 is a positive integer.

System Action: The application was terminated.

IWZ036W **Truncation of high order digit positions occurred in program** *program-name* **on line number** *line-number* **.**

Explanation: The generated code has truncated an intermediate result (that is, temporary storage used during an arithmetic calculation) to 30 digits; some of the truncated digits were not 0.

Programmer Response: See Appendix C, “Intermediate Results and Arithmetic Precision” on page 545 for a description of intermediate results.

System Action: No system action was taken.

IWZ037I **The flow of control in program** *program-name* **proceeded beyond the last line of the program. Control returned to the caller of the program** *program-name*

Explanation: The program did not have a terminator (STOP, GOBACK, or EXIT), and control fell through the last instruction.

Programmer Response: Check the logic of the program. Sometimes this error occurs because of one of the following logic errors:

- The last paragraph in the program was only supposed to receive control as the result of a PERFORM statement, but due to a logic error it was branched to by a GO TO statement.
- The last paragraph in the program was executed as the result of a “fall-through” path, and there was no statement at the end of the paragraph to end the program.

System Action: The application was terminated.

IWZ038S **A reference modification length value of** *reference-modification-value* **on line** *line-number* **which was not equal to 1 was found in a reference to data item** *data-item*

Explanation: The length value in a reference modification specification was not equal to 1. The length value must be equal to 1.

Programmer Response: Check the indicated line number in the program to ensure that any reference modified length values are (or will resolve to) 1.

System Action: The application was terminated.

IWZ039S **An invalid overpunched sign was detected.**

Explanation: The value in the sign position was not valid.

Given X'*sd*', where *s* is the sign representation and *d* represents the digit, the valid sign representations for external decimal (USAGE DISPLAY without the SIGN IS SEPARATE clause) are:

Positive:	0, 1, 2, 3, 8, 9, A, and B.
Negative:	4, 5, 6, 7, C, D, E, and F.

Signs generated internally are 3 for positive and unsigned, and 7 for negative.

Given X'*ds*', where *d* represents the digit and *s* is the sign representation, the valid sign representations for internal decimal (USAGE PACKED-DECIMAL) COBOL data are:

Positive:	A, C, E, and F.
Negative:	B and D.

Signs generated internally are C for positive and unsigned, and D for negative.

Programmer Response: This error might have occurred because of a REDEFINES clause involving the sign position or a group move involving the sign position, or the position was never initialized. Check for the above cases.

System Action: The application was terminated.

IWZ040S **An invalid separate sign was detected.**

Explanation: An operation was attempted on data defined with a separate sign. The value in the sign position was not a plus (+) or a minus (-).

Programmer Response: This error might have occurred because of a REDEFINES clause involving the sign position or a group move involving the sign position, or the position was never initialized. Check for the above cases.

System Action: The application was terminated.

IWZ045S **Unable to invoke method** *method-name* **on line number** *line number* **in program** *program-name***.**

Explanation: The specific method is not supported for the class of the current object reference.

Programmer Response: Check the indicated line number in the program to ensure that the class of the current object reference supports the method being invoked.

System Action: The application was terminated.

IWZ047S **Unable to invoke method** *method-name* **on line number** *line number* **in class** *class-name***.**

Explanation: The specific method is not supported for the class of the current object reference.

Programmer Response: Check the indicated line number in the class to ensure that the class of the current object reference supports the method being invoked.

System Action: The application was terminated.

IWZ048W **A negative base was raised to a fractional power in an exponentiation expression. The absolute value of the base was used.**

Explanation: A negative number raised to a fractional power occurred in a library routine.

The value of a negative number raised to a fractional power is undefined in COBOL. If a SIZE ERROR clause had appeared on the statement in question, the SIZE ERROR imperative would have been used. However, no SIZE ERROR clause was present, so the absolute value of the base was used in the exponentiation.

Programmer Response: Ensure that the program variables in the failing statement have been set correctly.

System Action: No system action was taken.

IWZ049W **A zero base was raised to a zero power in an exponentiation expression. The result was set to one.**

Explanation: The value of zero raised to the power zero occurred in a library routine.

The value of zero raised to the power zero is undefined in COBOL. If a SIZE ERROR clause had appeared on the statement in question, the SIZE ERROR imperative would have been used. However, no SIZE ERROR clause was present, so the value returned was one.

Programmer Response: Ensure that the program variables in the failing statement have been set correctly.

System Action: No system action was taken.

IWZ050S A zero base was raised to a negative power in an exponentiation expression.

Explanation: The value of zero raised to a negative power occurred in a library routine.

The value of zero raised to a negative number is not defined. If a SIZE ERROR clause had appeared on the statement in question, the SIZE ERROR imperative would have been used. However, no SIZE ERROR clause was present.

Programmer Response: Ensure that the program variables in the failing statement have been set correctly.

System Action: The application was terminated.

IWZ053S An overflow occurred on conversion to floating point.

Explanation: A number was generated in the program that is too large to be represented in floating point.

Programmer Response: You need to modify the program appropriately to avoid an overflow.

System Action: The application was terminated.

IWZ054S A floating point exception occurred.

Explanation: A floating point calculation has produced an illegal result. Floating point calculations are done using IEEE floating point arithmetic, which can produce results called NaN (Not a Number). For example, the result of 0 divided by 0 is NaN.

Programmer Response: Modify the program to test the arguments to this operation so that NaN is not produced.

System Action: The application was terminated.

IWZ055W An underflow occurred on conversion to floating point. The result was set to zero.

Explanation: On conversion to floating point, the negative exponent exceeded the limit of the hardware. The floating point value was set to zero.

Programmer Response: No action is necessary, although you may want to modify the program to avoid an underflow.

System Action: No system action was taken.

IWZ058S Exponent overflow occurred.

Explanation: Floating point exponent overflow occurred in a library routine.

Programmer Response: Ensure that the program variables in the failing statement have been set correctly.

System Action: The application was terminated.

IWZ059W An exponent with more than nine digits was truncated.

Explanation: Exponents in fixed point exponentiations may not contain more than nine digits. The exponent was truncated back to nine digits; some of the truncated digits were not 0.

Programmer Response: No action is necessary, although you may want to adjust the exponent in the failing statement.

System Action: No system action was taken.

IWZ060W Truncation of high order digit positions occurred.

Explanation: Code in a library routine has truncated an intermediate result (that is, temporary storage used during an arithmetic calculation) back to 30 digits; some of the truncated digits were not 0.

Programmer Response: See Appendix C, “Intermediate Results and Arithmetic Precision” on page 545 for a description of intermediate results.

System Action: No system action was taken.

IWZ061S Division by zero occurred.

Explanation: Division by zero occurred in a library routine. Division by zero is not defined. If a SIZE ERROR clause had appeared on the statement in question, the SIZE ERROR imperative would have been used. However, no SIZE ERROR clause was present.

Programmer Response: Ensure that the program variables in the failing statement have been set correctly.

System Action: The application was terminated.

IWZ063S An invalid sign was detected in a numeric edited sending field in *program-name* **on line number** *line-number***.**

Explanation: An attempt has been made to move a signed numeric edited field to a signed numeric or numeric edited receiving field in a MOVE statement. However, the sign position in the sending field contained a character that was not a valid sign character for the corresponding PICTURE.

Programmer Response: Ensure that the program variables in the failing statement have been set correctly.

System Action: The application was terminated.

IWZ064S A recursive call to active program *program-name* in compilation unit *compilation-unit* was attempted.

Explanation: COBOL does not allow reinvocation of an internal program which has begun execution, but has not yet terminated. For example, if internal programs A and B are siblings of a containing program, and A calls B and B calls A, this message will be issued.

Programmer Response: Examine your program to eliminate calls to active internal programs.

System Action: The application was terminated.

IWZ065I A CANCEL of active program *program-name* in compilation unit *compilation-unit* was attempted.

Explanation: An attempt was made to cancel an active internal program. For example, if internal programs A and B are siblings in a containing program and A calls B and B cancels A, this message will be issued.

Programmer Response: Examine your program to eliminate cancellation of active internal programs.

System Action: The application was terminated.

IWZ066S The length of external data record *data-record* in program *program-name* did not match the existing length of the record.

Explanation: While processing External data records during program initialization, it was determined that an External data record was previously defined in another program in the run-unit, and the length of the record as specified in the current program was not the same as the previously defined length.

Programmer Response: Examine the current file and ensure the External data records are specified correctly.

System Action: The application was terminated.

IWZ071S ALL subscripted table reference to table *table-name* by verb number *verb-number* on line *line-number* had an ALL subscript specified for an OCCURS DEPENDING ON dimension, and the object was less than or equal to 0.

Explanation: When the SSRANGE option is in effect, this message is issued to indicate that there are 0 occurrences of dimension subscripted by ALL.

The check is performed against the current value of the OCCURS DEPENDING ON OBJECT.

Programmer Response: Ensure that ODO object(s) of ALL-subscripted dimensions of any subscripted items in the indicated statement are positive.

System Action: The application was terminated.

IWZ072S **A reference modification start position value of** *reference-modification-value* **on line** *line-number* **referenced an area outside the region of data item** *data-item***.**

Explanation: The value of the starting position in a reference modification specification was less than 1, or was greater than the current length of the data item that was being reference modified. The starting position value must be a positive integer less than or equal to the number of characters in the reference modified data item.

Programmer Response: Check the value of the starting position in the reference modification specification.

System Action: The application was terminated.

IWZ073S **A non-positive reference modification length value of** *reference-modification-value* **on line** *line-number* **was found in a reference to data item** *data-item***.**

Explanation: The length value in a reference modification specification was less than or equal to 0. The length value must be a positive integer.

Programmer Response: Check the indicated line number in the program to ensure that any reference modified length values are (or will resolve to) positive integers.

System Action: The application was terminated.

IWZ074S **A reference modification start position value of** *reference-modification-value* **and length value of** *length* **on line** *line-number* **caused reference to be made beyond the rightmost character of data item** *data-item***.**

Explanation: The starting position and length value in a reference modification specification combine to address an area beyond the end of the reference modified data item. The sum of the starting position and length value minus one must be less than or equal to the number of characters in the reference modified data item.

Programmer Response: Check the indicated line number in the program to ensure that any reference modified start and length values are set such that a reference is not made beyond the rightmost character of the data item.

System Action: The application was terminated.

IWZ075S **Inconsistencies were found in EXTERNAL file** *file-name* **in program** *program-name***. The following file attributes did not match those of the established external file:** *attribute-1 attribute-2 attribute-3 attribute-4 attribute-5 attribute-6 attribute-7*

Explanation: One or more attributes of an external file did not match between two programs that defined it.

Programmer Response: Correct the external file. For a summary of file attributes which must match between definitions of the same external file, see the *COBOL Language Reference*.

System Action: The application was terminated.

IWZ076W The number of characters in the INSPECT REPLACING CHARACTERS BY data-name was not equal to one. The first character was used.

Explanation: A data item which appears in a CHARACTERS phrase within a REPLACING phrase in an INSPECT statement must be defined as being one character in length. Because of a reference modification specification for this data item, the resultant length value was not equal to one. The length value is assumed to be one.

Programmer Response: You may correct the reference modification specifications in the failing INSPECT statement to ensure that the reference modification length is (or will resolve to) 1; programmer action is not required.

System Action: No system action was taken.

IWZ077W The lengths of the INSPECT data items were not equal. The shorter length was used.

Explanation: The two data items which appear in a REPLACING or CONVERTING phrase in an INSPECT statement must have equal lengths, except when the second such item is a figurative constant. Because of the reference modification for one or both of these data items, the resultant length values were not equal. The shorter length value is applied to both items, and execution proceeds.

Programmer Response: You may adjust the operands of unequal length in the failing INSPECT statement; programmer action is not required.

System Action: No system action was taken.

IWZ078S ALL subscripted table reference to table ***table-name*** **by verb number** ***verb-number*** **on line** ***line-number*** **will exceed the upper bound of the table.**

Explanation: When the SSRANGE option is in effect, this message is issued to indicate that a multidimensional table with ALL specified as one or more of the subscripts will result in a reference beyond the upper limit of the table.

The range check was performed on the composite of the subscripts and the maximum occurrences for the ALL subscripted dimensions. For variable-length tables the address is outside the region of the table defined when all OCCURS DEPENDING ON objects are at their maximum values; the ODO object's current value is not considered. The check was not performed on individual subscripts.

Programmer Response: Ensure that OCCURS DEPENDING ON objects as evaluated at run-time do not exceed the maximum number of occurrences of the dimension for table items referenced in the failing statement.

System Action: The application was terminated.

IWZ096C **Dynamic call of program** *program-name* **failed. Message variants include:**

- **A load of module** *module-name* **failed with an error code of** *error-code***.**
- **A load of module** *module-name* **failed with a return code of** *return-code***.**
- **Dynamic call of program** *program-name* **failed. Insufficient resources.**
- **Dynamic call of program** *program-name* **failed. COBPATH not found in environment.**
- **Dynamic call of program** *program-name* **failed. Entry** *entry-name* **not found.**
- **Dynamic call failed. The name of the target program does not contain any valid characters.**
- **Dynamic call of program** *program-name* **failed. The load module** *load-module* **could not be found in the directories identified in the COBPATH environment variable.**

Explanation: A dynamic call failed due to one of the reasons listed in the message variants above. In the above, the value of *error-code* depends on the execution platform as follows:

AIX:	The errno set by `load`.
OS/2:	The return code from the `DosLoadModule` service.
Windows:	The last-error code value set by `LoadLibrary`.

Programmer Response: Check that you have COBPATH defined. Check that the module exists: AIX, OS/2, and Windows have graphical interfaces for showing directories and files. You can also use the *ls* command on AIX or the *dir* command on OS/2 and Windows. Check that the name of the module to be loaded matches the name of the entry called. Check that the module to be loaded is built correctly using the appropriate `cob2` options, for example, to build a DLL on Windows, the `-dll` option must be used.

System Action: The application was terminated.

IWZ097S **Argument-1 for function** *function-name* **contained no digits.**

Explanation: Argument-1 for the indicated function must contain at least 1 digit.

Programmer Response: Adjust the number of digits in Argument-1 in the failing statement.

System Action: The application was terminated.

IWZ100S **Argument-1 for function** *function* **was less than or equal to -1.**

Explanation: An illegal value was used for Argument-1.

Programmer Response: Ensure that argument-1 is greater than -1.

System Action: The application was terminated.

IWZ151S Argument-1 for function ***function-name*** **contained more than 18 digits.**

Explanation: The total number of digits in argument-1 of the indicated function exceeded 18 digits.

Programmer Response: Adjust the number of digits in argument-1 in the failing statement.

System Action: The application was terminated.

IWZ152S Invalid character ***character*** **was found in column** ***column-number*** **in argument-1 for function** ***function-name*** **.**

Explanation: A non-digit character other than a decimal point, comma, space or sign (+,-,CR,DB) was found in argument-1 for NUMVAL/NUMVAL-C function.

Programmer Response: Correct argument-1 for NUMVAL or NUMVAL-C in the indicated statement.

System Action: The application was terminated.

IWZ155S Invalid character ***character*** **was found in column** ***column-number*** **in argument-2 for function** ***function-name*** **.**

Explanation: Illegal character was found in argument-2 for NUMVAL-C function.

Programmer Response: Check that the function argument does follow the syntax rules.

System Action: The application was terminated.

IWZ156S Argument-1 for function ***function-name*** **was less than zero or greater than 28.**

Explanation: Input argument to function FACTORIAL is greater than 28 or less than 0.

Programmer Response: Check that the function argument is only one byte long.

System Action: The application was terminated.

IWZ157S The length of Argument-1 for function ***function-name*** **was not equal to 1.**

Explanation: The length of input argument to ORD function is not 1.

Programmer Response: Check that the function argument is only one byte long.

System Action: The application was terminated.

IWZ159S Argument-1 for function *function-name* was less than 1 or greater than 3067671.

Explanation: The input argument to DATE-OF-INTEGER or DAY-OF-INTEGER function is less than 1 or greater than 3067671.

Programmer Response: Check that the function argument is in the valid range.

System Action: The application was terminated.

IWZ160S Argument-1 for function *function-name* was less than 16010101 or greater than 99991231.

Explanation: The input argument to function INTEGER-OF-DATE is less than 16010101 or greater than 99991231.

Programmer Response: Check that the function argument is in the valid range.

System Action: The application was terminated.

IWZ161S Argument-1 for function *function-name* was less than 1601001 or greater than 9999365.

Explanation: The input argument to function INTEGER-OF-DAY is less than 1601001 or greater than 9999365.

Programmer Response: Check that the function argument is in the valid range.

System Action: The application was terminated.

IWZ162S Argument-1 for function *function-name* was less than 1 or greater than the number of positions in the program collating sequence.

Explanation: The input argument to function CHAR is less than 1 or greater than the highest ordinal position in the program collating sequence.

Programmer Response: Check that the function argument is in the valid range.

System Action: The application was terminated.

IWZ163S Argument-1 for function *function-name* was less than zero.

Explanation: The input argument to function RANDOM is less than 0.

Programmer Response: Correct the argument for function RANDOM in the failing statement.

System Action: The application was terminated.

IWZ165S A reference modification start position value of ***start-position-value*** **on line** ***line number*** **referenced an area outside the region of the function result of** ***function-result*****.**

Explanation: The value of the starting position in a reference modification specification was less than 1, or was greater than the current length of the function result that was being reference modified. The starting position value must be a positive integer less than or equal to the number of characters in the reference modified function result.

Programmer Response: Check the value of the starting position in the reference modification specification and the length of the actual function result.

System Action: The application was terminated.

IWZ166S A non-positive reference modification length value of ***length*** **on line** ***line-number*** **was found in a reference to the function result of** ***function-result*****.**

Explanation: The length value in a reference modification specification for a function result was less than or equal to 0. The length value must be a positive integer.

Programmer Response: Check the length value and make appropriate correction.

System Action: The application was terminated.

IWZ167S A reference modification start position value of ***start-position*** **and length value of** ***length*** **on line** ***line-number*** **caused reference to be made beyond the rightmost character of the function result of** ***function-result*****.**

Explanation: The starting position and length value in a reference modification specification combine to address an area beyond the end of the reference modified function result. The sum of the starting position and length value minus one must be less than or equal to the number of characters in the reference modified function result.

Programmer Response: Check the length of the reference modification specification against the actual length of the function result and make appropriate corrections.

System Action: The application was terminated.

IWZ168W SYSPUNCH/SYSPCH will default to the system logical output device. The corresponding environment variable has not been set.

Explanation: COBOL environment names (such as SYSPUNCH/SYSPCH) are used as the environment variable names corresponding to the mnemonic names used on ACCEPT and DISPLAY statements. Set them equal to files, not existing directory names. To set environment variables:

- On OS/2 or Windows, use the SET command.
- On AIX, use the EXPORT command.

You can set environment variables either temporarily or persistently. For more information, see “Setting Environment Variables” on page 134.

Programmer Response: If you do not want SYSPUNCH/SYSPCH to default to the screen, set the corresponding environment variable.

System Action: No system action was taken.

IWZ170S Illegal data type for DISPLAY operand.

Explanation: An invalid data type was specified as the target of the DISPLAY statement.

Programmer Response: Specify a valid data type. The following data types are **not** valid:

- Data items defined with USAGE IS PROCEDURE-POINTER
- Data items defined with USAGE IS OBJECT REFERENCE
- Data items or index names defined with USAGE IS INDEX

System Action: The application was terminated.

IWZ171I ***string-name*** **is not a valid run-time option.**

Explanation: *string-name* is not a valid option.

Programmer Response: See Chapter 12, "Run-Time Options" on page 240 for valid run-time options.

System Action: *string-name* is ignored.

IWZ172I The string ***string-name*** **is not a valid suboption of the run-time option** ***option-name*****.**

Explanation: *string-name* was not in the set of recognized values.

Programmer Response: Remove the invalid suboption *string* from the run-time option *option-name*. See Chapter 12, "Run-Time Options" on page 240 for valid suboptions for run-time option *option-name*.

System Action: The invalid suboption is ignored.

IWZ173I The suboption string ***string-name*** **of the run-time option** ***option-name*** **must be** ***number of*** **characters long. The default will be used.**

Explanation: The number of characters for the suboption string *string-name* of run-time option *option-name* is invalid.

Programmer Response: If you do not want to accept the default, specify a valid character length. See Chapter 12, "Run-Time Options" on page 240

System Action: The default value will be used.

IWZ174I The suboption string ***string-name*** **of the run-time option** ***option-name*** **contains one or more invalid characters. The default will be used.**

Explanation: At least one invalid character was detected in the specified suboption.

Programmer Response: If you do not want to accept the default, specify valid characters.

System Action: The default value will be used.

IWZ175S There is no support for routine *routine-name* on this system.

Explanation: *routine-name* is not supported.

Programmer Response:

System Action: The application was terminated.

IWZ176S Argument-1 for function *function-name* was greater than *decimal-value*.

Explanation: An illegal value for argument-1 was used.

Programmer Response: Ensure argument-1 is less than or equal to *decimal-value*.

System Action: The application was terminated.

IWZ177S Argument-2 for function *function-name* was equal to *decimal-value*.

Explanation: An illegal value for argument-2 was used.

Programmer Response: Ensure argument-1 is not equal to *decimal-value*.

System Action: The application was terminated.

IWZ178S Argument-1 for function *function-name* was less than or equal to *decimal-value*.

Explanation: An illegal value for argument-1 was used.

Programmer Response: Ensure argument-1 is greater than *decimal-value*.

System Action: The application was terminated.

IWZ179S Argument-1 for function *function-name* was less than *decimal-value*.

Explanation: An illegal value for argument-1 was used.

Programmer Response: Ensure argument-1 is equal to or greater than *decimal-value*.

System Action: The application was terminated.

IWZ180S Argument-1 for function *function-name* was not an integer.

Explanation: An illegal value for argument-1 was used.

Programmer Response: Ensure argument-1 is an integer.

System Action: The application was terminated.

IWZ181I **An invalid character was found in the numeric string** *string* **of the run-time option** *option-name***. The default will be used.**

Explanation: *string* did not contain all decimal numeric characters.

Programmer Response: If you do not want the default value, correct the run-time option's string to contain all numeric characters.

System Action: The default will be used.

IWZ182I **The number** *number* **of the run-time option** *option-name* **exceeded the range of** *min-range* **to** *max-range***. The default will be used.**

Explanation: *number* exceeded the range of *min-range* to *max-range*.

Programmer Response: Correct the run-time option's string to be within the valid range. See Chapter 12, “Run-Time Options” on page 240 for valid ranges.

System Action: The default will be used.

IWZ183S **The function name in _iwzCOBOLInit did a return.**

Explanation: The run unit termination exit routine returned to the invoker of the routine (the function specified in `function_code`).

Programmer Response: Rewrite the function so that the run unit termination exit routine does a longjump or exit() instead of return to the function.

System Action: The application was terminated.

IWZ200S **Message variants include:**

- **Error detected during** *I/O operation* **for file** *file-name***. File status is:** *file-status***.**
- **STOP or ACCEPT failed with an I/O error,** *error-code***. The run unit is terminated.**

Explanation: See messages below.

IWZ200S **Error detected during** *I/O operation* **for file** *file-name***. File status is:** *file-status***.**

Explanation: An error was detected during a file I/O operation. No file status was specified for the file and no applicable error declarative is in effect for the file.

Programmer Response: Correct the condition described in this message. You can specify the FILE STATUS clause for the file if you want to detect the error and take appropriate actions within your source program.

System Action: The application was terminated.

IWZ200S STOP or ACCEPT failed with an I/O error, *error-code***. The run unit is terminated.**

Explanation: A STOP or ACCEPT statement failed.

Programmer Response: Check that the STOP or ACCEPT refers to a legitimate file or terminal device.

System Action: The application was terminated.

IWZ201C Message variants include:

Access Intent List Error.
Concurrent Opens Exceeds Maximum.
Cursor Not Selecting a Record Position.
Data Stream Syntax Error.
Duplicate Key Different Index.
Duplicate Key Same Index.
Duplicate Record Number.
File Temporarily Not Available.
File system cannot be found.
File Space Not Available.
File Closed with Damage.
Invalid Key Definition.
Invalid Base File Name.
Key Update Not Allowed by Different Index.
Key Update Not Allowed by Same Index.
No Update Intent on Record.
Not Authorized to Use Access Method.
Not Authorized to Directory.
Not Authorized to Function.
Not authorized to File.
Parameter Value Not Supported.
Parameter Not Supported.
Record Number Out of Bounds.
Record Length Mismatch.
Resource Limits Reached in Target System.
Resource Limits Reached in Source System.
Address Error.
Command Check.
Duplicate File Name.
End of File Condition.
Existing Condition.
File Handle Not Found.
Field Length Error.
File Not Found.
File Damaged.
File is Full.
File In Use.
Function Not Supported.
Invalid Access Method.
Invalid Data Record.
Invalid Key Length.
Invalid File Name.
Invalid Request.
Invalid Flag.
Object Not Supported.
Record Not Available.
Record Not Found.
Record Inactive.
Record Damaged.
Record In Use.
Update Cursor Error.

Explanation: An error was detected during a file I/O operation for a VSAM file. No file status was specified for the file and no applicable error declarative is in effect for the file.

Programmer Response: Correct the condition described in this message. For details, see the SMARTdata Utilities VSAM manual for your platform:

- For OS/2: *VSAM in a Distributed Environment*
- For Windows: *VSAM API Reference*
- For AIX: *VSAM in a Distributed Environment*

System Action: The application was terminated.

IWZ203W The code page in effect is not a DBCS code page.

Explanation: References to DBCS data was made with a non-DBCS code page in effect.

Programmer Response: For DBCS data, specify a valid DBCS code page. Valid DBCS code pages are:

	OS/2	Windows (NT and 95)	AIX
Japan	IBM-932, IBM-942, IBM-943	IBM-943	IBM-932
Korea	IBM-942		IBM-1363
China (Simplified - Mainland)	IBM-1381, IBM-1861		IBM-1386

	OS/2	Windows (NT and 95)	AIX
China (Traditional - Taiwan)	IBM-950	IBM-950	

Note: The code pages listed above might not be supported for a specific version or release of that platform. For additional information, see "Locales and Code Sets Supported" on page 477.

System Action: No system action was taken.

IWZ204W An error occurred during conversion from ASCII DBCS to EBCDIC DBCS.

Explanation: A Kanji or DBCS class test failed due to an error detected during the ASCII character string EBCDIC string conversion.

Programmer Response: Verify that the locale in effect is consistent with the ASCII character string being tested. No action is likely to be required if the locale setting is correct. The class test is likely to indicate the string to be non-Kanji or non-DBCS correctly.

System Action: No system action was taken.

IWZ211S CBLTDLI detected a Remote DL/I error.

Explanation: The CBLTDLI routine invoked Remote DL/I and Remote DL/I returned with an error.

Programmer Response: Look for Remote DL/I messages that provide information about the error.

System Action: The application was terminated.

IWZ212S Too few arguments were passed to CBLTDLI.

Explanation: The CBLTDLI routine must be passed at least one argument.

Programmer Response: Add the missing arguments to the CBLTDLI call.

System Action: The application was terminated.

IWZ213S Too many arguments were passed to CBLTDLI.

Explanation: The CBLTDLI routine was passed more than 19 arguments.

Programmer Response: Remove the extra arguments from the CBLTDLI call.

System Action: The application was terminated.

IWZ214S No function code was passed to CBLTDLI.

Explanation: The CBLTDLI routine recognized the first argument as a parm count field and a second argument was was not provided.

Programmer Response: Add the extra arguments to the CBLTDLI call.

System Action: The application was terminated.

IWZ230x Message variants include:

- **The conversion table for the current codeset,** *ASCII codeset-id***, to the EBCDIC codeset,** *EBCDIC codeset-id***, is not available. The default ASCII to EBCDIC conversion table will be used.**
- **The EBCDIC codepage specified,** *EBCDIC codepage***, is not consistent with the locale** *locale***, but will be used as requested.**
- **The EBCDIC codepage specified,** *EBCDIC codepage***, is not supported. The default EBCDIC codepage,** *EBCDIC codepage***, will be used.**
- **The EBCDIC conversion table cannot be opened.**
- **The EBCDIC conversion table cannot be built.**

Explanation: See separate messages listed below.

IWZ230W The conversion table for the current codeset, *ASCII codeset-id***, to the EBCDIC codeset,** *EBCDIC codeset-id***, is not available. The default ASCII to EBCDIC conversion table will be used.**

Explanation: The application has a module which was compiled with the CHAR(EBCDIC) compiler option. At run-time a translation table will be built to handle the conversion from the current ASCII code page to an EBCDIC code page specified by the EBCDIC_CODEPAGE environment variable. This error occurred because either a conversion table is not available for the specified code pages, or the specification of the EBCDIC_CODE page is invalid. Execution will continue with a default conversion table based on ASCII code page IBM-850 and EBCDIC code page IBM-037.

Programmer Response: Verify that the EBCDIC_CODEPAGE environment variable has a valid value (see “Locales and Code Sets Supported” on page 477).

If EBCDIC_CODEPAGE is not set, the default value, IBM-037, will be used. This is the default code page used by IBM COBOL for OS/390 & VM.

System Action: No system action was taken.

IWZ230W The EBCDIC codepage specified, ***EBCDIC codepage*****, is not consistent with the locale** ***locale*****, but will be used as requested.**

Explanation: The application has a module which was compiled with the CHAR(EBCDIC) compiler option. This error occurred because the code page specified is not the same language as the current locale.

Programmer Response: Verify that the EBCDIC_CODEPAGE environment variable is valid for this locale (see “Locales and Code Sets Supported” on page 477).

System Action: No system action was taken.

IWZ230W The EBCDIC codepage specified, ***EBCDIC codepage*****, is not supported. The default EBCDIC codepage,** ***EBCDIC codepage*****, will be used.**

Explanation: The application has a module which was compiled with the CHAR(EBCDIC) compiler option. This error occurred because the specification of the EBCDIC_CODE page is invalid. Execution will continue with the default host code page that corresponds to the current locale.

Programmer Response: Verify that the EBCDIC_CODEPAGE environment variable has a valid value (see “Locales and Code Sets Supported” on page 477).

System Action: No system action was taken.

IWZ230S The EBCDIC conversion table cannot be opened.

Explanation: The current system installation does not include the translation table for the default ASCII and EBCDIC code pages.

Programmer Response: Reinstall the compiler and run time. If the problem still persists, call your IBM representative.

System Action: The application was terminated.

IWZ230S The EBCDIC conversion table cannot be built.

Explanation: The ASCII to EBCDIC conversion table has been opened, but the conversion has failed.

Programmer Response: Retry the execution from a new window.

System Action: The application was terminated.

IWZ231S Query of current locale setting failed.

Explanation: A query of the execution environment failed to identify a valid locale setting. The current locale needs to be established to access appropriate message files and set the collating order. It is also used by the date/time services and for EBCDIC character support.

Programmer Response: Check the settings for the following environment variables:

- LOCPATH
  - Not used on AIX.
  - On OS/2 and Windows, this environment variable should include the IBMCOBOL\LOCALE directory
- LANG
  - On OS/2, this environment variable should be set to the filename (without extension) of one of the DLLs located in the IBM\LOCALE directory. The default value is en_US.
  - On Windows this should be set to the name of one of the directories located in the IBMCOBW\LOCALE directory. The default value is en_US.
  - On AIX this should be set to a locale which has been installed on your machine. Type "locale -a" to get a list of the valid values. The default value is en_US.

System Action: The application was terminated.

IWZ240S The base year for program *program-name* was outside the valid range of 1900 through 1999. The sliding window value *window-value* resulted in a base year of *base-year*.

Explanation: When the 100-year window was computed using the current year and the sliding window value specified with the YEARWINDOW compiler option, the base year of the 100-year window was outside the valid range of 1900 through 1999.

For example, if a COBOL program had been compiled with YEARWINDOW(-99) and the COBOL program was run in the year 1998 this message would occur because the base year of the 100-year window would be 1899 (1998 - 99).

Programmer Response: Examine the application design to determine if it will support a change to the YEARWINDOW option value. If the application can run with a change to the YEARWINDOW option value, then compile the program with an appropriate YEARWINDOW option value. If the application cannot run with a change to the YEARWINDOW option value, then convert all date fields to expanded dates and compile the program with NODATEPROC.

System Action: The application was terminated.

IWZ241S The current year was outside the 100-year window, *year-start* through *year-end*, for program *program-name*.

Explanation: The current year was outside the 100-year fixed window specified by the YEARWINDOW compiler option value.

For example, if a COBOL program is compiled with YEARWINDOW(1920), the 100-year window for the program is 1920 through 2019. When the program is run in the year 2020, this error message would occur since the current year is not within the 100-year window.

Programmer Response: Examine the application design to determine if it will support a change to the YEARWINDOW option value. If the application can run with a change to the YEARWINDOW option value, then compile the program with an appropriate YEARWINDOW option value. If the application cannot run with a change to the YEARWINDOW option value, then convert all date fields to expanded dates and compile the program with NODATEPROC.

System Action: The application was terminated.

IWZ813S Insufficient storage was available to satisfy a get storage request.

Explanation: There was not enough free storage available to satisfy a get storage or reallocate request. This message indicates that storage management could not obtain sufficient storage from the operating system.

Programmer Response: Ensure that you have sufficient storage available to run your application.

System Action: No storage is allocated.

Symbolic Feedback Code: CEE0PD

IWZ901W Message variants include:

- **Program exits due to severe or critical error.**
- **Program exits: more than ERRCOUNT errors occurred.**

Explanation: Every severe or critical message is followed by an IWZ901 message. An IWZ901 message is also issued if you have used the ERRCOUNT run-time option and the number of warning messages exceeds ERRCOUNT.

Programmer Response: See the severe or critical message, or increase ERRCOUNT.

System Action: The application was terminated.

IWZ902W The system detected a decimal-divide exception.

Explanation: An attempt to divide a number by 0 was detected.

Programmer Response: Modify the program. For example, add ON SIZE ERROR to the flagged statement.

System Action: No system action was taken.

IWZ907W Message variants include:

- **Insufficient storage.**
- **Insufficient storage. Cannot get** *number-bytes* **bytes of space for** *storage***.**

Explanation: The run-time library requested virtual memory space and the operating system denied the request.

Programmer Response: Your program uses a large amount of virtual memory and it ran out of space. The problem is usually not due to a particular statement, but is associated with the program as a whole. Look at your use of OCCURS clauses and reduce the size of your tables.

System Action: No system action was taken.

IWZ993W Insufficient storage. Cannot find space for message *message-number***.**

Explanation: The run-time library requested virtual memory space and the operating system denied the request.

Programmer Response: Your program uses a large amount of virtual memory and it ran out of space. The problem is usually not due to a particular statement, but is associated with the program as a whole. Look at your use of OCCURS clauses and reduce the size of your tables.

System Action: No system action was taken.

IWZ994W Cannot find message *message-number* **in** *%s***.**

Explanation: The run-time library cannot find either the message catalog or a particular message in the message catalog.

Programmer Response: Check that the COBOL library and messages were correctly installed and that NLSPATH is specified correctly.

System Action: No system action was taken.

| **IWZ995C Message variants include:**

- *system exception* **signal received while executing routine**
- *system exception* **signal received while executing code at location 0x** *offset-value***.** *routine-name* **at offset 0x** *offset-value***.**

Explanation: The operating system has detected an illegal action, such as an attempt to store into a protected area of memory or the operating system has detected that you pressed the interrupt key (typically the Control-C key, but it can be reconfigured).

Programmer Response: If the signal was due to an illegal action, run the program under the debugger and it will give you more precise information as to where the error occurred. An example of this type of error is a pointer with an illegal value.

System Action: The application was terminated.

IWZ2502S The UTC/GMT was not available from the system.

Explanation: A call to CEEUTC or CEEGMT failed because the system clock was in an invalid state. The current time could not be determined.

Programmer Response: Notify systems support personnel that the system clock is in an invalid state.

System Action: All output values are set to 0.

Symbolic Feedback Code: CEE2E6

IWZ2503S The offset from UTC/GMT to local time was not available from the system.

Explanation: A call to CEEGMTO failed because either (1) the current operating system could not be determined, or (2) the time zone field in the operating system control block appears to contain invalid data.

Programmer Response: Notify systems support personnel that the local time offset stored in the operating system appears to contain invalid data.

System Action: All output values are set to 0.

Symbolic Feedback Code: CEE2E7

IWZ2505S The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.

Explanation: The input_seconds value passed in a call to CEEDATM or CEESECI was not a floating-point number between 86,400.0 and 265,621,679,999.999 The input parameter should represent the number of seconds elapsed since 00:00:00 on 14 October 1582, with 00:00:00.000 15 October 1582 being the first supported date/time, and 23:59:59.999 31 December 9999 being the last supported date/time.

Programmer Response: Verify that input parameter contains a floating-point value between 86,400.0 and 265,621,679,999.999.

System Action: For CEEDATM, the output value is set to blanks. For CEESECI, all output parameters are set to 0.

Symbolic Feedback Code: CEE2E9

IWZ2506S Japanese (<JJJJ>) or Republic of China (<CCCC> or <CCCCCCCC>) Era was used in a picture string passed to CEEDATM, but the input number-of-seconds value was not within the supported range. The era could not be determined.

Explanation: In a CEEDATM call, the picture string indicates that the input value is to be converted to a Japanese or Republic of China Era; however the input value that was specified lies outside the range of supported eras.

Programmer Response: Verify that the input value contains a valid number-of-seconds value within the range of supported eras.

System Action: The output value is set to blanks.

IWZ2507S Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.

Explanation: The picture string passed in a CEEDAYS or CEESECS call did not contain enough information. For example, it is an error to use the picture string ‘MM/DD’ (month and day only) in a call to CEEDAYS or CEESECS, because the year value is missing. The minimum information required to calculate a Lilian value is either (1) month, day and year, or (2) year and Julian day.

Programmer Response: Verify that the picture string specified in a call to CEEDAYS or CEESECS specifies, as a minimum, the location in the input string of either (1) the year, month, and day, or (2) the year and Julian day.

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2EB

IWZ2508S The date value passed to CEEDAYS or CEESECS was invalid.

Explanation: In a CEEDAYS or CEESECS call, the value in the DD or DDD field is not valid for the given year and/or month. For example, ‘MM/DD/YY’ with ‘02/29/90’, or ‘YYYY.DDD’ with ‘1990.366’ are invalid because 1990 is not a leap year. This code may also be returned for any non-existent date value such as June 31st, January 0.

Programmer Response: Verify that the format of the input data matches the picture string specification and that input data contains a valid date.

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2EC

IWZ2509S The Japanese or Republic of China Era passed to CEEDAYS or CEESECS was not recognized.

Explanation: The value in the <JJJJ>, <CCCC>, or <CCCCCCCC> field passed in a call to CEEDAYS or CEESECS does not contain a supported Japanese or Republic of China Era name.

Programmer Response: Verify that the format of the input data matches the picture string specification and that the spelling of the Japanese or ROC Era name is correct. Note that the era name must be a proper DBCS string where the '<' position must contain the first byte of the era name.

System Action: The output value is set to 0.

IWZ2510S The hours value in a call to CEEISEC or CEESECS was not recognized.

Explanation: (1) In a CEEISEC call, the hours parameter did not contain a number between 0 and 23, or (2) in a CEESECS call, the value in the HH (hours) field does not contain a number between 0 and 23, or the “AP” (a.m./p.m.) field is present and the HH field does not contain a number between 1 and 12.

Programmer Response: For CEEISEC, verify that the hours parameter contains an integer between 0 and 23. For CEESECS, verify that the format of the input data matches the picture string specification, and that the hours field contains a value between 0 and 23, (or 1 and 12 if the “AP” field is used).

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2EE

IWZ2511S The day parameter passed in a CEEISEC call was invalid for year and month specified.

Explanation: The day parameter passed in a CEEISEC call did not contain a valid day number. The combination of year, month, and day formed an invalid date value. Examples: year=1990, month=2, day=29; or month=6, day=31; or day=0.

Programmer Response: Verify that the day parameter contains an integer between 1 and 31, and that the combination of year, month, and day represents a valid date.

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2EF

IWZ2512S The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.

Explanation: The Lilian day number passed in a call to CEEDATE or CEEDYWK was not a number between 1 and 3,074,324.

Programmer Response: Verify that the input parameter contains an integer between 1 and 3,074,324.

System Action: The output value is set to blanks.

Symbolic Feedback Code: CEE2EG

IWZ2513S The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.

Explanation: The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was earlier than 15 October 1582, or later than 31 December 9999.

Programmer Response: For CEEISEC, verify that the year, month, and day parameters form a date greater than or equal to 15 October 1582. For CEEDAYS and CEESECS, verify that the format of the input date matches the picture string specification, and that the input date is within the supported range.

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2EH

IWZ2514S The year value passed in a CEEISEC call was not within the supported range.

Explanation: The year parameter passed in a CEEISEC call did not contain a number between 1582 and 9999.

Programmer Response: Verify that the year parameter contains valid data, and that the year parameter includes the century, for example, specify year 1990, not year 90.

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2EI

IWZ2515S The milliseconds value in a CEEISEC call was not recognized.

Explanation: In a CEEISEC call, the milliseconds parameter (*input_milliseconds*) did not contain a number between 0 and 999.

Programmer Response: Verify that the milliseconds parameter contains an integer between 0 and 999.

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2EJ

IWZ2516S The minutes value in a CEEISEC call was not recognized.

Explanation: (1) In a CEEISEC call, the minutes parameter (*input_minutes*) did not contain a number between 0 and 59, or (2) in a CEESECS call, the value in the MI (minutes) field did not contain a number between 0 and 59.

Programmer Response: For CEEISEC, verify that the minutes parameter contains an integer between 0 and 59. For CEESECS, verify that the format of the input data matches the picture string specification, and that the minutes field contains a number between 0 and 59.

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2EK

IWZ2517S The month value in a CEEISEC call was not recognized.

Explanation: (1) In a CEEISEC call, the month parameter (*input_month*) did not contain a number between 1 and 12, or (2) in a CEEDAYS or CEESECS call, the value in the MM field did not contain a number between 1 and 12, or the value in the MMM, MMMM, etc. field did not contain a correctly spelled month name or month abbreviation in the currently active National Language.

Programmer Response: For CEEISEC, verify that the month parameter contains an integer between 1 and 12. For CEEDAYS and CEESECS, verify that the format of the input data matches the picture string specification. For the MM field, verify that the input value is between 1 and 12. For spelled-out month names (MMM, MMMM, etc.), verify that the spelling or abbreviation of the month name is correct in the currently active National Language.

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2EL

IWZ2518S An invalid picture string was specified in a call to a date/time service.

Explanation: The picture string supplied in a call to one of the date/time services was invalid. Only one era character string can be specified.

Programmer Response: Verify that the picture string contains valid data. If the picture string contains more than one era descriptor, such as both Japanese (<JJJJ>) and Republic of China (<CCCC>) being specified, then change the picture string to use only one era.

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2EM

IWZ2519S The seconds value in a CEEISEC call was not recognized.

Explanation: (1) In a CEEISEC call, the seconds parameter (*input_seconds*) did not contain a number between 0 and 59, or (2) in a CEESECS call, the value in the SS (seconds) field did not contain a number between 0 and 59.

Programmer Response: For CEEISEC, verify that the seconds parameter contains an integer between 0 and 59. For CEESECS, verify that the format of the input data matches the picture string specification, and that the seconds field contains a number between 0 and 59.

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2EN

IWZ2520S CEEDAYS detected non-numeric data in a numeric field, or the date string did not match the picture string.

Explanation: The input value passed in a CEEDAYS call did not appear to be in the format described by the picture specification, for example, non-numeric characters appear where only numeric characters are expected.

Programmer Response: Verify that the format of the input data matches the picture string specification and that numeric fields contain only numeric data.

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2EO

IWZ2521S The Japanese (<JJJJ>) or Chinese (<CCCC>) year-within-Era value passed to CEEDAYS or CEESECS was zero.

Explanation: In a CEEDAYS or CEESECS call, if the YY or ZYY picture token is specified, and if the picture string contains one of the era tokens such as <CCCC> or <JJJJ>, then the year value must be greater than or equal to 1 and must be a valid year value for the era. In this context, the YY or ZYY field means year within Era.

Programmer Response: Verify that the format of the input data matches the picture string specification and that the input data is valid.

System Action: The output value is set to 0.

IWZ2522S Japanese (<JJJJ>) or Republic of China (<CCCC> or <CCCCCCCC>) Era was used in a picture string passed to CEEDATE, but the Lilian date value was not within the supported range. The era could not be determined.

Explanation: In a CEEDATE call, the picture string indicates that the Lilian date is to be converted to a Japanese or Republic of China Era, but the Lilian date lies outside the range of supported eras.

Programmer Response: Verify that the input value contains a valid Lilian day number within the range of supported eras.

System Action: The output value is set to blanks.

IWZ2525S CEESECS detected non-numeric data in a numeric field, or the timestamp string did not match the picture string.

Explanation: The input value passed in a CEESECS call did not appear to be in the format described by the picture specification. For example, non-numeric characters appear where only numeric characters are expected, or the a.m./p.m. field (AP, A.P., etc.) did not contain the strings ‘AM’ or ‘PM’.

Programmer Response: Verify that the format of the input data matches the picture string specification and that numeric fields contain only numeric data.

System Action: The output value is set to 0.

Symbolic Feedback Code: CEE2ET

| **IWZ2526S The date string returned by CEEDATE was truncated.**

Explanation: In a CEEDATE call, the output string was not large enough to contain the formatted date value.

Programmer Response: Verify that the output string variable is large enough to contain the entire formatted date. Ensure that the output parameter is at least as long as the picture string parameter.

System Action: The output value is truncated to the length of the output parameter.

Symbolic Feedback Code: CEE2EU

| **IWZ2527S The timestamp string returned by CEEDATM was truncated.**

Explanation: In a CEEDATM call, the output string was not large enough to contain the formatted timestamp value.

Programmer Response: Verify that the output string variable is large enough to contain the entire formatted timestamp. Ensure that the output parameter is at least as long as the picture string parameter.

System Action: The output value is truncated to the length of the output parameter.

Symbolic Feedback Code: CEE2EV

IWZ2531S The local time was not available from the system.

Explanation: A call to CEELOCT failed because the system clock was in an invalid state. The current time cannot be determined.

Programmer Response: Notify systems support personnel that the system clock is in an invalid state.

System Action: All output values are set to 0.

Symbolic Feedback Code: CEE2F3

IWZ2533S The value passed to CEESCEN was not between 0 and 100.

Explanation: The *century_start* value passed in a CEESCEN call was not between 0 and 100, inclusive.

Programmer Response: Ensure that the input parameter is within range.

System Action: No system action is taken; the 100-year window assumed for all 2-digit years is unchanged.

Symbolic Feedback Code: CEE2F5

IWZ2534W Insufficient field width was specified for a month or weekday name in a call to CEEDATE or CEEDATM. Output set to blanks.

Explanation: The CEEDATE or CEEDATM callable services issues this message whenever the picture string contained MMM, MMMMMZ, WWW, Wwww, etc., requesting a spelled out month name or weekday name, and the month name currently being formatted contained more characters than can fit in the indicated field.

Programmer Response: Increase the field width by specifying enough Ms or Ws to contain the longest month or weekday name being formatted.

System Action: The month name and weekday name fields that are of insufficient width are set to blanks. The rest of the output string is unaffected. Processing continues.

Symbolic Feedback Code: CEE2F6

Appendix G. Remote DL/I

Remote DL/I provides access to IMS full function databases and GSAM databases from programs using Data Language I (DL/I) calls running on workstations. Remote DL/I provides the support to develop and test mainframe COBOL programs on workstations that use DL/I calls in a subset of environments. Specifically, support for DL/I calls will be provided for:

- IMS batch applications that access IMS full function databases and GSAM databases

With Remote DL/I support and IBM VisualAge COBOL, customers that use CBLTDLI can develop, compile and test on workstations the following:

- COBOL programs that run in an IMS batch environment that use CBLTDLI calls.

  OS/2 ▶

- COBOL programs that run in an CICS environment that use CBLTDLI calls to access IMS full function databases.

  ◀ OS/2

Note:

- Remote DL/I does not provide access to IMS message queues or IMS fast path databases.
- Remote DL/I runs using only S/390 data types as input and output. It does not provide any data conversion function.

How Remote DL/I Works

Remote DL/I uses APPC and an IMS batch environment to provide the remote DL/I call support. When Remote DL/I is first initialized on the workstation, it asks the user for the OS/390 userid and password to be used when the remote job is started on OS/390. Then APPC is used by Remote DL/I to start a job on OS/390 which brings up an IMS batch environment. Once the IMS batch environment environment is available, then remote DL/I calls can be processed. The DL/I calls on the workstation are sent to the IMS batch environment environment and executed. The results of the DL/I calls are then sent back to the program running on the workstation.

Remote DL/I Utilities

This section describes utility commands provided by Remote DL/I.

DLICHECK Command

Remote DL/I provides a command called DLICHECK which is used to verify that your Remote DL/I connection is working.

```
►►──DLICHECK──────────────────────────────────────────────────────►◄
```

DLICHKP Command

Remote DL/I provides a command called DLICHKP which is used to verify that your Remote DL/I connection is working and that a specific PSB can be succesfully scheduled and terminated.

```
►►──DLICHKP──PSB_Name──────────────────────────────────────────────►◄
```

DLILOGIN Command

Remote DL/I provides a command called DLILOGIN which is used to update the userid and password used by Remote DL/I.

```
►►──DLILOGIN───────────────────────────────────────────────────────►◄
```

IMS Batch Support

This section describes the support that allows a programmer to develop and test S/390 IMS Batch COBOL programs on workstations.

DLIBATCH Command

Remote DL/I provides a command called DLIBATCH which is used to invoke IMS Batch COBOL programs on workstations.

```
►►──DLIBATCH──┬───────────────────┬──┬────────────────┬────────────────►
              └─/D:debugger_name──┘  └─/E:entry_name──┘

►──┬───────────────┬──program_name─────────────────────────────────►◄
   └─/P:PSB_name───┘
```

DLIBATCH Options

debugger_name Name of the debugger to give control before starting the program. If this option is not specified, program execution begins at *entry_name* or its default. When using COBOL, IDBUG is the only valid value for *debugger_name*.

entry_name Name of the entry point in *program_name* where program execution begins. If this option is not specified, it defaults to *program_name*.

PSB_name Name of the PSB to schedule. If this option is not specified, it defaults to *program_name*.

Usage notes:

1. The program name must be the name of a DLL.
2. The DLIBATCH command supports PSBs with up to 100 PCBs.

Preparing to use the DLIBATCH Command

In order to use the DLIBATCH command, the following need to be addressed:

- The programs that are going to be run have to be compiled and link edited with certain options. For example, since Remote DL/I runs using S/390 data types as input and output, specific compiler options that enable S/390 data type support must be used. See "Preparing IBM VisualAge COBOL Programs" on what needs to be done to prepare IBM VisualAge COBOL programs to run with Remote DL/I.
- The Remote DL/I Server Environment File or the JCL associated with the APPC TP profile that is used to bring up the remote IMS batch environment may need to be modified. For example, if a new PSB was created, the data set that contains the new PSB must be included in the IMS DD statement. For information on the Server Environment File, see "Remote DL/I Server Environment File" on page 642.
- The RMTDLI_PARTNER_LU and the RMTDLI_PARTNER_TP environment variables must be properly set.
- When the programs access files using native language (such as OPEN, READ, WRITE, etc), the programmer has to set up access to those files. The files could be set up as local files or remote files (remote file access is provided by SMARTdata Utilities.).

Preparing IBM VisualAge COBOL Programs

Compiling IBM VisualAge COBOL Programs: When you want to use Remote DL/I with IBM VisualAge COBOL, you must use certain COBOL compiler options. The following compiler options are required:

- CHAR(EBCDIC)
- COLLSEQ(EBCDIC)
- FLOAT(HEX)
- BINARY(S390)
- ENTRYINT(SYSTEM)
- NOTHREAD

Linking IBM VisualAge COBOL Programs: The IBM VisualAge COBOL import library IWZRLIB.LIB has the linkage information for CBLTDLI.

In order to use the DLIBATCH command, the IBM COBOL program it invokes must be in a DLL.

Windows ▶

To create a DLL on Windows, use the -dll option on the cob2 command.

◀ Windows

OS/2

To create a DLL on OS/2, you need to create a .DEF file for the program and specify the name of the .DEF file on the cob2 command. In the .DEF file, you need to EXPORT the program, and if the program uses the DLITCBL entry point, you also need to EXPORT DLITCBL.

Figure 131 shows an example of a .DEF file for a COBOL program called RDLIC03 that uses DLITCBL as its entry point.

```
;
; Link editor Module Definition file for Remote DL/I
; Cobol program RDLIC03 that has a DLITCBL entry point.
;
LIBRARY RDLIC03
PROTMODE
DATA SINGLE NONSHARED READWRITE LOADONCALL
CODE LOADONCALL
EXPORTS
   RDLIC03
   DLITCBL
```

Figure 131. Sample .DEF file for Remote DL/I DLL

For information on the .DEF file statements and their meanings, see Chapter 25, “Creating Module Definition Files” on page 448.

OS/2

Using DLIBATCH

This section discusses what you can expect when using the DLIBATCH command.

Once all of the preparation is complete, the DLIBATCH command can be used. There are a number of items that will be discussed in this section about the DLIBATCH command:

1. Obtaining the OS/390 userid and password
2. Invoking the target program
3. Supported IMS function codes
4. Syncpoint coordination
5. Diagnostics
6. Interaction with a debugger

OS/390 Userid and Password: If Remote DL/I is not already running, the DLIBATCH command will cause Remote DL/I to become active. The first time Remote DL/I is used on a workstation, Remote DL/I will prompt the user for the OS/390 userid and password that will be used to start the server job. The password information is saved until the workstation is shutdown.

Note: The userid and password information can be updated using the DLILOGIN command.

Invoking the Target Program: Once DLIBATCH has successfully started an IMS batch environment on the host with the specified PSB name, it invokes the target program on the workstation with the PCBs obtained from the IMS batch environment.

Supported Function Codes: All of the function codes that are supported in an IMS batch environment are supported when using DLIBATCH except the following:

- GSCD

Syncpoint Coordination: IMS/ESA provides syncpoint control with DB2 in an IMS batch environment. When using the DLIBATCH command, Remote DL/I does not provide any capability on the workstation to provide syncpoint coordination with DB2/2.

Diagnostics: Error messages produced by Remote DL/I have a prefix of IWZ. (See Appendix H, “Remote DL/I Run-Time Messages” on page 650)

In some cases the error messages will indicate that there was some sort of problem on the server (for example, there could be a JCL error). On the server, you will need to look at the OS/390 message log and the Remote DL/I message log. The OS/390 messages are written to the message data set defined in the TP profile. The Remote DL/I messages are written to the DD IWZRDOUT specified in the JCL in the TP profile.

Using a Debugger: DLIBATCH can be used with the IBM VisualAge COBOL debugger IDBUG. To do this, use the /D: option of the DLIBATCH command.

Remote DL/I Server Environment File

This section describes how you can alter the Remote DL/I server environment using a file on the workstation.

Within a Server Environment File you can specify DD names and the associated data set names that you want the Remote DL/I server to allocate before bringing up the IMS batch environment.

Remote DL/I gets the name of the Server Environment File from the environment variable RMTDLI_SERVER_ENV.

The syntax for specifying a DD name and an associated data set name is:

```
DD=ddname DSN=data_set_name
```

Usage notes:

1. Both the DD name and the data set name must be on the same line.
2. DISP=SHR is used when the data set is allocated.
3. Concatenation is supported only when the DD name is IMS. If more than one library is required for the IMS DD, then there needs to be one line per data set.
4. There is no support to provide anything but a data set name (that is, no support for DD DUMMY or SYSOUT).
5. If the DD name specified in the Server Environment File is is also specified in the JCL for the Remote DL/I TP profile, the dynamic allocation will fail.

Figure 132 shows an example Server Environment File.

```
DD=RDLIDSN  DSN=IMS.DATABASE.RDLIDSN
DD=RDLIDSNO DSN=IMS.DATABASE.RDLIDSNO
DD=IMS      DSN=MYTEST.PSBLIB
DD=IMS      DSN=IMS.PSBLIB
DD=IMS      DSN=IMS.DBDLIB
```

Figure 132. Example Remote DL/I Server Environment File

Checkpoint and Rollback Support

Remote DL/I can be set up to support the ability to checkpoint and rollback database updates. In order to enable Remote DL/I to support checkpoint and rollback, the following has to be done:

- The IMS batch environment that is used must be enabled for checkpoint and rollback calls. In order to enable the IMS batch environment for checkpoint and rollback calls, the following must be done:
  - The IEFRDER DD card must specify a system log that is on direct access storage.
  - Dynamic backout is specified using the IMS batch BKO execution parameter.

If the IMS batch environment is not set up as stated above, rollback calls will get an “AL” status code from IMS.

There is no requirement that the PSB be generated with CMPAT=YES in order to get checkpoint and rollback support.

See Figure 133 on page 644 for an example of the JCL used in a TP profile that enables Remote DL/I for checkpoint and rollback.

```
//RMTDLI   JOB
//*********************************************************************
//* Remote DL/I Server
//*
//*********************************************************************
//RMTDLI    EXEC PGM=IWZRDM01,REGION=8M,
//   PARM='DLI,PGMNAME,PSBNAME,,0000,,,,,,,,,,N,N,,Y'
//*                                                |
//* BKO parameter----------------------------------+
//*
//STEPLIB   DD DISP=SHR,DSN=REMOTE.DLI.LOADLIB
//          DD DISP=SHR,DSN=CEEV1R50.SCEERUN
//          DD DISP=SHR,DSN=IMSVS.IMS5.RESLIB
//DFSRESLB  DD DISP=SHR,DSN=IMSVS.IMS5.RESLIB
//IMS       DD DISP=SHR,DSN=IMS.PSBLIB
//          DD DISP=SHR,DSN=IMS.DBDLIB
//IEFRDER   DD DSN=IMS.RMTDLI.LOG,DISP=SHR        << System log to DASD
//IEFRDER2  DD DUMMY
//SYSUDUMP  DD SYSOUT=*
//DFSVSAMP  DD DISP=SHR,DSN=IMS.DFSVSAMP
//* IMS databases
//RDLIDSN   DD DISP=SHR,DSN=IMS.RDLI.RDLIDSN
//RDLIDSNO  DD DISP=SHR,DSN=IMS.RDLI.RDLIDSNO
//* DD Statements required by Remote IMS
//IWZRDOUT  DD SYSOUT=*
```

Figure 133. Example JCL for a Remote DL/I TP Profile Enabled for Checkpoint and Rollback

VisualAge CICS Support (OS/2 Only)

OS/2

This section describes the support that allows a programmer to develop and test S/390 CICS COBOL programs on VisualAge CICS that use DL/I calls.

Note: On OS/2, VisualAge CICS Enterprise Application Development is required to use Remote DL/I with CICS. On Windows, there is no support to use Remote DL/I with CICS.

Preparing to use Remote DL/I with VisualAge CICS

In order to use Remote DL/I with VisualAge CICS, the following things have to be done and/or considered:

- Any required set up to use VisualAge CICS has to be done. The CICS set up must include setting the EBCDIC code page field in the SIT to a recognized code page.
- A VisualAge CICS user exit program for user exit 15 (user task detach) must be provided which does two things:

1. Ensures that if a PSB is still scheduled when the transaction ends, the PSB is terminated.
2. Ensures that the connection to the Remote DL/I server is ended when the transaction ends.

A sample COBOL program that has the necessary code to do the above is provided in SAMPLES\COBOL\FAAEXP15.CBL. See *VisualAge CICS Customization* on how to build and implement user exits.

Note: When you link the user exit program, you will need to specify the IWZRDLI.LIB library to resolve the Remote DL/I API calls.

- The programs that are going to be run have to be translated, compiled, and link edited with certain options. For example, since Remote DL/I runs using S/390 data types as input and output, specific translator and compiler options that enable S/390 data type support must be used. See “Preparing COBOL Programs” on what needs to be done to prepare COBOL programs to run with Remote DL/I.
- When using VisualAge CICS the IMS batch environment that is used for the Remote DL/I calls must be enabled to do checkpoints and rollbacks. See “Checkpoint and Rollback Support” on page 643 on what needs to be done to enable the IMS batch environment to support checkpoint and rollback.
- The Remote DL/I Server Environment File or the JCL associated with the APPC TP profile that is used to bring up the remote IMS batch environment may need to be modified. For example, if a new PSB was created, the dataset that contains the new PSB must be included in the IMS DD statement. For information on the Server Environment File, see “Remote DL/I Server Environment File” on page 642.

  If you want to use a specific setting of an environment variable when using VisualAge CICS, you can do this by setting the environment variable in the CICSENV.CMD file. You may need to do this for the environment variables RMTDLI_PARTNER_LU, RMTDLI_PARTNER_TP, and RMTDLI_SEVER_ENV.

Preparing COBOL Programs

There are steps that have to be taken to prepare a COBOL program to run with VisualAge CICS. The *VisualAge CICS Programming Guide* has information on what steps need to be done. This section discusses the additional information you need to know to enable COBOL programs to use CBLTDLI with VisualAge CICS.

Translating COBOL Programs

When you translate COBOL programs, you must use the BINARY(S370), EBCDIC, and the LOADCALL translator options.

Compiling COBOL Programs

When you want to use Remote DL/I with COBOL, you must use certain COBOL compiler options. The following compiler options are required:

```
CHAR(EBCDIC)
COLLSEQ(EBCDIC)
FLOAT(HEX)
```

BINARY(S390)
NOTHREAD

A DLIUIB copy file is provided with Remote DL/I in SAMPLES\COBOL\DLIUIB.CBL. You will need ensure that this copy file is available to the COBOL compiler when compiling a COBOL program that contains CBLTDLI calls.

Linking COBOL Programs

The COBOL import library IWZRLIB.LIB has the linkage information for CBLTDLI. No changes have to be done to the VisualAge CICS environment to link COBOL programs that have calls to CBLTDLI.

User Interface Block

Just like CICS on S/390, when running on VisualAge CICS, Remote DL/I provides a UIB when a PSB is scheduled. The UIBRCODE values returned by Remote DL/I are a subset of the values returned when running CICS on S/390. Figure 134 and Figure 135 show the return codes from Remote DL/I when running with VisualAge CICS.

Figure 134. Return Codes in UIBFCTR

Condition	Value
NORESP (normal response)	X'00'
INVREQ (invalid request)	X'08'

Figure 135. Return Codes in UIBDLTR if UIBFCTR=X'08' (INVREQ)

Condition	Value
Invalid argument passed to DL/I	X'00'
PSBSCH (PSB already scheduled)	X'03'
PSBFAIL (the PSB could not be scheduled)	X'05'
TERMNS (termination not successful)	X'07'
FUNCNS (function unscheduled)	X'08'

Supported Function Codes

When using Remote DL/I with VisualAge CICS, the function codes listed in Figure 136 and in Figure 137 on page 647 are supported. When a function code is used that is not in the tables, UIBFCTR is set to X'08' (INVREQ), and UIBDLTR is set to X'00'.

Figure 136 (Page 1 of 2). Supported IMS system service calls under VisualAge CICS

Function Code	Description	Comments
INIT	Initialize. Application receives data availability and deadlock occurrence status codes.	Requires that the PSB is scheduled with the sysserve parameter set to IOPCB.

Figure 136 (Page 2 of 2). Supported IMS system service calls under VisualAge CICS

Function Code	Description	Comments
LOG	Write a message to the system log.	Requires that the PSB is scheduled with the sysserve parameter set to IOPCB.
PCB	Schedule a PSB.	Remote DL/I supports PSBs with up to 100 PCBs.
STAT	Retrieve IMS system statistics.	
TERM	Terminate a PSB. Commit database changes.	Causes a CICS syncpoint.

Figure 137. Supported IMS DB calls under VisualAge CICS

Function Code	Description	Comments
DEQ	Release segments reserved by Q command code.	
DLET	Delete a segment.	
GHN	Get Hold Next.	
GHNP	Get Hold Next in Parent.	
GHU	Get Hold Unique.	
GN	Get Next.	
GNP	Get Next in Parent.	
GU	Get Unique.	
ISRT	Insert.	

Scheduling a PSB

When you use a DL/I call to schedule a PSB, the PCB address list returned depends on the parameters used on the DL/I call.

The PCB call formats supported are as shown in the following syntax diagram:

```
►►──CALL──"CBLTDLI"──USING──function-PCB──,──PSB-name──,──uibptr──────►

►──┬──────────────┬─.──────────────────────────────────────────────►◄
   └─,──sysserve──┘
```

When the *sysserve* parameter is not specified or if specified has the value NOIOPCB, then the PCB list will include only the database PCBs. When the *sysserve* parameter is provided and contains the value IOPCB, then the PCB list will include the I/O PCB, any alternate PCBs, and the database PCBs.

Diagnostics

The PCB list returned will not have any GSAM PCBs even if GSAM PCBs are in the PSB.

Remote DL/I allows either a PSB name in the PSB name field or an asterisk. If the PSB name field is an asterisk, Remote DL/I will use the name of the program associated with the current transaction.

Syncpoint Coordination

Remote DL/I on VisualAge CICS supports syncpoint coordination that behaves like CICS and IMS on the S/390.

Note: In order for commit and rollback processing against the IMS databases to complete successfully, the remote IMS job has to be set up to support rollback. See "Checkpoint and Rollback Support" on page 643 for more information on what has to be done to set up the server to enable the support to checkpoint and rollback database updates.

In order to provide syncpoint coordination, Remote DL/I provides an external resource manager connection with VisualAge CICS. Having a resource manager connection with VisualAge CICS will cause Remote DL/I to get control from VisualAge CICS to process commit and rollback requests. The following list discuses the rollback and commit behavior when running Remote DL/I with VisualAge CICS:

- When a PSB is terminated with a DL/I call, an EXEC CICS SYNCPOINT is issued by Remote DL/I and the IMS database changes are committed.
- When a PSB is terminated implicitly due the normal termination of a CICS run unit (for example, due to an EXEC CICS RETURN), the IMS database changes are committed.
- When an EXEC CICS SYNCPOINT command is used, the PSB is terminated and the IMS database changes are commited.
- When there is an unhandled ABEND, the PSB is terminated and the IMS database changes are rolled back.
- When an EXEC CICS SYNCPOINT command with the ROLLBACK option is used, the PSB is terminated and the the IMS database changes are rolled back.

Diagnostics Using CBLTDLI

If there are any errors while using CBLTDLI with VisualAge CICS, the Remote DL/I messages are written where the COBOL runtime messages are written. If there is a severe error, the transaction ends with abend code 1099. In some cases the error messages will point to some sort of problem on the server (for example, there could be a JCL error). On the server, you will need to look at the OS/390 message log and the Remote DL/I message log. The OS/390 messages are written to the message data set defined in the TP profile. The Remote DL/I messages are written to the DD IWZRDOUT specified in the JCL in the TP profile.

Using a Debugger

The COBOL debugger IDBUG can be used on VisualAge CICS with programs that contain DL/I calls. See *VisualAge CICS Application Programming* for how to use the debugger on VisualAge CICS.

◄ OS/2

Appendix H. Remote DL/I Run-Time Messages

Messages for the Remote DL/I component of VisualAge COBOL contain a message prefix, message number, severity code, and descriptive text. The message prefix is always IWZ, followed by the message number. The severity code will be either I (Information), W (Warning), S (Severe), or C (Critical). The message text provides a brief explanation of the condition.

In the following example message:

```
IWZ301S DL/I-command detected an error.
```

- The message prefix is IWZ.
- The message number is 301.
- The severity code is S.
- The message text is "*DL/I-command* detected an error."

IWZ300S A communication error was encountered when processing a *DL/I-function* request.

Explanation: Remote DL/I encountered a communication error and could not continue.

Programmer Response: Look for other messages that provide additional detail about the communication error.

IWZ301S *DL/I-command* detected an error.

Explanation: The Remote DL/I command detected an error and could not continue.

Programmer Response: Look for other messages that provide additional detail about why the command failed.

IWZ302S An incorrect number of arguments was passed to *DL/I-command*.

Explanation: The Remote DL/I command was passed too few or too many arguments.

Programmer Response: Invoke the command with the correct number of arguments.

IWZ303S Could not load DLL *DLL-name*. *DLL-name2* failed to load.

Explanation: The DLIBATCH command was unable to load DLL *DLL-name*.

Programmer Response: Invoke the DLIBATCH command with a valid DLL name.

IWZ304S Could not find the entry point in DLL ***DLL-name*****.**

Explanation: The DLIBATCH command was unable to find the specified entry point in the DLL *DLL-name*.

Programmer Response: Invoke the DLIBATCH command with a valid entry name for the DLL. Make sure the program has been compiled with the PGMNAME(UPPER) option.

IWZ305S Could not free DLL ***DLL-name*****.**

Explanation: Remote DL/I was unable to free the DLL.

Programmer Response: Exit the process where Remote DL/I is being used and try again.

IWZ306S Could not schedule PSB ***PSB-name*****.**

Explanation: Remote DL/I was unable to schedule PSB *PSB-name*.

Programmer Response: Look for other messages that provide additional detail about the error.

IWZ307S An error occured while trying to obtain a PCB.

Explanation: Remote DL/I encountered an error when it was getting the PCBs associated with the currently scheduled PSB.

Programmer Response: Look for other messages that provide additional detail about the error.

IWZ308S Could not terminate the PSB.

Explanation: Remote DL/I was not able to terminate the PSB. The PSB may have been already terminated due to an IMS abend.

Programmer Response: Look for other messages that provide additional detail about the error. In addition, look at the Remote DL/I transaction program job log on the host system.

IWZ309S Could not allocate a connection with MVS.

Explanation: Remote DL/I was not able to establish a connection with the MVS or OS/390 system specified in the environment variable RMTDLI_PARTNER_TP.

Programmer Response: Look for other messages that provide additional detail about the error.

IWZ310S Generic network communication error message. The CPI-C function is ***CPI-C function*****. The CPI-C return code is** ***CPI-C return code*****.**

Explanation: Remote DL/I was not able to establish a connection with the host system due to a non-zero return code from a CPI-C function.

Programmer Response: Look up the CPI-C return code and correct the problem.

IWZ311S Unable to successfully start the Remote DL/I transaction program on MVS.

Explanation: Remote DL/I was able to connect with the host system, but the Remote DL/I transaction program could not be started.

Programmer Response: This error can occur if there is a JCL error in the transaction program definition. Look at the Remote DL/I transaction program job log on the host to determine the problem.

IWZ312S The userid and password was rejected by MVS.

Explanation: The Remote DL/I transaction program would not start because the MVS or OS/390 userid and password was not accepted as valid by the host system.

Programmer Response: Update your userid and password using the DLILOGIN command.

IWZ313S The Remote DL/I transaction program *TP-name* **could not be started.**

Explanation: The Remote DL/I transaction program would not start.

Programmer Response: Look at the definition of the TP name on the host and make sure ACTIVE is YES.

IWZ314S The Remote DL/I transaction program name *TP-name* **is not defined on MVS.**

Explanation: The Remote DL/I transaction program specified in the environment variable RMTDLI_PARTNER_TP does not exist on the host system.

Programmer Response: Set the environment variable RMTDLI_SERVER_TP to a valid transaction program name.

IWZ315S The environment variable *env-variable* **was not defined.**

Explanation: The environment variable *env-variable* must be set to a valid value.

Programmer Response: Set the environment variable to a valid value.

IWZ316S Invalid value for environment variable *env-variable*.

Explanation: The environment variable *env-variable* has an invalid value. For example, it may contain too many characters.

Programmer Response: Set the environment variable to a valid value.

IWZ317S The network software on the workstation is not active.

Explanation: Remote DL/I was unable to start communications because the network software on the workstation is not active.

Programmer Response: Start the network software on the workstation.

IWZ318S The PARM= value in the Remote DL/I transaction program JCL is not valid.

Explanation: The PARM= value in the Remote DL/I transaction program JCL must start with 'DLI,PGMNAME,PSBNAME'.

Programmer Response: Correct the Remote DL/I transaction program definition.

IWZ319S System error. *Service-name* service failed. Return code: *Service-rc*

Explanation: Remote DL/I encountered a system service error.

Programmer Response: Determine the reason for the system service error.

IWZ320S Unexpected response from the Remote DL/I transaction program. Request: *request-name*. Return code: *return-code*.

Explanation: The Remote DL/I transaction program sent an unexpected response. Possible return codes are:

```
100     ALLOCATION FAILED
200     CHECKPOINT FAILED
201     ROLLBACK FAILED
300     PSB NOT FOUND
301     BAD PARAMETER LIST
302     PCB NOT ALLOWED
500     NO PCBS PASSED
501     OPEN TRACE FILE FAILED
502     UNSUPPORTED IMS ENVIRONMENT
600     COMMUNICATION PUT FAILED
997     IMS ABEND
998     ALLOCATION ERROR
999     UNRECOGNIZED REQUEST
1000    TOO MANY PARMAMETERS
1001    I/O AREA TOO BIG
2001    LOCATE CONTROL BLOCK FAILED
2002    COMMUNICATIONS LOAD FAILED
2003    ALLOCATION IOAREA FAILED
2004    ALLOCATION PCBI FAILED
2005    ALLOCATION PCBR FAILED
```

Programmer Response: Determine the reason for the unexpected response.

IWZ330S A previously established connection is still active.

Explanation: There was an attempt to connect with Remote DL/I more than once.

Programmer Response: Exit the process where Remote DL/I is being used and try again.

IWZ331S A communication link failed.

Explanation: Remote DL/I lost the communication link with the host.

Programmer Response: Exit the process where Remote DL/I is being used and try again.

IWZ332S A connection does not exist.

Explanation: There was an attempt to use Remote DL/I without first establishing a connection.

Programmer Response: Determine the reason for the error.

IWZ333S A PSB is currently scheduled.

Explanation: There was an attempt to schedule a PSB when a PSB is currently scheduled.

Programmer Response: Exit the process where Remote DL/I is being used and try again.

IWZ334S A request to allocate memory failed.

Explanation: Remote DL/I was unable to allocate the memory it needed.

Programmer Response: End other processes that are using memory and try again.

IWZ335S A PSB is not currently scheduled.

Explanation: There was an attempt to perform a Remote DL/I request, but a PSB was not scheduled.

Programmer Response: Schedule the PSB before attempting the Remote DL/I request.

IWZ336S The IMS standard checkpoint call failed.

Explanation: An IMS standard checkpoint failed.

Programmer Response: Look at the Remote DL/I transaction program job log on the host to determine the problem.

IWZ337S The IMS rolllback call failed.

Explanation: An IMS rollback failed.

Programmer Response: Look at the Remote DL/I transaction program job log on the host to determine the problem. Also, make sure the BKO parameter used to bring up the Remote DL/I transaction program is set to 'Y'.

IWZ340S The IMS batch region ended with abend *abend-code***. Make sure the PSB name you specified is correct.**

Explanation: The Remote DL/I transaction program failed with start the IMS batch region.

Programmer Response: Make sure the PSB name is correct. If it is correct, look at the Remote DL/I transaction program job log on the host to determine the problem.

IWZ341S The IMS batch region ended with abend *abend-code***.**

Explanation: The IMS batch region used by the Remote DL/I transaction program ended with an abend.

Programmer Response: Look at the Remote DL/I transaction program job log on the host to determine the problem.

IWZ350S The I/O area provided on a DL/I call was *number-bytes1* **bytes long but** *number-bytes2* **bytes are required to contain the I/O area returned by IMS.**

Explanation: The I/O area provided the caller is not big enough to hold the data returned by IMS.

Programmer Response: Increase the size of the I/O area.

IWZ351S The PCB passed is not a valid PCB.

Explanation: The PCB passed to Remote DL/I is not a valid PCB.

Programmer Response: Change the DL/I call to pass in a valid PCB.

IWZ352S Allocation on MVS failed for DD name *DD-name* with dataset name *dataset-name*.

Explanation: Remote DL/I was unable to dynamically allocate a file specified in the server environment file. Typical problems include data set not found.

Programmer Response: Look at the Remote DL/I transaction program job log on the host to determine the problem.

IWZ353S Invalid specification of the DD name or the dataset name on line *line-number* in file *file-name*.

Explanation: Remote DL/I found an invalid specification of a DD name or a dataset name. Both DD= and DSN= are required on the same line.

Programmer Response: Update the server environment file using the correct specification. Provide both the DD= and DSN= values in upper case on the same line.

IWZ354S Unable to open file *file-name* specified in the RMTDLI_SERVER_ENV environment variable.

Explanation: Remote DL/I was unable to open the file.

Programmer Response: Make sure the file name specified in the environment variable RMTDLI_SERVER_ENV is correct.

IWZ355S Unable to open the Remote DL/I server trace file.

Explanation: The Remote DL/I transaction program was unable to open the server trace file.

Programmer Response: Include a DD for IWZRDTRC in the Remote DL/I transaction program definition.

IWZ356S The Xparm area provided on a DL/I call is not big enough to contain the Xparm area returned by IMS.

Explanation: The caller to DLICall failed to provide an area big enough to hold the data returned by IMS

Programmer Response: Increase the size of the Xparm area.

IWZ360S Could not disconnect.

Explanation: Remote DL/I was unable to successfully disconnect from the Remote DL/I transaction program. This may occur if the communications link was lost.

Programmer Response: Exit the process where Remote DL/I is being used and try again.

IWZ380S Error occured when trying to communicate with the Remote DL/I server program.

Explanation: Remote DL/I encounted an error.

Programmer Response: Look for other messages that provide additional detail about the error.

IWZ382S Errors occured when scheduling PSB *PSB-name*

Explanation: Remote DL/I encounted an error when scheduling a PSB.

Programmer Response: Look for other messages that provide additional detail about the error.

IWZ383S Starting a process for DLIBATCH failed with errno *error-number***.**

Explanation: Remote DL/I encounted an error when starting a process.

Programmer Response: Determine the reason for the error.

IWZ384S *command* **was invoked with an invalid option,** *option-value***.**

Explanation: An invalid option was used in the specified command.

Programmer Response: Remove the incalide option.

IWZ390S An error occured while processing the *parameter-type* **parameter.**

Explanation: Invalid data or an invalid address of a parameter was passed to Remote DL/I.

Programmer Response: Validate that all the parameters passed are correct.

IWZ391S The *parameter-type* **is invalid.**

Explanation: Invalid data or an invalid address of a parameter was passed to Remote DL/I.

Programmer Response: Validate that all the parameters passed are correct.

Bibliography

VisualAge COBOL

Building Parts for Fun and Profit, GC26-9038

COBOL Resource Catalog, GC26-8488

Fact Sheet, GC26-9052

Getting Started on OS/2, GC26-9051

Getting Started on Windows, GC26-8944

Introducing Redeveloper, SC26-9056

Language Reference, SC26-9046

Programming Guide, SC26-9050

Technology Brochure, GC26-9060

Visual Builder User's Guide, SC26-9053

Related Publications

COBOL for OS/390 & VM

Compiler and Run-Time Migration Guide, GC26-4764

Debug Tool User's Guide and Reference, SC09-2137

Diagnosis Guide, GC26-9047

Fact Sheet, GC26-9048

Installation and Customization under OS/390, GC26-9045

Language Reference, SC26-9046

Licensed Program Specifications, GC26-9044

Programming Guide, SC26-9049

COBOL Set for AIX

Fact Sheet, GC26-8484

Getting Started, GC26-8425

Language Reference, SC26-9046

LPEX User's Guide and Reference, SC09-2202

Program Builder User's Guide, SC09-2201

Programming Guide, SC26-8423

VisualAge CICS Enterprise Application Development

Installation, GC34-5356

Customization, SC34-5357

Operation, SC34-5358

Reference Summary, SX33-6109

Intercommunication, SC34-5359

Problem Determination, GC34-5360

Performance, SC34-5363

Application Programming, SC34-5361

Messages and Codes, GC34-5362

CICS for OS/2

Application Programming, SC33-1585

Customization, SC33-1581

Installation, GC33-1580

Intercommunication, SC33-1583

Messages & Codes, SC33-1586

Operation, SC33-1582

Problem Determination, SC33-1584

Reference Summary, SX33-6100

CICS for Windows NT

Application Programming Guide, SC33-1888

Installation Guide, GC33-1880

Intercommunication Guide, SC33-1882

Messages & Codes, SC33-1886

Problem Determination Guide, SC33-1883

DB2

Application Programming Guide, S20H-4643

DATABASE 2 Command Reference for Common Servers, S20H-4645

SQL Reference, S20H-4665

SMARTdata Utilities for OS/2

Data Description and Conversion A Data Language Reference, SC26-7092

Data Description and Conversion, SC26-7091

VSAM in a Distributed Environment, SC26-7063

SMARTdata Utilities for Windows

Data Description and Conversion A Data Language Reference, SC26-7092

Data Description and Conversion, SC26-7091

User's Guide, SC26-7134

VSAM Reference, SC26-7133

SOMobjects Developer's Toolkit

SOMobjects Developer's Toolkit Programmer's Reference

SOMobjects Developer's Toolkit Programming Guide

SOMobjects Developer's Toolkit User's Guide

Other

Btrieve Programmer's Manual

Glossary

The terms in this glossary are defined in accordance with their meaning in COBOL. These terms may or may not have the same meaning in other languages.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the following publications:

- *American National Standard Programming Language COBOL, ANSI X3.23-1985* (Copyright 1985 American National Standards Institute, Inc.), which was prepared by Technical Committee X3J4, which had the task of revising American National Standard COBOL, X3.23-1974.
- *American National Dictionary for Information Processing Systems* (Copyright 1982 by the Computer and Business Equipment Manufacturers Association).

American National Standard definitions are preceded by an asterisk (*).

A

*** abbreviated combined relation condition**. The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

abend. Abnormal termination of program.

*** access mode**. The manner in which records are to be operated upon within a file.

*** actual decimal point**. The physical representation, using the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

*** alphabet-name**. A user-defined word, in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION, that assigns a name to a specific character set and/or collating sequence.

*** alphabetic character**. A letter or a space character.

*** alphanumeric character**. Any character in the computer's character set.

alphanumeric-edited character. A character within an alphanumeric character-string that contains at least one B, 0 (zero), or / (slash).

*** alphanumeric function**. A function whose value is composed of a string of one or more characters from the computer's character set.

*** alternate record key**. A key, other than the prime record key, whose contents identify a record within an indexed file.

ANSI (American National Standards Institute). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

*** argument**. An identifier, a literal, an arithmetic expression, or a function-identifier that specifies a value to be used in the evaluation of a function.

*** arithmetic expression**. An identifier of a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

*** arithmetic operation**. The process caused by the execution of an arithmetic statement, or the evaluation of an arithmetic expression, that results in a mathematically correct solution to the arguments presented.

*** arithmetic operator**. A single character, or a fixed two-character combination that belongs to the following set:

Character	Meaning
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

*** arithmetic statement**. A statement that causes an arithmetic operation to be executed. The arithmetic statements are the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements.

array. In Language Environment, an aggregate consisting of data objects, each of which may be uniquely referenced by subscripting. Roughly analogous to a COBOL table.

*** ascending key**. A key upon the values of which data is ordered, starting with the lowest value of the key up to the highest value of the key, in accordance with the rules for comparing data items.

ASCII. American National Standard Code for Information Interchange. The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange between data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

Extension: IBM has defined an extension to ASCII code (characters 128-255).

assignment-name. A name that identifies the organization of a COBOL file and the name by which it is known to the system.

*** assumed decimal point**. A decimal point position that does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning with no physical representation.

*** AT END condition**. A condition caused:

1. During the execution of a READ statement for a sequentially accessed file, when no next logical record exists in the file, or when the number of significant digits in the relative record number is larger than the size of the relative key data item, or when an optional input file is not present.
2. During the execution of a RETURN statement, when no next logical record exists for the associated sort or merge file.
3. During the execution of a SEARCH statement, when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

B

big-endian. Default format used by the mainframe and the AIX workstation to store binary data. In this format, the least significant digit is on the highest address. Compare with “little-endian.”

binary item. A numeric data item represented in binary notation (on the base 2 numbering system). Binary items have a decimal equivalent consisting of the decimal digits 0 through 9, plus an operational sign. The leftmost bit of the item is the operational sign.

binary search. A dichotomizing search in which, at each step of the search, the set of data elements is divided by two; some appropriate action is taken in the case of an odd number.

*** block**. A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block may contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical record(s) that are either contained within the block or that overlap the block. The term is synonymous with physical record.

breakpoint. A place in a computer program, usually specified by an instruction, where its execution may be interrupted by external intervention or by a monitor program.

Btrieve. A key-indexed record management system that allows applications to manage records by key value, sequential access method, or random access method. IBM COBOL supports COBOL sequential and indexed file I-O language through Btrieve.

buffer. A portion of storage used to hold input or output data temporarily.

built-in function. See “intrinsic function.”

byte. A string consisting of a certain number of bits, usually eight, treated as a unit, and representing a character.

C

callable services. In Language Environment, a set of services that can be invoked by a COBOL program using the conventional Language Environment-defined call interface, and usable by all programs sharing the Language Environment conventions.

called program. A program that is the object of a CALL statement.

*** calling program**. A program that executes a CALL to another program.

case structure. A program processing logic in which a series of conditions is tested in order to make a choice between a number of resulting actions.

century window. A century window is a 100-year interval within which any 2-digit year is unique. There are several types of century window available to COBOL programmers:

1. For windowed date fields, the YEARWINDOW compiler option
2. For windowing intrinsic functions DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD, and YEAR-TO-YYYY, it is specified by argument-2
3. For date and time callable services, it is specified in CEESCEN

*** character**. The basic indivisible unit of the language.

character position. The amount of physical storage required to store a single standard data format character described as USAGE IS DISPLAY.

character set. All the valid characters for a programming language or a computer system.

*** character-string**. A sequence of contiguous characters that form a COBOL word, a literal, a PICTURE character-string, or a comment-entry. Must be delimited by separators.

checkpoint. A point at which information about the status of a job and the system can be recorded so that the job step can be later restarted.

*** class**. The entity that defines common behavior and implementation for zero, one, or more objects. The objects that share the same implementation are considered to be objects of the same class.

*** class condition**. The proposition, for which a truth value can be determined, that the content of an item is wholly alphabetic, is wholly numeric, or consists exclusively of those characters listed in the definition of a class-name.

*** Class Definition**. The COBOL source unit that defines a class.

*** class identification entry**. An entry in the CLASS-ID paragraph of the IDENTIFICATION DIVISION which contains clauses that specify the class-name and assign selected attributes to the class definition.

*** class-name**. A user-defined word defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION that assigns a name to the proposition for which a truth value can be defined, that the content of a data item consists exclusively of those characters listed in the definition of the class-name.

class object. The run-time object representing a SOM class.

*** clause**. An ordered set of consecutive COBOL character-strings whose purpose is to specify an attribute of an entry.

CMS (Conversational Monitor System). A virtual machine operating system that provides general interactive, time-sharing, problem solving, and program development capabilities, and that operates only under the control of the VM/SP control program.

*** COBOL character set**. The complete COBOL character set consists of the characters listed below:

Character	Meaning
0,1...,9	digit
A,B,...,Z	uppercase letter
a,b,...,z	lowercase letter
ƀ	space
+	plus sign
-	minus sign (hyphen)
*	asterisk
/	slant (virgule, slash)
=	equal sign
$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point, full stop)
"	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol
:	colon

*** COBOL word**. See “word.”

code page. An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for 8-bit code, assignment of characters and meanings to 128 code points for 7-bit code.

*** collating sequence**. The sequence in which the characters that are acceptable to a computer are ordered for purposes of sorting, merging, comparing, and for processing indexed files sequentially.

*** column**. A character position within a print line. The columns are numbered from 1, by 1, starting at the leftmost character position of the print line and extending to the rightmost position of the print line.

*** combined condition**. A condition that is the result of connecting two or more conditions with the AND or the OR logical operator.

*** comment-entry**. An entry in the IDENTIFICATION DIVISION that may be any combination of characters from the computer's character set.

*** comment line**. A source program line represented by an asterisk (*) in the indicator area of the line and any characters from the computer's character set in area A and area B of that line. The comment line serves only for documentation in a program. A special form of comment line represented by a slant (/) in the indicator area of the line and any characters from the computer's character set in area A and area B of that line causes page ejection prior to printing the comment.

*** common program**. A program which, despite being directly contained within another program, may be called from any program directly or indirectly contained in that other program.

compatible dates. The meaning of the term "compatible," when applied to date fields, depends on the COBOL division in which the usage occurs:

- **Data Division**

  Two date fields are compatible if they have identical USAGE and meet at least one of the following conditions:

  - They have the same date format.
  - Both are windowed date fields, where one consists only of a windowed year, date format YY.
  - Both are expanded date fields, where one consists only of an expanded year, date format YYYY.
  - One has date format YYXXXX, the other, YYXX.
  - One has date format YYYYXXXX, the other, YYYYXX.

- **Procedure Division**

  Two date fields are compatible if they have the same date format except for the year part, which may be windowed or expanded. For example, a windowed date field with date format YYXXX is compatible with:

  - Another windowed date field with date format YYXXX
  - An expanded date field with date format YYYYXXX

*** compile**. (1) To translate a program expressed in a high-level language into a program expressed in an intermediate language, assembly language, or a computer language. (2) To prepare a machine language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one computer instruction for each symbolic statement, or both, as well as performing the function of an assembler.

*** compile time**. The time at which a COBOL source program is translated, by a COBOL compiler, to a COBOL object program.

compiler. A program that translates a program written in a higher level language into a machine language object program.

compiler directing statement. A statement, beginning with a compiler directing verb, that causes the compiler to take a specific action during compilation.

compiler directing statement. A statement that specifies actions to be taken by the compiler during processing of a COBOL source program. Compiler directives are contained in the COBOL source program. Thus, you can specify different suboptions of the directive within the source program by using multiple compiler directive statements in the program.

*** complex condition**. A condition in which one or more logical operators act upon one or more conditions. (See also "negated simple condition," "combined condition," and "negated combined condition.")

*** computer-name**. A system-name that identifies the computer upon which the program is to be compiled or run.

condition. An exception that has been enabled, or recognized, by Language Environment and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and results in an interrupt. They can also be detected by language-specific generated code or language library code.

*** condition**. A status of a program at run time for which a truth value can be determined. Where the term ‘condition’ (condition-1, condition-2,...) appears in these language specifications in or in reference to ‘condition’ (condition-1, condition-2,...) of a general format, it is a conditional expression consisting of either a simple condition optionally parenthesized, or a combined condition consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses, for which a truth value can be determined.

*** conditional expression**. A simple condition or a complex condition specified in an EVALUATE, IF, PERFORM, or SEARCH statement. (See also “simple condition” and “complex condition.”)

*** conditional phrase**. A conditional phrase specifies the action to be taken upon determination of the truth value of a condition resulting from the execution of a conditional statement.

*** conditional statement**. A statement specifying that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value.

*** conditional variable**. A data item one or more values of which has a condition-name assigned to it.

*** condition-name**. A user-defined word that assigns a name to a subset of values that a conditional variable may assume; or a user-defined word assigned to a status of an implementor defined switch or device. When ‘condition-name’ is used in the general formats, it represents a unique data item reference consisting of a syntactically correct combination of a ‘condition-name’, together with qualifiers and subscripts, as required for uniqueness of reference.

*** condition-name condition**. The proposition, for which a truth value can be determined, that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

*** CONFIGURATION SECTION**. A section of the ENVIRONMENT DIVISION that describes overall specifications of source and object programs and class definitions.

CONSOLE. A COBOL environment-name associated with the operator console.

*** contiguous items**. Items that are described by consecutive entries in the Data Division, and that bear a definite hierarchic relationship to each other.

copybook. A file or library member containing a sequence of code that is included in the source program at compile time using the COPY statement. The file can be created by the user, supplied by COBOL, or supplied by another product.

CORBA. The Common Object Request Broker Architecture established by the Object Management Group. IBM's *Interface Definition Language* used to describe the *interface* for SOM classes is fully compliant with CORBA standards.

*** counter**. A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

cross-reference listing. The portion of the compiler listing that contains information on where files, fields, and indicators are defined, referenced, and modified in a program.

currency sign. The character ‘$’ of the COBOL character set or that character defined by the CURRENCY compiler option. If the NOCURRENCY compiler option is in effect, the currency sign is defined as the character ‘$’.

currency symbol. The character defined by the CURRENCY compiler option or by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. If the NOCURRENCY compiler option is in effect for a COBOL source program and the CURRENCY SIGN clause is also **not** present in the source program, the currency symbol is identical to the currency sign.

*** current record**. In file processing, the record that is available in the record area associated with a file.

*** current volume pointer**. A conceptual entity that points to the current volume of a sequential file.

D

*** data clause**. A clause, appearing in a data description entry in the DATA DIVISION of a COBOL program, that provides information describing a particular attribute of a data item.

*** data description entry** . An entry in the DATA DIVISION of a COBOL program that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

DATA DIVISION. One of the four main components of a COBOL program, class definition, or method definition. The DATA DIVISION describes the data to be processed by the object program, class, or method: files to be used and the records contained within them; internal working-storage records that will be needed; data to be made available in more than one program in the COBOL run unit. (Note, the Class DATA DIVISION contains only the WORKING-STORAGE SECTION.)

*** data item**. A unit of data (excluding literals) defined by a COBOL program or by the rules for function evaluation.

*** data-name**. A user-defined word that names a data item described in a data description entry. When used in the general formats, 'data-name' represents a word that must not be reference-modified, subscripted or qualified unless specifically permitted by the rules for the format.

date field. Any of the following:

- A data item whose data description entry includes a DATE FORMAT clause.
- A value returned by one of the following intrinsic functions:

  DATE-OF-INTEGER
  DATE-TO-YYYYMMDD
  DATEVAL
  DAY-OF-INTEGER
  DAY-TO-YYYYDDD
  YEAR-TO-YYYY
  YEARWINDOW
- The conceptual data items DATE, DATE YYYYMMDD, DAY, and DAY YYYYDDD in the Format 2 ACCEPT statement.
- The result of certain arithmetic operations.

The term date field refers to both "expanded date field" and "windowed date field." See also "non-date."

date format. The date pattern of a date field, specified either:

- Explicitly, by the DATE FORMAT clause or DATEVAL intrinsic function

or

- Implicitly, by statements and intrinsic functions that return date fields.

DBCS (Double-Byte Character Set). See "Double-Byte Character Set (DBCS)."

*** debugging line**. A debugging line is any line with a 'D' in the indicator area of the line.

*** debugging section**. A section that contains a USE FOR DEBUGGING statement.

*** declarative sentence**. A compiler directing sentence consisting of a single USE statement terminated by the separator period.

*** declaratives**. A set of one or more special purpose sections, written at the beginning of the Procedure Division, the first of which is preceded by the key word DECLARATIVES and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler directing sentence, followed by a set of zero, one, or more associated paragraphs.

*** de-edit**. The logical removal of all editing characters from a numeric edited data item in order to determine that item's unedited numeric value.

*** delimited scope statement**. Any statement that includes its explicit scope terminator.

*** delimiter**. A character or a sequence of contiguous characters that identify the end of a string of characters and separate that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

*** descending key**. A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

digit. Any of the numerals from 0 through 9. In COBOL, the term is not used in reference to any other symbol.

*** digit position**. The amount of physical storage required to store a single digit. This amount may vary depending on the usage specified in the data description entry that defines the data item.

*** direct access**. The facility to obtain data from storage devices or to enter data into a storage device in such a way that the process depends only on the location of that data and not on a reference to data previously accessed.

*** division**. A collection of zero, one or more sections or paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules.

Each division consists of the division header and the related division body. There are four (4) divisions in a COBOL program: Identification, Environment, Data, and Procedure.

*** division header**. A combination of words followed by a separator period that indicates the beginning of a division. The division headers are:

IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.

DLL. See “dynamic link library.”

do construction. In structured programming, a DO statement is used to group a number of statements in a procedure. In COBOL, an in-line PERFORM statement functions in the same way.

do-until. In structured programming, a do-until loop will be executed at least once, and until a given condition is true. In COBOL, a TEST AFTER phrase used with the PERFORM statement functions in the same way.

do-while. In structured programming, a do-while loop will be executed if, and while, a given condition is true. In COBOL, a TEST BEFORE phrase used with the PERFORM statement functions in the same way.

Double-Byte Character Set (DBCS). A set of characters in which each character is represented by two bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require Double-Byte Character Sets. Because each character requires two bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software that are DBCS-capable.

*** dynamic access**. An access mode in which specific logical records can be obtained from or placed into a mass storage file in a nonsequential manner and obtained from a file in a sequential manner during the scope of the same OPEN statement.

dynamic link library. A file containing executable code and data bound to a program at load time or run time, rather than during linking. The code and data in a dynamic link library can be shared by several applications simultaneously.

Dynamic Storage Area (DSA). Dynamically acquired storage composed of a register save area and an area available for dynamic storage allocation (such as program variables). DSAs are generally allocated within STACK segments managed by Language Environment.

E

*** EBCDIC (Extended Binary-Coded Decimal Interchange Code)**. A coded character set consisting of 8-bit coded characters.

EBCDIC character. Any one of the symbols included in the 8-bit EBCDIC (Extended Binary-Coded-Decimal Interchange Code) set.

edited data item. A data item that has been modified by suppressing zeroes and/or inserting editing characters.

*** editing character**. A single character or a fixed two-character combination belonging to the following set:

Character	**Meaning**
ƀ	space
0	zero
+	plus
-	minus
CR	credit
DB	debit
Z	zero suppress
*	check protect
$	currency sign
,	comma (decimal point)
.	period (decimal point)
/	slant (virgule, slash)

element (text element). One logical unit of a string of text, such as the description of a single data item or verb, preceded by a unique code identifying the element type.

*** elementary item**. A data item that is described as not being further logically subdivided.

enclave. When running under the Language Environment product, an enclave is analogous to a run unit. An enclave can create other enclaves on OS/390 and CMS by a LINK, on CMS by CMSCALL, and the use of the system () function of C.

***end class header**. A combination of words, followed by a separator period, that indicates the end of a COBOL class definition. The end class header is:

```
END CLASS class-name.
```

***end method header**. A combination of words, followed by a separator period, that indicates the end of a COBOL method definition. The end method header is:

```
END METHOD method-name.
```

*** end of Procedure Division**. The physical position of a COBOL source program after which no further procedures appear.

*** end program header**. A combination of words, followed by a separator period, that indicates the end of a COBOL source program. The end program header is:

```
END PROGRAM program-name.
```

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

*** environment clause**. A clause that appears as part of an ENVIRONMENT DIVISION entry.

ENVIRONMENT DIVISION. One of the four main component parts of a COBOL program, class definition, or method definition. The ENVIRONMENT DIVISION describes the computers upon which the source program is compiled and those on which the object program is executed, and provides a linkage between the logical concept of files and their records, and the physical aspects of the devices on which files are stored.

environment-name. A name, specified by IBM, that identifies system logical units, printer and card punch control characters, report codes, and/or program switches. When an environment-name is associated with a mnemonic-name in the ENVIRONMENT DIVISION, the mnemonic-name may then be substituted in any format in which such substitution is valid.

environment variable. Any of a number of variables that describe the way an operating system is going to run and the devices it is going to recognize.

execution time. See “run time.”

execution-time environment. See “run-time environment.”

expanded date field. A date field containing an expanded (4-digit) year. See also “date field” and “expanded year.”

expanded year. Four digits representing a year, including the century (for example, 1998). Appears in expanded date fields. Compare with “windowed year.”

*** explicit scope terminator**. A reserved word that terminates the scope of a particular Procedure Division statement.

exponent. A number, indicating the power to which another number (the base) is to be raised. Positive exponents denote multiplication, negative exponents denote division, fractional exponents denote a root of a quantity. In COBOL, an exponential expression is indicated with the symbol ‘**’ followed by the exponent.

*** expression**. An arithmetic or conditional expression.

*** extend mode**. The state of a file after execution of an OPEN statement, with the EXTEND phrase specified for that file, and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

extensions. Certain COBOL syntax and semantics supported by IBM compilers in addition to those described in ANSI Standard.

*** external data**. The data described in a program as external data items and external file connectors.

*** external data item**. A data item which is described as part of an external record in one or more programs of a run unit and which itself may be referenced from any program in which it is described.

*** external data record**. A logical record which is described in one or more programs of a run unit and whose constituent data items may be referenced from any program in which they are described.

external decimal item. A format for representing numbers in which the digit is contained in bits 4 through 7 and the sign is contained in bits 0 through 3 of the rightmost byte. Bits 0 through 3 of all other bytes contain 1’s (hex F). For example, the decimal value of +123 is represented as 1111 0001 1111 0010 1111 0011. (Also know as “zoned decimal item.”)

*** external file connector**. A file connector which is accessible to one or more object programs in the run unit.

external floating-point item. A format for representing numbers in which a real number is represented by a pair of distinct numerals. In a floating-point representation, the real number is the product of the fixed-point part (the first numeral), and a value obtained by raising the implicit floating-point base to a power denoted by the exponent (the second numeral).

For example, a floating-point representation of the number 0.0001234 is: 0.1234 -3, where 0.1234 is the mantissa and -3 is the exponent.

external program. The outermost program. A program that is not nested.

*** external switch**. A hardware or software device, defined and named by the implementor, which is used to indicate that one of two alternate states exists.

F

*** figurative constant**. A compiler-generated value referenced through the use of certain reserved words.

*** file**. A collection of logical records.

*** file attribute conflict condition**. An unsuccessful attempt has been made to execute an input-output operation on a file and the file attributes, as specified for that file in the program, do not match the fixed attributes for that file.

*** file clause**. A clause that appears as part of any of the following DATA DIVISION entries: file description entry (FD entry) and sort-merge file description entry (SD entry).

*** file connector**. A storage area which contains information about a file and is used as the linkage between a file-name and a physical file and between a file-name and its associated record area.

File-Control. The name of an ENVIRONMENT DIVISION paragraph in which the data files for a given source program are declared.

*** file control entry**. A SELECT clause and all its subordinate clauses which declare the relevant physical attributes of a file.

*** file description entry**. An entry in the File Section of the DATA DIVISION that is composed of the level indicator FD, followed by a file-name, and then followed by a set of file clauses as required.

*** file-name**. A user-defined word that names a file connector described in a file description entry or a sort-merge file description entry within the File Section of the DATA DIVISION.

*** file organization**. The permanent logical file structure established at the time that a file is created.

***file position indicator**. A conceptual entity that contains the value of the current key within the key of reference for an indexed file, or the record number of the current record for a sequential file, or the relative record number of the current record for a relative file, or indicates that no next logical record exists, or that an optional input file is not present, or that the at end condition already exists, or that no valid next record has been established.

*** File Section**. The section of the DATA DIVISION that contains file description entries and sort-merge file description entries together with their associated record descriptions.

file system. The collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

*** fixed file attributes**. Information about a file which is established when a file is created and cannot subsequently be changed during the existence of the file. These attributes include the organization of the file (sequential, relative, or indexed), the prime record key, the alternate record keys, the code set, the minimum and maximum record size, the record type (fixed or variable), the collating sequence of the keys for indexed files, the blocking factor, the padding character, and the record delimiter.

*** fixed length record**. A record associated with a file whose file description or sort-merge description entry requires that all records contain the same number of character positions.

fixed-point number. A numeric data item defined with a PICTURE clause that specifies the location of an optional sign, the number of digits it contains, and the location of an optional decimal point. The format may be either binary, packed decimal, or external decimal.

floating-point number. A numeric data item containing a fraction and an exponent. Its value is obtained by multiplying the fraction by the base of the numeric data item raised to the power specified by the exponent.

*** format**. A specific arrangement of a set of data.

*** function**. A temporary data item whose value is determined at the time the function is referenced during the execution of a statement.

*** function-identifier**. A syntactically correct combination of character-strings and separators that references a function. The data item represented by a function is uniquely identified by a function-name with its arguments, if any. A function-identifier may include a reference-modifier. A function-identifier that references an alphanumeric function may be specified anywhere in the general formats that an identifier may be specified, subject to certain restrictions. A function-identifier that references an integer or numeric function may be referenced anywhere in the general formats that an arithmetic expression may be specified.

function-name. A word that names the mechanism whose invocation, along with required arguments, determines the value of a function.

G

*** global name**. A name which is declared in only one program but which may be referenced from that program and from any program contained within that program. Condition-names, data-names, file-names, record-names, report-names, and some special registers may be global names.

*** group item**. A data item that is composed of subordinate data items.

H

header label. (1) A file label or data set label that precedes the data records on a unit of recording media. (2) Synonym for beginning-of-file label.

*** high order end**. The leftmost character of a string of characters.

I

IBM COBOL extension. Certain COBOL syntax and semantics supported by IBM compilers in addition to those described in ANSI Standard.

IDENTIFICATION DIVISION. One of the four main component parts of a COBOL program, class definition, or method definition. The IDENTIFICATION DIVISION identifies the program name, class name, or method name. The IDENTIFICATION DIVISION may include the following documentation: author name, installation, or date.

*** identifier**. A syntactically correct combination of character-strings and separators that names a data item. When referencing a data item that is not a function, an identifier consists of a data-name, together with its qualifiers, subscripts, and reference-modifier, as required for uniqueness of reference. When referencing a data item which is a function, a function-identifier is used.

IGZCBSN. The COBOL/370 Release 1 bootstrap routine. It must be link-edited with any module that contains a COBOL/370 Release 1 program.

IGZCBSO. The COBOL for MVS & VM Release 2 and IBM COBOL for OS/390 & VM bootstrap routine. It must be link-edited with any module that contains a COBOL for MVS & VM Release 2 or IBM COBOL for OS/390 & VM program.

*** imperative statement**. A statement that either begins with an imperative verb and specifies an unconditional action to be taken or is a conditional statement that is delimited by its explicit scope terminator (delimited scope statement). An imperative statement may consist of a sequence of imperative statements.

*** implicit scope terminator**. A separator period which terminates the scope of any preceding unterminated statement, or a phrase of a statement which by its occurrence indicates the end of the scope of any statement contained within the preceding phrase.

*** index**. A computer storage area or register, the content of which represents the identification of a particular element in a table.

*** index data item**. A data item in which the values associated with an index-name can be stored in a form specified by the implementor.

indexed data-name. An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

*** indexed file**. A file with indexed organization.

*** indexed organization**. The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

indexing. Synonymous with subscripting using index-names.

*** index-name**. A user-defined word that names an index associated with a specific table.

*** inheritance (for classes)**. A mechanism for using the implementation of one or more *classes* as the basis for another class. A *sub-class* inherits from one or more *super-classes*. By definition the inheriting class conforms to the inherited classes.

*** initial program**. A program that is placed into an initial state every time the program is called in a run unit.

*** initial state**. The state of a program when it is first called in a run unit.

inline. In a program, instructions that are executed sequentially, without branching to routines, subroutines, or other programs.

*** input file**. A file that is opened in the INPUT mode.

*** input mode**. The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

*** input-output file**. A file that is opened in the I-O mode.

*** INPUT-OUTPUT SECTION**. The section of the ENVIRONMENT DIVISION that names the files and the external media required by an object program or method and that provides information required for transmission and handling of data during execution of the object program or method definition.

*** Input-Output statement**. A statement that causes files to be processed by performing operations upon individual records or upon the file as a unit. The input-output statements are: ACCEPT (with the identifier phrase), CLOSE, DELETE, DISPLAY, OPEN, READ, REWRITE, SET (with the TO ON or TO OFF phrase), START, and WRITE.

*** input procedure**. A set of statements, to which control is given during the execution of a SORT statement, for the purpose of controlling the release of specified records to be sorted.

instance data. Data defining the state of an object. The instance data introduced by a class is defined in the WORKING-STORAGE SECTION of the DATA DIVISION of the class definition. The state of an object also includes the state of the instance variables introduced by base classes that are inherited by the current class. A separate copy of the instance data is created for each object instance.

*** integer**. (1) A numeric literal that does not include any digit positions to the right of the decimal point.

(2) A numeric data item defined in the DATA DIVISION that does not include any digit positions to the right of the decimal point.

(3) A numeric function whose definition provides that all digits to the right of the decimal point are zero in the returned value for any possible evaluation of the function.

integer function. A function whose category is numeric and whose definition does not include any digit positions to the right of the decimal point.

interface. The information that a *client* must know to use a *class*—the names of its *attributes* and the signatures of its *methods*. With direct-to-SOM compilers such as COBOL, the interface to a class may be defined by native language syntax for class definitions. Classes implemented in other languages might have their interfaces defined directly in SOM Interface Definition Language (IDL). The COBOL compiler has a compiler option, IDLGEN, to automatically generate IDL for a COBOL class.

Interface Definition Language (IDL). The formal language (independent of any programming language) by which the *interface* for a class of *objects* is defined in a IDL file, which the SOM compiler then interprets to create an implementation template file and binding files. SOM's Interface Definition Language is fully compliant with standards established by the Object Management Group's Common Object Request Broker Architecture (*CORBA*).

interlanguage communication (ILC). The ability of routines written in different programming languages to communicate. ILC support allows the application writer to readily build applications from component routines written in a variety of languages.

intermediate result. An intermediate field containing the results of a succession of arithmetic operations.

*** internal data**. The data described in a program excluding all external data items and external file connectors. Items described in the LINKAGE SECTION of a program are treated as internal data.

*** internal data item**. A data item which is described in one program in a run unit. An internal data item may have a global name.

internal decimal item. A format in which each byte in a field except the rightmost byte represents two numeric digits. The rightmost byte contains one digit and the sign. For example, the decimal value +123 is represented as 0001 0010 0011 1111. (Also known as packed decimal.)

*** internal file connector**. A file connector which is accessible to only one object program in the run unit.

*** intra-record data structure**. The entire collection of groups and elementary data items from a logical record which is defined by a contiguous subset of the data description entries which describe that record. These data description entries include all entries whose level-number is greater than the level-number of the first data description entry describing the intra-record data structure.

intrinsic function. A pre-defined function, such as a commonly used arithmetic function, called by a built-in function reference.

*** invalid key condition**. A condition, at object time, caused when a specific value of the key associated with an indexed or relative file is determined to be invalid.

*** I-O-CONTROL**. The name of an ENVIRONMENT DIVISION paragraph in which object program requirements for rerun points, sharing of same areas by several data files, and multiple file storage on a single input-output device are specified.

*** I-O-CONTROL entry**. An entry in the I-O-CONTROL paragraph of the ENVIRONMENT DIVISION which contains clauses that provide information required for the transmission and handling of data on named files during the execution of a program.

*** I-O-Mode**. The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phase for that file.

*** I-O status**. A conceptual entity which contains the two-character value indicating the resulting status of an input-output operation. This value is made available to the program through the use of the FILE STATUS clause in the file control entry for the file.

ISPF. Interactive System Productivity Facility. An IBM software product that provides a menu-driven interface to the TSO or VM user. Includes library utilities, a powerful editor, and dialog management.

iteration structure. A program processing logic in which a series of statements is repeated while a condition is true or until a condition is true.

K

K. When referring to storage capacity, two to the tenth power; 1024 in decimal notation.

*** key**. A data item that identifies the location of a record, or a set of data items which serve to identify the ordering of data.

*** key of reference**. The key, either prime or alternate, currently being used to access records within an indexed file.

*** key word**. A reserved word or function-name whose presence is required when the format in which the word appears is used in a source program.

kilobyte (KB). One kilobyte equals 1024 bytes.

L

*** language-name**. A system-name that specifies a particular programming language.

Language Environment-conforming. A characteristic of compiler products (COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, AD/Cycle C/370, C/C++ for MVS and VM, PL/I for MVS and VM) that produce object code conforming to the Language Environment format.

last-used state. A program is in last-used state if its internal values remain the same as when the program was exited (are not reset to their initial values).

*** letter**. A character belonging to one of the following two sets:

1. Uppercase letters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
2. Lowercase letters: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

*** level indicator**. Two alphabetic characters that identify a specific type of file or a position in a hierarchy. The level indicators in the DATA DIVISION are: CD, FD, and SD.

*** level-number**. A user-defined word, expressed as a two digit number, which indicates the hierarchical position of a data item or the special properties of a data description entry. Level-numbers in the range from 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. Level-numbers in the range 1 through 9 may be written either as a single digit or as a zero followed by a significant digit. Level-numbers 66, 77 and 88 identify special properties of a data description entry.

*** library-name**. A user-defined word that names a COBOL library that is to be used by the compiler for a given source program compilation.

*** library text**. A sequence of text words, comment lines, the separator space, or the separator pseudo-text delimiter in a COBOL library.

LILIAN DATE. The number of days since the beginning of the Gregorian calendar. Day one is Friday, October 15, 1582.

*** LINAGE-COUNTER**. A special register whose value points to the current position within the page body.

LINKAGE SECTION. The section in the DATA DIVISION of the called program that describes data items available from the calling program. These data items may be referred to by both the calling and called program.

literal. A character-string whose value is specified either by the ordered set of characters comprising the string, or by the use of a figurative constant.

little-endian. Default format used by the PC to store binary data. In this format, the most significant digit is on the highest address. Compare with “big-endian.”

locale. A set of attributes for a program execution environment indicating culturally sensitive considerations, such as: character code page, collating sequence, date/time format, monetary value representation, numeric value representation, or language.

*** LOCAL-STORAGE SECTION**. The section of the DATA DIVISION that defines storage that is allocated and freed on a per-invocation basis, depending on the value assigned in their VALUE clauses.

*** logical operator**. One of the reserved words AND, OR, or NOT. In the formation of a condition, either AND, or OR, or both can be used as logical connectives. NOT can be used for logical negation.

*** logical record**. The most inclusive data item. The level-number for a record is 01. A record may be either an elementary item or a group of items. The term is synonymous with record.

*** low order end**. The rightmost character of a string of characters.

M

main program. In a hierarchy of programs and subroutines, the first program to receive control when the programs are run.

*** mass storage**. A storage medium in which data may be organized and maintained in both a sequential and nonsequential manner.

*** mass storage device**. A device having a large storage capacity; for example, magnetic disk, magnetic drum.

*** mass storage file**. A collection of records that is assigned to a mass storage medium.

*** megabyte (M)**. One megabyte equals 1,048,576 bytes.

*** merge file**. A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

metaclass. A SOM class whose instances are SOM class-objects. The methods defined in metaclasses are executed without requiring any object instances of the class to exist, and are frequently used to create instances of the class.

method. Procedural code that defines one of the operations supported by an object, and that is executed by an INVOKE statement on that object.

*** Method Definition**. The COBOL source unit that defines a method.

*** method identification entry**. An entry in the METHOD-ID paragraph of the IDENTIFICATION DIVISION which contains clauses that specify the method-name and assign selected attributes to the method definition.

*** method-name**. A user-defined word that identifies a method.

*** mnemonic-name**. A user-defined word that is associated in the ENVIRONMENT DIVISION with a specified implementor-name.

| **MLE**. See “millennium language extensions.”

millennium language extensions. IBM extension to COBOL, enabling compiler-assisted date processing for dates containing 2-digit and 4-digit years. Language elements in support of the millennium language extensions are:

- DATE FORMAT clause in data description entries
- Intrinsic functions:

  DATEVAL
  UNDATE
  YEARWINDOW

multitasking. Mode of operation that provides for the concurrent, or interleaved, execution of two or more tasks. When running under the Language Environment product, multitasking is synonymous with *multithreading*.

N

name. A word composed of not more than 30 characters that defines a COBOL operand.

*** native character set**. The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

*** native collating sequence**. The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

*** negated combined condition**. The 'NOT' logical operator immediately followed by a parenthesized combined condition.

*** negated simple condition**. The 'NOT' logical operator immediately followed by a simple condition.

nested program. A program that is directly contained within another program.

*** next executable sentence**. The next sentence to which control will be transferred after execution of the current statement is complete.

*** next executable statement**. The next statement to which control will be transferred after execution of the current statement is complete.

*** next record**. The record that logically follows the current record of a file.

*** noncontiguous items**. Elementary data items in the WORKING-STORAGE and LINKAGE SECTIONs that bear no hierarchic relationship to other data items.

non-date. Any of the following:

- A data item whose data description entry does not include the DATE FORMAT clause
- A literal
- A reference-modified date field
- The result of certain arithmetic operations that may include date field operands; for example, the difference between two compatible dates.

The value of a non-date may or may not represent a date.

*** non-numeric item**. A data item whose description permits its content to be composed of any combination of characters taken from the computer's character set. Certain categories of non-numeric items may be formed from more restricted character sets.

*** non-numeric literal**. A literal bounded by quotation marks. The string of characters may include any character in the computer's character set.

null. Figurative constant used to assign the value of an invalid address to pointer data items. NULLS can be used wherever NULL can be used.

*** numeric character**. A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

numeric-edited item. A numeric item that is in such a form that it may be used in printed output. It may consist of external decimal digits from 0 through 9, the decimal point, commas, the dollar sign, editing sign control symbols, plus other editing symbols.

*** numeric function**. A function whose class and category are numeric but which for some possible evaluation does not satisfy the requirements of integer functions.

*** numeric item**. A data item whose description restricts its content to a value represented by characters chosen from the digits from '0' through '9'; if signed, the item may also contain a '+', '-', or other representation of an operational sign.

*** numeric literal**. A literal composed of one or more numeric characters that may contain either a decimal point, or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

O

object. An entity that has state (its data values) and operations (its methods). An object is a way to encapsulate state and behavior.

object code. Output from a compiler or assembler that is itself executable machine code or is suitable for processing to produce executable machine code.

*** OBJECT-COMPUTER**. The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, within which the object program is executed, is described.

*** object computer entry**. An entry in the OBJECT-COMPUTER paragraph of the ENVIRONMENT DIVISION which contains clauses that describe the computer environment in which the object program is to be executed.

object deck. A portion of an object program suitable as input to a linkage editor. Synonymous with *object module* and *text deck*.

object module. Synonym for *object deck* or *text deck*.

*** object of entry**. A set of operands and reserved words, within a DATA DIVISION entry of a COBOL program, that immediately follows the subject of the entry.

*** object program**. A set or group of executable machine language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program. Where there is no danger of ambiguity, the word ‘program’ alone may be used in place of the phrase ‘object program.’

*** object time**. The time at which an object program is executed. The term is synonymous with execution time.

*** obsolete element**. A COBOL language element in Standard COBOL that is to be deleted from the next revision of Standard COBOL.

ODBC. Open Database Connectivity that provides you access to data from a variety of databases and file systems.

ODO object. In the example below,

```
WORKING-STORAGE SECTION
01  TABLE-1.
    05  X                  PICS9.
    05  Y OCCURS 3 TIMES
           DEPENDING ON X    PIC X.
```

X is the object of the OCCURS DEPENDING ON clause (ODO object). The value of the ODO object determines how many of the ODO subject appear in the table.

ODO subject. In the example above, Y is the subject of the OCCURS DEPENDING ON clause (ODO subject). The number of Y ODO subjects that appear in the table depends on the value of X.

*** open mode**. The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O or EXTEND.

*** operand**. Whereas the general definition of operand is “that component which is operated upon,” for the purposes of this document, any lowercase word (or words) that appears in a statement or entry format may be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

*** operational sign**. An algebraic sign, associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

*** optional file**. A file which is declared as being not necessarily present each time the object program is executed. The object program causes an interrogation for the presence or absence of the file.

*** optional word**. A reserved word that is included in a specific format only to improve the readability of the language and whose presence is optional to the user when the format in which the word appears is used in a source program.

OS/2 (Operating System/2*). A multi-tasking operating system for the IBM Personal Computer family that allows you to run both DOS mode and OS/2 mode programs.

*** output file**. A file that is opened in either the OUTPUT mode or EXTEND mode.

*** output mode**. The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file.

*** output procedure**. A set of statements to which control is given during execution of a SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function reaches a point at which it can select the next record in merged order when requested.

overflow condition. A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

P

packed decimal item. See “internal decimal item.”

*** padding character**. An alphanumeric character used to fill the unused character positions in a physical record.

page. A vertical division of output data representing a physical separation of such data, the separation being based on internal logical requirements and/or external characteristics of the output medium.

*** page body**. That part of the logical page in which lines can be written and/or spaced.

*** paragraph**. In the Procedure Division, a paragraph-name followed by a separator period and by zero, one, or more sentences. In the IDENTIFICATION and ENVIRONMENT DIVISIONs, a paragraph header followed by zero, one, or more entries.

*** paragraph header**. A reserved word, followed by the separator period, that indicates the beginning of a paragraph in the IDENTIFICATION and ENVIRONMENT DIVISIONs. The permissible paragraph headers in the IDENTIFICATION DIVISION are:

```
    PROGRAM-ID. (Program IDENTIFICATION DIVISION)
    CLASS-ID. (Class IDENTIFICATION DIVISION)
    METHOD-ID. (Method IDENTIFICATION DIVISION)
    AUTHOR.
    INSTALLATION.
    DATE-WRITTEN.
    DATE-COMPILED.
    SECURITY.
```

The permissible paragraph headers in the ENVIRONMENT DIVISION are:

```
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
REPOSITORY. (Program or Class CONFIGURATION SECTION)
FILE-CONTROL.
I-O-CONTROL.
```

*** paragraph-name**. A user-defined word that identifies and begins a paragraph in the Procedure Division.

parameter. Parameters are used to pass data values between calling and called programs.

password. A unique string of characters that a program, computer operator, or user must supply to meet security requirements before gaining access to data.

*** phrase**. A phrase is an ordered set of one or more consecutive COBOL character-strings that form a portion of a COBOL procedural statement or of a COBOL clause.

*** physical record**. See “block.”

pointer data item. A data item in which address values can be stored. Data items are explicitly defined as pointers with the USAGE IS POINTER clause. ADDRESS OF special registers are implicitly defined as pointer data items. Pointer data items can be compared for equality or moved to other pointer data items.

portability. The ability to transfer an application program from one application platform to another with relatively few changes to the source program.

preloaded. In COBOL this refers to COBOL programs that remain resident in storage under IMS instead of being loaded each time they are called.

*** prime record key**. A key whose contents uniquely identify a record within an indexed file.

*** priority-number**. A user-defined word which classifies sections in the Procedure Division for purposes of segmentation. Segment-numbers may contain only the characters '0','1', ... , '9'. A segment-number may be expressed either as a one- or two-digit number.

*** procedure**. A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the Procedure Division.

*** procedure branching statement**. A statement that causes the explicit transfer of control to a statement other than the next executable statement in the sequence in which the statements are written in the source program. The procedure branching statements are: ALTER, CALL, EXIT, EXIT PROGRAM, GO TO, MERGE, (with the OUTPUT PROCEDURE phrase), PERFORM and SORT (with the INPUT PROCEDURE or OUTPUT PROCEDURE phrase).

Procedure Division. One of the four main component parts of a COBOL program, class definition, or method definition. The Procedure Division contains instructions for solving a problem. The Program and Method Procedure Divisions may contain imperative statements, conditional statements, compiler directing statements, paragraphs, procedures, and sections. The Class Procedure Division contains only method definitions.

procedure integration. One of the functions of the COBOL optimizer is to simplify calls to performed procedures or contained programs.

PERFORM procedure integration is the process whereby a PERFORM statement is replaced by its performed procedures. Contained program procedure integration is the process where a CALL to a contained program is replaced by the program code.

*** procedure-name**. A user-defined word that is used to name a paragraph or section in the Procedure Division. It consists of a paragraph-name (which may be qualified) or a section-name.

procedure-pointer data item. A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point.

*** program identification entry**. An entry in the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION which contains clauses that specify the program-name and assign selected program attributes to the program.

*** program-name**. In the IDENTIFICATION DIVISION and the end program header, a user-defined word that identifies a COBOL source program.

*** pseudo-text**. A sequence of text words, comment lines, or the separator space in a source program or COBOL library bounded by, but not including, pseudo-text delimiters.

*** pseudo-text delimiter**. Two contiguous equal sign characters (==) used to delimit pseudo-text.

*** punctuation character**. A character that belongs to the following set:

Character	Meaning
,	comma
;	semicolon
:	colon
.	period (full stop)
"	quotation mark
(	left parenthesis
)	right parenthesis
ƀ	space
=	equal sign

Q

QSAM (Queued Sequential Access Method). An extended version of the basic sequential access method (BSAM). When this method is used, a queue is formed of input data blocks that are awaiting processing or of output data blocks that have been processed and are awaiting transfer to auxiliary storage or to an output device.

*** qualified data-name**. An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

*** qualifier**.

1. A data-name or a name associated with a level indicator which is used in a reference either together with another data-name which is the name of an item that is subordinate to the qualifier or together with a condition-name.
2. A section-name that is used in a reference together with a paragraph-name specified in that section.
3. A library-name that is used in a reference together with a text-name associated with that library.

R

*** random access**. An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

*** record**. See “logical record.”

*** record area**. A storage area allocated for the purpose of processing the record described in a record description entry in the File Section of the DATA DIVISION. In the File Section, the current number of character positions in the record area is determined by the explicit or implicit RECORD clause.

*** record description**. See “record description entry.”

*** record description entry**. The total set of data description entries associated with a particular record. The term is synonymous with record description.

recording mode. The format of the logical records in a file. Recording mode can be F (fixed-length), V (variable-length), S (spanned), or U (undefined).

record key. A key whose contents identify a record within an indexed file.

*** record-name**. A user-defined word that names a record described in a record description entry in the DATA DIVISION of a COBOL program.

*** record number**. The ordinal number of a record in the file whose organization is sequential.

recursion. A program calling itself or being directly or indirectly called by a one of its called programs.

recursively capable. A program is recursively capable (can be called recursively) if the RECURSIVE attribute is on the PROGRAM-ID statement.

reel. A discrete portion of a storage medium, the dimensions of which are determined by each implementor that contains part of a file, all of a file, or any number of files. The term is synonymous with unit and volume.

reentrant. The attribute of a program or routine that allows more than one user to share a single copy of a load module.

*** reference format**. A format that provides a standard method for describing COBOL source programs.

reference modification. A method of defining a new alphanumeric data item by specifying the leftmost character and length relative to the leftmost character of another alphanumeric data item.

*** reference-modifier**. A syntactically correct combination of character-strings and separators that defines a unique data item. It includes a delimiting left parenthesis separator, the leftmost character position, a colon separator, optionally a length, and a delimiting right parenthesis separator.

*** relation**. See “relational operator” or “relation condition.”

*** relational operator**. A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meanings are:

Operator	**Meaning**
IS GREATER THAN	Greater than
IS >	Greater than
IS NOT GREATER THAN	Not greater than
IS NOT >	Not greater than
IS LESS THAN	Less than
IS <	Less than
IS NOT LESS THAN	Not less than
IS NOT <	Not less than
IS EQUAL TO	Equal to
IS =	Equal to
IS NOT EQUAL TO	Not equal to
IS NOT =	Not equal to
IS GREATER THAN OR EQUAL TO	Greater than or equal to
IS >=	Greater than or equal to
IS LESS THAN OR EQUAL TO	Less than or equal to
IS <=	Less than or equal to

*** relation character**. A character that belongs to the following set:

Character	**Meaning**
>	greater than
<	less than
=	equal to

*** relation condition**. The proposition, for which a truth value can be determined, that the value of an arithmetic expression, data item, non-numeric literal, or index-name has a specific relationship to the value of another arithmetic expression, data item, non-numeric literal, or index name. (See also “relational operator.”)

*** relative file**. A file with relative organization.

*** relative key**. A key whose contents identify a logical record in a relative file.

*** relative organization**. The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the record’s logical ordinal position in the file.

*** relative record number**. The ordinal number of a record in a file whose organization is relative. This number is treated as a numeric literal which is an integer.

*** reserved word**. A COBOL word specified in the list of words that may be used in a COBOL source program, but that must not appear in the program as user-defined words or system-names.

*** resource**. A facility or service, controlled by the operating system, that can be used by an executing program.

*** resultant identifier**. A user-defined data item that is to contain the result of an arithmetic operation.

reusable environment. A reusable environment is when you establish an assembler program as the main program by using either ILBOSTP0 programs, IGZERRE programs, or the RTEREUS run-time option.

routine. A set of statements in a COBOL program that causes the computer to perform an operation or series of related operations. In Language Environment, refers to either a procedure, function, or subroutine.

*** routine-name**. A user-defined word that identifies a procedure written in a language other than COBOL.

*** run time**. The time at which an object program is executed. The term is synonymous with object time.

run-time environment. The environment in which a COBOL program executes.

*** run unit**. A stand-alone object program, or several object programs, that interact via COBOL CALL statements, which function at run time as an entity.

S

SBCS (Single Byte Character Set). See “Single Byte Character Set (SBCS).”

scope terminator. A COBOL reserved word that marks the end of certain Procedure Division statements. It may be either explicit (END-ADD, for example) or implicit (separator period).

*** section**. A set of zero, one or more paragraphs or entities, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

*** section header**. A combination of words followed by a separator period that indicates the beginning of a section in the Environment, Data, and Procedure Divisions. In the ENVIRONMENT and DATA DIVISIONs, a section header is composed of reserved words followed by a separator period. The permissible section headers in the ENVIRONMENT DIVISION are:

```
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
```

The permissible section headers in the DATA DIVISION are:

```
FILE SECTION.
WORKING-STORAGE SECTION.
LOCAL-STORAGE SECTION.
LINKAGE SECTION.
```

In the Procedure Division, a section header is composed of a section-name, followed by the reserved word SECTION, followed by a separator period.

*** section-name**. A user-defined word that names a section in the Procedure Division.

selection structure. A program processing logic in which one or another series of statements is executed, depending on whether a condition is true or false.

*** sentence**. A sequence of one or more statements, the last of which is terminated by a separator period.

*** separately compiled program**. A program which, together with its contained programs, is compiled separately from all other programs.

*** separator**. A character or two contiguous characters used to delimit character-strings.

*** separator comma**. A comma (,) followed by a space used to delimit character-strings.

*** separator period**. A period (.) followed by a space used to delimit character-strings.

*** separator semicolon**. A semicolon (;) followed by a space used to delimit character-strings.

sequence structure. A program processing logic in which a series of statements is executed in sequential order.

*** sequential access**. An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

*** sequential file**. A file with sequential organization.

*** sequential organization**. The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

serial search. A search in which the members of a set are consecutively examined, beginning with the first member and ending with the last.

*** 77-level-description-entry**. A data description entry that describes a noncontiguous data item with the level-number 77.

*** sign condition**. The proposition, for which a truth value can be determined, that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

*** simple condition**. Any single condition chosen from the set:

Relation condition
Class condition
Condition-name condition
Switch-status condition
Sign condition

Single Byte Character Set (SBCS). A set of characters in which each character is represented by a single byte. See also "EBCDIC (Extended Binary-Coded Decimal Interchange Code)."

slack bytes. Bytes inserted between data items or records to ensure correct alignment of some numeric items. Slack bytes contain no meaningful data. In some cases, they are inserted by the compiler; in others, it is the responsibility of the programmer to insert them. The SYNCHRONIZED clause instructs the compiler to insert slack bytes when they are needed for proper alignment. Slack bytes between records are inserted by the programmer.

SOM. See “System Object Model”

*** sort file**. A collection of records to be sorted by a SORT statement. The sort file is created and can be used by the sort function only.

*** sort-merge file description entry**. An entry in the File Section of the DATA DIVISION that is composed of the level indicator SD, followed by a file-name, and then followed by a set of file clauses as required.

*** SOURCE-COMPUTER**. The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, within which the source program is compiled, is described.

*** source computer entry**. An entry in the SOURCE-COMPUTER paragraph of the ENVIRONMENT DIVISION which contains clauses that describe the computer environment in which the source program is to be compiled.

*** source item**. An identifier designated by a SOURCE clause that provides the value of a printable item.

source program. Although it is recognized that a source program may be represented by other forms and symbols, in this document it always refers to a syntactically correct set of COBOL statements. A COBOL source program commences with the IDENTIFICATION DIVISION or a COPY statement. A COBOL source program is terminated by the end program header, if specified, or by the absence of additional source program lines.

*** special character**. A character that belongs to the following set:

Character	**Meaning**
+	plus sign
-	minus sign (hyphen)
*	asterisk
/	slant (virgule, slash)
=	equal sign
$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point, full stop)
"	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol
:	colon

*** special-character word**. A reserved word that is an arithmetic operator or a relation character.

SPECIAL-NAMES. The name of an ENVIRONMENT DIVISION paragraph in which environment-names are related to user-specified mnemonic-names.

*** special names entry**. An entry in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION which provides means for specifying the currency sign; choosing the decimal point; specifying symbolic characters; relating implementor-names to user-specified mnemonic-names; relating alphabet-names to character sets or collating sequences; and relating class-names to sets of characters.

*** special registers**. Certain compiler generated storage areas whose primary use is to store information produced in conjunction with the use of a specific COBOL feature.

*** standard data format**. The concept used in describing the characteristics of data in a COBOL DATA DIVISION under which the characteristics or properties of the data are expressed in a form oriented to the appearance of the data on a printed page of infinite length and breadth, rather than a form oriented to the manner in which the data is stored internally in the computer, or on a particular external medium.

*** statement**. A syntactically valid combination of words, literals, and separators, beginning with a verb, written in a COBOL source program.

STL. Standard Language file system: native workstation and PC file system for COBOL and PL/I. Supports sequential, relative, and indexed files, including the full ANSI 85 COBOL standard I/O language and all of the extensions described in *IBM COBOL Language Reference*, unless exceptions are explicitly noted.

structured programming. A technique for organizing and coding a computer program in which the program comprises a hierarchy of segments, each segment having a single entry point and a single exit point. Control is passed downward through the structure without unconditional branches to higher levels of the hierarchy.

*** sub-class**. A class that inherits from another class. When two classes in an inheritance relationship are considered together, the sub-class is the inheritor or inheriting class; the *super-class* is the inheritee or inherited class.

*** subject of entry**. An operand or reserved word that appears immediately following the level indicator or the level-number in a DATA DIVISION entry.

*** subprogram**. See “called program.”

*** subscript**. An occurrence number represented by either an integer, a data-name optionally followed by an integer with the operator + or -, or an index-name optionally followed by an integer with the operator + or -, that identifies a particular element in a table. A subscript may be the word ALL when the subscripted identifier is used as a function argument for a function allowing a variable number of arguments.

*** subscripted data-name**. An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.

*** super-class**. A class that is inherited by another class. See also *sub-class*.

switch-status condition. The proposition, for which a truth value can be determined, that an UPSI switch, capable of being set to an ‘on’ or ‘off’ status, has been set to a specific status.

*** symbolic-character**. A user-defined word that specifies a user-defined figurative constant.

syntax. (1) The relationship among characters or groups of characters, independent of their meanings or the manner of their interpretation and use. (2) The structure of expressions in a language. (3) The rules governing the structure of a language. (4) The relationship among symbols. (5) The rules for the construction of a statement.

*** system-name**. A COBOL word that is used to communicate with the operating environment.

System Object Model (SOM). IBM's object-oriented programming technology for building, packaging, and manipulating class libraries. SOM conforms to the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) standards.

T

*** table**. A set of logically consecutive items of data that are defined in the DATA DIVISION by means of the OCCURS clause.

*** table element**. A data item that belongs to the set of repeated items comprising a table.

text deck. Synonym for *object deck* or *object module*.

*** text-name**. A user-defined word that identifies library text.

*** text word**. A character or a sequence of contiguous characters between margin A and margin R in a COBOL library, source program, or in pseudo-text which is:

- A separator, except for: space; a pseudo-text delimiter; and the opening and closing delimiters for nonnumeric literals. The right parenthesis and left parenthesis characters, regardless of context within the library, source program, or pseudo-text, are always considered text words.

- A literal including, in the case of non-numeric literals, the opening quotation mark and the closing quotation mark that bound the literal.
- Any other sequence of contiguous COBOL characters except comment lines and the word 'COPY' bounded by separators that are neither a separator nor a literal.

top-down design. The design of a computer program using a hierarchic structure in which related functions are performed at each level of the structure.

top-down development. See "structured programming."

trailer-label. (1) A file or data set label that follows the data records on a unit of recording medium. (2) Synonym for end-of-file label.

*** truth value**. The representation of the result of the evaluation of a condition in terms of one of two values: true or false.

U

*** unary operator**. A plus (+) or a minus (-) sign, that precedes a variable or a left parenthesis in an arithmetic expression and that has the effect of multiplying the expression by +1 or -1, respectively.

unit. A module of direct access, the dimensions of which are determined by IBM.

universal object reference. A data-name that can refer to an object of any class.

*** unsuccessful execution**. The attempted execution of a statement that does not result in the execution of all the operations specified by that statement. The unsuccessful execution of a statement does not affect any data referenced by that statement, but may affect status indicators.

UPSI switch. A program switch that performs the functions of a hardware switch. Eight are provided: UPSI-0 through UPSI-7.

*** user-defined word**. A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

V

*** variable**. A data item whose value may be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

*** variable length record**. A record associated with a file whose file description or sort-merge description entry permits records to contain a varying number of character positions.

*** variable occurrence data item**. A variable occurrence data item is a table element which is repeated a variable number of times. Such an item must contain an OCCURS DEPENDING ON clause in its data description entry, or be subordinate to such an item.

*** variably located group.**. A group item following, and not subordinate to, a variable-length table in the same level-01 record.

*** variably located item.**. A data item following, and not subordinate to, a variable-length table in the same level-01 record.

*** verb**. A word that expresses an action to be taken by a COBOL compiler or object program.

VM/SP (Virtual Machine/System Product). An IBM-licensed program that manages the resources of a single computer so that multiple computing systems appear to exist. Each virtual machine is the functional equivalent of a "real" machine.

volume. A module of external storage. For tape devices it is a reel; for direct-access devices it is a unit.

volume switch procedures. System specific procedures executed automatically when the end of a unit or reel has been reached before end-of-file has been reached.

VSAM/2. A file system that supports COBOL sequential, relative, and indexed organizations. This file system is available as part of IBM VisualAge COBOL.

W

windowed date field. A date field containing a windowed (2-digit) year. See also “date field” and “windowed year.”

windowed year. Two digits representing a year within a century window (for example, 98). Appears in windowed date fields. See also “century window.”

Compare with “expanded year.”

*** word**. A character-string of not more than 30 characters which forms a user-defined word, a system-name, a reserved word, or a function-name.

*** WORKING-STORAGE SECTION**. The section of the DATA DIVISION that describes working storage data items, composed either of noncontiguous items or working storage records or of both.

Z

zoned decimal item. See “external decimal item.”

Index

We'd Like to Hear from You

VisualAge COBOL
Programming Guide
Version 2.2

Publication No. SC26-9050-02

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773.
- Electronic mail—Use one of the following network IDs:
  - IBMMail: USIB2VVG at IBMMAIL
  - IBMLink: COBPUBS at STLVM27
  - Internet: COMMENTS@VNET.IBM.COM

  Be sure to include the following with your comments:

  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

VisualAge COBOL
Programming Guide
Version 2.2

Publication No. SC26-9050-02

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□
Grammatically correct and consistent	□	□	□	□	□
Graphically well designed	□	□	□	□	□
Overall satisfaction	□	□	□	□	□

Please tell us how we can improve this book:

May we contact you to discuss your comments? □ Yes □ No

Name

Address

Company or Organization

Phone No.

Readers' Comments
SC26-9050-02

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

Department J58
International Business Machines Corporation
PO BOX 49023
SAN JOSE CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

IBM®

Program Number: 5639-B92

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

VisualAge COBOL

GC26-9038	Building Parts for Fun and Profit
GC26-8488	Resource Catalog
GC26-9052	Fact Sheet
GC26-9051	Getting Started on OS/2
GC26-8944	Getting Started on Windows
SC26-9056	Introducing Redeveloper
SC26-9046	Language Reference
SC26-9050	Programming Guide
GC26-9060	Technology Brochure
SC26-9053	Visual Builder User's Guide

SC26-9050-02

Spine information:

IBM VisualAge COBOL Programming Guide *Version 2.2*