Enterprise COBOL for z/OS

Programming Guide

Version 3 Release 4

SC27-1412-05

Enterprise COBOL for z/OS

Programming Guide

Version 3 Release 4

SC27-1412-05

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 787.

Sixth Edition (November 2006)

This edition applies to Version 3 Release 4 of IBM Enterprise COBOL for z/OS (program number 5655-G53) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure that you are using the correct edition for the level of the product.

You can order publications online at www.ibm.com/shop/publications/order/, or order by phone or fax. IBM Software Manufacturing Solutions takes publication orders between 8:30 a.m. and 7:00 p.m. Eastern Standard Time (EST). The phone number is (800)879-2755. The fax number is (800)445-9269.

You can also order publications through your IBM representative or the IBM branch office serving your locality.

Contents

Tables

Preface

About this document

Welcome to IBM(R) Enterprise COBOL for z/OS(R), IBM's latest host COBOL compiler!

This version of IBM COBOL adds new COBOL function to help integrate COBOL business processes and Web-oriented business processes by:

- Simplifying the componentization of COBOL programs and enabling interoperability with Java(TM) components
- Promoting the exchange and use of data in standardized formats, including XML and Unicode

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The accessibility features in z/OS provide accessibility for Enterprise COBOL.

The major accessibility features in z/OS enable users to:

- Use assistive technology products such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using only the keyboard.
- Customize display attributes such as color, contrast, and font size.

Using assistive technologies

Assistive technology products work with the user interfaces that are found in z/OS. For specific guidance information, consult the documentation for the assistive technology product that you use to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces by using TSO/E or ISPF. For information about accessing TSO/E or ISPF interfaces, refer to the following publications:

- z/OS TSO/E Primer
- z/OS TSO/E User's Guide
- z/OS ISPF User's Guide Volume I

These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Accessibility of this document

The English-language XHTML format of this document that will be provided at the IBM Problem Determination Tools information center is accessible to visually impaired individuals who use a screen reader.

To enable your screen reader to accurately read syntax diagrams, source code examples, and text that contains the period or comma `PICTURE` symbols, you must set the screen reader to speak all punctuation.

When you use JAWS for Windows(R), the links to the accessible syntax diagrams might not work. Use IBM Home Page Reader to read the accessible syntax diagrams.

How this document will help you

This document will help you write and compile Enterprise COBOL programs. It will also help you define object-oriented classes and methods, invoke methods, and refer to objects in your programs.

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

For information on migrating programs to Enterprise COBOL, see *IBM Enterprise COBOL Compiler and Runtime Migration Guide*.

IBM z/OS Language Environment(R) provides the runtime environment and runtime services that are required to run your Enterprise COBOL programs. You will find information on link-editing and running programs in the *IBM z/OS Language Environment Programming Guide* and *IBM z/OS Language Environment Programming Reference*.

For a comparison of commonly used Enterprise COBOL and IBM z/OS Language Environment terms, see “Comparison of commonly used terms” on page xvii.

Abbreviated terms

Certain terms are used in a shortened form in this document. Abbreviations for the product names used most frequently are listed alphabetically in the following table.

Term used	**Long form**
CICS(R)	CICS Transaction Server
Enterprise COBOL	IBM Enterprise COBOL for z/OS
Language Environment	IBM z/OS Language Environment
MVS(TM)	MVS/ESA(TM)
UNIX(R)	z/OS UNIX System Services

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

- ISO 1989:1985, Programming languages - COBOL
- ISO/IEC 1989/AMD1:1992, Programming languages - COBOL - Intrinsic function module
- ISO/IEC 1989/AMD2:1994, Programming languages - COBOL - Correction and clarification amendment for COBOL
- ANSI INCITS 23-1985, Programming Languages - COBOL
- ANSI INCITS 23a-1989, Programming Languages - Intrinsic Function Module for COBOL
- ANSI INCITS 23b-1993, Programming Language - Correction Amendment for COBOL

The ISO standards are identical to the American National standards.

Other terms, if not commonly understood, are shown in *italics* the first time that they appear, and are listed in the glossary at the back of this document.

Comparison of commonly used terms

To better understand the terms used throughout the IBM z/OS Language Environment and IBM Enterprise COBOL for z/OS publications and what terms are meant to be equivalent, see the following table.

Language Environment term	Enterprise COBOL equivalent
Aggregate	Group item
Array	A table created using the `OCCURS` clause
Array element	Table element
Enclave	Run unit
External data	`WORKING-STORAGE` data defined with `EXTERNAL` clause
Local data	Any non-`EXTERNAL` data item
Pass parameters directly, by value	`BY VALUE`
Pass parameters indirectly, by reference	`BY REFERENCE`
Pass parameters indirectly, by value	`BY CONTENT`
Routine	Program
Scalar	Elementary item

How to read syntax diagrams

Use the following description to read the syntax diagrams in this information.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The **>>—-** symbol indicates the beginning of a syntax diagram.

  The **—->** symbol indicates that the syntax diagram is continued on the next line.

  The **>—-** symbol indicates that the syntax diagram is continued from the previous line.

  The **—-><** symbol indicates the end of a syntax diagram.

  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):

```
►►──required_item──────────────────────────────────────────────────────►◄
```

- Optional items appear below the main path:

```
►►──required_item──┬───────────────┬───────────────────────────────────►◄
                   └─optional_item─┘
```

- If 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:

```
►►──required_item──┬─required_choice1─┬──────────────────────────────────►◄
                   └─required_choice2─┘
```

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

```
►►──required_item──┬──────────────────┬──────────────────────────────────►◄
                   ├─optional_choice1─┤
                   └─optional_choice2─┘
```

If one of the items is the default, it appears above the main path and the remaining choices are shown below:

```
                   ┌─default_choice──┐
►►──required_item──┼─────────────────┼───────────────────────────────────►◄
                   ├─optional_choice─┤
                   └─optional_choice─┘
```

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

```
                   ┌─────────────────┐
►►──required_item──▼─repeatable_item─┴───────────────────────────────────►◄
```

If the repeat arrow contains a comma, you must separate repeated items with a comma:

```
                   ┌─,───────────────┐
►►──required_item──▼─repeatable_item─┴───────────────────────────────────►◄
```

- Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in lowercase italics (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

How examples are shown

This document shows numerous examples of sample COBOL statements, program fragments, and small programs to illustrate the coding techniques 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 ways.

To more clearly separate some examples from the explanatory text, they are presented in a `monospace font`.

COBOL keywords and compiler options that appear in text are generally shown in SMALL UPPERCASE. Other terms such as program variable names are sometimes shown in *an italic font* for clarity.

Accessing softcopy documentation and support information

IBM Enterprise COBOL for z/OS provides PDF and BookManager versions of the library on the product site at www.ibm.com/software/awdtools/cobol/zos/library/.

You can check that Web site for the latest editions of the documents. In the BookManager(R) version of a document, the content of some tables and syntax diagrams might be aligned improperly due to variations in the display technology.

Support information is also available on the product site at www.ibm.com/software/awdtools/cobol/zos/support/.

Summary of changes

This section lists the key changes that have been made to Enterprise COBOL. The changes that are described in this document have an associated cross-reference for your convenience.

Technical changes after the Version 3 Release 4 release of July, 2005, are marked by a plus sign (+) in the left margin in the PDF and BookManager versions. Technical changes and editorial changes to terminology that occurred in Version 3 Release 4 are marked by a vertical bar (|) in the left margin in the PDF and BookManager versions.

Version 3 Release 4 update (November 2006)

- A new compiler option, `SQLCCSID`, which works in conjunction with the DB2 coprocessor, determines whether the `CODEPAGE` compiler option influences the processing of SQL statements in your COBOL programs ("SQLCCSID" on page 337).

  For full details about the determination of the code page used for string host variables in SQL statements, see "COBOL and DB2 CCSID determination" on page 411.

Version 3 Release 4 (July 2005)

- Several limits on COBOL data-item size have been significantly raised, for example:
  - The maximum data-item size has been raised from 16 MB to 128 MB.
  - The maximum `PICTURE` symbol replication has been raised to 134,217,727.
  - The maximum `OCCURS` integer has been raised to 134,217,727.

  This support facilitates programming with large amounts of data, for example:
  - DB2(R)/COBOL applications that use DB2 BLOB and CLOB data types
  - COBOL XML applications that parse or generate large XML documents

  For full details about changed compiler limits, see Compiler limits (*Enterprise COBOL Language Reference*).
- Support for national (Unicode UTF-16) data has been enhanced. Several additional kinds of data items can now be described implicitly or explicitly as `USAGE NATIONAL`:
  - External decimal (*national decimal*) items ("Defining numeric data" on page 45)
  - External floating-point (*national floating-point*) items ("Formats for numeric data" on page 49)
  - Numeric-edited items ("Displaying numeric data" on page 47)

  - National-edited items ("Using national data (Unicode) in COBOL" on page 126)
  - Group (*national group*) items, supported by the GROUP-USAGE NATIONAL clause ("Using national groups" on page 130)
- Many COBOL language elements support the new kinds of UTF-16 data, or newly support the processing of national data:
  - Numeric data with USAGE NATIONAL (national decimal and national floating point) can be used in arithmetic operations and in any language constructs that support numeric operands ("Formats for numeric data" on page 49).
  - Edited data with USAGE NATIONAL is supported in the same language constructs as any existing edited type, including editing and de-editing operations associated with moves ("Displaying numeric data" on page 47 and "Using national data (Unicode) in COBOL" on page 126).
  - Group items that contain all national data can be defined with the GROUP-USAGE NATIONAL clause, which results in the group behaving as an elementary item in most language constructs. This support facilitates use of national groups in statements such as STRING, UNSTRING, and INSPECT ("National groups" on page 129).
  - The XML GENERATE statement supports national groups as receiving data items, and national-edited, numeric-edited of USAGE NATIONAL, national decimal, national floating-point, and national group items as sending data items ("Generating XML output" on page 511).
  - The NUMVAL and NUMVAL-C intrinsic functions can take a national literal or national data item as an argument ("Converting to numbers (NUMVAL, NUMVAL-C)" on page 113).

  Using these new national data capabilities, it is now practical to develop COBOL programs that exclusively use Unicode for all application data.
- The REDEFINES clause has been enhanced such that for data items that are not level 01, the subject of the entry can be larger than the data item being redefined.
- A new compiler option, MDECK, causes the output from library-processing statements to be written to a file ("MDECK" on page 321).
- DB2 coprocessor support has been enhanced: COBOL zoned decimal (USAGE DISPLAY SIGN LEADING SEPARATE) data items and numeric Unicode (USAGE NATIONAL SIGN LEADING SEPARATE) data items can be used as DB2 host variables. The characters @, #, and $ can be used in EXEC SQL INCLUDE file-names. XREF is improved.
- The literal in a VALUE clause for a data item of class national can be alphanumeric ("Initializing a table at the group level" on page 78).
- The IBM SDK for z/OS, Java 2 Technology Edition, V1.4 is now supported ("Using IBM SDK for z/OS, Java 2 Technology Edition, V1.4" on page 295).

These terminology changes were also made in this release:
- The term *alphanumeric group* is introduced to refer specifically to groups other than national groups.
- The term *group* means both alphanumeric groups and national groups except when used in a context that obviously refers to only an alphanumeric group or only a national group.
- The term *external decimal* refers to both zoned decimal items and national decimal items.
- The term *display floating point* is introduced to refer to an external floating-point item that has USAGE DISPLAY.

- The term *external floating point* refers to both display floating-point items and national floating-point items.

Version 3 Release 3 (February 2004)

- XML support has been enhanced. A new statement, `XML GENERATE`, converts the content of COBOL data records to XML format. `XML GENERATE` creates XML documents encoded in Unicode UTF-16 or in one of several single-byte EBCDIC code pages (Chapter 29, "Producing XML output," on page 511).
- The `XML PARSE` statement supports XML files encoded in Unicode UTF-16 or any of several single-byte EBCDIC code pages. Support is not provided for ASCII code pages.
- There are new and improved features of Debug Tool:
  - Performance is improved when you use COBOL SYSDEBUG files.
  - You can more easily debug programs that use national data: When you display national data in a formatted dump or by using the Debug Tool `LIST` command, the data is automatically converted to EBCDIC representation using the code page specified by the `CODEPAGE` compiler option. You can use the Debug Tool `MOVE` command to assign values to national data items, and you can move national data items to or from group data items. You can use national data as a comparand in Debug Tool conditional commands such as `IF` and `EVALUATE`.
  - You can debug mixed COBOL-Java applications, COBOL class definitions, and COBOL programs that contain object-oriented syntax.

  For more details about these enhancements to debugging support, see *Debug Tool User's Guide*.
- DB2 Version 8 SQL features are supported when you use the integrated DB2 coprocessor.
- The syntax for specifying options in the COBJVMINITOPTIONS environment variable has changed ("Running OO applications that start with a COBOL program" on page 291).

Version 3 Release 2 (September 2002)

- The compiler has been enhanced to support new features of Debug Tool, and features of Debug Tool Utilities and Advanced Functions:
  - Playback support lets you record and replay application execution paths and data values.
  - Automonitor support displays the values of variables that are referenced in the current statement during debugging.
  - Programs that have been compiled with the `OPTIMIZE` and `TEST(NONE,SYM,. . .)` options are supported for debugging ("TEST" on page 338).
  - The Debug Tool `GOTO` command is enabled for programs that have been compiled with the `NOOPTIMIZE` option and the `TEST` option with any of its suboptions ("TEST" on page 338). (In earlier releases, the `GOTO` command was not supported for programs compiled with `TEST(NONE,. . .)`.)

  For more details about these enhancements to debugging support, see *Debug Tool User's Guide*.
- Extending Java interoperability to IMS(TM): Object-oriented COBOL programs can run in an IMS Java dependent region. The object-oriented COBOL and Java languages can be mixed in a single application ("Using object-oriented COBOL and Java under IMS" on page 418).

- Enhanced support for Java interoperability:
  - The OPTIMIZE compiler option is fully supported for programs that contain OO syntax for Java interoperability.
  - Object references of type jobjectArray are supported for interoperation between COBOL and Java ("Declaring arrays and strings for Java" on page 575).
  - OO applications that begin with a COBOL main factory method can be invoked with the java command ("Structuring OO applications" on page 566).
  - A new environment variable, COBJVMINITOPTIONS, is provided, enabling the user to specify options that will be used when COBOL initializes a Java virtual machine (JVM) ("Running OO applications under UNIX" on page 289).
  - OO applications that begin with a COBOL program can, with some limitations, be bound as modules in a PDSE and run using batch JCL ("Preparing and running OO applications using JCL or TSO/E" on page 292).
- Unicode enhancement for working with DB2: The code pages for host variables are handled implicitly when you use the DB2 integrated coprocessor. SQL DECLARE statements are necessary only for variables described with USAGE DISPLAY or USAGE DISPLAY-1 when COBOL and DB2 code pages do not match ("Coding SQL statements" on page 406).

Version 3 Release 1 (November 2001)

- Interoperation of COBOL and Java by means of object-oriented syntax, permitting COBOL programs to instantiate Java classes, invoke methods on Java objects, and define Java classes that can be instantiated in Java or COBOL and whose methods can be invoked in Java or COBOL (Chapter 30, "Writing object-oriented programs," on page 525)
- Ability to call services provided by the Java Native Interface (JNI) to obtain additional Java capabilities, with a copybook JNI.cpy and special register JNIEnvPtr to facilitate access ("Accessing JNI services" on page 569)
- XML support, including a high-speed XML parser that allows programs to consume inbound XML messages, verify that they are well formed, and transform their contents into COBOL data structures; with support for XML documents encoded in Unicode UTF-16 or several single-byte EBCDIC or ASCII code pages (Chapter 28, "Processing XML input," on page 487)
- Support for compilation of programs that contain CICS statements, without the need for a separate translation step ("Integrated CICS translator" on page 399)
  - Compiler option CICS, enabling integrated CICS translation and specification of CICS options ("CICS" on page 303)
- Support for Unicode provided by NATIONAL data type and national (N, NX) literals, intrinsic functions DISPLAY-OF and NATIONAL-OF for character conversions, and compiler options NSYMBOL and CODEPAGE (Chapter 7, "Processing data in an international environment," on page 121)
  - Compiler option CODEPAGE to specify the code page used for encoding national literals, and alphanumeric and DBCS data items and literals ("CODEPAGE" on page 305)
  - Compiler option NSYMBOL to control whether national or DBCS processing should be in effect for literals and data items that use the N symbol ("NSYMBOL" on page 323)

- Multithreading support: support of POSIX threads and asynchronous signals, permitting applications with COBOL programs to run on multiple threads within a process (Chapter 27, "Preparing COBOL programs for multithreading," on page 477)
  - Compiler option `THREAD`, enabling programs to run in Language Environment enclaves with multiple POSIX threads or PL/I subtasks ("THREAD" on page 342)
- `VALUE` clauses for `BINARY` data items that permit numeric literals to have a value of magnitude up to the capacity of the native binary representation, rather than being limited to the value implied by the number of `9`s in the `PICTURE` clause ("Formats for numeric data" on page 49)
- A 4-byte `FUNCTION-POINTER` data item that can contain the address of a COBOL or non-COBOL entry point, providing easier interoperability with C function pointers ("Using procedure and function pointers" on page 447)
- The following support is no longer provided (as documented in *Enterprise COBOL Compiler and Runtime Migration Guide*):
  - SOM-based object-oriented syntax and services
  - Compiler options `CMPR2`, `ANALYZE`, `FLAGMIG`, `TYPECHK`, and `IDLGEN`
- Changed default values for the following compiler options: `DBCS` ("DBCS" on page 309), `FLAG(I,I)` ("FLAG" on page 314), `RENT` ("RENT" on page 331), and `XREF(FULL)` ("XREF" on page 347)

For a history of changes to previous COBOL compilers, see *Enterprise COBOL Compiler and Runtime Migration Guide*.

How to send your comments

Your feedback is important in helping us to provide accurate, high-quality information. If you have comments about this document or any other Enterprise COBOL documentation, contact us in one of these ways:

- Fill out the Readers' Comments Form at the back of this document, and return it by mail or give it to an IBM representative. If there is no form at the back of this document, address your comments to:

  IBM Corporation
  Reader Comments
  DTX/E269
  555 Bailey Avenue
  San Jose, CA 95141-1003
  USA
- Use the Online Readers' Comments Form at www.ibm.com/software/awdtools/rcf/.
- Send your comments to the following e-mail address: comments@us.ibm.com.

Be sure to include the name of the document, the publication number of the document, the version of Enterprise COBOL, and, if applicable, the specific location (for example, the page number or section heading) of the text that you are commenting on.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way that IBM believes appropriate without incurring any obligation to you.

Part 1. Coding your program

Chapter 1. Structuring your program

COBOL programs consist of four divisions: IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION. Each division has a specific logical function.

To define a program, only the IDENTIFICATION DIVISION is required.

To define a COBOL class or method, you need to define some divisions differently than you do for a program.

RELATED TASKS
"Identifying a program"
"Describing the computing environment" on page 7
"Describing the data" on page 12
"Processing the data" on page 19
"Defining a class" on page 528
"Defining a class instance method" on page 533
"Structuring OO applications" on page 566

Identifying a program

Use the IDENTIFICATION DIVISION to name a program and optionally provide other identifying information.

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

```
IDENTIFICATION DIVISION.
Program-ID.    Helloprog.
Author.        A. Programmer.
Installation.  Computing Laboratories.
Date-Written.  06/21/2005.
Date-Compiled. 06/30/2005.
```

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

- Other programs use that name to call your program.
- The name appears in the header on each page, except the first, of the program listing that is generated when you compile the program.
- If you use the NAME compiler option, the name is placed on the NAME linkage-editor or binder control statement to identify the object module that the compilation creates.

  Tip: Do not use program-names that start with prefixes used by IBM products. If you use program-names that start with any of the following prefixes, your CALL statements might resolve to IBM library or compiler routines rather than to your intended program:

  - AFB
  - AFH
  - CBC
  - CEE

- IBM
- IFY
- IGY
- IGZ
- ILB

Tip: When a program-name is case sensitive, avoid mismatches with the name the compiler is looking for. Verify that the appropriate setting of the PGMNAME compiler option is in effect.

RELATED TASKS
"Changing the header of a source listing" on page 7
"Identifying a program as recursive"
"Marking a program as callable by containing programs"
"Setting a program to an initial state"

RELATED REFERENCES
Compiler limits (*Enterprise COBOL Language Reference*)
Conventions for program-names (*Enterprise COBOL Language Reference*)

Identifying a program as recursive

Code the RECURSIVE attribute on the PROGRAM-ID clause to specify that a program can be recursively reentered while a previous invocation is still active.

You can code RECURSIVE only on the outermost program of a compilation unit. Neither nested subprograms nor programs that contain nested subprograms can be recursive. You must code RECURSIVE for programs that you compile with the THREAD option.

RELATED TASKS
"Sharing data in recursive or multithreaded programs" on page 18
"Making recursive calls" on page 447

Marking a program as callable by containing programs

Use the COMMON attribute in the PROGRAM-ID paragraph to specify that a program can be called by the containing program or by any program in the containing program. The COMMON program cannot be called by any program contained in itself.

Only contained programs can have the COMMON attribute.

RELATED CONCEPTS
"Nested programs" on page 444

Setting a program to an initial state

Use the INITIAL attribute to specify that whenever a program is called, that program and any nested programs that it contains are to be placed in their initial state.

When a program is in its initial state:

- Data items that have VALUE clauses are set to the specified values.
- Changed GO TO statements and PERFORM statements are in their initial states.
- Non-EXTERNAL files are closed.

RELATED TASKS
"Ending and reentering main programs or subprograms" on page 434
"Making static calls" on page 436
"Making dynamic calls" on page 437

Changing the header of a source listing

The header on the first page of a source listing contains the identification of the compiler and the current release level, the date and time of compilation, and the page number.

The following example shows these five elements:

```
PP 5655-G53 IBM Enterprise COBOL for z/OS  3.4.0   Date 06/30/2005 Time 15:05:19  Page    1
```

The header indicates the compilation platform. You can customize the header on succeeding pages of the listing by using the compiler-directing `TITLE` statement.

RELATED REFERENCES
TITLE statement (*Enterprise COBOL Language Reference*)

Describing the computing environment

In the `ENVIRONMENT DIVISION` of a program, you describe the aspects of the program that depend on the computing environment.

Use the `CONFIGURATION SECTION` to specify the following items:

- Computer for compiling the program (in the `SOURCE-COMPUTER` paragraph)
- Computer for running the program (in the `OBJECT-COMPUTER` paragraph)
- Special items such as the currency sign and symbolic characters (in the `SPECIAL-NAMES` paragraph)
- User-defined classes (in the `REPOSITORY` paragraph)

Use the `FILE-CONTROL` and `I-O-CONTROL` paragraphs of the `INPUT-OUTPUT SECTION` to:

- Identify and describe the characteristics of the files in the program.
- Associate your files with the external QSAM, VSAM, or HFS (hierarchical file system) data sets where they physically reside.

  The terms *file* in COBOL terminology and *data set* or *HFS file* in operating-system terminology have essentially the same meaning and are used interchangeably in this information.

  For Customer Information Control System (CICS) and online Information Management System (IMS) message processing programs (MPP), code only the `ENVIRONMENT DIVISION` header and, optionally, the `CONFIGURATION SECTION`. CICS does not allow COBOL definition of files. IMS allows COBOL definition of files only for batch programs.
- Provide information to control efficient transmission of the data records between your program and the external medium.

"Example: FILE-CONTROL entries" on page 8

RELATED TASKS
"Specifying the collating sequence" on page 8
"Defining symbolic characters" on page 10

"Defining a user-defined class" on page 10
"Defining files to the operating system" on page 10

RELATED REFERENCES
Sections and paragraphs (*Enterprise COBOL Language Reference*)

Example: FILE-CONTROL entries

The following table shows example FILE-CONTROL entries for a QSAM sequential file, a VSAM indexed file, and a line-sequential file.

Table 1. **FILE-CONTROL entries**

QSAM file	VSAM file	Line-sequential file
SELECT PRINTFILE[1] ASSIGN TO UPDPRINT[2] ORGANIZATION IS SEQUENTIAL[3] ACCESS IS SEQUENTIAL.[4]	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.	SELECT PRINTFILE[1] ASSIGN TO UPDPRINT[2] ORGANIZATION IS LINE SEQUENTIAL[3] ACCESS IS SEQUENTIAL.[4]

1. The SELECT clause chooses a file in the COBOL program to be associated with an external data set.
2. The ASSIGN clause associates the program's name for the file with the external name for the actual data file. You can define the external name with a DD statement or an environment variable.
3. The ORGANIZATION clause describes the file's organization. For QSAM files, the ORGANIZATION clause is optional.
4. The ACCESS MODE clause defines the manner in which the records are made available for processing: sequential, random, or dynamic. For QSAM and line-sequential files, the ACCESS MODE clause is optional. These files always have sequential organization.
5. For VSAM files, you might have additional statements in the FILE-CONTROL paragraph depending on the type of VSAM file you use.

RELATED TASKS
Chapter 9, "Processing QSAM files," on page 151
Chapter 10, "Processing VSAM files," on page 179
Chapter 11, "Processing line-sequential files," on page 207
"Describing the computing environment" on page 7

Specifying the collating sequence

You can use the PROGRAM COLLATING SEQUENCE clause and the ALPHABET clause of the SPECIAL-NAMES paragraph to establish the collating sequence that is used in several operations on alphanumeric items.

These clauses specify the collating sequence for the following operations on alphanumeric items:

- Nonnumeric 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 in the SORT or MERGE statement

"Example: specifying the collating sequence" on page 9

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

- EBCDIC: references the collating sequence associated with the EBCDIC character set
- NATIVE: references the same collating sequence as EBCDIC
- STANDARD-1: references the collating sequence associated with the ASCII character set defined by *ANSI INCITS X3.4, Coded Character Sets - 7-bit American National Standard Code for Information Interchange (7-bit ASCII)*
- STANDARD-2: references the collating sequence associated with the character set defined by *ISO/IEC 646 — Information technology — ISO 7-bit coded character set for information interchange, International Reference Version*
- An alteration of the EBCDIC sequence that you define in the `SPECIAL-NAMES` paragraph

The `PROGRAM COLLATING SEQUENCE` clause does not affect comparisons that involve national or DBCS operands.

RELATED TASKS
"Choosing alternate collating sequences" on page 223
"Comparing national (UTF-16) data" on page 138

Example: specifying the collating sequence

The following example shows the `ENVIRONMENT DIVISION` coding that you can use to specify a collating sequence in which uppercase and lowercase letters are similarly handled in comparisons and in sorting and merging.

When you change the EBCDIC sequence in the `SPECIAL-NAMES` paragraph, the overall collating sequence is affected, not just the collating sequence of the characters that are included in the `SPECIAL-NAMES` paragraph.

```
IDENTIFICATION DIVISION.
. . .
ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
   Source-Computer. IBM-390.
   Object-Computer. IBM-390.
      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".
```

RELATED TASKS
"Specifying the collating sequence" on page 8

Defining symbolic characters

Use the `SYMBOLIC CHARACTERS` clause to give symbolic names to any character of the specified alphabet. Use ordinal position to identify the character, where position 1 corresponds to character X'00'.

For example, to give a name to the backspace character (X'16' in the EBCDIC alphabet), code:

```
SYMBOLIC CHARACTERS BACKSPACE IS 23
```

Defining a user-defined class

Use the `CLASS` clause to give a name to a set of characters that you list in the clause.

For example, name the set of digits by coding the following clause:

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

You can reference the class-name only in a class condition. (This user-defined class is not the same as an object-oriented class.)

Defining files to the operating system

For all files that you process in your COBOL program, you need to define the files to the operating system with an appropriate system data definition.

Depending on the operating system, this system data definition can take any of the following forms:

- `DD` statement for MVS JCL.
- `ALLOCATE` command under TSO.
- Environment variable for z/OS or UNIX. The contents can define either an MVS data set or a file in the HFS (hierarchical file system).

The following examples show the relationship of a `FILE-CONTROL` entry to the system data definition and to the `FD` entry in the `FILE SECTION`:

- JCL `DD` statement:

  ```
       (1)
  //OUTFILE  DD  DSNAME=MY.OUT171,UNIT=SYSDA,SPACE=(TRK,(50,5))
  /*
  ```

- Environment variable (`export` command):

  ```
            (1)
  export OUTFILE=DSN(MY.OUT171),UNIT(SYSDA),SPACE(TRK,(50,5))
  ```

- COBOL code:

  ```
  ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
      SELECT CARPOOL
        ASSIGN TO OUTFILE (1)
        ORGANIZATION IS SEQUENTIAL.
  ```

```
. . .
DATA DIVISION.
FILE SECTION.
FD CARPOOL       (2)
    LABEL  RECORD STANDARD
    BLOCK  CONTAINS 0 CHARACTERS
    RECORD CONTAINS 80 CHARACTERS
```

(1) The *assignment-name* in the `ASSIGN` clause points to the *ddname* `OUTFILE` in the `DD` statement or the environment variable `OUTFILE` in the `export` command:

- `//OUTFILE  DD DSNAME=OUT171 . . .`, or
- `export OUTFILE= . . .`

(2) When you specify a file *file-name* in a `FILE-CONTROL` entry, you must describe the file in an `FD` entry:

```
SELECT CARPOOL
. . .
FD CARPOOL
```

RELATED TASKS
"Optimizing buffer and device space" on page 12

RELATED REFERENCES
"FILE SECTION entries" on page 14
File section (*Enterprise COBOL Language Reference*)

Varying the input or output file at run time

The *file-name* that you code in a `SELECT` clause is used as a constant throughout your COBOL program, but you can associate the name of the file with a different actual file at run time.

Changing a file-name in a COBOL program would require changing the input statements and output statements and recompiling the program. Alternatively, you can change the `DSNAME` value in the `DD` statement or the `DSN` or `PATH` value in the `export` command to use a different file at run time.

Environment variable values that are in effect at the time of the `OPEN` statement are used for associating COBOL file-names to the system file-names (including any path specifications).

The name that you use in the *assignment-name* of the `ASSIGN` clause must be the same as the ddname in the `DD` statement or the environment variable in the `export` command.

The *file-name* that you use in the `SELECT` clause (such as `SELECT MASTER`) must be the same as in the `FD` *file-name* entry.

Two files should not use the same ddname or environment variable name in their `SELECT` clauses; otherwise, results could be unpredictable. For example, if `DISPLAY` output is directed to `SYSOUT`, do not use `SYSOUT` as the ddname or environment variable name in the `SELECT` clause for a file.

"Example: using different input files"

Example: using different input files: This example shows that you use the same COBOL program to access different files by coding a `DD` statement or an `export` command before the programs runs.

Consider a COBOL program that contains the following SELECT clause:

```
SELECT MASTER ASSIGN TO DA-3330-S-MASTERA
```

Assume the three possible input files are MASTER1, MASTER2, and MASTER3. Before running the program, code one of the following DD statements in the job step that calls for program execution, or issue one of the following export commands from the same shell from which you run the program:

```
//MASTERA     DD   DSNAME=MY.MASTER1,. . .
export MASTERA=DSN(MY.MASTER1),. . .

//MASTERA     DD   DSNAME=MY.MASTER2,. . .
export MASTERA=DSN(MY.MASTER2),. . .

//MASTERA     DD   DSNAME=MY.MASTER3,. . .
export MASTERA=DSN(MY.MASTER3),. . .
```

Any reference in the program to MASTER will therefore be a reference to the file currently assigned to the ddname or environment-variable name MASTERA.

Notice that in this example, you cannot use the PATH(*path*) form of the export command to reference a line-sequential file in the HFS, because you cannot specify an organization field (S- or AS-) with a line-sequential file.

Optimizing buffer and device space

Use the APPLY WRITE-ONLY clause to make optimum use of buffer and device space when you create a sequential file with blocked variable-length records.

With APPLY WRITE-ONLY specified, a buffer is truncated only when the next record does not fit in the unused portion of the buffer. Without APPLY WRITE-ONLY specified, a buffer is truncated when it does not have enough space for a maximum-size record.

The APPLY WRITE-ONLY clause has meaning only for sequential files that have variable-length records and are blocked.

The AWO compiler option applies an implicit APPLY WRITE-ONLY clause to all eligible files. The NOAWO compiler option has no effect on files that have the APPLY WRITE-ONLY clause specified. The APPLY WRITE-ONLY clause takes precedence over the NOAWO compiler option.

The APPLY-WRITE ONLY clause can cause input files to use a record area rather than process the data in the buffer. This use might affect the processing of both input files and output files.

RELATED REFERENCES
"AWO" on page 302

Describing the data

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

You can use these sections for defining the following types of data:

- Data used in input-output operations (FILE SECTION)
- Data developed for internal processing:

  - To have storage be statically allocated and exist for the life of the *run unit* (WORKING-STORAGE SECTION)
  - To have storage be allocated each time a program is entered, and deallocated on return from the program (LOCAL-STORAGE SECTION)
- Data from another program (LINKAGE SECTION)

The Enterprise COBOL compiler limits the maximum size of DATA DIVISION elements.

RELATED CONCEPTS
"Comparison of WORKING-STORAGE and LOCAL-STORAGE" on page 15

RELATED TASKS
"Using data in input and output operations"
"Using data from another program" on page 17

RELATED REFERENCES
Compiler limits (*Enterprise COBOL Language Reference*)

Using data in input and output operations

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

Provide the following information about the data:

- Name the input and output files that the program will use. Use the FD entry to give names to the files that the input-output statements in the PROCEDURE DIVISION can refer to.

  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 that follows the FD entry, describe the fields of the records in the file:
  - You can code a level-01 description of the entire record, and then in the WORKING-STORAGE SECTION code a working copy that describes the fields of the record in more detail. Use the READ INTO statement to bring the records into WORKING-STORAGE. Processing occurs on the copy of data in WORKING-STORAGE. A WRITE FROM statement writes processed data into the record area defined in the FILE SECTION.
  - The record-name established is the object of WRITE and REWRITE statements.
  - For QSAM files only, you can set the record format in the RECORDING MODE clause. If you omit the RECORDING MODE clause, the compiler determines the record format based on the RECORD clause and on the level-01 record descriptions.
  - For QSAM files, you can set a blocking factor for the file in the BLOCK CONTAINS clause. If you omit the BLOCK CONTAINS clause, the file defaults to unblocked. However, you can override this with z/OS data management facilities (including a DD file job-control statement).
  - For line-sequential files, you can set a blocking factor for the file in the BLOCK CONTAINS clause. When you code BLOCK CONTAINS 1 RECORDS, or BLOCK CONTAINS *n* CHARACTERS, where *n* is the length of one logical record in bytes, WRITE statements result in the record being transferred immediately to the file rather than being buffered. This technique is useful when you want each record written immediately, such as to an error log.

Programs in the same run unit can share, or have access to, common files. The method for doing this depends on whether the programs are part of a nested (contained) structure or are separately compiled (including programs compiled as part of a batch sequence).

You can use the EXTERNAL clause 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.

You can use the GLOBAL clause 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.

RELATED CONCEPTS
"Nested programs" on page 444

RELATED TASKS
"Sharing files between programs (external files)" on page 461

RELATED REFERENCES
"FILE SECTION entries"

FILE SECTION entries

The entries that you can use in the FILE SECTION are summarized in the table below.

Table 2. **FILE SECTION entries**

Clause	To define	Notes
FD	The *file-name* to be referred to in PROCEDURE DIVISION input-output statements (OPEN, CLOSE, READ, also START and DELETE for VSAM)	Must match *file-name* in the SELECT clause. *file-name* is associated with a *ddname* through the *assignment-name*.
BLOCK CONTAINS	Size of physical records	If the CHARACTERS phrase is specified, size indicates the number of bytes in a record regardless of the USAGE of the data items in the record. QSAM: If provided, must match information on JCL or data-set label. If specified as BLOCK CONTAINS 0, or not provided, the system determines the optimal block size for you. Line sequential: Can be specified to control buffering for WRITE statements. VSAM: Syntax-checked, but has no effect on execution.
RECORD CONTAINS *n*	Size of logical records (fixed length)	Integer size indicates the number of bytes in a record regardless of the USAGE of the data items in the record. If the clause is provided, it must match information on JCL or data-set label. If *n* is equal to 0, LRECL must be coded on JCL or data-set label.

Table 2. **FILE SECTION entries** *(continued)*

Clause	To define	Notes
`RECORD IS VARYING`	Size of logical records (variable length)	Integer size or sizes, if specified, indicate the number of bytes in a record regardless of the `USAGE` of the data items in the record. If the clause is provided, it must match information on JCL or data-set label; compiler checks that record descriptions match.
`RECORD CONTAINS` *n* `TO` *m*	Size of logical records (variable length)	The integer sizes indicate the number of bytes in a record regardless of the `USAGE` of the data items in the record. If the clause is provided, it must match information on JCL or data-set label; compiler checks that record descriptions match.
`LABEL RECORDS`	Labels for QSAM files	VSAM: Handled as comments
`STANDARD`	Labels exist	QSAM: Handled as comments
`OMITTED`	Labels do not exist	QSAM: Handled as comments
data-name	Labels defined by the user	QSAM: Allowed for (optional) tape or disk
`VALUE OF`	An item in the label records associated with file	Comments only
`DATA RECORDS`	Names of records associated with file	Comments only
`LINAGE`	Depth of logical page	QSAM only
`CODE-SET`	ASCII or EBCDIC files	QSAM only. When an ASCII file is identified with the `CODE-SET` clause, the corresponding `DD` statement might need to have `DCB=(OPTCD=Q. . .)` or `DCB=(RECFM=D. . .)` coded if the file was not created using VS COBOL II, COBOL for OS/390 & VM, or IBM Enterprise COBOL for z/OS.
`RECORDING MODE`	Physical record description	QSAM only

RELATED CONCEPTS
"Labels for QSAM files" on page 173

RELATED REFERENCES
File section (*Enterprise COBOL Language Reference*)

Comparison of WORKING-STORAGE and LOCAL-STORAGE

How data items are allocated and initialized varies depending on whether the items are in the `WORKING-STORAGE SECTION` or `LOCAL-STORAGE SECTION`.

`WORKING-STORAGE` for programs is allocated at the start of the run unit.

Any data items that have `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 in the PROGRAM-ID paragraph

  In this case, WORKING-STORAGE data items are reinitialized each time that the program is entered.
- A subprogram that is dynamically called and then canceled

  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.

See the related tasks for information about WORKING-STORAGE in COBOL class definitions.

A separate copy of LOCAL-STORAGE data is allocated for each call of a program or invocation of a method, and is freed on return from the program or method. If you specify a VALUE clause for a LOCAL-STORAGE item, the item is initialized to that value on each call or invocation. If a VALUE clause is not specified, the initial value of the item is undefined.

Threading: Each invocation of a program that runs simultaneously on multiple threads shares access to a single copy of WORKING-STORAGE data. Each invocation has a separate copy of LOCAL-STORAGE data.

"Example: storage sections"

RELATED TASKS
"Ending and reentering main programs or subprograms" on page 434
Chapter 27, "Preparing COBOL programs for multithreading," on page 477
"WORKING-STORAGE SECTION for defining class instance data" on page 532

RELATED REFERENCES
Working-storage section (*Enterprise COBOL Language Reference*)
Local-storage section (*Enterprise COBOL Language Reference*)

Example: storage sections

The following is an example of a recursive program that uses both WORKING-STORAGE and LOCAL-STORAGE.

```
CBL 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.
```

The program produces the following output:

```
0000! = 00000001
0001! = 00000001
0002! = 00000002
0003! = 00000006
0004! = 00000024
0005! = 00000120
```

The following tables show the changing values of the data items in `LOCAL-STORAGE` and `WORKING-STORAGE` in the successive recursive calls of the program, and in the ensuing gobacks. During the gobacks, `fact` progressively accumulates the value of 5! (five factorial).

Recursive calls	**Value for `num` in `LOCAL-STORAGE`**	**Value for `numb` in `WORKING-STORAGE`**	**Value for `fact` in `WORKING-STORAGE`**
Main	5	5	0
1	4	4	0
2	3	3	0
3	2	2	0
4	1	1	0
5	0	0	0

Gobacks	**Value for `num` in `LOCAL-STORAGE`**	**Value for `numb` in `WORKING-STORAGE`**	**Value for `fact` in `WORKING-STORAGE`**
5	0	0	1
4	1	0	1
3	2	0	2
2	3	0	6
1	4	0	24
Main	5	0	120

RELATED CONCEPTS
"Comparison of WORKING-STORAGE and LOCAL-STORAGE" on page 15

Using data from another program

How you share data depends on the type of program. You share data differently in programs that are separately compiled than you do for programs that are nested or for programs that are recursive or multithreaded.

RELATED TASKS
"Sharing data in separately compiled programs" on page 18
"Sharing data in nested programs" on page 18
"Sharing data in recursive or multithreaded programs" on page 18
"Passing data" on page 451

Sharing data in separately compiled programs

Many applications consist of separately compiled programs that call and pass data to one another. Use the `LINKAGE SECTION` in the called program to describe the data passed from another program.

In the calling program, use a `CALL . . . USING` or `INVOKE . . . USING` statement to pass the data.

RELATED TASKS
"Passing data" on page 451

Sharing data in nested programs

Some applications consist of nested programs, that is, programs that are contained in other programs. Level-01 data items can include the `GLOBAL` attribute. This attribute allows any nested program that includes the declarations to access these 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.

RELATED CONCEPTS
"Nested programs" on page 444

Sharing data in recursive or multithreaded programs

If your program has the `RECURSIVE` attribute or is compiled with the `THREAD` compiler option, data that is defined in the `LINKAGE SECTION` is not accessible on subsequent invocations of the program.

To address a record in the `LINKAGE SECTION`, use either of these techniques:

- Pass an argument to the program and specify the record in an appropriate position in the `USING` phrase in the program.
- Use the format-5 `SET` statement.

If your program has the `RECURSIVE` attribute or is compiled with the `THREAD` compiler option, the address of the record is valid for a particular instance of the program invocation. The address of the record in another execution instance of the same program must be reestablished for that execution instance. Unpredictable results will occur if you refer to a data item for which the address has not been established.

RELATED CONCEPTS
"Multithreading" on page 478

RELATED TASKS
"Making recursive calls" on page 447
"Processing files with multithreading" on page 480

RELATED REFERENCES
"THREAD" on page 342
SET statement (*Enterprise COBOL Language Reference*)

Processing the data

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

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

```
PROCEDURE DIVISION.
```

You can code the division header to receive parameters by using the `USING` phrase, or to return a value by using the `RETURNING` phrase.

To receive an argument that was passed by reference (the default) or by content, code the division header for a program in either of these ways:

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

Be sure to define *dataname* in the `LINKAGE SECTION` of the `DATA DIVISION`.

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

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

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

```
PROCEDURE DIVISION RETURNING dataname2
```

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

```
PROCEDURE DIVISION USING dataname RETURNING dataname2
```

Be sure to define *dataname* and *dataname2* in the `LINKAGE SECTION`.

RELATED CONCEPTS
"How logic is divided in the PROCEDURE DIVISION"

RELATED TASKS
"Eliminating repetitive coding" on page 639

RELATED REFERENCES
The procedure division header (*Enterprise COBOL Language Reference*)
The USING phrase (*Enterprise COBOL Language Reference*)
CALL statement (*Enterprise COBOL Language Reference*)

How logic is divided in the PROCEDURE DIVISION

The `PROCEDURE DIVISION` of a program is divided into sections and paragraphs, which contain sentences, statements, and phrases.

Section

Logical subdivision of your processing logic.

A section has a section header and is optionally followed by one or more paragraphs.

A section can be the subject of a `PERFORM` statement. One type of section is for declaratives.

Paragraph
Subdivision of a section, procedure, or program.

A paragraph has a name followed by a period and zero or more sentences.

A paragraph can be the subject of a statement.

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

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

A statement is a valid combination of words, and begins with a COBOL verb. Statements are imperative (indicating unconditional action), conditional, or compiler-directing. Using explicit scope terminators instead of periods to show the logical end of a statement is preferred.

Phrase
A subdivision of a statement.

RELATED CONCEPTS
"Compiler-directing statements" on page 21
"Scope terminators" on page 22
"Imperative statements"
"Conditional statements"
"Declaratives" on page 23

RELATED REFERENCES
PROCEDURE DIVISION structure (*Enterprise COBOL Language Reference*)

Imperative statements

An imperative statement (such as `ADD`, `MOVE`, `INVOKE`, or `CLOSE`) indicates an unconditional action to be taken.

You can end an imperative statement with an implicit or explicit scope terminator.

A conditional statement that ends with an explicit scope terminator becomes an imperative statement called a *delimited scope statement*. Only imperative statements (or delimited scope statements) can be nested.

RELATED CONCEPTS
"Conditional statements"
"Scope terminators" on page 22

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.

You can end a conditional statement with an implicit or explicit scope terminator. If you end a conditional statement explicitly, it becomes a delimited scope statement (which is an imperative statement).

You can use a delimited scope statement 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 phrase 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.

For additional program control, you can use the NOT phrase 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. Nested statements must be imperative statements (or delimited scope statements) and must follow the rules for imperative statements.

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

- Arithmetic statement with ON SIZE ERROR
- Data-manipulation statements with ON OVERFLOW
- CALL statements with ON OVERFLOW
- I/O statements with INVALID KEY, AT END, or AT END-OF-PAGE
- RETURN with AT END

RELATED CONCEPTS
"Imperative statements" on page 20
"Scope terminators" on page 22

RELATED TASKS
"Selecting program actions" on page 89

RELATED REFERENCES
Conditional statements (*Enterprise COBOL Language Reference*)

Compiler-directing statements

A compiler-directing statement causes the compiler to take specific action about the program structure, COPY processing, listing control, or control flow.

A compiler-directing statement is not part of the program logic.

RELATED REFERENCES
Chapter 18, "Compiler-directing statements," on page 351
Compiler-directing statements (*Enterprise COBOL Language Reference*)

Scope terminators

A scope terminator ends a verb or statement. Scope terminators can be explicit or implicit.

Explicit scope terminators end a verb without ending a sentence. They consist of END followed by a hyphen and the name of the verb being terminated, such as END-IF. An implicit scope terminator is a period (.) that ends the scope of all previous statements not yet ended.

Each of the two periods in the following program fragment ends an IF statement, making the code equivalent to the code after it that instead uses explicit scope terminators:

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

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

If you use implicit terminators, the end of statements can be unclear. 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 MOVE statement and the DISPLAY statement after it are 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, use explicit scope terminators, especially within paragraphs. Use implicit scope terminators only at the end of a paragraph or the end of a program.

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
```

RELATED CONCEPTS
"Conditional statements" on page 20
"Imperative statements" on page 20

Declaratives

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

Start each declarative section with a USE statement that identifies the function of the section. In the procedures, specify the actions to be taken when the condition occurs.

RELATED TASKS
"Finding and handling input-output errors" on page 357

RELATED REFERENCES
Declaratives (*Enterprise COBOL Language Reference*)

Chapter 2. Using data

This information is intended to help non-COBOL programmers relate terms for data used in other programming languages to COBOL terms. It introduces COBOL fundamentals for variables, structures, literals, and constants; assigning and displaying values; intrinsic (built-in) functions, and tables (arrays) and pointers.

RELATED CONCEPTS
"Storage and its addressability" on page 41

RELATED TASKS
"Using variables, structures, literals, and constants"
"Assigning values to data items" on page 29
"Displaying values on a screen or in a file (DISPLAY)" on page 37
"Using intrinsic functions (built-in functions)" on page 40
"Using tables (arrays) and pointers" on page 41
Chapter 7, "Processing data in an international environment," on page 121

Using variables, structures, literals, and constants

Most high-level programming languages share the concept of data being represented as variables, structures (group items), literals, or constants.

| The data in a COBOL program can be alphabetic, alphanumeric, double-byte
| character set (DBCS), national, or numeric. You can also define index-names and data items described as `USAGE POINTER`, `USAGE FUNCTION-POINTER`, `USAGE PROCEDURE-POINTER`, or `USAGE OBJECT REFERENCE`. You place all data definitions in the `DATA DIVISION` of your program.

RELATED TASKS
"Using variables"
"Using data items and group items" on page 26
"Using literals" on page 27
"Using constants" on page 28
"Using figurative constants" on page 28

RELATED REFERENCES
Classes and categories of data (*Enterprise COBOL Language Reference*)

Using variables

A *variable* is a data item whose value can change during a program. The value is restricted, however, to the data type that you define when you specify a name and a length for the data item.

| For example, if a customer name is an alphanumeric data item in your program, you could define and use the customer name as shown below:

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

| You could instead declare the customer names above as national data items by
| specifying their PICTURE clauses as Pic N(20) and specifying the USAGE NATIONAL
| clause for the items. National data items are represented in Unicode UTF-16, in
| which most characters are represented in 2 bytes of storage.

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125

RELATED TASKS
"Using national data (Unicode) in COBOL" on page 126

RELATED REFERENCES
"NSYMBOL" on page 323
"Storage of national data" on page 133
PICTURE clause (*Enterprise COBOL Language Reference*)

Using data items and group items

Related data items can be parts of a hierarchical data structure. A data item that does not have subordinate data items is called an *elementary item*. A data item that is composed of one or more subordinate data items is called a *group item*.

| A record can be either an elementary item or a group item. A group item can be
| either an *alphanumeric group item* or a *national group item*.

| For example, Customer-Record below is an alphanumeric group item that is
| composed of two subordinate alphanumeric group items (Customer-Name and
| Part-Order), each of which contains elementary data items. These groups items
| implicitly have USAGE DISPLAY. You can refer to an entire group item or to parts of
a group item in MOVE statements in the PROCEDURE DIVISION as shown below:

```
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
    . . .
```

| You could instead define Customer-Record as a national group item that is
| composed of two subordinate national group items by changing the declarations in
| the DATA DIVISION as shown below. National group items behave in the same way
| as elementary category national data items in most operations. The GROUP-USAGE
| NATIONAL clause indicates that a group item and any group items subordinate to it
| are national groups. Subordinate elementary items in a national group must be
| explicitly or implicitly described as USAGE NATIONAL.

```
Data Division.
File Section.
FD  Customer-File
    Record Contains 90 Characters.
01  Customer-Record        Group-Usage National.
    05  Customer-Name.
        10 Last-Name       Pic n(17).
        10 Filler          Pic n.
        10 Initials        Pic nn.
    05  Part-Order.
        10 Part-Name       Pic n(15).
        10 Part-Color      Pic n(10).
Working-Storage Section.
01  Orig-Customer-Name     Group-Usage National.
    05  Surname            Pic n(17).
    05  Initials           Pic n(3).
01  Inventory-Part-Name    Pic n(15)  Usage National.
. . .
Procedure Division.
    Move Customer-Name to Orig-Customer-Name
    Move Part-Name to Inventory-Part-Name
    . . .
```

In the example above, the group items could instead specify the USAGE NATIONAL clause at the group level. A USAGE clause at the group level applies to each elementary data item in a group (and thus serves as a convenient shorthand notation). However, a group that specifies the USAGE NATIONAL clause is *not* a national group despite the representation of the elementary items within the group. Groups that specify the USAGE clause are alphanumeric groups and behave in many operations, such as moves and compares, like elementary data items of USAGE DISPLAY (except that no editing or conversion of data occurs).

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125
"National groups" on page 129

RELATED TASKS
"Using national data (Unicode) in COBOL" on page 126
"Using national groups" on page 130

RELATED REFERENCES
"FILE SECTION entries" on page 14
"Storage of national data" on page 133
Classes and categories of group items (*Enterprise COBOL Language Reference*)
PICTURE clause (*Enterprise COBOL Language Reference*)
MOVE statement (*Enterprise COBOL Language Reference*)
USAGE clause (*Enterprise COBOL Language Reference*)

Using literals

A *literal* is a character string whose value is given by the characters themselves. If you know the value you want a data item to have, you can use a literal representation of the data value in the PROCEDURE DIVISION.

You do not need to declare a data item for the value nor refer to it by using a data-name. For example, you can prepare an error message for an output file by moving an alphanumeric literal:

```
Move "Name is not valid" To Customer-Name
```

You can compare a data item to a specific integer value by using a numeric literal. In the example below, "Name is not valid" is an alphanumeric literal, and 03519 is a numeric literal:

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

| You can use the opening delimiter N" or N' to designate a national literal if the
| NSYMBOL(NATIONAL) compiler option is in effect, or to designate a DBCS literal if the
| NSYMBOL(DBCS) compiler option is in effect.

| You can use the opening delimiter NX" or NX' to designate national literals in
| hexadecimal notation (regardless of the setting of the NSYMBOL compiler option).
| Each group of four hexadecimal digits designates a single national character.

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125

RELATED TASKS
"Using national literals" on page 127
"Using DBCS literals" on page 141

RELATED REFERENCES
"NSYMBOL" on page 323
Literals (*Enterprise COBOL Language Reference*)

Using constants

A *constant* is a data item that has only one value. COBOL does not define a
| construct for constants. However, you can define a data item with an initial value
| by coding a VALUE clause in the data description (instead of coding an INITIALIZE
statement).

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

| The example above initializes an alphanumeric and a numeric data item. You can
| likewise use a VALUE clause in defining a national or DBCS constant.

RELATED TASKS
"Using national data (Unicode) in COBOL" on page 126
"Coding for use of DBCS support" on page 141

Using figurative constants

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

For example:

```
Move Spaces To Report-Header
```

RELATED TASKS
"Using national-character figurative constants" on page 128
"Coding for use of DBCS support" on page 141

RELATED REFERENCES
Figurative constants (*Enterprise COBOL Language Reference*)

Assigning values to data items

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 what you want to do.

Table 3. **Assignment to data items in a program**

What you want to do	How to do it
Assign values to a data item or large data area.	Use 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 initial state)
Assign the results of arithmetic.	Use `COMPUTE`, `ADD`, `SUBTRACT`, `MULTIPLY`, or `DIVIDE` statements.
Examine or replace characters or groups of characters in a data item.	Use the `INSPECT` statement.
Receive values from a file.	Use the `READ` (or `READ INTO`) statement.
Receive values from a system input device or a file.	Use the `ACCEPT` statement.
Establish a constant.	Use the `VALUE` clause in the definition of the data item, and do not use the data item as a receiver. Such an item is in effect a constant even though the compiler does not enforce read-only constants.
One of these actions: • Place a value associated with a table element in an index. • Set the status of an external switch to `ON` or `OFF`. • Move data to a condition-name to make the condition true. • Set a `POINTER`, `PROCEDURE-POINTER`, or `FUNCTION-POINTER` data item to an address. • Associate an `OBJECT REFERENCE` data item with an object instance.	Use the `SET` statement.

"Examples: initializing data items" on page 30

RELATED TASKS
"Initializing a structure (INITIALIZE)" on page 32
"Assigning values to elementary data items (MOVE)" on page 34
"Assigning values to group data items (MOVE)" on page 35
"Assigning input from a screen or file (ACCEPT)" on page 36
"Joining data items (STRING)" on page 101
"Splitting data items (UNSTRING)" on page 103
"Assigning arithmetic results (MOVE or COMPUTE)" on page 36
"Tallying and replacing data items (INSPECT)" on page 111
Chapter 7, "Processing data in an international environment," on page 121

Examples: initializing data items

The following examples show how you can initialize many kinds of data items, including alphanumeric, national-edited, and numeric-edited data items, by using INITIALIZE statements.

An INITIALIZE statement is functionally equivalent to one or more MOVE statements. The related tasks about initializing show how you can use an INITIALIZE statement on a group item to conveniently initialize all the subordinate data items that are in a given data category.

Initializing a data item to blanks or zeros:

```
INITIALIZE identifier-1
```

identifier-1 **PICTURE**	*identifier-1* **before**	*identifier-1* **after**
9(5)	12345	00000
X(5)	AB123	*bbbbb*[1]
N(3)	004100420031[2]	002000200020[3]
99XX9	12AB3	*bbbbb*[1]
XXBX/XX	ABbC/DE	*bbbb*/*bb*[1]
**99.9CR	1234.5CR	**00.0*bb*[1]
A(5)	ABCDE	*bbbbb*[1]
+99.99E+99	+12.34E+02	+00.00E+00

1. The symbol *b* represents a blank space.
2. Hexadecimal representation of the national (UTF-16) characters 'AB1'. The example assumes that *identifier-1* has Usage National.
3. Hexadecimal representation of the national (UTF-16) characters '   ' (three blank spaces). Note that if *identifier-1* were not defined as Usage National, and if NSYMBOL(DBCS) were in effect, INITIALIZE would instead store DBCS spaces ('4040') into *identifier-1*.

Initializing an alphanumeric data item:

```
01  ALPHANUMERIC-1    PIC X    VALUE "y".
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 an alphanumeric right-justified data item:

```
01  ANJUST            PIC X(8)  VALUE SPACES  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	*bbbbbbbb*[1]	*bbbb*ABCD[1]

1. The symbol *b* represents a blank space.

Initializing an alphanumeric-edited data item:

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

ALPHANUM-EDIT-3	ALPHANUM-EDIT-1 before	ALPHANUM-EDIT-1 after
M/*bb*[1]	AB*b*C/DEF[1]	M/*bb*/*bbb*[1]

1. The symbol *b* represents a blank space.

Initializing a national data item:

```
01  NATIONAL-1        PIC NN  USAGE NATIONAL  VALUE N"AB".
01  NATIONAL-3        PIC NN  USAGE NATIONAL  VALUE N"CD".
. . .
    INITIALIZE NATIONAL-1
        REPLACING NATIONAL DATA BY NATIONAL-3
```

NATIONAL-3	NATIONAL-1 before	NATIONAL-1 after
00430044[1]	00410042[2]	00430044[1]

1. Hexadecimal representation of the national characters 'CD'
2. Hexadecimal representation of the national characters 'AB'

Initializing a national-edited data item:

```
01  NATL-EDIT-1       PIC 0NN  USAGE NATIONAL  VALUE N"123".
01  NATL-3            PIC NNN  USAGE NATIONAL  VALUE N"456".
. . .
    INITIALIZE NATL-EDIT-1
        REPLACING NATIONAL-EDITED DATA BY NATL-3
```

NATL-3	NATL-EDIT-1 before	NATL-EDIT-1 after
003400350036[1]	003100320033[2]	003000340035[3]

1. Hexadecimal representation of the national characters '456'
2. Hexadecimal representation of the national characters '123'
3. Hexadecimal representation of the national characters '045'

Initializing a numeric (zoned decimal) data item:

```
01  NUMERIC-1         PIC 9(8)        VALUE 98765432.
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 a numeric (national decimal) data item:

```
01  NAT-DEC-1         PIC 9(3)  USAGE  NATIONAL VALUE 987.
01  NUM-INT-BIN-3     PIC 9(2)  BINARY VALUE 12.
. . .
    INITIALIZE NAT-DEC-1
        REPLACING NUMERIC DATA BY NUM-INT-BIN-3
```

NUM-INT-BIN-3	NAT-DEC-1 before	NAT-DEC-1 after
12	003900380037[1]	003000310032[2]

1. Hexadecimal representation of the national characters '987'
2. Hexadecimal representation of the national characters '012'

Initializing a numeric-edited (USAGE DISPLAY) data item:

```
01  NUM-EDIT-DISP-1   PIC $ZZ9V  VALUE "$127".
01  NUM-DISP-3        PIC 999V   VALUE 12.
. . .
    INITIALIZE NUM-EDIT-DISP-1
        REPLACING NUMERIC DATA BY NUM-DISP-3
```

NUM-DISP-3	NUM-EDIT-DISP-1 before	NUM-EDIT-DISP-1 after
012	$127	$ 12

Initializing a numeric-edited (USAGE NATIONAL) data item:

```
01  NUM-EDIT-NATL-1   PIC $ZZ9V  NATIONAL VALUE N"$127".
01  NUM-NATL-3        PIC 999V   NATIONAL VALUE 12.
. . .
    INITIALIZE NUM-EDIT-NATL-1
        REPLACING NUMERIC DATA BY NUM-NATL-3
```

NUM-NATL-3	NUM-EDIT-NATL-1 before	NUM-EDIT-NATL-1 after
003000310032[1]	0024003100320037[2]	0024002000310032[3]

1. Hexadecimal representation of the national characters '012'
2. Hexadecimal representation of the national characters '$127'
3. Hexadecimal representation of the national characters '$ 12'

RELATED TASKS
"Initializing a structure (INITIALIZE)"
"Initializing a table (INITIALIZE)" on page 75
"Defining numeric data" on page 45

RELATED REFERENCES
"NSYMBOL" on page 323

Initializing a structure (INITIALIZE)

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

The following example shows how you can reset fields to spaces and zeros in transaction records that a program produces. The values of the fields are not identical in each record that is produced. (The transaction record is defined as an alphanumeric group item, TRANSACTION-OUT.)

```
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
```

Record	**TRANSACTION-OUT before**	**TRANSACTION-OUT after**
1	R001383000240000000000000000	*b*000000000000000000000000000[1]
2	R001390000480000000000000000	*b*000000000000000000000000000[1]
3	S001410000120000000000000000	*b*000000000000000000000000000[1]
4	C001383000000000425000000000	*b*000000000000000000000000000[1]
5	C002010000000000000100000000	*b*000000000000000000000000000[1]
1. The symbol *b* represents a blank space.		

You can likewise reset the values of all the subordinate data items in a national group item by applying the INITIALIZE statement to that group item. The following structure is similar to the preceding structure, but instead uses Unicode UTF-16 data:

```
01  TRANSACTION-OUT GROUP-USAGE NATIONAL.
    05  TRANSACTION-CODE        PIC N.
    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
```

Regardless of the previous contents of the transaction record, after the INITIALIZE statement above is executed:

- TRANSACTION-CODE contains NX"0020" (a national space).
- Each of the remaining 27 national character positions of TRANSACTION-OUT contains NX"0030" (a national-decimal zero).

When you use an INITIALIZE statement to initialize an alphanumeric or national group data item, the data item is processed as a group item, that is, with group semantics. The elementary data items within the group are recognized and processed, as shown in the examples above. If you do not code the REPLACING phrase of the INITIALIZE statement:

- SPACE is the implied sending item for alphabetic, alphanumeric, alphanumeric-edited, DBCS, category national, and national-edited receiving items.
- ZERO is the implied sending item for numeric and numeric-edited receiving items.

RELATED CONCEPTS
"National groups" on page 129

RELATED TASKS
"Initializing a table (INITIALIZE)" on page 75
"Using national groups" on page 130

RELATED REFERENCES
INITIALIZE statement (*Enterprise COBOL Language Reference*)

Assigning values to elementary data items (MOVE)

Use a `MOVE` statement to assign a value to an elementary data item.

The following statement assigns the contents of an elementary data item, `Customer-Name`, to the elementary data item `Orig-Customer-Name`:

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

If `Customer-Name` is longer than `Orig-Customer-Name`, truncation occurs on the right. If `Customer-Name` is shorter, the extra character positions on the right in `Orig-Customer-Name` are filled with spaces.

For data items that contain numbers, moves can be more complicated than with character data items because there are several ways in which numbers can be represented. In general, the algebraic values of numbers are moved if possible, as opposed to the digit-by-digit moves that are performed with character data. For example, after the `MOVE` statement below, `Item-x` contains the value 3.0, represented as 0030:

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

You can move an alphabetic, alphanumeric, alphanumeric-edited, DBCS, integer, or numeric-edited data item to a category national or national-edited data item; the sending item is converted. You can move a national data item to a category national or national-edited data item. If the content of a category national data item has a numeric value, you can move that item to a numeric, numeric-edited, external floating-point, or internal floating-point data item. You can move a national-edited data item only to a category national data item or another national-edited data item. Padding or truncation might occur.

For complete details about elementary moves, see the related reference below about the `MOVE` statement.

The following example shows an alphanumeric data item in the Greek language that is moved to a national data item:

```
CBL CODEPAGE(00875)
. . .
01 Data-in-Unicode   Pic N(100) usage national.
01 Data-in-Greek     Pic X(100).
. . .
    Read Greek-file into Data-in-Greek
    Move Data-in-Greek to Data-in-Unicode
```

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125

RELATED TASKS
"Assigning values to group data items (MOVE)" on page 35
"Converting to or from national (Unicode) representation" on page 133

RELATED REFERENCES
"CODEPAGE" on page 305
Classes and categories of data (*Enterprise COBOL Language Reference*)
MOVE statement (*Enterprise COBOL Language Reference*)

Assigning values to group data items (MOVE)

Use the MOVE statement to assign values to group data items.

You can move a national group item (a data item that is described with the GROUP-USAGE NATIONAL clause) to another national group item. The compiler processes the move as though each national group item were an elementary item of category national, that is, as if each item were described as PIC N(*m*), where *m* is the length of that item in national character positions.

You can move an alphanumeric group item to an alphanumeric group item or to a national group item. You can also move a national group item to an alphanumeric group item. The compiler performs such moves as group moves, that is, without consideration of the individual elementary items in the sending or receiving group, and without conversion of the sending data item. Be sure that the subordinate data descriptions in the sending and receiving group items are compatible. The moves occur even if a destructive overlap could occur at run time.

You can code the CORRESPONDING phrase in a MOVE statement to move subordinate elementary items from one group item to the identically named corresponding subordinate elementary items in another group item:

```
01  Group-X.
    02 T-Code    Pic X    Value "A".
    02 Month     Pic 99   Value 04.
    02 State     Pic XX   Value "CA".
    02 Filler    PIC X.
01  Group-N      Group-Usage National.
    02 State     Pic NN.
    02 Month     Pic 99.
    02 Filler    Pic N.
    02 Total     Pic 999.
. . .
    MOVE CORR Group-X TO Group-N
```

In the example above, State and Month within Group-N receive the values in national representation of State and Month, respectively, from Group-X. The other data items in Group-N are unchanged. (Filler items in a receiving group item are unchanged by a MOVE CORRESPONDING statement.)

In a MOVE CORRESPONDING statement, sending and receiving group items are treated as group items, not as elementary data items; group semantics apply. That is, the elementary data items within each group are recognized, and the results are the same as if each pair of corresponding data items were referenced in a separate MOVE statement. Data conversions are performed according to the rules for the MOVE statement as specified in the related reference below. For details about which types of elementary data items correspond, see the related reference about the CORRESPONDING phrase.

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125
"National groups" on page 129

RELATED TASKS
"Assigning values to elementary data items (MOVE)" on page 34
"Using national groups" on page 130
"Converting to or from national (Unicode) representation" on page 133

RELATED REFERENCES
Classes and categories of group items (*Enterprise COBOL Language Reference*)
MOVE statement (*Enterprise COBOL Language Reference*)
CORRESPONDING phrase (*Enterprise COBOL Language Reference*)

Assigning arithmetic results (MOVE or COMPUTE)

When assigning a number to a data item, consider using the `COMPUTE` statement instead of the `MOVE` statement.

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

In the example above, the two statements in most cases have the same effect. The `MOVE` statement however carries out the assignment with truncation. You can use the `DIAGTRUNC` compiler option to request that the compiler issue a warning for `MOVE` statements that might truncate numeric receivers.

When significant left-order digits would be lost in execution, the `COMPUTE` statement can detect the condition and allow you to handle it. If 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 you do not specify the `ON SIZE ERROR` phrase, the assignment is carried out with truncation. There is no `ON SIZE ERROR` support for the `MOVE` statement.

You can also use the `COMPUTE` statement to assign the result of an arithmetic expression or intrinsic function to a data item. For example:

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

You can assign the results of date, time, mathematical, and other calculations to data items by using Language Environment callable services. Language Environment services are available through a standard COBOL `CALL` statement, and the values they return are passed in the parameters of the `CALL` statement. For example, you can call the Language Environment service CEESIABS to find the absolute value of a data item by coding the following statement:

```
Call 'CEESIABS' Using Arg, Feedback-code, Result.
```

As a result of this call, data item `Result` is assigned the absolute value of the value in data item `Arg`; data item `Feedback-code` contains the return code that indicates whether the service completed successfully. You have to define all the data items in the `DATA DIVISION` using the correct descriptions according to the requirements of the particular callable service. For the example above, the data items could be defined as follows:

```
77 Arg            Pic s9(9)  Binary.
77 Feedback-code  Pic x(12)  Display.
77 Result         Pic s9(9)  Binary.
```

RELATED REFERENCES
"DIAGTRUNC" on page 310
Intrinsic functions (*Enterprise COBOL Language Reference*)
Callable services (*Language Environment Programming Reference*)

Assigning input from a screen or file (ACCEPT)

One way to assign a value to a data item is to read the value from a screen or a file.

To enter data from the screen, first associate the monitor with a mnemonic-name in the SPECIAL-NAMES paragraph. Then use ACCEPT to assign the line of input entered at the screen to a data item. For example:

```
Environment Division.
Configuration Section.
Special-Names.
    Console is Names-Input.
. . .
    Accept Customer-Name From Names-Input
```

To read from a file instead of the screen, make the following change:

- Change Console to *device*, where *device* is any valid system device (for example, SYSIN). For example:

  ```
  SYSIN is Names-Input
  ```

  device can be a ddname that references a hierarchical file system (HFS) path. If this DD is not defined and your program is running in a UNIX environment, stdin is the input source. If this DD is not defined and your program is not running in a UNIX environment, the ACCEPT statement fails.

When you use the ACCEPT statement, you can assign a value to an alphanumeric or national group item, or to an elementary data item that has USAGE DISPLAY, USAGE DISPLAY-1, or USAGE NATIONAL.

When you assign a value to a USAGE NATIONAL data item, input data from the console is converted from the EBCDIC code page specified in the CODEPAGE compiler option to national (Unicode UTF-16) representation. This is the only case where conversion of national data is done when you use the ACCEPT statement. Conversion is done in this case because the input is known to be coming from a screen.

To have conversion done when the input data is from any other device, use the NATIONAL-OF intrinsic function.

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125

RELATED TASKS
"Converting alphanumeric and DBCS data to national data (NATIONAL-OF)" on page 135

RELATED REFERENCES
"CODEPAGE" on page 305
ACCEPT statement (*Enterprise COBOL Language Reference*)
SPECIAL-NAMES paragraph (*Enterprise COBOL Language Reference*)

Displaying values on a screen or in a file (DISPLAY)

You can display the value of a data item on a screen or write it to a file by using the DISPLAY statement.

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

In the example above, if the content of data item *Customer-Name* is JOHNSON, then the statement displays the following message on the system logical output device:

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

To write data to a destination other than the system logical output device, use the UPON phrase with a destination other than SYSOUT. For example, the following statement writes to the file specified in the SYSPUNCH DD statement:

```
Display "Hello" upon syspunch.
```

You can specify a file in the HFS by using the SYSPUNCH DD statement. For example, the following definition causes DISPLAY output to be written to the file /u/*userid*/cobol/demo.lst:

```
//SYSPUNCH DD PATH='/u/userid/cobol/demo.lst',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU,
// FILEDATA=TEXT
```

The following statement writes to the job log or console and to the TSO screen if you are running under TSO:

```
Display "Hello" upon console.
```

| When you display the value of a USAGE NATIONAL data item to the console, it is converted from Unicode (UTF-16) representation to EBCDIC based on the value of the CODEPAGE option. This is the only case where conversion of national data is done when you use the DISPLAY statement. Conversion is done in this case because the output is known to be directed to a screen.

To have a national data item be converted when you direct output to a different device, use the DISPLAY-OF intrinsic function, such as in the following example:

```
01 Data-in-Unicode pic N(10) usage national.
. . .
    Display function Display-of(Data-in-Unicode, 00037)
```

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125

RELATED TASKS
"Displaying data on the system logical output device"
"Using WITH NO ADVANCING" on page 39
"Converting national data to alphanumeric data (DISPLAY-OF)" on page 135
"Coding COBOL programs to run under CICS" on page 393

RELATED REFERENCES
"CODEPAGE" on page 305
DISPLAY statement (*Enterprise COBOL Language Reference*)

Displaying data on the system logical output device

To write data to the system logical output device, either omit the UPON clause or use the UPON clause with destination SYSOUT.

```
Display "Hello" upon sysout.
```

The output is directed to the ddname that you specify in the OUTDD compiler option. You can specify a file in the hierarchical file system with this ddname.

If the OUTDD ddname is not allocated and you are not running in a UNIX environment, a default DD of SYSOUT=* is allocated. If the OUTDD ddname is not allocated and you are running in a UNIX environment, the _IGZ_SYSOUT environment variable is used as follows:

Undefined or set to stdout
    Output is routed to stdout (file descriptor 1).

Set to stderr
Output is routed to stderr (file descriptor 2).

Otherwise (set to something other than stdout or stderr)
The DISPLAY statement fails; a severity-3 Language Environment condition is raised.

When DISPLAY output is routed to stdout or stderr, the output is not subdivided into records. The output is written as a single stream of characters without line breaks.

If OUTDD and the Language Environment runtime option MSGFILE specify the same ddname, both DISPLAY output and Language Environment runtime diagnostics are routed to the Language Environment message file.

RELATED TASKS
"Setting and accessing environment variables" on page 424

RELATED REFERENCES
"OUTDD" on page 328
DISPLAY statement (*Enterprise COBOL Language Reference*)

Using WITH NO ADVANCING

If you specify the WITH NO ADVANCING phrase, and output is going to a ddname, the printer control character + (plus) is placed into the first output position from the *next* DISPLAY statement. + is the ANSI-defined printer control character that suppresses line spacing before a record is printed.

If you specify the WITH NO ADVANCING phrase and the output is going to stdout or stderr, a new-line character is not appended to the end of the stream. A subsequent DISPLAY statement might add additional characters to the end of the stream.

If you do not specify WITH NO ADVANCING, and the output is going to a ddname, the printer control character ' ' (space) is placed into the first output position from the next DISPLAY statement, indicating single-spaced output.

```
DISPLAY "ABC"
DISPLAY "CDEF" WITH NO ADVANCING
DISPLAY "GHIJK" WITH NO ADVANCING
DISPLAY "LMNOPQ"
DISPLAY "RSTUVWX"
```

If you code the statements above, the result sent to the output device is:

```
 ABC
 CDEF
+GHIJK
+LMNOPQ
 RSTUVMX
```

The output that is printed depends on how the output device interprets printer control characters.

If you do not specify the WITH NO ADVANCING phrase and the output is going to stdout or stderr, a new-line character is appended to the end of the stream.

RELATED REFERENCES
DISPLAY statement (*Enterprise COBOL Language Reference*)

Using intrinsic functions (built-in functions)

Some high-level programming languages have built-in functions that you can reference in your program as if they were variables that have defined attributes and a predetermined value. In COBOL, these functions are called *intrinsic functions*. They provide capabilities for manipulating strings and numbers.

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.

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). For example, the groups of highlighted words below are function-identifiers:

```
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 invocation of the function and the data value returned by the function. Because it actually represents a data item, you can use a function-identifier in most places in the PROCEDURE DIVISION where a data item that has the attributes of the returned value can be used.

The COBOL word function is a reserved word, but the function-names are not reserved. You can use them in other contexts, such as for the name of a data item. For example, you could use Sqrt to invoke an intrinsic function and to name a data item in your program:

```
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)
    . . .
```

A function-identifier represents a value that is of one of these types: alphanumeric, national, numeric, or integer. You can include a substring specification (reference modifier) in a function-identifier for alphanumeric or national functions. Numeric intrinsic functions are further classified according to the type of numbers they return.

The functions MAX, MIN, DATEVAL, and UNDATE can return either type of value depending on the type of arguments you supply.

The functions DATEVAL, UNDATE, and YEARWINDOW are provided with the millennium language extensions to assist with manipulating and converting windowed date fields.

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, Function Sqrt(5) returns a numeric value. Thus, the three arguments to the MAX function below are all numeric, which is an allowable argument type for this function:

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

RELATED TASKS
"Processing table items using intrinsic functions" on page 86
"Converting data items (intrinsic functions)" on page 112
"Evaluating data items (intrinsic functions)" on page 115

Using tables (arrays) and pointers

In COBOL, arrays are called *tables*. A table is a set of logically consecutive data items that you define in the DATA DIVISION by using the OCCURS clause.

Pointers are data items that contain virtual storage addresses. You define them either explicitly with the USAGE IS POINTER clause in the DATA DIVISION or implicitly as ADDRESS OF special registers.

You can perform the following operations with pointer data items:
- Pass them between programs by using the CALL . . . BY REFERENCE statement.
- Move them to other pointers by using the SET statement.
- Compare them to other pointers for equality by using a relation condition.
- Initialize them to contain an invalid address by using VALUE IS NULL.

Use pointer data items to:
- Accomplish limited base addressing, particularly if you want to pass and receive addresses of a record area that is defined with OCCURS DEPENDING ON and is therefore variably located.
- Handle a chained list.

RELATED TASKS
"Defining a table (OCCURS)" on page 69
"Using procedure and function pointers" on page 447

Storage and its addressability

When you run COBOL programs, the programs and the data that they use reside in virtual storage. Storage that you use with COBOL can be either below the 16-MB line or above the 16-MB line but below the 2-GB bar. Two modes of addressing are available to address this storage: 24-bit and 31-bit.

You can address storage below (but not above) the 16-MB line with 24-bit addressing. You can address storage either above or below the 16-MB line with 31-bit addressing. *Unrestricted storage* is addressable by 31-bit addressing and therefore encompasses all the storage available to your program, both above and below the 16-MB line.

Enterprise COBOL does not directly exploit the 64-bit virtual addressing capability of z/OS; however, COBOL applications running in 31-bit or 24-bit addressing mode are fully supported on 64-bit z/OS systems.

Addressing mode (AMODE) is the attribute that tells which hardware addressing mode is supported by your program: 24-bit addressing, 31-bit addressing, or either 24-bit or 31-bit addressing. This attribute is AMODE 24, AMODE 31, or AMODE ANY, respectively. The object program, the load module, and the executing program each has an AMODE attribute. All Enterprise COBOL object programs are AMODE ANY.

Residency mode (RMODE) is the attribute of a program load module that identifies where in virtual storage the program will reside: below the 16-MB line, or either below or above. This attribute is RMODE 24 or RMODE ANY.

Enterprise COBOL uses Language Environment services to control the storage used at run time. Thus COBOL compiler options and Language Environment runtime options influence the AMODE and RMODE attributes of your program and data, alone and in combination:

DATA Compiler option that influences the location of storage for WORKING-STORAGE data, I-O buffers, and parameter lists for programs compiled with RENT.

RMODE Compiler option that influences the residency mode and also influences the location of storage for WORKING-STORAGE data, I-O buffers, and parameter lists for programs compiled with NORENT.

RENT Compiler option to generate a reentrant program.

HEAP Runtime option that controls storage for the runtime heap. For example, COBOL WORKING-STORAGE is allocated from heap storage.

STACK Runtime option that controls storage for the runtime stack. For example, COBOL LOCAL-STORAGE is allocated from stack storage.

ALL31 Runtime option that specifies whether an application can run entirely in AMODE 31.

Settings for RMODE

The RMODE and RENT options determine the RMODE attribute of your program:

Table 4. **Effect of RMODE and RENT compiler options on the RMODE attribute**

RMODE compiler option	RENT compiler option	RMODE attribute
RMODE(AUTO)	NORENT	RMODE 24
RMODE(AUTO)	RENT	RMODE ANY
RMODE(24)	RENT or NORENT	RMODE 24
RMODE(ANY)	RENT or NORENT	RMODE ANY

| **Link-edit considerations:** When the object code that COBOL generates has an
| attribute of RMODE 24, you must link-edit it with RMODE 24. When the object code
| that COBOL generates has an attribute of RMODE ANY, you can link-edit it with
| RMODE ANY or RMODE 24.

Storage restrictions for passing data

Do not pass parameters that are allocated in storage above the 16-MB line to AMODE 24 subprograms. Force the WORKING-STORAGE data and parameter lists below the line for programs that run in 31-bit addressing mode and pass data to programs that run in AMODE 24:

- Compile reentrant programs (RENT) with DATA(24).
- Compile nonreentrant programs (NORENT) with RMODE(24) or RMODE(AUTO).
- Nonreentrant programs (NORENT) compiled with RMODE(ANY) must be link-edited with RMODE 24. The data areas for NORENT programs are above the 16-MB line or

below the 16-MB line depending on where the program is loaded, even if the program was compiled with DATA(24). The DATA option does not affect programs compiled with NORENT.

Location of data areas

For reentrant programs, the DATA compiler option and the HEAP runtime option control whether storage for data areas such as WORKING-STORAGE SECTION and FD record areas is obtained from below the 16-MB line or from unrestricted storage. Compile programs with RENT or RMODE(ANY) if they will be run with 31-bit addressing in virtual storage addresses above the 16-MB line. The DATA option does not affect programs compiled with NORENT.

When you specify the runtime option HEAP(,,BELOW), the DATA compiler option has no effect; the storage for WORKING-STORAGE SECTION data areas is allocated from below the 16-MB line. However, with HEAP(,,ANYWHERE) as the runtime option, storage for data areas is allocated from below the 16-MB line if you compiled the program with the DATA(24) compiler option, or from unrestricted storage if you compiled with the DATA(31) compiler option.

Storage for LOCAL-STORAGE data

The location of LOCAL-STORAGE data items is controlled by the STACK runtime option and the AMODE of the program. LOCAL-STORAGE data items are acquired in unrestricted storage when the STACK(,,ANYWHERE) runtime option is in effect and the program is running in AMODE 31. Otherwise LOCAL-STORAGE is acquired below the 16-MB line. The DATA compiler option does not influence the location of LOCAL-STORAGE data.

Storage for external data

In addition to affecting how storage is obtained for dynamic data areas (WORKING-STORAGE, FD record areas, and parameter lists), the DATA compiler option can also influence where storage for EXTERNAL data is obtained. Storage required for EXTERNAL data is obtained from unrestricted storage if the following conditions are met:

- The program is compiled with the DATA(31) and RENT compiler options or the RMODE(ANY) and NORENT compiler options.
- The HEAP(,,ANYWHERE) runtime option is in effect.
- The ALL31(ON) runtime option is in effect.

In all other cases, the storage for EXTERNAL data is obtained from below the 16-MB line. When you specify the ALL31(ON) runtime option, all the programs in the run unit must be capable of running in 31-bit addressing mode.

Storage for QSAM input-output buffers

The DATA compiler option can also influence where input-output buffers for QSAM files are obtained. See the related references below for information about allocation of buffers for QSAM files and the DATA compiler option.

RELATED CONCEPTS
"AMODE switching" on page 439
Heap storage overview: AMODE considerations (*Language Environment Programming Guide*)

RELATED TASKS
Chapter 24, "Using subprograms," on page 433
Chapter 25, "Sharing data," on page 451

RELATED REFERENCES
"Allocation of buffers for QSAM files" on page 172
"DATA" on page 307
"RENT" on page 331
"RMODE" on page 332
"Performance-related compiler options" on page 633
HEAP (*Language Environment Programming Reference*)
STACK (*Language Environment Programming Reference*)
ALL31 (*Language Environment Programming Reference*)
MVS Program Management: User's Guide and Reference

Chapter 3. Working with numbers and arithmetic

In general, you can view COBOL numeric data as a series of decimal digit positions. However, numeric items can also have special properties such as an arithmetic sign or a currency sign.

To define, display, and store numeric data so that you can perform arithmetic operations efficiently:

- Use the `PICTURE` clause and the characters `9`, `+`, `-`, `P`, `S`, and `V` to define numeric data.
- Use the `PICTURE` clause and editing characters (such as `Z`, comma, and period) along with `MOVE` and `DISPLAY` statements to display numeric data.
- Use the `USAGE` clause with various formats to control how numeric data is stored.
- Use the numeric class test to validate that data values are appropriate.
- Use `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, and `COMPUTE` statements to perform arithmetic.
- Use the `CURRENCY SIGN` clause and appropriate `PICTURE` characters to designate the currency you want.

RELATED TASKS
"Defining numeric data"
"Displaying numeric data" on page 47
"Controlling how numeric data is stored" on page 48
"Checking for incompatible data (numeric class test)" on page 56
"Performing arithmetic" on page 57
"Using currency signs" on page 66

Defining numeric data

Define numeric items by using the `PICTURE` clause with the character `9` in the data description to represent the decimal digits of the number. Do not use an `X`, which is for alphanumeric data items.

| For example, `Count-y` below is a numeric data item, an external decimal item that
| has `USAGE DISPLAY` (a *zoned decimal item*):

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

| You can similarly define numeric data items to hold national characters (UTF-16).
| For example, `Count-n` below is an external decimal data item that has `USAGE`
| `NATIONAL` (a *national decimal item*):

```
05  Count-n       Pic 9(4)  Value 25  Usage National.
```

You can code up to 18 digits in the `PICTURE` clause when you compile using the default compiler option `ARITH(COMPAT)` (referred to as *compatibility mode*). When you compile using `ARITH(EXTEND)` (referred to as *extended mode*), you can code up to 31 digits in the `PICTURE` clause.

Other characters of special significance that you can code are:

P Indicates leading or trailing zeroes

S	Indicates a sign, positive or negative
V	Implies a decimal point

The s in the following example means that the value is signed:

```
05  Price  Pic s99v99.
```

The field can therefore hold a positive or a negative value. The v indicates the position of an implied decimal point, but does not contribute to the size of the item because it does not require a storage position. An s usually does not contribute to the size of a numeric item, because by default s does not require a storage position.

However, if you plan to port your program or data to a different machine, you might want to code the sign for a zoned decimal data item as a separate position in storage. In the following case, the sign takes 1 byte:

```
05  Price  Pic s99V99  Sign Is Leading, Separate.
```

This coding ensures that the convention your machine uses for storing a nonseparate sign will not cause unexpected results on a machine that uses a different convention.

Separate signs are also preferable for zoned decimal data items that will be printed or displayed.

Separate signs are required for national decimal data items that are signed. The sign takes 2 bytes of storage, as in the following example:

```
05 Price Pic s99V99 Usage National Sign Is Leading, Separate.
```

You cannot use the PICTURE clause with internal floating-point data (COMP-1 or COMP-2). However, you can use the VALUE clause to provide an initial value for an internal floating-point literal:

```
05  Compute-result  Usage Comp-2  Value 06.23E-24.
```

For information about external floating-point data, see the examples referenced below and the related concept about formats for numeric data.

"Examples: numeric data and internal representation" on page 52

RELATED CONCEPTS
"Formats for numeric data" on page 49
Appendix A, "Intermediate results and arithmetic precision," on page 647

RELATED TASKS
"Displaying numeric data" on page 47
"Controlling how numeric data is stored" on page 48
"Performing arithmetic" on page 57
"Defining national numeric data items" on page 129

RELATED REFERENCES
"Sign representation of zoned and packed-decimal data" on page 55
"Storage of national data" on page 133
"ARITH" on page 302
"NUMPROC" on page 325
SIGN clause (*Enterprise COBOL Language Reference*)

Displaying numeric data

You can define numeric items with certain editing symbols (such as decimal points, commas, dollar signs, and debit or credit signs) to make the items easier to read and understand when you display or print them.

For example, in the code below, Edited-price is a numeric-edited item that has USAGE DISPLAY. (You can specify the clause USAGE IS DISPLAY for numeric-edited items; however, it is implied. It means that the items are stored in character format.)

```
05  Price          Pic     9(5)v99.
05  Edited-price   Pic  $zz,zz9.99.
. . .
Move Price To Edited-price
Display Edited-price
```

If the contents of Price are 0150099 (representing the value 1,500.99), `$ 1,500.99` is displayed when you run the code. The z in the PICTURE clause of Edited-price indicates the suppression of leading zeros.

You can define numeric-edited data items to hold national (UTF-16) characters instead of alphanumeric characters. To do so, declare the numeric-edited items as USAGE NATIONAL. The effect of the editing symbols is the same for numeric-edited items that have USAGE NATIONAL as it is for numeric-edited items that have USAGE DISPLAY, except that the editing is done with national characters. For example, if Edited-price is declared as USAGE NATIONAL in the code above, the item is edited and displayed using national characters.

To display numeric or numeric-edited data items that have USAGE NATIONAL in EBCDIC, direct them to CONSOLE. For example, if Edited-price in the code above has USAGE NATIONAL, `$ 1,500.99` is displayed when you run the program if the last statement above is:

```
Display Edited-price Upon Console
```

You can cause an elementary numeric or numeric-edited item to be filled with spaces when a value of zero is stored into it by coding the BLANK WHEN ZERO clause for the item. For example, each of the DISPLAY statements below causes blanks to be displayed instead of zeroes:

```
05 Price           Pic     9(5)v99.
05 Edited-price-D  Pic  $99,999.99
       Blank When Zero.
05 Edited-price-N  Pic  $99,999.99 Usage National
       Blank When Zero.
. . .
Move 0 to Price
Move Price to Edited-price-D
Move Price to Edited-price-N
Display Edited-price-D
Display Edited-price-N upon console
```

You cannot use numeric-edited items as sending operands in arithmetic expressions or in ADD, SUBTRACT, MULTIPLY, DIVIDE, or COMPUTE statements. (Numeric editing takes place when a numeric-edited item is the receiving field for one of these statements, or when a MOVE statement has a numeric-edited receiving field and a numeric-edited or numeric sending field.) You use numeric-edited items primarily for displaying or printing numeric data.

| You can move numeric-edited items to numeric or numeric-edited items. In the following example, the value of the numeric-edited item (whether it has USAGE DISPLAY or USAGE NATIONAL) is moved to the numeric item:

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

If these two statements immediately followed the statements in the first example above, then Price would be displayed as 0150099, representing the value 1,500.99. Price would also be displayed as 0150099 if Edited-price had USAGE NATIONAL.

You can also move numeric-edited items to alphanumeric, alphanumeric-edited, floating-point, and national data items. For a complete list of the valid receiving items for numeric-edited data, see the related reference about the MOVE statement.

"Examples: numeric data and internal representation" on page 52

RELATED TASKS
"Displaying values on a screen or in a file (DISPLAY)" on page 37
"Controlling how numeric data is stored"
"Defining numeric data" on page 45
"Performing arithmetic" on page 57
"Defining national numeric data items" on page 129
"Converting to or from national (Unicode) representation" on page 133

RELATED REFERENCES
MOVE statement (*Enterprise COBOL Language Reference*)
BLANK WHEN ZERO clause (*Enterprise COBOL Language Reference*)

Controlling how numeric data is stored

You can control how the computer stores numeric data items by coding the USAGE clause in your data description entries.

You might want to control the format for any of several reasons such as these:
- Arithmetic performed with computational data types is more efficient than with USAGE DISPLAY or USAGE NATIONAL data types.
- Packed-decimal format requires less storage per digit than USAGE DISPLAY or USAGE NATIONAL data types.
- Packed-decimal format converts to and from DISPLAY or NATIONAL format more efficiently than binary format does.
- Floating-point format is well suited for arithmetic operands and results with widely varying scale, while maintaining the maximal number of significant digits.
- You might need to preserve data formats when you move data from one machine to another.

The numeric data you use in your program will have one of the following formats available with COBOL:
- External decimal (USAGE DISPLAY or USAGE NATIONAL)
- External floating point (USAGE DISPLAY or USAGE NATIONAL)
- Internal decimal (USAGE PACKED-DECIMAL)
- Binary (USAGE BINARY)
- Native binary (USAGE COMP-5)

- Internal floating point (USAGE COMP-1 or USAGE COMP-2)

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

The compiler converts displayable numbers to the internal representation of their numeric values before using them in arithmetic operations. Therefore it is often more efficient if you define data items as BINARY or PACKED-DECIMAL than as DISPLAY or NATIONAL. For example:

```
05  Initial-count   Pic S9(4)  Usage Binary   Value 1000.
```

Regardless of which USAGE clause you use to control the internal representation of a value, you use the same PICTURE clause conventions and decimal value in the VALUE clause (except for internal floating-point data, for which you cannot use a PICTURE clause).

"Examples: numeric data and internal representation" on page 52

RELATED CONCEPTS
"Formats for numeric data"
"Data format conversions" on page 54
Appendix A, "Intermediate results and arithmetic precision," on page 647

RELATED TASKS
"Defining numeric data" on page 45
"Displaying numeric data" on page 47
"Performing arithmetic" on page 57

RELATED REFERENCES
"Conversions and precision" on page 54
"Sign representation of zoned and packed-decimal data" on page 55

Formats for numeric data

Several formats are available for numeric data.

External decimal (DISPLAY and NATIONAL) items

When USAGE DISPLAY is in effect for a category numeric data item (either because you have coded it, or by default), each position (byte) of storage contains one decimal digit. This means that the items are stored in displayable form. External decimal items that have USAGE DISPLAY are referred to as *zoned decimal* data items.

When USAGE NATIONAL is in effect for a category numeric data item, 2 bytes of storage are required for each decimal digit. The items are stored in UTF-16 format. External decimal items that have USAGE NATIONAL are referred to as *national decimal* data items.

National decimal data items, if signed, must have the SIGN SEPARATE clause in effect. All other rules for zoned decimal items apply to national decimal items. You can use national decimal items anywhere that other category numeric data items can be used.

External decimal (both zoned decimal and national decimal) data items are primarily intended for receiving and sending numbers between your program and files, terminals, or printers. You can also use external decimal items as operands and receivers in arithmetic processing. However, if your program performs a lot of intensive arithmetic, and efficiency is a high priority, COBOL's computational numeric types might be a better choice for the data items used in the arithmetic.

External floating-point (DISPLAY and NATIONAL) items

When `USAGE DISPLAY` is in effect for a floating-point data item (either because you have coded it, or by default), each `PICTURE` character position (except for `v`, an implied decimal point, if used) takes 1 byte of storage. The items are stored in displayable form. External floating-point items that have `USAGE DISPLAY` are referred to as *display floating-point* data items in this information when necessary to distinguish them from external floating-point items that have `USAGE NATIONAL`.

In the following example, `Compute-Result` is implicitly defined as a display floating-point item:

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

The minus signs (`-`) do not mean that the mantissa and exponent must necessarily be negative numbers. Instead, they mean that when the number is displayed, the sign appears as a blank for positive numbers or a minus sign for negative numbers. If you instead code a plus sign (`+`), the sign appears as a plus sign for positive numbers or a minus sign for negative numbers.

When `USAGE NATIONAL` is in effect for a floating-point data item, each `PICTURE` character position (except for `v`, if used) takes 2 bytes of storage. The items are stored as national characters (UTF-16). External floating-point items that have `USAGE NATIONAL` are referred to as *national floating-point* data items.

The existing rules for display floating-point items apply to national floating-point items.

In the following example, `Compute-Result-N` is a national floating-point item:

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

If `Compute-Result-N` is displayed, the signs appear as described above for `Compute-Result`, but in national characters. To instead display `Compute-Result-N` in EBCDIC characters, direct it to the console:

```
Display Compute-Result-N Upon Console
```

You cannot use the `VALUE` clause for external floating-point items.

As with external decimal numbers, external floating-point numbers have to be converted (by the compiler) to an internal representation of their numeric value before they can be used in arithmetic operations. If you compile with the default option `ARITH (COMPAT)`, external floating-point numbers are converted to long (64-bit) floating-point format. If you compile with `ARITH (EXTEND)`, they are instead converted to extended-precision (128-bit) floating-point format.

Binary (COMP) items

`BINARY`, `COMP`, and `COMP-4` are synonyms. Binary-format numbers occupy 2, 4, or 8 bytes of storage. If the `PICTURE` clause specifies that an item is signed, the leftmost bit is used as the operational sign.

A binary number with a `PICTURE` description of four or fewer decimal digits occupies 2 bytes; five to nine decimal digits, 4 bytes; and 10 to 18 decimal digits, 8 bytes. Binary items with nine or more digits require more handling by the compiler. Testing them for the SIZE ERROR condition and rounding is more cumbersome than with other types.

You can use binary items, for example, for indexes, subscripts, switches, and arithmetic operands or results.

Use the TRUNC(STD|OPT|BIN) compiler option to indicate how binary data (BINARY, COMP, or COMP-4) is to be truncated.

Native binary (COMP-5) items

Data items that you declare as USAGE COMP-5 are represented in storage as binary data. However, unlike USAGE COMP items, they can contain values of magnitude up to the capacity of the native binary representation (2, 4, or 8 bytes) rather than being limited to the value implied by the number of 9s in the PICTURE clause.

When you move or store numeric data into a COMP-5 item, truncation occurs at the binary field size rather than at the COBOL PICTURE size limit. When you reference a COMP-5 item, the full binary field size is used in the operation.

COMP-5 is thus particularly useful for binary data items that originate in non-COBOL programs where the data might not conform to a COBOL PICTURE clause.

The table below shows the ranges of possible values for COMP-5 data items.

Table 5. **Ranges in value of COMP-5 data items**

PICTURE	Storage representation	Numeric values
S9(1) through S9(4)	Binary halfword (2 bytes)	-32768 through +32767
S9(5) through S9(9)	Binary fullword (4 bytes)	-2,147,483,648 through +2,147,483,647
S9(10) through S9(18)	Binary doubleword (8 bytes)	-9,223,372,036,854,775,808 through +9,223,372,036,854,775,807
9(1) through 9(4)	Binary halfword (2 bytes)	0 through 65535
9(5) through 9(9)	Binary fullword (4 bytes)	0 through 4,294,967,295
9(10) through 9(18)	Binary doubleword (8 bytes)	0 through 18,446,744,073,709,551,615

You can specify scaling (that is, decimal positions or implied integer positions) in the PICTURE clause of COMP-5 items. If you do so, you must appropriately scale the maximal capacities listed above. For example, a data item you describe as PICTURE S99V99 COMP-5 is represented in storage as a binary halfword, and supports a range of values from -327.68 through +327.67.

Large literals in VALUE clauses: Literals specified in VALUE clauses for COMP-5 items can, with a few exceptions, contain values of magnitude up to the capacity of the native binary representation. See *Enterprise COBOL Language Reference* for the exceptions.

Regardless of the setting of the TRUNC compiler option, COMP-5 data items behave like binary data does in programs compiled with TRUNC(BIN).

Packed-decimal (COMP-3) items

`PACKED-DECIMAL` and `COMP-3` are synonyms. Packed-decimal items occupy 1 byte of storage for every two decimal digits you code in the `PICTURE` description, except that the rightmost byte contains only one digit and the sign. This format is most efficient when you code an odd number of digits in the `PICTURE` description, so that the leftmost byte is fully used. Packed-decimal items are handled as fixed-point numbers for arithmetic purposes.

Internal floating-point (COMP-1 and COMP-2) items

`COMP-1` refers to short floating-point format and `COMP-2` refers to long floating-point format, which occupy 4 and 8 bytes of storage, respectively. The leftmost bit contains the sign and the next 7 bits contain the exponent; the remaining 3 or 7 bytes contain the mantissa.

`COMP-1` and `COMP-2` data items are stored in zSeries hexadecimal format.

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125
Appendix A, "Intermediate results and arithmetic precision," on page 647

RELATED TASKS
"Defining numeric data" on page 45
"Defining national numeric data items" on page 129

RELATED REFERENCES
"Storage of national data" on page 133
"TRUNC" on page 343
Classes and categories of data (*Enterprise COBOL Language Reference*)
SIGN clause (*Enterprise COBOL Language Reference*)
VALUE clause (*Enterprise COBOL Language Reference*)

Examples: numeric data and internal representation

The following table shows the internal representation of numeric items.

Table 6. **Internal representation of numeric items**

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
External decimal	PIC S9999 DISPLAY	+ 1234	F1 F2 F3 C4
		- 1234	F1 F2 F3 D4
		1234	F1 F2 F3 C4
	PIC 9999 DISPLAY	1234	F1 F2 F3 F4
	PIC 9999 NATIONAL	1234	00 31 00 32 00 33 00 34
	PIC S9999 DISPLAY SIGN LEADING	+ 1234	C1 F2 F3 F4
		- 1234	D1 F2 F3 F4
	PIC S9999 DISPLAY SIGN LEADING SEPARATE	+ 1234	4E F1 F2 F3 F4
		- 1234	60 F1 F2 F3 F4
	PIC S9999 DISPLAY SIGN TRAILING SEPARATE	+ 1234	F1 F2 F3 F4 4E
		- 1234	F1 F2 F3 F4 60
	PIC S9999 NATIONAL SIGN LEADING SEPARATE	+ 1234	00 2B 00 31 00 32 00 33 00 34
		- 1234	00 2D 00 31 00 32 00 33 00 34
	PIC S9999 NATIONAL SIGN TRAILING SEPARATE	+ 1234	00 31 00 32 00 33 00 34 00 2B
		- 1234	00 31 00 32 00 33 00 34 00 2D
Binary	PIC S9999 BINARY PIC S9999 COMP PIC S9999 COMP-4	+ 1234	04 D2
		- 1234	FB 2E
	PIC S9999 COMP-5	+ 12345[1]	30 39
		- 12345[1]	CF C7
	PIC 9999 BINARY PIC 9999 COMP PIC 9999 COMP-4	1234	04 D2
	PIC 9999 COMP-5	60000[1]	EA 60
Internal decimal	PIC S9999 PACKED-DECIMAL PIC S9999 COMP-3	+ 1234	01 23 4C
		- 1234	01 23 4D
	PIC 9999 PACKED-DECIMAL PIC 9999 COMP-3	1234	01 23 4F
Internal floating point	COMP-1	+ 1234	43 4D 20 00
		- 1234	C3 4D 20 00
	COMP-2	+ 1234	43 4D 20 00 00 00 00 00
		- 1234	C3 4D 20 00 00 00 00 00
External floating point	PIC +9(2).9(2)E+99 DISPLAY	+ 12.34E+02	4E F1 F2 4B F3 F4 C5 4E F0 F2
		- 12.34E+02	60 F1 F2 4B F3 F4 C5 4E F0 F2
	PIC +9(2).9(2)E+99 NATIONAL	+ 12.34E+02	00 2B 00 31 00 32 00 2E 00 33 00 34 00 45 00 2B 00 30 00 32
		- 12.34E+02	00 2D 00 31 00 32 00 2E 00 33 00 34 00 45 00 2B 00 30 00 32

1. The example demonstrates that COMP-5 data items can contain values of magnitude up to the capacity of the native binary representation (2, 4, or 8 bytes), rather than being limited to the value implied by the number of 9s in the PICTURE clause.

Data format conversions

When the code in your program involves the interaction of items that have different data formats, the compiler converts those items either temporarily, for comparisons and arithmetic operations, or permanently, for assignment to the receiver in a `MOVE` or `COMPUTE` statement.

A conversion is actually a move of a value from one data item to another. The compiler performs any conversions that are required during the execution of arithmetic or comparisons by using the same rules that are used for `MOVE` and `COMPUTE` statements.

When possible, the compiler performs a move to preserve numeric value instead of a direct digit-for-digit move.

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

A loss of precision is possible in conversions between fixed-point data formats and floating-point data formats (short floating point, long floating point, or external floating point). These conversions happen during arithmetic evaluations that have a mixture of both fixed-point and floating-point operands.

RELATED REFERENCES
"Conversions and precision"
"Sign representation of zoned and packed-decimal data" on page 55

Conversions and precision

In some numeric conversions, a loss of precision is possible; other conversions preserve precision or result in rounding.

Because both fixed-point and external floating-point items have decimal characteristics, references to fixed-point items in the following examples include external floating-point items unless stated otherwise.

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

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

Conversions that lose precision

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

When a USAGE COMP-2 data item is moved to a fixed-point data item that has more than 18 digits, the fixed-point data item will receive only 18 significant digits, and the remaining digits will be zero.

Conversions that preserve precision

If a fixed-point data item that has six 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 nine or more digits and then returned to the USAGE COMP-1 data item, the original value is recovered.

If a fixed-point data item that has 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 or more 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 and 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.

RELATED CONCEPTS
Appendix A, "Intermediate results and arithmetic precision," on page 647

Sign representation of zoned and packed-decimal data

| Sign representation affects the processing and interaction of zoned decimal and
| internal decimal data.

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

Positive:
C, A, E, and F

Negative:
D and B

| The COBOL NUMPROC compiler option affects sign processing for zoned decimal and
| internal decimal data. NUMPROC has no effect on binary data, national decimal data, or floating-point data.

NUMPROC(PFD)

Given `X'`*sd*`'`, where *s* is the sign representation and *d* represents the digit, when you use `NUMPROC(PFD)`, the compiler assumes that the sign in your data is one of three preferred signs:

Signed positive or 0:
`X'C'`

Signed negative:
`X'D'`

Unsigned or alphanumeric:
`X'F'`

Based on this assumption, the compiler uses whatever sign it is given to process data. The preferred sign is generated only where necessary (for example, when unsigned data is moved to signed data). Using the `NUMPROC(PFD)` option can save processing time, but you must use preferred signs with your data for correct processing.

NUMPROC(NOPFD)

When the `NUMPROC(NOPFD)` compiler option is in effect, the compiler accepts any valid sign configuration. The preferred sign is always generated in the receiver. `NUMPROC(NOPFD)` is less efficient than `NUMPROC(PFD)`, but you should use it whenever data that does not use preferred signs might exist.

| If an unsigned, zoned-decimal sender is moved to an alphanumeric receiver, the sign is unchanged (even with `NUMPROC(NOPFD)` in effect).

NUMPROC(MIG)

When `NUMPROC(MIG)` is in effect, the compiler generates code that is similar to that produced by OS/VS COBOL. This option can be especially useful if you migrate OS/VS COBOL programs to IBM Enterprise COBOL for z/OS.

RELATED REFERENCES
"NUMPROC" on page 325
"ZWB" on page 349

Checking for incompatible data (numeric class test)

The compiler assumes that values you supply for a data item are valid for the `PICTURE` and `USAGE` clauses, and does not check their validity. Ensure that the contents of a data item conform to the `PICTURE` and `USAGE` clauses before using the item in additional processing.

It can happen that values are passed into your program and assigned to items that have incompatible data descriptions for those values. For example, nonnumeric data might be moved or passed into a field that is defined as numeric, or a signed number might be passed into a field that is defined as unsigned. In either case, the receiving fields contain invalid data. When you give an item a value that is incompatible with its data description, references to that item in the `PROCEDURE DIVISION` are undefined and your results are unpredictable.

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 PICTURE and USAGE of the data item. For example, a packed-decimal item is checked for hexadecimal values X'0' through X'9' in the digit positions and for a valid sign value in the sign position (whether separate or nonseparate).

| For zoned decimal and packed-decimal items, the numeric class test is affected by the NUMPROC compiler option and the NUMCLS option (which is set at installation time). To determine the NUMCLS setting used at your installation, consult your system programmer.

If NUMCLS(PRIM) is in effect at your installation, use the following table to find the values that the compiler considers valid for the sign.

Table 7. **NUMCLS(PRIM) and valid signs**

	NUMPROC(NOPFD)	NUMPROC(PFD)	NUMPROC(MIG)
Signed	C, D, F	C, D, +0 (positive zero)	C, D, F
Unsigned	F	F	F
Separate sign	+, -	+, -, +0 (positive zero)	+, -

If NUMCLS(ALT) is in effect at your installation, use the following table to find the values that the compiler considers valid for the sign.

Table 8. **NUMCLS(ALT) and valid signs**

	NUMPROC(NOPFD)	NUMPROC(PFD)	NUMPROC(MIG)
Signed	A to F	C, D, +0 (positive zero)	A to F
Unsigned	F	F	F
Separate sign	+, -	+, -, +0 (positive zero)	+, -

RELATED REFERENCES
"NUMPROC" on page 325

Performing arithmetic

You can use any of several COBOL language features (including COMPUTE, arithmetic expressions, numeric intrinsic functions, and math and date callable services) to perform arithmetic. Your choice depends on whether a feature meets your particular needs.

For most common arithmetic evaluations, the COMPUTE statement is appropriate. If you need to use numeric literals, numeric data, or arithmetic operators, you might want to use arithmetic expressions. In places where numeric expressions are allowed, you can save time by using numeric intrinsic functions. Language

Environment callable services for mathematical functions and for date and time operations also provide a means of assigning arithmetic results to data items.

RELATED TASKS
"Using COMPUTE and other arithmetic statements"
"Using arithmetic expressions"
"Using numeric intrinsic functions" on page 59
"Using math-oriented callable services" on page 60
"Using date callable services" on page 62

Using COMPUTE and other arithmetic statements

Use the COMPUTE statement for most arithmetic evaluations rather than ADD, SUBTRACT, MULTIPLY, and DIVIDE statements. Often you can code only one COMPUTE statement instead of several individual arithmetic statements.

The COMPUTE statement assigns the result of an arithmetic expression to one or more data items:

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

Some arithmetic calculations might be more intuitive using arithmetic statements other than COMPUTE. For example:

COMPUTE	Equivalent arithmetic statements
`Compute Increment = Increment + 1`	`Add 1 to Increment`
`Compute Balance = Balance - Overdraft`	`Subtract Overdraft from Balance`
`Compute IncrementOne = IncrementOne + 1` `Compute IncrementTwo = IncrementTwo + 1` `Compute IncrementThree = IncrementThree + 1`	`Add 1 to IncrementOne, IncrementTwo, IncrementThree`

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.

When you perform arithmetic calculations, you can use national decimal data items as operands just as you use zoned decimal data items. You can also use national floating-point data items as operands just as you use display floating-point operands.

RELATED CONCEPTS
"Fixed-point contrasted with floating-point arithmetic" on page 64
Appendix A, "Intermediate results and arithmetic precision," on page 647

RELATED TASKS
"Defining numeric data" on page 45

Using arithmetic expressions

You can use arithmetic expressions in many (but not all) places in statements 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. . .
```

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.

Arithmetic operators are evaluated in the following order of precedence:

Table 9. **Order of evaluation of arithmetic operators**

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 of precedence are evaluated from left to right; however, you can use parentheses to change the order of evaluation. Expressions in parentheses are evaluated before the individual operators are evaluated. Parentheses, whether necessary or not, make your program easier to read.

RELATED CONCEPTS
"Fixed-point contrasted with floating-point arithmetic" on page 64
Appendix A, "Intermediate results and arithmetic precision," on page 647

Using numeric intrinsic functions

You can use numeric intrinsic functions only in places where numeric expressions are allowed. These functions can save you time because you don't have to code the many common types of calculations that they provide.

Numeric intrinsic functions return a signed numeric value, and are treated as temporary numeric data items.

Numeric functions are classified into the following categories:

Integer
Those that return an integer

Floating point
Those that return a long (64-bit) or extended-precision (128-bit) floating-point value (depending on whether you compile using the default option `ARITH(COMPAT)` or using `ARITH(EXTEND)`)

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

You can use intrinsic functions to perform several different arithmetic operations, as outlined in the following table.

Table 10. **Numeric intrinsic functions**

Number handling	Date and 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	DATEVAL		COS	RANDOM
NUMVAL-C	DAY-OF-INTEGER		FACTORIAL	RANGE
ORD-MAX	DAY-TO-YYYYDDD		INTEGER	STANDARD-DEVIATION
ORD-MIN	INTEGER-OF-DATE		INTEGER-PART	VARIANCE
	INTEGER-OF-DAY		LOG	
	UNDATE		LOG10	
	WHEN-COMPILED		MOD	
	YEAR-TO-YYYY		REM	
	YEARWINDOW		SIN	
			SQRT	
			SUM	
			TAN	

"Examples: numeric intrinsic functions" on page 62

You can reference one function as the argument of another. A nested function is evaluated independently of the outer function (except when the compiler determines whether a mixed function should be evaluated using fixed-point or floating-point instructions).

You can also nest an arithmetic expression as an argument to a numeric function. For example, in the statement below, there are three function arguments (`a`, `b`, and the arithmetic expression `(c / d)`):

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

You can reference all the elements of a table (or array) as function arguments by using the `ALL` subscript.

You can also use the integer special registers as arguments wherever integer arguments are allowed.

Many of the capabilities of numeric intrinsic functions are also provided by Language Environment callable services.

RELATED CONCEPTS
"Fixed-point contrasted with floating-point arithmetic" on page 64
Appendix A, "Intermediate results and arithmetic precision," on page 647

RELATED REFERENCES
"ARITH" on page 302

Using math-oriented callable services

Most COBOL intrinsic functions have corresponding math-oriented callable services that you can use to produce the same results.

When you compile with the default option `ARITH(COMPAT)`, COBOL floating-point intrinsic functions return long (64-bit) results. When you compile with option `ARITH(EXTEND)`, COBOL floating-point intrinsic functions (with the exception of `RANDOM`) return extended-precision (128-bit) results.

For example (considering the first row of the table below), if you compile using ARITH(COMPAT), CEESDACS returns the same result as ACOS. If you compile using ARITH(EXTEND), CEESQACS returns the same result as ACOS.

Table 11. **Compatibility of math intrinsic functions and callable services**

COBOL intrinsic function	Corresponding long-precision Language Environment callable service	Corresponding extended-precision Language Environment callable service	Results same for intrinsic function and callable service?
ACOS	CEESDACS	CEESQACS	Yes
ASIN	CEESDASN	CEESQASN	Yes
ATAN	CEESDATN	CEESQATN	Yes
COS	CEESDCOS	CEESQCOS	Yes
LOG	CEESDLOG	CEESQLOG	Yes
LOG10	CEESDLG1	CEESQLG1	Yes
RANDOM[1]	CEERAN0	none	No
REM	CEESDMOD	CEESQMOD	Yes
SIN	CEESDSIN	CEESQSIN	Yes
SQRT	CEESDSQT	CEESQSQT	Yes
TAN	CEESDTAN	CEESQTAN	Yes

1. RANDOM returns a long (64-bit) floating-point result even if you pass it a 31-digit argument and compile with ARITH(EXTEND).

Both the RANDOM intrinsic function and CEERAN0 service generate random numbers between zero and one. However, because each uses its own algorithm, RANDOM and CEERAN0 produce different random numbers from the same seed.

Even for functions that produce the same results, how you use intrinsic functions and Language Environment callable services differs. The rules for the data types required for intrinsic function arguments are less restrictive. For numeric intrinsic functions, you can use arguments that are of any numeric data type. When you invoke a Language Environment callable service with a CALL statement, however, you must ensure that the parameters match the numeric data types (generally COMP-1 or COMP-2) required by that service.

The error handling of intrinsic functions and Language Environment callable services sometimes differs. If you pass an explicit feedback token when calling the Language Environment math services, you must check the feedback code after each call and take explicit action to deal with errors. However, if you call with the feedback token explicitly OMITTED, you do not need to check the token; Language Environment automatically signals any errors.

RELATED CONCEPTS
"Fixed-point contrasted with floating-point arithmetic" on page 64
Appendix A, "Intermediate results and arithmetic precision," on page 647

RELATED TASKS
"Using Language Environment callable services" on page 641

RELATED REFERENCES
"ARITH" on page 302

Using date callable services

Both the COBOL date intrinsic functions and the Language Environment date callable services are based on the Gregorian calendar. However, the starting dates can differ depending on the setting of the `INTDATE` compiler option.

When `INTDATE(LILIAN)` is in effect, COBOL uses October 15, 1582 as day 1. Language Environment always uses October 15, 1582 as day 1. If you use `INTDATE(LILIAN)`, you get equivalent results from COBOL intrinsic functions and Language Environment date callable services. The following table compares the results when `INTDATE(LILIAN)` is in effect.

Table 12. **INTDATE(LILIAN) and compatibility of date intrinsic functions and callable services**

COBOL intrinsic function	Language Environment callable service	Results
`DATE-OF-INTEGER`	CEEDATE with picture string `YYYYMMDD`	Compatible
`DAY-OF-INTEGER`	CEEDATE with picture string `YYYYDDD`	Compatible
`INTEGER-OF-DATE`	CEEDAYS	Compatible
`INTEGER-OF-DATE`	CEECBLDY	Incompatible

When the default setting of `INTDATE(ANSI)` is in effect, COBOL uses January 1, 1601 as day 1. The following table compares the results when `INTDATE(ANSI)` is in effect.

Table 13. **INTDATE(ANSI) and compatibility of date intrinsic functions and callable services**

COBOL intrinsic function	Language Environment callable service	Results
`INTEGER-OF-DATE`	CEECBLDY	Compatible
`DATE-OF-INTEGER`	CEEDATE with picture string `YYYYMMDD`	Incompatible
`DAY-OF-INTEGER`	CEEDATE with picture string `YYYYDDD`	Incompatible
`INTEGER-OF-DATE`	CEEDAYS	Incompatible

RELATED TASKS
"Using Language Environment callable services" on page 641

RELATED REFERENCES
"INTDATE" on page 317

Examples: numeric intrinsic functions

The following examples and accompanying explanations show intrinsic functions in each of several categories.

| Where the examples below show zoned decimal data items, national decimal items
| could instead be used. (Signed national decimal items, however, require that the
| `SIGN SEPARATE` clause be in effect.)

General number handling

Suppose you want to find the maximum value of two prices (represented below 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 can use

| NUMVAL-C (a function that returns the numeric value of an alphanumeric or national
| literal, or an alphanumeric or national data item) and the MAX and LENGTH functions to do so:

```
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 can use the following statement:

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

Date and 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 four-digit year, two-digit month, and two-digit day format (YYYYMMDD). The 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 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 that 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 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)
```

You can use the ANNUITY function 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 can calculate the monthly payment required to repay a $15,000 loan in three years at a 12% annual interest rate (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 that you can nest intrinsic functions, use arithmetic expressions as arguments, and perform previously complex calculations simply:

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

Here in the addend the intrinsic function REM (instead of a DIVIDE statement with a REMAINDER clause) returns the remainder of dividing X by 2.

Statistics

Intrinsic functions make calculating statistical information 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)
```

RELATED TASKS
"Converting to numbers (NUMVAL, NUMVAL-C)" on page 113

Fixed-point contrasted with floating-point arithmetic

How you code arithmetic in a program (whether an arithmetic statement, an intrinsic function, an expression, or some combination of these nested within each other) determines whether the evaluation is done with floating-point or fixed-point arithmetic.

Many statements in a program could involve arithmetic. For example, each of the following types of COBOL statements requires some 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)
```

Floating-point evaluations

In general, if your arithmetic coding has either of the characteristics listed below, it is evaluated in floating-point arithmetic:

- An operand or result field is floating point.

  An operand is floating point if you code it as a floating-point literal or if you code it as a data item that is defined as `USAGE COMP-1`, `USAGE COMP-2`, or external floating point (`USAGE DISPLAY` or `USAGE NATIONAL` with a floating-point `PICTURE`).

  An operand that is a nested arithmetic expression or a reference to a numeric intrinsic function results in floating-point arithmetic when any of the following conditions is true:

  – 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.

  An exponent contains decimal places if you use a literal that contains decimal places, give the item a `PICTURE` that contains decimal places, or use an arithmetic expression or function whose result has decimal places.

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

Fixed-point evaluations

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

Arithmetic comparisons (relation conditions)

When you compare numeric expressions using a relational operator, the numeric expressions (whether they are data items, arithmetic expressions, function references, or some combination of these) are comparands in the context of the entire evaluation. That is, the attributes of each can influence the evaluation of the other: both expressions are evaluated in fixed point, or both are evaluated in floating point. This is also true of abbreviated comparisons even though one comparand does not explicitly appear in the comparison. For example:

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

This statement has two comparisons: `(a + d) = (b + e)`, and `(a + d) = c`. Although `(a + d)` does not explicitly appear in the second comparison, it is a comparand in that comparison. Therefore, the attributes of `c` can influence the evaluation of `(a + d)`.

The compiler handles comparisons (and the evaluation of any arithmetic expressions nested in comparisons) in floating-point arithmetic if either comparand is a floating-point value or resolves to a floating-point value.

The compiler handles comparisons (and the evaluation of any arithmetic expressions nested in comparisons) in fixed-point arithmetic if both comparands are fixed-point values or resolve to fixed-point values.

Implicit comparisons (no relational operator used) are not handled as a unit, however; the two comparands are treated separately as to their evaluation in floating-point or fixed-point arithmetic. In the following example, five arithmetic expressions are evaluated independently of one another's attributes, and then are compared to each other.

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

"Examples: fixed-point and floating-point evaluations"

RELATED REFERENCES
"Arithmetic expressions in nonarithmetic statements" on page 655

Examples: fixed-point and floating-point evaluations

The following example shows statements that are evaluated using fixed-point arithmetic and using floating-point arithmetic.

Assume that you define the data items for an employee table 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 statements are evaluated using 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
```

These statements are evaluated using 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)
```

Using currency signs

Many programs need to process financial information and present that information using the appropriate currency signs. With COBOL currency support (and the appropriate code page for your printer or display unit), you can use several currency signs in a program.

You can use one or more of the following signs:

- Symbols such as the dollar sign ($)
- Currency signs of more than one character (such as USD or EUR)
- Euro sign, established by the Economic and Monetary Union (EMU)

To specify the symbols for displaying financial information, use the CURRENCY SIGN clause (in the SPECIAL-NAMES paragraph in the CONFIGURATION SECTION) with the PICTURE characters that relate to those symbols. In the following example, the PICTURE character $ indicates that the currency sign $US is to be used:

```
    Currency Sign is "$US" with Picture Symbol "$".
    . . .
77  Invoice-Amount      Pic $$,$$9.99.
. . .
    Display "Invoice amount is " Invoice-Amount.
```

In this example, if Invoice-Amount contained 1500.00, the display output would be:

```
Invoice amount is  $US1,500.00
```

By using more than one CURRENCY SIGN clause in your program, you can allow for multiple currency signs to be displayed.

You can use a hexadecimal literal to indicate the currency sign value. Using a hexadecimal literal could be useful if the data-entry method for the source program does not allow the entry of the intended characters easily. The following example shows the hexadecimal value X'9F' used as the currency sign:

```
    Currency Sign X'9F' with Picture Symbol 'U'.
    . . .
01  Deposit-Amount       Pic UUUUU9.99.
```

If there is no corresponding character for the euro sign on your keyboard, you need to specify it as a hexadecimal value in the CURRENCY SIGN clause. The hexadecimal value for the euro sign is either X'9F' or X'5A' depending on the code page in use, as shown in the following table.

Table 14. **Hexadecimal values of the euro sign**

Code page CCSID	Applicable countries	Modified from	Euro sign
1140	USA, Canada, Netherlands, Portugal, Australia, New Zealand	037	X'9F'
1141	Austria, Germany	273	X'9F'
1142	Denmark, Norway	277	X'5A'
1143	Finland, Sweden	278	X'5A'
1144	Italy	280	X'9F'
1145	Spain, Latin America - Spanish	284	X'9F'
1146	UK	285	X'9F'
1147	France	297	X'9F'
1148	Belgium, Canada, Switzerland	500	X'9F'
1149	Iceland	871	X'9F'

"Example: multiple currency signs" on page 68

Example: multiple currency signs

The following example shows how you can display values in both euro currency (as EUR) and Swiss francs (as CHF).

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EuroSamp.
Environment Division.
Configuration Section.
Special-Names.
    Currency Sign is "CHF "  with Picture Symbol "F"
    Currency Sign is "EUR "  with Picture Symbol "U".
Data Division.
Working-Storage Section.
01  Deposit-in-Euro       Pic S9999V99 Value 8000.00.
01  Deposit-in-CHF        Pic S99999V99.
01  Deposit-Report.
    02  Report-in-Franc    Pic -FFFFF9.99.
    02  Report-in-Euro     Pic -UUUUU9.99.
01  EUR-to-CHF-Conv-Rate   Pic 9V99999  Value 1.53893.
. . .
PROCEDURE DIVISION.
Report-Deposit-in-CHF-and-EUR.
    Move Deposit-in-Euro to Report-in-Euro
    Compute Deposit-in-CHF Rounded
          = Deposit-in-Euro * EUR-to-CHF-Conv-Rate
      On Size Error
        Perform Conversion-Error
      Not On Size Error
        Move Deposit-in-CHF to Report-in-Franc
        Display "Deposit in euro  = " Report-in-Euro
        Display "Deposit in franc = " Report-in-Franc
    End-Compute
    Goback.
Conversion-Error.
           Display "Conversion error from EUR to CHF"
           Display "Euro value: " Report-in-Euro.
```

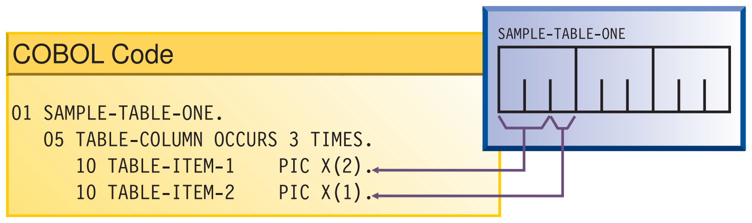
The above example produces the following display output:

```
Deposit in euro  =  EUR 8000.00
Deposit in franc = CHF 12311.44
```

The exchange rate used in this example is for illustrative purposes only.

Chapter 4. Handling tables

A *table* is a collection of data items that have the same description, such as account totals or monthly averages; it consists of a table name and subordinate items called *table elements*. A table is the COBOL equivalent of an array.



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

In the example above, 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.

Rather than defining repetitious items as separate, consecutive entries in the DATA DIVISION, you use the OCCURS clause in the DATA DIVISION entry to define a table. This practice has these advantages:

- The code clearly shows the unity of the items (the table elements).
- You can use subscripts and indexes to refer to the table elements.
- You can easily repeat data items.

Tables are important for increasing the speed of a program, especially one that looks up records.

RELATED TASKS
"Nesting tables" on page 71
"Defining a table (OCCURS)"
"Referring to an item in a table" on page 72
"Putting values into a table" on page 75
"Creating variable-length tables (DEPENDING ON)" on page 80
"Searching a table" on page 83
"Processing table items using intrinsic functions" on page 86
"Handling tables efficiently" on page 627

Defining a table (OCCURS)

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

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

In the example above, table-name is the name of an alphanumeric group item. 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.

If a table is to contain only Unicode (UTF-16) data, and you want the group item that contains the table to behave like an elementary category national item in most operations, code the GROUP-USAGE NATIONAL clause for the group item:

```
01  table-nameN Group-Usage National.
    05  element-nameN OCCURS m TIMES.
        10  elementN1  Pic nn.
        10  elementN2  Pic S99 Sign Is Leading, Separate.
        . . .
```

Any elementary item that is subordinate to a national group must be explicitly or implicitly described as USAGE NATIONAL, and any subordinate numeric data item that is signed must be implicitly or explicitly described with the SIGN IS SEPARATE clause.

To create tables of two to seven dimensions, use nested OCCURS clauses.

To create a variable-length table, code the DEPENDING ON phrase of the OCCURS clause.

To specify that table elements will be arranged in ascending or descending order based on the values in one or more key fields of the table, code the ASCENDING or DESCENDING KEY phrases of the OCCURS clause, or both. Specify the names of the keys in decreasing order of significance. Keys can be of class alphabetic, alphanumeric, DBCS, national, or numeric. (If it has USAGE NATIONAL, a key can be of category national, or can be a national-edited, numeric-edited, national decimal, or national floating-point item.)

You must code the ASCENDING or DESCENDING KEY phrase of the OCCURS clause to do a binary search (SEARCH ALL) of a table.

"Example: binary search" on page 85

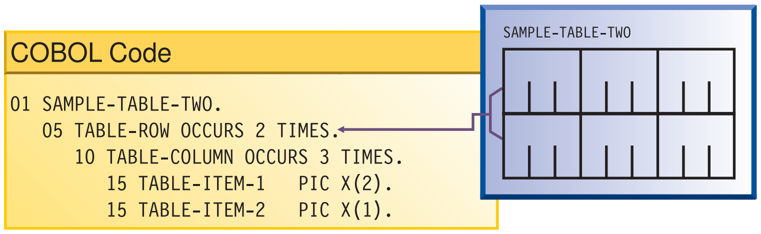
RELATED CONCEPTS
"National groups" on page 129

RELATED TASKS
"Nesting tables" on page 71
"Referring to an item in a table" on page 72
"Putting values into a table" on page 75
"Creating variable-length tables (DEPENDING ON)" on page 80
"Using national groups" on page 130
"Doing a binary search (SEARCH ALL)" on page 85
"Defining numeric data" on page 45

RELATED REFERENCES
OCCURS clause (*Enterprise COBOL Language Reference*)
SIGN clause (*Enterprise COBOL Language Reference*)
ASCENDING KEY and DESCENDING KEY phrases (*Enterprise COBOL Language Reference*)

Nesting tables

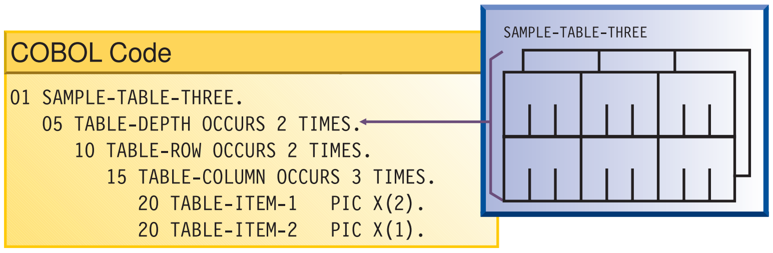
To create a two-dimensional table, define a one-dimensional table in each occurrence of another one-dimensional table.



```
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).
```

For example, in SAMPLE-TABLE-TWO above, 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.

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:



```
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).
```

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.

In a two-dimensional table, the two subscripts correspond to the row and column numbers. In a three-dimensional table, the three subscripts correspond to the depth, row, and column numbers.

"Example: subscripting" on page 72
"Example: indexing" on page 72

RELATED TASKS
"Defining a table (OCCURS)" on page 69
"Referring to an item in a table" on page 72
"Putting values into a table" on page 75
"Creating variable-length tables (DEPENDING ON)" on page 80
"Searching a table" on page 83
"Processing table items using intrinsic functions" on page 86
"Handling tables efficiently" on page 627

RELATED REFERENCES
OCCURS clause (*Enterprise COBOL Language Reference*)

Example: subscripting

The following example shows valid references to SAMPLE-TABLE-THREE that use literal subscripts. The spaces are required in the second example.

```
TABLE-COLUMN (2, 2, 1)
TABLE-COLUMN (2 2 1)
```

In either table reference, 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.

The following reference to SAMPLE-TABLE-TWO uses variable subscripts. The reference is valid if SUB1 and SUB2 are data-names that contain positive integer values within the range of the table.

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

RELATED TASKS
"Subscripting" on page 73

Example: indexing

The following example shows how displacements to elements that are referenced with indexes are calculated.

Consider the following three-dimensional table, SAMPLE-TABLE-FOUR:

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

Suppose you code the following relative indexing reference to SAMPLE-TABLE-FOUR:

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

This reference causes the following computation of the displacement to the TABLE-COLUMN element:

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

This calculation is based on the following element lengths:

- Each occurrence of TABLE-DEPTH is 256 bytes in length (4 * 8 * 8).
- Each occurrence of TABLE-ROW is 64 bytes in length (8 * 8).
- Each occurrence of TABLE-COLUMN is 8 bytes in length.

RELATED TASKS
"Indexing" on page 74

Referring to an item in a table

A table element has a collective name, but the individual items within it do not have unique data-names.

To refer to an item, you have a choice of three techniques:

- Use the data-name of the table element, along with its occurrence number (called a *subscript*) in parentheses. This technique is called *subscripting*.

- Use the data-name of the table element, along with a value (called an *index*) that is added to the address of the table to locate an item (as a displacement from the beginning of the table). This technique is called *indexing*, or subscripting using index-names.
- Use both subscripts and indexes together.

RELATED TASKS
"Subscripting"
"Indexing" on page 74

Subscripting

The lowest possible subscript value is 1, which references the first occurrence of a table element. In a one-dimensional table, the subscript corresponds to the row number.

You can use a literal or a data-name as a subscript. If a data item that has a literal subscript is of fixed length, the compiler resolves the location of the data item.

When you use a data-name as a variable subscript, you must describe the data-name as an elementary numeric integer. The most efficient format is COMPUTATIONAL (COMP) with a PICTURE size that is smaller than five digits. You cannot use a subscript with a data-name that is used as a subscript. The code generated for the application resolves the location of a variable subscript at run time.

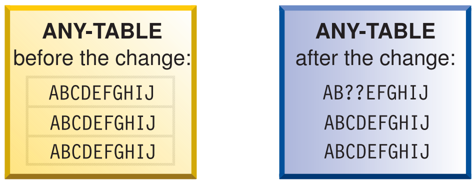
You can increment or decrement a literal or variable subscript by a specified integer amount. For example:

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

You can change part of a table element rather than the whole element. To do so, refer to the character position and length of the substring to be changed. For example:

```
01  ANY-TABLE.
    05 TABLE-ELEMENT     PIC X(10)
        OCCURS 3 TIMES   VALUE "ABCDEFGHIJ".
. . .
    MOVE "??" TO TABLE-ELEMENT (1) (3 : 2).
```

The MOVE statement in the example above moves the string '??' into table element number 1, beginning at character position 3, for a length of 2 characters.



"Example: subscripting" on page 72

RELATED TASKS
"Indexing" on page 74
"Putting values into a table" on page 75
"Searching a table" on page 83
"Handling tables efficiently" on page 627

Indexing

You create an index by using the INDEXED BY phrase of the OCCURS clause to identify an index-name.

For example, INX-A in the following code is an index-name:

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

The compiler calculates the value contained in the index as the occurrence number (subscript) minus 1, multiplied by the length of the table element. Therefore, for the fifth occurrence of TABLE-ITEM, the binary value contained in INX-A is (5 - 1) * 8, or 32.

You can use an index-name to reference another table only if both table descriptions have the same number of table elements, and the table elements are of the same length.

You can use the USAGE IS INDEX clause to create an index data item, and can use an index data item with any table. For example, INX-B in the following code is an index data item:

```
77  INX-B  USAGE IS INDEX.
. . .
    SET INX-A TO 10
    SET INX-B TO INX-A.
    PERFORM VARYING INX-A FROM 1 BY 1 UNTIL INX-A > INX-B
        DISPLAY TABLE-ITEM (INX-A)
        . . .
    END-PERFORM.
```

The index-name INX-A is used to traverse table TABLE-ITEM above. The index data item INX-B is used to hold the index of the last element of the table. The advantage of this type of coding is that calculation of offsets of table elements is minimized, and no conversion is necessary for the UNTIL condition.

You can use the SET statement to assign to an index data item the value that you stored in an index-name, as in the statement SET INX-B TO INX-A above. For example, when you load records into a variable-length table, you can store the index value of the last record into a data item defined as USAGE IS INDEX. Then you can test for the end of the table by comparing the current index value with the index value of the last record. This technique is useful when you look through or process a table.

You can increment or decrement an index-name by an elementary integer data item or a nonzero integer literal, for example:

```
SET INX-A DOWN BY 3
```

The integer represents a number of occurrences. It is converted to an index value before being added to or subtracted from the index.

Initialize the index-name by using a SET, PERFORM VARYING, or SEARCH ALL statement. You can then use the index-name in SEARCH or relational condition statements. To change the value, use a PERFORM, SEARCH, or SET statement.

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

"Example: indexing" on page 72

RELATED TASKS
"Subscripting" on page 73
"Putting values into a table"
"Searching a table" on page 83
"Processing table items using intrinsic functions" on page 86
"Handling tables efficiently" on page 627

RELATED REFERENCES
INDEXED BY phrase (*Enterprise COBOL Language Reference*)
INDEX phrase (*Enterprise COBOL Language Reference*)
SET statement (*Enterprise COBOL Language Reference*)

Putting values into a table

You can put values into a table by loading the table dynamically, initializing the table with the `INITIALIZE` statement, or assigning values with the `VALUE` clause when you define the table.

RELATED TASKS
"Loading a table dynamically"
"Loading a variable-length table" on page 82
"Initializing a table (INITIALIZE)"
"Assigning values when you define a table (VALUE)" on page 77
"Assigning values to a variable-length table" on page 83

Loading a table dynamically

If the initial values of a table are different with each execution of your program, you can define the table without initial values. You can instead read the changed values into the table dynamically before the program refers to the table.

To load a table, use the `PERFORM` statement and 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 (rather than a literal) for the maximum item count. Then, if you make the table bigger, you need to change only one value instead of all references to a literal.

"Example: PERFORM and subscripting" on page 78
"Example: PERFORM and indexing" on page 79

RELATED REFERENCES
PERFORM with VARYING phrase (*Enterprise COBOL Language Reference*)

Initializing a table (INITIALIZE)

| You can load a table by coding one or more `INITIALIZE` statements.

| For example, to move the value 3 into each of the elementary numeric data items
| in a table called `TABLE-ONE`, shown below, you can code the following statement:

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

| To move the character 'X' into each of the elementary alphanumeric data items in
| `TABLE-ONE`, you can code the following statement:

```
| INITIALIZE TABLE-ONE REPLACING ALPHANUMERIC DATA BY "X".
```

When you use the INITIALIZE statement to initialize a table, the table is processed as a group item (that is, with group semantics); elementary data items within the group are recognized and processed. For example, suppose that TABLE-ONE is an alphanumeric group that is defined like this:

```
01  TABLE-ONE.
  02  Trans-out  Occurs 20.
      05  Trans-code       Pic X    Value "R".
      05  Part-number      Pic XX   Value "13".
      05  Trans-quan       Pic 99   Value 10.
      05  Price-fields.
          10  Unit-price   Pic 99V  Value 50.
          10  Discount     Pic 99V  Value 25.
          10  Sales-Price  Pic 999  Value 375.
      . . .
      Initialize TABLE-ONE Replacing Numeric Data By 3
                                Alphanumeric Data By "X"
```

The table below shows the content that each of the twenty 12-byte elements Trans-out(*n*) has before execution and after execution of the INITIALIZE statement shown above:

Trans-out(*n*) before	**Trans-out(*n*) after**
R13105025375	XX*b*030303003[1]

1. The symbol *b* represents a blank space.

You can similarly use an INITIALIZE statement to load a table that is defined as a national group. For example, if TABLE-ONE shown above specified the GROUP-USAGE NATIONAL clause, and Trans-code and Part-number had N instead of X in their PICTURE clauses, the following statement would have the same effect as the INITIALIZE statement above, except that the data in TABLE-ONE would instead be encoded in UTF-16:

```
Initialize TABLE-ONE Replacing Numeric  Data By 3
                               National Data By N"X"
```

The REPLACING NUMERIC phrase initializes floating-point data items also.

You can use the REPLACING phrase of the INITIALIZE statement similarly to initialize all of the elementary ALPHABETIC, DBCS, ALPHANUMERIC-EDITED, NATIONAL-EDITED, and NUMERIC-EDITED data items in a table.

The INITIALIZE statement cannot assign values to a variable-length table (that is, a table that was defined using the OCCURS DEPENDING ON clause).

"Examples: initializing data items" on page 30

RELATED TASKS
"Initializing a structure (INITIALIZE)" on page 32
"Assigning values when you define a table (VALUE)" on page 77
"Assigning values to a variable-length table" on page 83
"Looping through a table" on page 99
"Using data items and group items" on page 26
"Using national groups" on page 130

RELATED REFERENCES
INITIALIZE statement (*Enterprise COBOL Language Reference*)

Assigning values when you define a table (VALUE)

If a table is to contain stable values (such as days and months), you can set the specific values when you define the table.

Set static values in tables in one of these ways:

- Initialize each table item individually.
- | Initialize an entire table at the group level.
- Initialize all occurrences of a given table element to the same value.

RELATED TASKS
"Initializing each table item individually"
"Initializing a table at the group level" on page 78
"Initializing all occurrences of a given table element" on page 78
"Initializing a structure (INITIALIZE)" on page 32

Initializing each table item individually

If a table is small, you can set the value of each item individually by using a VALUE clause.

Use the following technique, which is shown in the example code below:

1. Declare a record (such as Error-Flag-Table below) that contains the items that are to be in the table.
2. Set the initial value of each item in a VALUE clause.
3. Code a REDEFINES entry to make the record into a table.

```
***********************************************************
***           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.
 01  Filler Redefines Error-Flag-Table.
     05 Error-Flag Occurs 8 Times
         Indexed By Flag-Index          Pic X.
```

| In the example above, the VALUE clause at the 01 level initializes each of the table
| items to the same value. Each table item could instead be described with its own
| VALUE clause to initialize that item to a distinct value.

To initialize larger tables, use MOVE, PERFORM, or INITIALIZE statements.

RELATED TASKS
"Initializing a structure (INITIALIZE)" on page 32
"Assigning values to a variable-length table" on page 83

RELATED REFERENCES
REDEFINES clause (*Enterprise COBOL Language Reference*)
OCCURS clause (*Enterprise COBOL Language Reference*)

Initializing a table at the group level

| Code an alphanumeric or national group data item and assign to it, through the VALUE clause, the contents of the whole table. Then, in a subordinate data item, use an OCCURS clause to define the individual table items.

| In the following example, the alphanumeric group data item TABLE-ONE uses a
| VALUE clause that initializes each of the four elements of TABLE-TWO:

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

| In the following example, the national group data item Table-OneN uses a VALUE
| clause that initializes each of the three elements of the subordinate data item
| Table-TwoN (each of which is implicitly USAGE NATIONAL). Note that you can
| initialize a national group data item with a VALUE clause that uses an alphanumeric
| literal, as shown below, or a national literal.

```
| 01  Table-OneN  Group-Usage National  Value "AB12CD34EF56".
|     05  Table-TwoN   Occurs 3 Times   Indexed By MyI.
|         10  ElementOneN  Pic nn.
|         10  ElementTwoN  Pic 99.
```

| After Table-OneN is initialized, ElementOneN(1) contains NX"00410042" (the UTF-16
| representation of 'AB'), the national decimal item ElementTwoN(1) contains
| NX"00310032" (the UTF-16 representation of '12'), and so forth.

RELATED REFERENCES
OCCURS clause (*Enterprise COBOL Language Reference*)
GROUP-USAGE clause (*Enterprise COBOL Language Reference*)

Initializing all occurrences of a given table element

You can use the VALUE clause in the data description of a table element to initialize all instances of that element to the specified value.

```
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".
```

For example, the code above 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.

RELATED TASKS
"Assigning values to a variable-length table" on page 83

RELATED REFERENCES
OCCURS clause (*Enterprise COBOL Language Reference*)

Example: PERFORM and subscripting

This example traverses an error-flag table using subscripting until an error code that has been set is found. If an error code is found, the corresponding error message is moved to a print report field.

```
***********************************************************
***          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.
01  Filler Redefines Error-Flag-Table.
    05 Error-Flag Occurs 8 Times
          Indexed By Flag-Index         Pic X.
77  Error-on                            Pic X  Value "E".
***********************************************************
***        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.".
    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".
01  Filler Redefines Error-Message-Table.
    05  Error-Message Occurs 8 Times
           Indexed By Message-Index      Pic X(25).
. . .
PROCEDURE DIVISION.
    . . .
    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
    . . .
```

Example: PERFORM and indexing

This example traverses an error-flag table using indexing until an error code that has been set is found. If an error code is found, the corresponding error message is moved to a print report field.

```
***********************************************************
***          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.
01  Filler Redefines Error-Flag-Table.
    05 Error-Flag Occurs 8 Times
         Indexed By Flag-Index         Pic X.
77  Error-on                           Pic X  Value "E".
**********************************************************
***         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.".
    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".
01  Filler Redefines Error-Message-Table.
    05  Error-Message Occurs 8 Times
           Indexed By Message-Index      Pic X(25).
. . .
PROCEDURE DIVISION.
    . . .
    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
    . . .
```

Creating variable-length tables (DEPENDING ON)

If you do not know before run time how many times a table element occurs, define a variable-length table. To do so, use the OCCURS DEPENDING ON (ODO) clause.

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

In the example above, X is called the *ODO subject*, and Y is called the *ODO object*.

Two factors affect the successful manipulation of variable-length records:

- Correct calculation of record lengths

  The length of the variable portions of a group item is the product of the object of the DEPENDING ON phrase and the length of the subject of the OCCURS clause.
- Conformance of the data in the object of the OCCURS DEPENDING ON clause to its PICTURE clause

  If the content of the ODO object does not match its PICTURE clause, the program could terminate abnormally. You must ensure that the ODO object correctly specifies the current number of occurrences of table elements.

The following example shows a group item (REC-1) that contains both the subject and object of the OCCURS DEPENDING ON clause. The way the length of the group item is determined depends on whether it is sending or receiving data.

```
WORKING-STORAGE SECTION.
01  MAIN-AREA.
    03 REC-1.
        05 FIELD-1                     PIC 9.
        05 FIELD-2 OCCURS 1 TO 5 TIMES
           DEPENDING ON FIELD-1        PIC X(05).
01  REC-2.
    03 REC-2-DATA                      PIC X(50).
```

If you want to move REC-1 (the sending item in this case) to REC-2, the length of REC-1 is determined immediately before the move, using the current value in FIELD-1. If the content of FIELD-1 conforms to its PICTURE clause (that is, if FIELD-1 contains a zoned decimal item), the move can proceed based on the actual length of REC-1. Otherwise, the result is unpredictable. You must ensure that the ODO object has the correct value before you initiate the move.

When you do a move to REC-1 (the receiving item in this case), the length of REC-1 is determined using the maximum number of occurrences. In this example, five occurrences of FIELD-2, plus FIELD-1, yields a length of 26 bytes. In this case, you do not need to set the ODO object (FIELD-1) before referencing REC-1 as a receiving item. However, the sending field's ODO object (not shown) must 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.

However, if you do a move to REC-1 (again the receiving item) where REC-1 is followed by a variably located group (a type of *complex ODO*), the actual length of REC-1 is calculated immediately before the move, using the current value of the ODO object (FIELD-1). In the following example, REC-1 and REC-2 are in the same record, but REC-2 is not subordinate to REC-1 and is therefore variably located:

```
01  MAIN-AREA
    03 REC-1.
       05 FIELD-1                     PIC 9.
       05 FIELD-3                     PIC 9.
       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).
```

The compiler issues a message that lets you know that the actual length was used. This case requires that you set the value of the ODO object before using the group item as a receiving field.

The following example shows how to define a variable-length table when the ODO object (LOCATION-TABLE-LENGTH below) is outside the group:

```
 DATA DIVISION.
 FILE SECTION.
 FD  LOCATION-FILE
     RECORDING MODE F
     BLOCK 0 RECORDS
     RECORD 80 CHARACTERS
     LABEL RECORD STANDARD.
 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).
```

RELATED CONCEPTS
Appendix B, “Complex OCCURS DEPENDING ON,” on page 657

RELATED TASKS
“Assigning values to a variable-length table” on page 83
“Loading a variable-length table”
“Preventing overlay when adding elements to a variable table” on page 659
“Finding the length of data items” on page 118
Enterprise COBOL Compiler and Runtime Migration Guide

RELATED REFERENCES
OCCURS DEPENDING ON clause (*Enterprise COBOL Language Reference*)

Loading a variable-length table

You can use a *do-until* structure (a TEST AFTER loop) to control the loading of a variable-length table. For example, after the following code runs, LOCATION-TABLE-LENGTH contains the subscript of the last item in the table.

```
 DATA DIVISION.
 FILE SECTION.
 FD  LOCATION-FILE
     RECORDING MODE F
     BLOCK 0 RECORDS
     RECORD 80 CHARACTERS
     LABEL RECORD STANDARD.
 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 "YES".
 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).
 . . .
 PROCEDURE DIVISION.
     . . .
     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
```

Assigning values to a variable-length table

You can code a VALUE clause for an alphanumeric or national group item that has a subordinate data item that contains the OCCURS clause with the DEPENDING ON phrase. Each subordinate structure that contains the DEPENDING ON phrase is initialized using the maximum number of occurrences.

If you define the entire table by using the DEPENDING ON phrase, all the elements are initialized using the maximum defined value of the ODO (OCCURS DEPENDING ON) object.

If the ODO object is initialized by a VALUE clause, it is logically initialized after the ODO subject has been initialized.

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

For example, in the code above, the ODO subject Y(1) is initialized to 'A', Y(2) to 'B', . . ., Y(5) to 'E', and finally the ODO object X is initialized to 3. Any subsequent reference to TABLE-THREE (such as in a DISPLAY statement) refers to X and the first three elements, Y(1) through Y(3), of the table.

RELATED TASKS
"Assigning values when you define a table (VALUE)" on page 77

RELATED REFERENCES
OCCURS DEPENDING ON clause (*Enterprise COBOL Language Reference*)

Searching a table

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

To do serial searches, use SEARCH and indexing. For variable-length tables, you can use PERFORM with subscripting or indexing.

To do binary searches, use SEARCH ALL and indexing.

A binary search can be considerably more efficient than a serial search. For a serial search, the number of comparisons is of the order of *n*, the number of entries in the table. For a binary search, the number of comparisons is of the order of only the logarithm (base 2) of *n*. A binary search, however, requires that the table items already be sorted.

RELATED TASKS
"Doing a serial search (SEARCH)" on page 84
"Doing a binary search (SEARCH ALL)" on page 85

Doing a serial search (SEARCH)

Use the SEARCH statement to do a serial (sequential) search beginning at the current index setting. To modify the index setting, use the SET statement.

The conditions in the WHEN phrase are evaluated in the order in which they appear:

- 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 did not code AT END, control passes to the next statement in the program.

You can reference only one level of a table (a table element) with each SEARCH statement. To search multiple levels of a table, use nested SEARCH statements. Delimit each nested SEARCH statement with END-SEARCH.

Performance: If the found condition comes after some intermediate point in the table, 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 of the table also enables more efficient serial searching. If the table is large and is presorted, a binary search is more efficient.

"Example: serial search"

RELATED REFERENCES
SEARCH statement (*Enterprise COBOL Language Reference*)

Example: serial search

The following example shows how you might find a particular string in the innermost table of a three-dimensional table.

Each dimension of the table has its own index (set to 1, 4, and 1, respectively). The innermost table (TABLE-ENTRY3) has an ascending key.

```
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 item
          that equals "A1234AAAAAAAA00"
RETURN-CODE = 0
```

Doing a binary search (SEARCH ALL)

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

| To use the SEARCH ALL statement to search a table, the table must specify the
| ASCENDING or DESCENDING KEY phrases of the OCCURS clause, or both, and must
| already be ordered on the key or keys that are specified in the ASCENDING and
| DESCENDING KEY phrases.

| In the WHEN phrase of the SEARCH ALL statement, you can test any key that is named
| in the ASCENDING or DESCENDING KEY phrases for the table, but you must test all
| preceding keys, if any. The test must be an equal-to condition, and the WHEN phrase
| must specify either a key (subscripted by the first index-name associated with the
| table) or a condition-name that is associated with the key. The WHEN condition can
be a compound condition that is formed from simple conditions that use AND as the only logical connective.

| Each key and its object of comparison must be compatible according to the rules
for comparison of data items. Note though that if a key is compared to a national
| literal or identifier, the key must be a national data item.

"Example: binary search"

RELATED TASKS
"Defining a table (OCCURS)" on page 69

RELATED REFERENCES
SEARCH statement (*Enterprise COBOL Language Reference*)
General relation conditions (*Enterprise COBOL Language Reference*)

Example: binary search

The following example shows how you can code a binary search of a table.

Suppose you define a table that contains 90 elements of 40 bytes each, and three 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:

```
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).
```

You can search this table by using the following statements:

```
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
```

If an entry is found in which each of the three keys is equal to the value to which it is compared (VALUE-1, VALUE-2, and VALUE-3, respectively), PART-1 of that entry is moved to OUTPUT-AREA. If no matching key is found in the entries in TABLE-A, the NOENTRY routine is performed.

Processing table items using intrinsic functions

| You can use intrinsic functions to process alphabetic, alphanumeric, national, or
| numeric table items. (You can process DBCS data items only with the NATIONAL-OF
| intrinsic function.) The data descriptions of the table items must be compatible with the requirements for the function arguments.

Use a subscript or index to reference an individual data item as a function argument. For example, assuming that Table-One is a 3 x 3 array of numeric items, you can find the square root of the middle element by using this statement:

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

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

You can mix scalars and array arguments for functions that accept multiple arguments:

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

"Example: processing tables using intrinsic functions"

RELATED TASKS
"Using intrinsic functions (built-in functions)" on page 40
"Converting data items (intrinsic functions)" on page 112
"Evaluating data items (intrinsic functions)" on page 115

RELATED REFERENCES
Intrinsic functions (*Enterprise COBOL Language Reference*)

Example: processing tables using intrinsic functions

These examples show how you can apply an intrinsic function to some or all of the elements in a table by using the ALL subscript.

Assuming that `Table-Two` is a 2 x 3 x 2 array, the following statement adds the values in elements `Table-Two(1,3,1)`, `Table-Two(1,3,2)`, `Table-Two(2,3,1)`, and `Table-Two(2,3,2)`:

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

The following example computes various salary values for all the employees whose salaries are encoded in `Employee-Table`:

```
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))
```

Chapter 5. Selecting and repeating program actions

Use COBOL control language to choose program actions based on the outcome of logical tests, to iterate over selected parts of your program and data, and to identify statements to be performed as a group.

These controls include the IF, EVALUATE, and PERFORM statements, and the use of switches and flags.

RELATED TASKS
"Selecting program actions"
"Repeating program actions" on page 97

Selecting program actions

You can provide for different program actions depending on the tested value of one or more data items.

The IF and EVALUATE statements in COBOL test one or more data items by means of a conditional expression.

RELATED TASKS
"Coding a choice of actions"
"Coding conditional expressions" on page 94

RELATED REFERENCES
IF statement (*Enterprise COBOL Language Reference*)
EVALUATE statement (*Enterprise COBOL Language Reference*)

Coding a choice of actions

Use IF . . . ELSE to code a choice between two processing actions. (The word THEN is optional.) Use the EVALUATE statement to code a choice among three or more possible actions.

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

When one of two processing choices is no action, code the IF statement with or without ELSE. Because the ELSE clause is optional, you can code the IF statement as follows:

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

Such coding is suitable for simple cases. For complex logic, you probably need to use the ELSE clause. For example, suppose you have nested IF statements in which there is an action for only one of the processing choices. You could 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
```

The `EVALUATE` statement is an expanded form of the `IF` statement that allows you to avoid nesting `IF` statements, a common source of logic errors and debugging problems.

RELATED TASKS
"Using nested IF statements"
"Using the EVALUATE statement" on page 91
"Coding conditional expressions" on page 94

Using nested IF statements

When an `IF` statement contains an `IF` statement as one of its possible branches, the `IF` statements are said to be *nested*. Theoretically, there is no limit to the depth of nested `IF` statements.

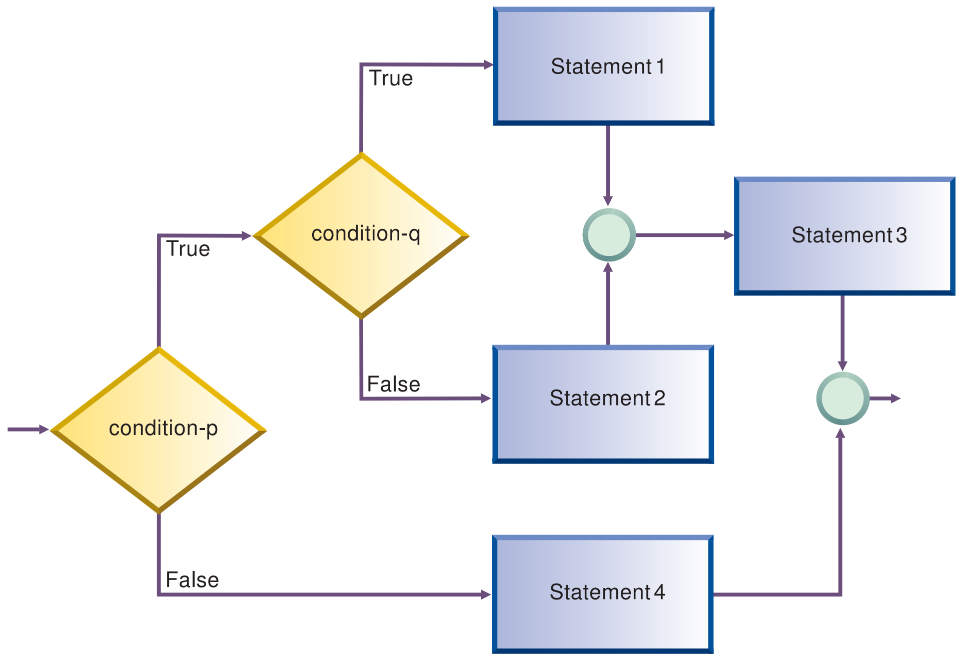
However, use nested `IF` statements sparingly. The logic can be difficult to follow, although explicit scope terminators and indentation help. When a program has to test a variable for more than two values, `EVALUATE` is probably a better choice.

The following pseudocode depicts a nested `IF` statement:

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

In the pseudocode above, an `IF` statement and a sequential structure are nested in one branch of the outer `IF`. In this structure, the `END-IF` that closes the nested `IF` is very important. Use `END-IF` instead of a period, because a period would end the outer `IF` structure also.

The following figure shows the logic structure of the pseudocode above.



RELATED TASKS
"Coding a choice of actions" on page 89

RELATED REFERENCES
Explicit scope terminators (*Enterprise COBOL Language Reference*)

Using the EVALUATE statement

You can use the EVALUATE statement instead of a series of nested IF statements to test several conditions and specify a different action for each. Thus you can use the EVALUATE statement to implement a *case structure* or decision table.

You can also use the EVALUATE statement to cause multiple conditions to lead to the same processing, as shown in these examples:

"Example: EVALUATE using THRU phrase" on page 92
"Example: EVALUATE using multiple WHEN phrases" on page 92

In an EVALUATE statement, the operands before the WHEN phrase are referred to as *selection subjects*, and the operands in the WHEN phrase are called the *selection objects*. Selection subjects can be identifiers, literals, conditional expressions, or the word TRUE or FALSE. Selection objects can be identifiers, literals, conditional or arithmetic expressions, or the word TRUE, FALSE, or ANY.

You can separate multiple selection subjects with the ALSO phrase. You can separate multiple selection objects with the ALSO phrase. The number of selection objects within each set of selection objects must be equal to the number of selection subjects, as shown in this example:

"Example: EVALUATE testing several conditions" on page 93

Identifiers, literals, or arithmetic expressions that appear within a selection object must be valid operands for comparison to the corresponding operand in the set of selection subjects. Conditions or the word TRUE or FALSE that appear in a selection

| object must correspond to a conditional expression or the word TRUE or FALSE in
| the set of selection subjects. (You can use the word ANY as a selection object to
| correspond to any type of selection subject.)

The execution of the EVALUATE statement ends when one of the following conditions occurs:

- The statements associated with the selected WHEN phrase are performed.
- The statements associated with the WHEN OTHER phrase are performed.
- No WHEN conditions are satisfied.

WHEN phrases are tested in the order that they appear in the source program. Therefore, you should order these phrases for the best performance. First code the WHEN phrase that contains selection objects that are most likely to be satisfied, then the next most likely, and so on. An exception is the WHEN OTHER phrase, which must come last.

RELATED TASKS
"Coding a choice of actions" on page 89

RELATED REFERENCES
EVALUATE statement (*Enterprise COBOL Language Reference*)
General relation conditions (*Enterprise COBOL Language Reference*)

Example: EVALUATE using THRU phrase: This example shows how you can code several conditions in a range of values to lead to the same processing action
| by coding the THRU phrase. Operands in a THRU phrase must be of the same class.

In this example, CARPOOL-SIZE is the *selection subject*; 1, 2, and 3 THRU 6 are the *selection objects*:

```
EVALUATE CARPOOL-SIZE
  WHEN 1
    MOVE "SINGLE" TO PRINT-CARPOOL-STATUS
  WHEN 2
    MOVE "COUPLE" TO PRINT-CARPOOL-STATUS
  WHEN 3 THRU 6
    MOVE "SMALL GROUP" TO PRINT-CARPOOL STATUS
  WHEN OTHER
    MOVE "BIG GROUP" TO PRINT-CARPOOL STATUS
END-EVALUATE
```

The following nested IF statements represent the same logic:

```
IF CARPOOL-SIZE = 1 THEN
  MOVE "SINGLE" TO PRINT-CARPOOL-STATUS
ELSE
  IF CARPOOL-SIZE = 2 THEN
    MOVE "COUPLE" TO PRINT-CARPOOL-STATUS
  ELSE
    IF CARPOOL-SIZE >= 3 and CARPOOL-SIZE <= 6 THEN
      MOVE "SMALL GROUP" TO PRINT-CARPOOL-STATUS
    ELSE
      MOVE "BIG GROUP" TO PRINT-CARPOOL-STATUS
    END-IF
  END-IF
END-IF
```

Example: EVALUATE using multiple WHEN phrases: The following example shows that you can code multiple WHEN phrases if several conditions should lead to the same action. Doing so gives you more flexibility than using only the THRU phrase, because the conditions do not have to evaluate to values in a range or have the same class.

```
EVALUATE MARITAL-CODE
  WHEN "M"
    ADD 2 TO PEOPLE-COUNT
  WHEN "S"
  WHEN "D"
  WHEN "W"
    ADD 1 TO PEOPLE-COUNT
END-EVALUATE
```

The following nested IF statements represent the same logic:

```
IF MARITAL-CODE = "M" THEN
  ADD 2 TO PEOPLE-COUNT
ELSE
  IF MARITAL-CODE = "S" OR
     MARITAL-CODE = "D" OR
     MARITAL-CODE = "W" THEN
       ADD 1 TO PEOPLE-COUNT
  END-IF
END-IF
```

Example: EVALUATE testing several conditions: This example shows the use of the ALSO phrase to separate two selection subjects (True ALSO True) and to separate the two corresponding selection objects within each set of selection objects (for example, When A + B < 10 Also C = 10).

Both selection objects in a WHEN phrase must satisfy the TRUE, TRUE condition before the associated action is performed. If both objects do not evaluate to TRUE, the next WHEN phrase is processed.

```
Identification Division.
  Program-ID. MiniEval.
Environment Division.
  Configuration Section.
  Source-Computer. IBM-390.
Data Division.
  Working-Storage Section.
  01   Age            Pic  999.
  01   Sex            Pic  X.
  01   Description    Pic  X(15).
  01   A              Pic  999.
  01   B              Pic  9999.
  01   C              Pic  9999.
  01   D              Pic  9999.
  01   E              Pic  99999.
  01   F              Pic  999999.
Procedure Division.
  PN01.
    Evaluate True Also True
      When Age < 13 Also Sex = "M"
        Move "Young Boy" To Description
      When Age < 13 Also Sex = "F"
        Move "Young Girl" To Description
      When Age > 12 And Age < 20 Also Sex = "M"
        Move "Teenage Boy" To Description
      When Age > 12 And Age < 20 Also Sex = "F"
        Move "Teenage Girl" To Description
      When Age > 19 Also Sex = "M"
        Move "Adult Man" To Description
      When Age > 19 Also Sex = "F"
        Move "Adult Woman" To Description
      When Other
        Move "Invalid Data" To Description
    End-Evaluate
    Evaluate True Also True
      When A + B < 10 Also C = 10
        Move "Case 1" To Description
      When A + B > 50 Also C = ( D + E ) / F
        Move "Case 2" To Description
      When Other
        Move "Case Other" To Description
    End-Evaluate
    Stop Run.
```

Coding conditional expressions

Using the `IF` and `EVALUATE` statements, you can code program actions that will be performed depending on the truth value of a conditional expression.

The following are some of the conditions that you can specify:

- Relation conditions, such as:
  - Numeric comparisons
  - Alphanumeric comparisons
  - DBCS comparisons
  - National comparisons
- Class conditions; for example, to test whether a data item:
  - `IS NUMERIC`
  - `IS ALPHABETIC`
  - `IS DBCS`
  - `IS KANJI`
  - `IS NOT KANJI`
- Condition-name conditions, to test the value of a conditional variable that you define
- Sign conditions, to test whether a numeric operand `IS POSITIVE`, `NEGATIVE`, or `ZERO`
- Switch-status conditions, to test the status of UPSI switches that you name in the `SPECIAL-NAMES` paragraph
- Complex conditions, such as:
  - Negated conditions; for example, `NOT (A IS EQUAL TO B)`
  - Combined conditions (conditions combined with logical operators `AND` or `OR`)

RELATED CONCEPTS
"Switches and flags" on page 95

RELATED TASKS
"Defining switches and flags" on page 95
"Resetting switches and flags" on page 96
"Checking for incompatible data (numeric class test)" on page 56
"Comparing national (UTF-16) data" on page 138
"Testing for valid DBCS characters" on page 142

RELATED REFERENCES
General relation conditions (*Enterprise COBOL Language Reference*)
Class condition (*Enterprise COBOL Language Reference*)
Rules for condition-name entries (*Enterprise COBOL Language Reference*)

Sign condition (*Enterprise COBOL Language Reference*)
Combined conditions (*Enterprise COBOL Language Reference*)

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. Control these two-way decisions by using level-88 items with meaningful names (*condition-names*) to act as switches.

Other program decisions depend on the particular value or range of values of a data item. When you use condition-names to give more than just on or off values to a field, the field is generally referred to as a *flag*.

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 value of that level-88 condition-name.

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

RELATED TASKS
"Defining switches and flags"
"Resetting switches and flags" on page 96

Defining switches and flags

In the `DATA DIVISION`, define level-88 items that will act as switches or flags, and give them meaningful names.

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

The reader can easily follow your code if you choose meaningful condition-names and if the values assigned to them have some association with logical values.

"Example: switches"
"Example: flags" on page 96

Example: switches

The following examples show how you can use level-88 items to test for various binary-valued (on-off) conditions in your program.

For example, to test for the end-of-file condition for an input file named Transaction-File, you can 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 that a condition named `Transaction-EOF` is turned on when `Transaction-EOF-Switch` has value 'y'. Referencing `Transaction-EOF` in the `PROCEDURE DIVISION` expresses the same condition as testing `Transaction-EOF-Switch = "y"`. For example, the following statement causes a report to be printed only if `Transaction-EOF-Switch` has been set to 'y':

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

Example: flags

The following examples show how you can use several level-88 items together with an EVALUATE statement to determine which of several conditions in a program is true.

Consider for example a program that updates a master file. The updates are read from a transaction file. The records in the file contain a field that indicates which of the three functions is to be performed: add, change, or delete. In the record description of the input file, 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 to determine which function is to be performed might look like this:

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

Resetting switches and flags

Throughout your program, you might need to reset switches or flags to the original values they had in their data descriptions. To do so, either use a SET statement or define a data item to move to the switch or flag.

When you use the SET *condition-name* TO TRUE statement, the switch or flag is set to the original value that it was assigned in its data description. For a level-88 item that has multiple values, SET *condition-name* TO TRUE assigns the first value (A in the example below):

```
88 Record-is-Active Value "A" "O" "S"
```

Using the SET statement and meaningful condition-names makes it easier for readers to follow your code.

"Example: set switch on"
"Example: set switch off" on page 97

Example: set switch on

The following examples show how you can set a switch on by coding a SET statement that moves the value TRUE to a level-88 item.

For example, the SET statement in the following example has the same effect as coding the statement Move "y" to Transaction-EOF-Switch:

```
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 to assign a value to 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
```

Example: set switch off

The following example shows how you can set a switch off by coding a MOVE statement that moves a value to a level-88 item.

For example, you can use a data item called SWITCH-OFF to set an on-off switch to off, as in the following code, which resets a switch to indicate that end-of-file has not been reached:

```
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
```

Repeating program actions

Use a PERFORM statement to repeat the same code (that is, loop) either a specified number of times or based on the outcome of a decision.

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

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

RELATED TASKS
"Choosing inline or out-of-line PERFORM" on page 98
"Coding a loop" on page 99
"Looping through a table" on page 99
"Executing multiple paragraphs or sections" on page 100

RELATED REFERENCES
PERFORM statement (*Enterprise COBOL Language Reference*)

Choosing inline or out-of-line PERFORM

An inline PERFORM is an imperative statement that is executed in the normal flow of a program; an out-of-line PERFORM entails a branch to a named paragraph and an implicit return from that paragraph.

To determine whether to code an inline or out-of-line PERFORM statement, answer the following questions:

- Is the PERFORM statement used in several places?

  Use an out-of-line PERFORM when you want to use the same portion of code in several places in your program.
- Which placement of the statement will be easier to read?

  If the code to be performed is short, an inline PERFORM can be easier to read. But if the code extends over several screens, the logical flow of the program might be clearer if you use an out-of-line PERFORM. (Each paragraph in structured programming should perform one logical function, however.)
- What are the efficiency tradeoffs?

  An inline PERFORM avoids the overhead of branching that occurs with an out-of-line PERFORM. But even out-of-line PERFORM coding can improve code optimization, so efficiency gains should not be overemphasized.

In the 1974 COBOL standard, the PERFORM statement is out-of-line and thus requires a branch to a separate paragraph and an implicit return. If the performed paragraph is in the subsequent sequential flow of your program, it is also executed in that logic flow. To avoid this additional execution, place the paragraph outside the normal sequential flow (for example, after the GOBACK) or code a branch around it.

The subject of an inline PERFORM is an imperative statement. Therefore, you must code statements (other than imperative statements) within an inline PERFORM with explicit scope terminators.

"Example: inline PERFORM statement"

Example: inline PERFORM statement

This example shows the structure of an inline PERFORM statement that has the required scope terminators and the required END-PERFORM phrase.

```
   Perform 100-Initialize-Paragraph
* The following statement is an inline PERFORM:
   Perform Until Transaction-EOF
      Read Update-Transaction-File Into WS-Transaction-Record
         At End
            Set Transaction-EOF To True
         Not At End
            Perform 200-Edit-Update-Transaction
            If No-Errors
               Perform 300-Update-Commuter-Record
            Else
               Perform 400-Print-Transaction-Errors
* End-If is a required scope terminator
            End-If
            Perform 410-Re-Initialize-Fields
* End-Read is a required scope terminator
      End-Read
   End-Perform
```

Coding a loop

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

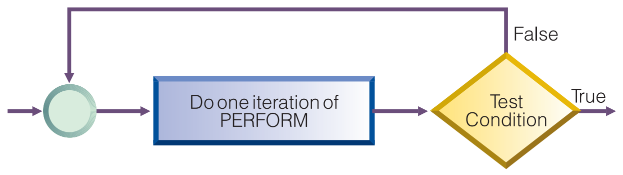
```
PERFORM 010-PROCESS-ONE-MONTH 12 TIMES
INSPECT . . .
```

In the example above, 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.

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 . . .
```

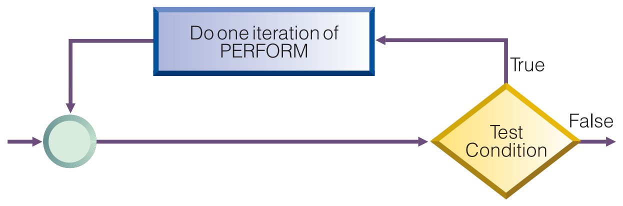
Use the PERFORM . . . WITH TEST AFTER . . . UNTIL statement if you want to execute the paragraph at least once, and test before any subsequent execution. This statement is equivalent to a do-until structure:



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

```
PERFORM  010-PROCESS-ONE-MONTH
  UNTIL MONTH GREATER THAN 12
INSPECT . . .
```

| When control reaches the PERFORM statement, the condition MONTH GREATER THAN 12 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.)



Looping through a table

You can use the PERFORM . . . VARYING statement to initialize a table. In this form of the PERFORM statement, a variable is increased or decreased and tested until a condition is satisfied.

Thus you use the PERFORM statement to control looping through a table. You can use either of these forms:

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

The following section of code shows an example of looping through a table to check for invalid 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 . . .
```

When control reaches the `PERFORM` statement above, `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 the condition 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.

The loop above controls input-checking for the 12 fields of item `WS-DATA`. Empty fields are not allowed in the application, so the section of code loops and issues error messages as appropriate.

Executing multiple paragraphs or sections

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

When you use the `PERFORM . . . THRU` statement, code a paragraph-`EXIT` statement to clearly indicate the end point of a series of paragraphs.

RELATED TASKS
"Processing table items using intrinsic functions" on page 86

Chapter 6. Handling strings

COBOL provides language constructs for performing many different operations on string data items.

For example, you can:
- Join or split data items.
- Manipulate null-terminated strings, such as count or move characters.
- Refer to substrings by their ordinal position and, if needed, length.
- Tally and replace data items, such as count the number of times a specific character occurs in a data item.
- Convert data items, such as change to uppercase or lowercase.
- Evaluate data items, such as determine the length of a data item.

RELATED TASKS
"Joining data items (STRING)"
"Splitting data items (UNSTRING)" on page 103
"Manipulating null-terminated strings" on page 106
"Referring to substrings of data items" on page 107
"Tallying and replacing data items (INSPECT)" on page 111
"Converting data items (intrinsic functions)" on page 112
"Evaluating data items (intrinsic functions)" on page 115
Chapter 7, "Processing data in an international environment," on page 121

Joining data items (STRING)

Use the `STRING` statement to join all or parts of several data items or literals into one data item. One `STRING` statement can take the place of several `MOVE` statements.

The `STRING` statement transfers data into a receiving data item in the order that you indicate. In the `STRING` statement you also specify:
- A delimiter for each set of sending fields that, if encountered, causes those sending fields to stop being transferred (`DELIMITED BY` phrase)
- (Optional) Action to be taken if the receiving field is filled before all of the sending data has been processed (`ON OVERFLOW` phrase)
- (Optional) An integer data item that indicates the leftmost character position within the receiving field into which data should be transferred (`WITH POINTER` phrase)

The receiving data item must not be an edited item, or a display or national floating-point item. If the receiving data item has:
- `USAGE DISPLAY`, each identifier in the statement except the `POINTER` identifier must have `USAGE DISPLAY`, and each literal in the statement must be alphanumeric
- `USAGE NATIONAL`, each identifier in the statement except the `POINTER` identifier must have `USAGE NATIONAL`, and each literal in the statement must be national
- `USAGE DISPLAY-1`, each identifier in the statement except the `POINTER` identifier must have `USAGE DISPLAY-1`, and each literal in the statement must be DBCS

| Only that portion of the receiving field into which data is written by the STRING
| statement is changed.

"Example: STRING statement"

RELATED TASKS
"Handling errors in joining and splitting strings" on page 234

RELATED REFERENCES
STRING statement (*Enterprise COBOL Language Reference*)

Example: STRING statement

The following example shows the STRING statement selecting and formatting information from a record into an output line.

The FILE SECTION defines the following record:

```
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).
```

The WORKING-STORAGE SECTION defines the following fields:

```
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 RCD-01 contains the following information (the symbol *b* indicates a blank space):

```
J.B.bSMITHbbbbb
444bSPRINGbST.,bCHICAGO,bILL.bbbbbb
A14275
$4,736.85
$2,400.00
09/22/76
$2,336.85
10/22/76
```

In the PROCEDURE DIVISION, these settings occur before the STRING statement:

- RPT-LINE is set to SPACES.
- LINE-POS, the data item to be used as the POINTER field, is set to 4.

Here is the 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.
```

Because the POINTER field LINE-POS has value 4 before the STRING statement is performed, data is moved into the receiving field RPT-LINE beginning at character position 4. Characters in positions 1 through 3 are unchanged.

The sending items that specify DELIMITED BY SIZE are moved in their entirety to the receiving field. Because BAL-DUE is delimited by DEC-POINT, the moving of BAL-DUE to the receiving field stops when a decimal point (the value of DEC-POINT) is encountered.

STRING results

When the STRING statement is performed, items are moved into RPT-LINE as shown in the table below.

Item	Positions
LINE-NO	4 - 8
Space	9
CUST-INFO	10 - 59
INV-NO	60 - 65
Space	66
DATE-DUE	67 - 74
Space	75
Portion of BAL-DUE that precedes the decimal point	76 - 81

After the STRING statement is performed, the value of LINE-POS is 82, and RPT-LINE has the values shown below.

```
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)

Use the UNSTRING statement to split a sending field into several receiving fields. One UNSTRING statement can take the place of several MOVE statements.

In the UNSTRING statement you can specify:

- Delimiters that, when one of them is encountered in the sending field, cause the current receiving field to stop receiving and the next, if any, to begin receiving (DELIMITED BY phrase)
- A field for the delimiter that, when encountered in the sending field, causes the current receiving field to stop receiving (DELIMITER IN phrase)
- An integer data item that stores the number of characters placed in the current receiving field (COUNT IN phrase)
- An integer data item that indicates the leftmost character position within the sending field at which UNSTRING processing should begin (WITH POINTER phrase)
- An integer data item that stores a tally of the number of receiving fields that are acted on (TALLYING IN phrase)

- Action to be taken if all of the receiving fields are filled before the end of the sending data item is reached (`ON OVERFLOW` phrase)

The sending data item and the delimiters in the `DELIMITED BY` phrase must be of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, national, or national-edited.

Receiving data items can be of category alphabetic, alphanumeric, numeric, DBCS, or national. If numeric, a receiving data item must be zoned decimal or national decimal. If a receiving data item has:
- `USAGE DISPLAY`, the sending item and each delimiter item in the statement must have `USAGE DISPLAY`, and each literal in the statement must be alphanumeric
- `USAGE NATIONAL`, the sending item and each delimiter item in the statement must have `USAGE NATIONAL`, and each literal in the statement must be national
- `USAGE DISPLAY-1`, the sending item and each delimiter item in the statement must have `USAGE DISPLAY-1`, and each literal in the statement must be DBCS

"Example: UNSTRING statement"

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125

RELATED TASKS
"Handling errors in joining and splitting strings" on page 234

RELATED REFERENCES
UNSTRING statement (*Enterprise COBOL Language Reference*)
Classes and categories of data (*Enterprise COBOL Language Reference*)

Example: UNSTRING statement

The following example shows the `UNSTRING` statement transferring selected information from an input record. Some information is organized for printing and some for further processing.

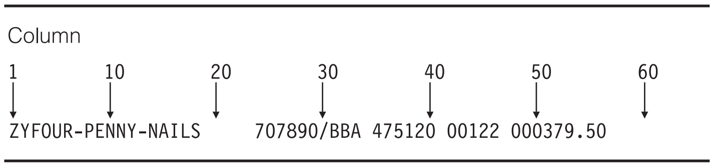
The `FILE SECTION` defines the following records:

```
*  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, these settings occur before the UNSTRING statement:

- A period (.) is placed in DBY-1 for use as a delimiter.
- CHAR-CT (the POINTER field) is set to 3.
- The value zero (0) is placed in FLDS-FILLED (the TALLYING field).
- Data is read into record INV-RCD, whose format is as shown below.



```
ZYFOUR-PENNY-NAILS     707890/BBA 475120 00122 000379.50
```

Here is the UNSTRING statement:

```
* 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.
```

Because the POINTER field CHAR-CT has value 3 before the UNSTRING statement is performed, the two character positions of the CONTROL-CHARS field in INV-RCD are ignored.

UNSTRING 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 four 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 five contiguous space characters in positions 19 through 23 are considered to be one occurrence of the delimiter.
3. Positions 24 through 29 (707890) are placed in INV-NO. The delimiter character slash (/) is placed in DLTR-1, and the value 6 is placed in CTR-2.

4. Positions 31 through 33 (BBA) are placed in INV-CLASS. The delimiter is SPACE, but because no field has been defined as a receiving area for delimiters, the space in position 34 is bypassed.
5. Positions 35 through 40 (475120) are placed in M-UNITS. The value 6 is placed in CTR-3. The delimiter is SPACE, but because no field has been defined as a receiving area for delimiters, the space in position 41 is bypassed.
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 zero (0). The delimiter is SPACE, but because no field was defined as a receiving area for delimiters, the space in position 47 is bypassed.
7. Positions 48 through 53 (000379) are placed in DISPLAY-DOLS. The period (.) delimiter 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 two characters in INV-RCD have not been examined, the ON OVERFLOW statement is executed. Execution of the UNSTRING statement is completed.

After the UNSTRING statement is performed, the fields contain the values shown below.

Field	**Value**
DISPLAY-REC	707890 FOUR-PENNY-NAILS 000379
WORK-REC	475120000122BBA
CHAR-CT (the POINTER field)	55
FLDS-FILLED (the TALLYING field)	6

Manipulating null-terminated strings

You can construct and manipulate null-terminated strings (for example, strings that are passed to or from a C program) by various mechanisms.

For example, you can:
- Use null-terminated literal constants (Z". . . ").
- Use an INSPECT statement to count the number of characters in a null-terminated string:

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

- Use an 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
```

- Use a SEARCH statement to locate trailing null or space characters. Define the string being examined as a table of single characters.
- Check each character in a field in a loop (PERFORM). You can examine each character in a field by using a reference modifier such as `source-field (I:1)`.

"Example: null-terminated strings"

RELATED TASKS
"Handling null-terminated strings" on page 456

RELATED REFERENCES
Alphanumeric literals (*Enterprise COBOL Language Reference*)

Example: null-terminated strings

The following example shows several ways in which you can process null-terminated strings.

```
 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 to produce another
     String L     delimited by x'00'
            M     delimited by x'00'
            X'00' delimited by size
            into N.
```

Referring to substrings of data items

| Refer to a substring of a data item that has USAGE DISPLAY, DISPLAY-1, or NATIONAL by using a reference modifier. You can also refer to a substring of an alphanumeric or national character string that is returned by an intrinsic function by using a reference modifier.

The following example shows how to use a reference modifier to refer to a twenty-character substring of a data item called Customer-Record:

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

You code a reference modifier in parentheses immediately after the data item. As the example shows, a reference modifier can contain two values that are separated by a colon:

1. Ordinal position (from the left) of the character that you want the substring to start with
2. (Optional) Length of the desired substring in *character positions*

The reference-modifier position and length for an item that has USAGE DISPLAY are expressed in terms of single-byte characters. The reference-modifier position and length for items that have USAGE DISPLAY-1 or NATIONAL are expressed in terms of DBCS character positions and national character positions, respectively.

If you omit the length in a reference modifier (coding only the ordinal position of the first character, followed by a colon), the substring extends to the end of the item. Omit the length where possible as a simpler and less error-prone coding technique.

You can refer to substrings of USAGE DISPLAY data items, including alphanumeric groups, alphanumeric-edited data items, numeric-edited data items, display floating-point data items, and zoned decimal data items, by using reference modifiers. When you reference-modify any of these data items, the result is of category alphanumeric. When you reference-modify an alphabetic data item, the result is of category alphabetic.

You can refer to substrings of USAGE NATIONAL data items, including national groups, national-edited data items, numeric-edited data items, national floating-point data items, and national decimal data items, by using reference modifiers. When you reference-modify any of these data items, the result is of category national. For example, suppose that you define a national decimal data item as follows:

```
01  NATL-DEC-ITEM  Usage National  Pic 999  Value 123.
```

You can use NATL-DEC-ITEM in an arithmetic expression because NATL-DEC-ITEM is of category numeric. But you cannot use NATL-DEC-ITEM(2:1) (the national character 2, which in hexadecimal notation is NX"0032") in an arithmetic expression, because it is of category national.

You can refer to substrings of table entries, including variable-length entries, by using reference modifiers. To refer to a substring of a table entry, code the subscript expression before the reference modifier. For example, assume that PRODUCT-TABLE is a properly coded table of character strings. To move D to the fourth character in the second string in the table, you can code this statement:

```
MOVE 'D' to PRODUCT-TABLE (2), (4:1)
```

You can code either or both of the two values in a reference modifier as a variable or as an arithmetic expression.

"Example: arithmetic expressions as reference modifiers" on page 110

Because numeric function identifiers can be used anywhere that arithmetic expressions can be used, you can code a numeric function identifier in a reference modifier as the leftmost character position or as the length, or both.

"Example: intrinsic functions as reference modifiers" on page 110

Each number in the reference modifier must have a value of at least 1. The sum of the two numbers must not exceed the total length of the data item by more than 1 character position so that you do not reference beyond the end of the substring.

If the leftmost character position or the length value is a fixed-point noninteger, truncation occurs to create an integer. If either is a floating-point noninteger, rounding occurs to create an integer.

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

- `SSRANGE` compiler option
- `CHECK` runtime option

RELATED CONCEPTS
"Reference modifiers"
"Unicode and the encoding of language characters" on page 125

RELATED TASKS
"Referring to an item in a table" on page 72

RELATED REFERENCES
"SSRANGE" on page 337
Reference modification (*Enterprise COBOL Language Reference*)
Function definitions (*Enterprise COBOL Language Reference*)

Reference modifiers

Reference modifiers let you easily refer to a substring of a data item.

For example, 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 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 modifiers, you would have to define data items for both formats. You would also have to write code to convert from one format to the other.

With reference modifiers, you do not need to provide names for the subfields that describe the `TIME` elements. The only data definition you need is for the time as returned by the system. For example:

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

The following code retrieves and expands the time value:

```
     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)
```

"Example: arithmetic expressions as reference modifiers" on page 110
"Example: intrinsic functions as reference modifiers" on page 110

RELATED TASKS
"Assigning input from a screen or file (ACCEPT)" on page 36
"Referring to substrings of data items" on page 107
"Using national data (Unicode) in COBOL" on page 126

RELATED REFERENCES
Reference modification (*Enterprise COBOL Language Reference*)

Example: arithmetic expressions as reference modifiers

Suppose that a field contains some right-justified characters, and you want to move those characters to another field where they will be left justified. You can do so by using reference modifiers and an INSPECT statement.

Suppose a program has the following data:

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

The program counts the number of leading spaces and, using arithmetic expressions in a reference modifier, moves the right-justified characters into another field, justified to the left:

```
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 as I + 1 for a length that is computed as LENGTH OF RIGHTY - I, into the field LEFTY.

Example: intrinsic functions as reference modifiers

You can use intrinsic functions in reference modifiers if you do not know the leftmost position or length of a substring at compile time.

For example, the following code fragment causes a substring of Customer-Record to be moved into the data item WS-name. The substring is determined at run time.

```
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
```

If you want to use a noninteger function in a position that requires 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
```

RELATED REFERENCES
INTEGER (*Enterprise COBOL Language Reference*)
INTEGER-PART (*Enterprise COBOL Language Reference*)

Tallying and replacing data items (INSPECT)

Use the INSPECT statement to inspect characters or groups of characters in a data item and to optionally replace them.

Use the INSPECT statement to do the following tasks:

- Count the number of times a specific character occurs in a data item (TALLYING phrase).
- Fill a data item or selected portions of a data item with specified characters such as spaces, asterisks, or zeros (REPLACING phrase).
- Convert all occurrences of a specific character or string of characters in a data item to replacement characters that you specify (CONVERTING phrase).

You can specify one of the following data items as the item to be inspected:

- An elementary item described explicitly or implicitly as USAGE DISPLAY, USAGE DISPLAY-1, or USAGE NATIONAL
- An alphanumeric group item or national group item

If the inspected item has:

- USAGE DISPLAY, each identifier in the statement (except the TALLYING count field) must have USAGE DISPLAY, and each literal in the statement must be alphanumeric
- USAGE NATIONAL, each identifier in the statement (except the TALLYING count field) must have USAGE NATIONAL, and each literal in the statement must be national
- USAGE DISPLAY-1, each identifier in the statement (except the TALLYING count field) must have USAGE DISPLAY-1, and each literal in the statement must be a DBCS literal

"Examples: INSPECT statement"

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125

RELATED REFERENCES
INSPECT statement (*Enterprise COBOL Language Reference*)

Examples: INSPECT statement

The following examples show some uses of the INSPECT statement to examine and replace characters.

In the following example, the INSPECT statement examines and replaces characters in data item DATA-2. The number of times a leading zero (0) occurs in the data item is accumulated in COUNTR. The first instance of the character A that follows the first instance of the character C is replaced by the character 2.

```
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

DATA-2 before	COUNTR after	DATA-2 after
0000ALABAMA	4	0000ALABAMA
CHATHAM0000	0	CH2THAM0000

In the following example, the INSPECT statement examines and replaces characters in data item DATA-3. Each character that precedes the first instance of a quotation mark (") is replaced by the character 0.

```
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

The following example shows the use of INSPECT CONVERTING with AFTER and BEFORE phrases to examine and replace characters in data item DATA-4. All characters that follow the first instance of the character / but that precede the first instance of the character ? (if any) are translated from lowercase to uppercase.

```
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)

You can use intrinsic functions to convert character-string data items to several other formats, for example, to uppercase or lowercase, to reverse order, to numbers, or to one code page from another.

You can use the NATIONAL-OF and DISPLAY-OF intrinsic functions to convert to and from national (Unicode) strings.

You can also use the INSPECT statement to convert characters.

"Examples: INSPECT statement" on page 111

RELATED TASKS
"Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE)" on page 113

"Transforming to reverse order (REVERSE)"
"Converting to numbers (NUMVAL, NUMVAL-C)"
"Converting from one code page to another" on page 114

Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE)

| You can use the UPPER-CASE and LOWER-CASE intrinsic functions to easily change the
| case of alphanumeric, alphabetic, or national strings.

```
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
```

The code above displays the following messages on the system logical output device:

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

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

RELATED TASKS
"Assigning input from a screen or file (ACCEPT)" on page 36
"Displaying values on a screen or in a file (DISPLAY)" on page 37

Transforming to reverse order (REVERSE)

You can reverse the order of the characters in a string by using the REVERSE intrinsic function.

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

For example, the statement above reverses the order of the characters in Orig-cust-name. If the starting value is JOHNSON*bbb*, the value after the statement is performed is *bbb*NOSNHOJ, where *b* represents a blank space.

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125

Converting to numbers (NUMVAL, NUMVAL-C)

| The NUMVAL and NUMVAL-C functions convert character strings (alphanumeric or
| national literals, or class alphanumeric or class national data items) to numbers. Use these functions to convert free-format character-representation numbers to numeric form so that you can process them numerically.

```
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)
```

Use NUMVAL-C when the argument includes a currency symbol or comma or both, as shown in the example above. You can also place an algebraic sign before or after the character string, and the sign will be processed. The arguments must not exceed 18 digits when you compile with the default option ARITH(COMPAT) (*compatibility mode*) nor 31 digits when you compile with ARITH(EXTEND) (*extended mode*), not including the editing symbols.

NUMVAL and NUMVAL-C return long (64-bit) floating-point values in compatibility mode, and return extended-precision (128-bit) floating-point values in extended mode. A reference to either of these functions represents a reference to a numeric data item.

| At most 15 decimal digits can be converted accurately to long-precision floating
| point (as described in the related reference below about conversions and precision).
| If the argument to NUMVAL or NUMVAL-C has more than 15 digits, it is recommended
| that you specify the ARITH(EXTEND) compiler option so that an extended-precision
| function result that can accurately represent the value of the argument is returned.

When you use NUMVAL or NUMVAL-C, you do not need to statically declare numeric data in a fixed format nor input data in a precise manner. For example, suppose you define numbers to be entered as follows:

```
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)
```

The input could then be:

```
1.23
-300
```

RELATED CONCEPTS
"Formats for numeric data" on page 49
"Data format conversions" on page 54
"Unicode and the encoding of language characters" on page 125

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 133

RELATED REFERENCES
"Conversions and precision" on page 54
"ARITH" on page 302

Converting from one code page to another

You can nest the DISPLAY-OF and NATIONAL-OF intrinsic functions to easily convert from any code page to any other code page.

For example, the following code converts an EBCDIC string to an ASCII string:

```
  77  EBCDIC-CCSID PIC 9(4) BINARY VALUE 1140.
  77  ASCII-CCSID  PIC 9(4) BINARY VALUE 819.
  77  Input-EBCDIC PIC X(80).
  77  ASCII-Output PIC X(80).
  . . .
* Convert EBCDIC to ASCII
      Move Function Display-of
          (Function National-of (Input-EBCDIC EBCDIC-CCSID),
              ASCII-CCSID)
      to ASCII-output
```

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 133

Evaluating data items (intrinsic functions)

You can use intrinsic functions to determine the ordinal position of a character in the collating sequence, to find the largest or smallest item in a series, to find the length of data item, or to determine when a program was compiled.

Use these intrinsic functions:

| • `CHAR` and `ORD` to evaluate integers and single alphabetic or alphanumeric characters with respect to the collating sequence used in a program
• `MAX`, `MIN`, `ORD-MAX`, and `ORD-MIN` to find the largest and smallest items in a series
| of data items, including `USAGE NATIONAL` data items
| • `LENGTH` to find the length of data items, including `USAGE NATIONAL` data items
• `WHEN-COMPILED` to find the date and time when a program was compiled

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125

RELATED TASKS
"Evaluating single characters for collating sequence"
"Finding the largest or smallest data item" on page 116
"Finding the length of data items" on page 118
"Finding the date of compilation" on page 119

Evaluating single characters for collating sequence

| To find out the ordinal position of a given alphabetic or alphanumeric character in the collating sequence, use the `ORD` function with the character as the argument. `ORD` returns an integer that represents that ordinal position.

You can use a one-character substring of a data item as the argument to `ORD`:

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

If you know the ordinal position in the collating sequence of a character, and want to find the character that it corresponds to, use the `CHAR` function with the integer ordinal position as the argument. `CHAR` returns the desired character. For example:

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

RELATED REFERENCES
CHAR (*Enterprise COBOL Language Reference*)
ORD (*Enterprise COBOL Language Reference*)

Finding the largest or smallest data item

To determine which of two or more alphanumeric, alphabetic, or national data items has the largest value, use the MAX or ORD-MAX intrinsic function. To determine which item has the smallest value, use MIN or ORD-MIN. These functions evaluate according to the collating sequence.

| To compare numeric items, including those that have USAGE NATIONAL, you can use
| MAX, ORD-MAX, MIN, or ORD-MIN. With these intrinsic functions, the algebraic values of the arguments are compared.

The MAX and MIN functions return the content of one of the arguments that you supply. For example, suppose that your program has the following 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 assigns VALLEJO*bbb* to the first 10 character positions of Customer-record, where *b* represents a blank space:

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

If you used MIN instead, then THOMAS*bbbb* would be assigned.

The functions ORD-MAX and ORD-MIN return an integer that represents the ordinal position (counting from the left) of the argument that has the largest or smallest value in the list of arguments that you supply. If you used the ORD-MAX function in the example above, the compiler would issue an error message because the reference to a numeric function is not in a valid place. The following statement is a valid use of ORD-MAX:

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

The statement above assigns the integer 3 to x if the same arguments are used as in the previous example. If you used ORD-MIN instead, the integer 2 would be returned. The examples above might be more realistic if Arg1, Arg2, and Arg3 were successive elements of an array (table).

If you specify a national item for any argument, you must specify all arguments as
| class national.

RELATED TASKS
"Performing arithmetic" on page 57
"Processing table items using intrinsic functions" on page 86
"Returning variable-length results with alphanumeric or national functions" on page 117

RELATED REFERENCES
MAX (*Enterprise COBOL Language Reference*)
MIN (*Enterprise COBOL Language Reference*)
ORD-MAX (*Enterprise COBOL Language Reference*)
ORD-MIN (*Enterprise COBOL Language Reference*)

Returning variable-length results with alphanumeric or national functions

The results of alphanumeric or national functions could 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))
```

This code has the following results:

- R2 is evaluated to be larger than R1.
- The string '*fbbbb*' is moved to R3, where *b* represents a blank space. (The unfilled character positions in R3 are padded with spaces.)
- L evaluates to the value 5.

If R1 contained 'g' instead of 'e', the code would have the following results:

- R1 would evaluate as larger than R2.
- The string '*gbbbbbbbbb*' would be moved to R3. (The unfilled character positions in R3 would be padded with spaces.)
- The value 10 would be assigned to L.

If a program uses national data for function arguments, the lengths and values of the function results could likewise vary. For example, the following code is identical to the fragment above, but uses national data instead of alphanumeric data.

```
01  R1    Pic n(10) national value "e".
01  R2    Pic n(05) national value "f".
01  R3    Pic n(20) national value spaces.
01  L     Pic 99    national.
. . .
    Move Function Max(R1 R2) to R3
    Compute L = Function Length(Function Max(R1 R2))
```

This code has the following results, which are similar to the first set of results except that these are for national characters:

- R2 is evaluated to be larger than R1.
- The string NX"0066 0020 0020 0020 0020" (the equivalent in national characters of '*fbbbb*', where *b* represents a blank space), shown here in hexadecimal notation with added spaces for readability, is moved to R3. The unfilled character positions in R3 are padded with national spaces.
- L evaluates to the value 5, the length in national character positions of R2.

You might be dealing with variable-length output from alphanumeric or national functions. Plan your program accordingly. For example, you might need to think about using variable-length files when the records that you are writing could be of different lengths:

```
File Section.
FD  Output-File Recording Mode V.
01  Short-Customer-Record  Pic X(50).
01  Long-Customer-Record   Pic X(70).
```

```
Working-Storage Section.
01  R1    Pic x(50).
01  R2    Pic x(70).
. . .
    If R1 > R2
      Write Short-Customer-Record from R1
    Else
      Write Long-Customer-Record from R2
    End-if
```

RELATED TASKS
"Finding the largest or smallest data item" on page 116
"Performing arithmetic" on page 57

RELATED REFERENCES
MAX (*Enterprise COBOL Language Reference*)

Finding the length of data items

You can use the LENGTH function in many contexts (including tables and numeric
| data) to determine the length of an item. For example, you can use the LENGTH
| function to determine the length of an alphanumeric or national literal, or a data
| item of any type except DBCS.

| The LENGTH function returns the length of a national item (a literal, or any item that
| has USAGE NATIONAL, including national group items) as an integer equal to the
| length of the argument in national character positions. It returns the length of any
| other data item as an integer equal to the length of the argument in alphanumeric
| character positions.

The following COBOL statement demonstrates moving a data item into the field in a record that holds customer names:

```
Move Customer-name To Customer-record(1:Function Length(Customer-name))
```

You can also use the LENGTH OF special register, which returns the length in bytes even for national data. Coding either Function Length(Customer-name) or LENGTH OF Customer-name returns the same result for alphanumeric items: the length of Customer-name in bytes.

You can use the LENGTH function only where arithmetic expressions are allowed. However, you can use the LENGTH OF special register in a greater variety of contexts. For example, you can use the LENGTH OF special register as an argument to an intrinsic function that allows integer arguments. (You cannot use an intrinsic function as an operand to the LENGTH OF special register.) You can also use the LENGTH OF special register as a parameter in a CALL statement.

RELATED TASKS
"Performing arithmetic" on page 57
"Creating variable-length tables (DEPENDING ON)" on page 80
"Processing table items using intrinsic functions" on page 86

RELATED REFERENCES
LENGTH (*Enterprise COBOL Language Reference*)
LENGTH OF (*Enterprise COBOL Language Reference*)

Finding the date of compilation

You can use the `WHEN-COMPILED` intrinsic function to determine when a program was compiled. The 21-character result indicates the four-digit year, month, day, and time (in hours, minutes, seconds, and hundredths of seconds) of compilation, and the difference in hours and minutes from Greenwich mean time.

The first 16 positions are in the following format:

```
YYYYMMDDhhmmsshh
```

You can instead use the `WHEN-COMPILED` special register to determine the date and time of compilation in the following 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. You can use this special register only as the sending field in a `MOVE` statement.

RELATED REFERENCES
WHEN-COMPILED (*Enterprise COBOL Language Reference*)

Chapter 7. Processing data in an international environment

Enterprise COBOL supports Unicode UTF-16 as national character data at run time. UTF-16 provides a consistent and efficient way to encode plain text. Using UTF-16, you can develop software that will work with various national languages.

Use these COBOL facilities to code and compile programs that process national data:

- Data types and literals:
  - Character data types, defined with the `USAGE NATIONAL` clause and a `PICTURE` clause that defines data of category national, national-edited, or numeric-edited
  - Numeric data types, defined with the `USAGE NATIONAL` clause and a `PICTURE` clause that defines a numeric data item (a *national decimal item*) or an external floating-point data item (a *national floating-point item*)
  - National literals, specified with literal prefix `N` or `NX`
  - Figurative constant `ALL` *national-literal*
  - Figurative constants `QUOTE`, `SPACE`, `HIGH-VALUE`, `LOW-VALUE`, or `ZERO`, which have national character (UTF-16) values when used in national-character contexts
- The COBOL statements shown in the related reference below about COBOL statements and national data
- Intrinsic functions:
  - `NATIONAL-OF` to convert an alphanumeric or double-byte character set (DBCS) character string to `USAGE NATIONAL` (UTF-16)
  - `DISPLAY-OF` to convert a national character string to `USAGE DISPLAY` in a selected code page (EBCDIC, ASCII, EUC, or UTF-8)
  - The other intrinsic functions shown in the related reference below about intrinsic functions and national data
- The `GROUP-USAGE NATIONAL` clause to define groups that contain only `USAGE NATIONAL` data items and that behave like elementary category national items in most operations
- Compiler options:
  - `CODEPAGE` to specify the code page to use for alphanumeric and DBCS data in your program
  - `NSYMBOL` to control whether national or DBCS processing is used for the `N` symbol in literals and `PICTURE` clauses

You can also take advantage of implicit conversions of alphanumeric or DBCS data items to national representation. The compiler performs such conversions (in most cases) when you move these items to national data items, or compare these items with national data items.

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125
"National groups" on page 129

RELATED TASKS
"Using national data (Unicode) in COBOL" on page 126
"Converting to or from national (Unicode) representation" on page 133

"Processing UTF-8 data" on page 137
"Processing Chinese GB 18030 data" on page 137
"Comparing national (UTF-16) data" on page 138
"Coding for use of DBCS support" on page 141
Appendix C, "Converting double-byte character set (DBCS) data," on page 663

RELATED REFERENCES
"COBOL statements and national data"
"Intrinsic functions and national data" on page 124
"CODEPAGE" on page 305
"NSYMBOL" on page 323
Classes and categories of data (*Enterprise COBOL Language Reference*)
Data categories and PICTURE rules (*Enterprise COBOL Language Reference*)
MOVE statement (*Enterprise COBOL Language Reference*)
General relation conditions (*Enterprise COBOL Language Reference*)

COBOL statements and national data

| You can use national data with the `PROCEDURE DIVISION` and compiler-directing
| statements shown in the table below.

Table 15. **COBOL statements and national data**

COBOL statement	Can be national	Comment	For more information
`ACCEPT`	*identifier-1*, *identifier-2*	*identifier-1* is converted from the native code page specified in the `CODEPAGE` compiler option only if input is from `CONSOLE`.	"Assigning input from a screen or file (ACCEPT)" on page 36
`ADD`	All identifiers can be numeric items that have `USAGE NATIONAL`. *identifier-3* (`GIVING`) can be numeric-edited with `USAGE NATIONAL`.		"Using COMPUTE and other arithmetic statements" on page 58
`CALL`	*identifier-2*, *identifier-3*, *identifier-4*, *identifier-5*; *literal-2*, *literal-3*		"Passing data" on page 451
`COMPUTE`	*identifier-1* can be numeric or numeric-edited with `USAGE NATIONAL`. *arithmetic-expression* can contain numeric items that have `USAGE NATIONAL`.		"Using COMPUTE and other arithmetic statements" on page 58
`COPY . . . REPLACING`	*operand-1*, *operand-2* of the `REPLACING` phrase		Chapter 18, "Compiler-directing statements," on page 351
`DISPLAY`	*identifier-1*	*identifier-1* is converted to EBCDIC only if the `CONSOLE` mnemonic-name is specified directly or indirectly.	"Displaying values on a screen or in a file (DISPLAY)" on page 37

Table 15. **COBOL statements and national data** *(continued)*

COBOL statement	Can be national	Comment	For more information
DIVIDE	All identifiers can be numeric items that have USAGE NATIONAL. *identifier-3* (GIVING) and *identifier-4* (REMAINDER) can be numeric-edited with USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 58
INITIALIZE	*identifier-1*; *identifier-2* or *literal-1* of the REPLACING phrase	If you specify REPLACING NATIONAL or REPLACING NATIONAL-EDITED, *identifier-2* or *literal-1* must be valid as a sending operand in a move to *identifier-1*.	"Examples: initializing data items" on page 30
INSPECT	All identifiers and literals. (*identifier-2*, the TALLYING integer data item, can have USAGE NATIONAL.)	If any of these (other than *identifier-2*, the TALLYING identifier) have USAGE NATIONAL, all must be national.	"Tallying and replacing data items (INSPECT)" on page 111
INVOKE	Method-name as *identifier-2* or *literal-1*; *identifier-3* or *literal-2* in the BY VALUE phrase		"Invoking methods (INVOKE)" on page 546
MERGE	Merge keys	The COLLATING SEQUENCE phrase does not apply.	"Setting sort or merge criteria" on page 221
MOVE	Both the sender and receiver, or only the receiver	Implicit conversions are performed for valid MOVE operands.	"Assigning values to elementary data items (MOVE)" on page 34 "Assigning values to group data items (MOVE)" on page 35
MULTIPLY	All identifiers can be numeric items that have USAGE NATIONAL. *identifier-3* (GIVING) can be numeric-edited with USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 58
SEARCH ALL (binary search)	Both the key data item and its object of comparison	The key data item and its object of comparison must be compatible according to the rules of comparison. If the object of comparison is of class national, the key must be also.	"Doing a binary search (SEARCH ALL)" on page 85
SORT	Sort keys	The COLLATING SEQUENCE phrase does not apply.	"Setting sort or merge criteria" on page 221
STRING	All identifiers and literals. (*identifier-4*, the POINTER integer data item, can have USAGE NATIONAL.)	If *identifier-3*, the receiving data item, is national, all identifiers and literals (other than *identifier-4*, the POINTER identifier) must be national.	"Joining data items (STRING)" on page 101

Table 15. **COBOL statements and national data** *(continued)*

COBOL statement	Can be national	Comment	For more information
SUBTRACT	All identifiers can be numeric items that have USAGE NATIONAL. *identifier-3* (GIVING) can be numeric-edited with USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 58
UNSTRING	All identifiers and literals. (*identifier-6* and *identifier-7*, the COUNT and TALLYING integer data items, respectively, can have USAGE NATIONAL.)	If *identifier-4*, a receiving data item, has USAGE NATIONAL, the sending data item and each delimiter must have USAGE NATIONAL, and each literal must be national.	"Splitting data items (UNSTRING)" on page 103
XML GENERATE	*identifier-1* (the generated XML document); *identifier-2* (the source field or fields)		Chapter 29, "Producing XML output," on page 511
XML PARSE	*identifier-1* (the XML document)	The XML-NTEXT special register contains national character document fragments during parsing.	Chapter 28, "Processing XML input," on page 487

RELATED TASKS
"Defining numeric data" on page 45
"Displaying numeric data" on page 47
"Using national data (Unicode) in COBOL" on page 126
"Comparing national (UTF-16) data" on page 138

RELATED REFERENCES
"CODEPAGE" on page 305
Classes and categories of data (*Enterprise COBOL Language Reference*)

Intrinsic functions and national data

You can use arguments of class national with the intrinsic functions shown in the table below.

Table 16. **Intrinsic functions and national character data**

Intrinsic function	Function type	For more information
DISPLAY-OF	Alphanumeric	"Converting national data to alphanumeric data (DISPLAY-OF)" on page 135
LENGTH	Integer	"Finding the length of data items" on page 118
LOWER-CASE, UPPER-CASE	National	"Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE)" on page 113
NUMVAL, NUMVAL-C	Numeric	"Converting to numbers (NUMVAL, NUMVAL-C)" on page 113
MAX, MIN	National	"Finding the largest or smallest data item" on page 116
ORD-MAX, ORD-MIN	Integer	"Finding the largest or smallest data item" on page 116
REVERSE	National	"Transforming to reverse order (REVERSE)" on page 113

You can use national decimal arguments wherever zoned decimal arguments are allowed. You can use national floating-point arguments wherever display floating-point arguments are allowed. (See the related reference below about arguments for a complete list of intrinsic functions that can take integer or numeric arguments.)

RELATED TASKS
"Defining numeric data" on page 45
"Using national data (Unicode) in COBOL" on page 126

RELATED REFERENCES
Arguments (*Enterprise COBOL Language Reference*)
Classes and categories of data (*Enterprise COBOL Language Reference*)

Unicode and the encoding of language characters

Enterprise COBOL provides basic runtime support for Unicode, which can handle tens of thousands of characters that cover all commonly used characters and symbols in the world.

A *character set* is a defined set of characters, but is not associated with a coded representation. A *coded character set* (also referred to in this documentation as a *code page*) is a set of unambiguous rules that relate the characters of the set to their coded representation. Each code page has a name and is like a table that sets up the symbols for representing a character set; each symbol is associated with a unique bit pattern, or *code point*. Each code page also has a *coded character set identifier* (*CCSID*), which is a value from 1 to 65,536.

Unicode has several encoding schemes, called *Unicode Transformation Format (UTF)*, such as UTF-8, UTF-16, and UTF-32. Enterprise COBOL uses UTF-16 (CCSID 1200) in big-endian format as the representation for national literals and data items that have `USAGE NATIONAL`.

UTF-8 represents ASCII invariant characters a-z, A-Z, 0-9, and certain special characters such as ' @ , . + - = / * ( ) the same way that they are represented in ASCII. UTF-16 represents these characters as `NX'00`*`nn`*`'`, where `X'`*`nn`*`'` is the representation of the character in ASCII.

For example, the string `'ABC'` is represented in UTF-16 as `NX'004100420043'`. In UTF-8, `'ABC'` is represented as `X'414243'`.

One or more *encoding units* are used to represent a character from a coded character set. For UTF-16, an encoding unit takes 2 bytes of storage. Any character defined in any EBCDIC, ASCII, or EUC code page is represented in one UTF-16 encoding unit when the character is converted to the national data representation.

Cross-platform considerations: Enterprise COBOL and COBOL for AIX$^{(R)}$ support UTF-16 in big-endian format in national data. COBOL for Windows supports UTF-16 in little-endian format (UTF-16LE) in national data. If you are porting Unicode data that is encoded in UTF-16LE representation to Enterprise COBOL from another platform, you must convert that data to UTF-16 in big-endian format to process the data as national data.

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 133

RELATED REFERENCES
"Storage of national data" on page 133
Character sets and code pages (*Enterprise COBOL Language Reference*)

Using national data (Unicode) in COBOL

In Enterprise COBOL, you can specify national (UTF-16) data in any of several ways.

These types of national data are available:

- National data items (categories national, national-edited, and numeric-edited)
- National literals
- Figurative constants as national characters
- Numeric data items (national decimal and national floating-point)

In addition, you can define national groups that contain only data items that explicitly or implicitly have `USAGE NATIONAL`, and that behave in the same way as elementary category national data items in most operations.

These declarations affect the amount of storage that is needed.

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125
"National groups" on page 129

RELATED TASKS
"Defining national data items"
"Using national literals" on page 127
"Using national-character figurative constants" on page 128
"Defining national numeric data items" on page 129
"Using national groups" on page 130
"Converting to or from national (Unicode) representation" on page 133
"Comparing national (UTF-16) data" on page 138

RELATED REFERENCES
"Storage of national data" on page 133
Classes and categories of data (*Enterprise COBOL Language Reference*)

Defining national data items

Define national data items with the `USAGE NATIONAL` clause to hold national (UTF-16) character strings.

You can define national data items of the following categories:

- National
- National-edited
- Numeric-edited

To define a category national data item, code a `PICTURE` clause that contains only one or more `PICTURE` symbols `N`.

To define a national-edited data item, code a `PICTURE` clause that contains at least one of each of the following symbols:

- Symbol `N`

| • Simple insertion editing symbol B, 0, or /

+ To define a numeric-edited data item of class national, code a PICTURE clause that
+ defines a numeric-edited item (for example, -$999.99) and code a USAGE NATIONAL
+ clause. You can use a numeric-edited data item that has USAGE NATIONAL in the
| same way that you use a numeric-edited item that has USAGE DISPLAY.

| You can also define a data item as numeric-edited by coding the BLANK WHEN ZERO
| clause for an elementary item that is defined as numeric by its PICTURE clause.

If you code a PICTURE clause but do not code a USAGE clause for data items that contain only one or more PICTURE symbols N, you can use the compiler option NSYMBOL(NATIONAL) to ensure that such items are treated as national data items instead of as DBCS items.

RELATED TASKS
"Displaying numeric data" on page 47

RELATED REFERENCES
"NSYMBOL" on page 323
BLANK WHEN ZERO clause (*Enterprise COBOL Language Reference*)

Using national literals

To specify national literals, use the prefix character N and compile with the option NSYMBOL(NATIONAL).

You can use either of these notations:
- N"*character-data*"
- N'*character-data*'

If you compile with the option NSYMBOL(DBCS), the literal prefix character N specifies a DBCS literal, not a national literal.

To specify a national literal as a hexadecimal value, use the prefix NX. You can use either of these notations:
- NX"*hexadecimal-digits*"
- NX'*hexadecimal-digits*'

Each of the following MOVE statements sets the national data item Y to the UTF-16 value of the characters 'AB':

```
01 Y pic NN usage national.
. . .
    Move NX"00410042" to Y
    Move N"AB"        to Y
    Move "AB"         to Y
```

Do not use alphanumeric hexadecimal literals in contexts that call for national literals, because such usage is easily misunderstood. For example, the following statement also results in moving the UTF-16 characters 'AB' (not the hexadecimal bit pattern C1C2) to Y, where Y is defined as USAGE NATIONAL:

```
Move X"C1C2" to Y
```

| You cannot use national literals in the SPECIAL-NAMES paragraph or as
| program-names. You can use a national literal to name an object-oriented method
| in the METHOD-ID paragraph or to specify a method-name in an INVOKE statement.

RELATED TASKS
"Using literals" on page 27

RELATED REFERENCES
"NSYMBOL" on page 323
National literals (*Enterprise COBOL Language Reference*)

Using national-character figurative constants

You can use the figurative constant ALL *national-literal* in a context that requires national characters. ALL *national-literal* represents all or part of the string that is generated by successive concatenations of the encoding units that make up the national literal.

You can use the figurative constants QUOTE, SPACE, HIGH-VALUE, LOW-VALUE, or ZERO in a context that requires national characters, such as a MOVE statement, an implicit move, or a relation condition that has national operands. In these contexts, the figurative constant represents a national-character (UTF-16) value.

When you use the figurative constant HIGH-VALUE in a context that requires national characters, its value is NX'FFFF'. When you use LOW-VALUE in a context that requires national characters, its value is NX'0000'.

Restrictions: You must not use HIGH-VALUE or the value assigned from HIGH-VALUE in a way that results in conversion of the value from one data representation to another (for example, between USAGE DISPLAY and USAGE NATIONAL). X'FF' (the value of HIGH-VALUE in an alphanumeric context when the EBCDIC collating sequence is being used) does not represent a valid EBCDIC character, and NX'FFFF' does not represent a valid national character. Conversion of such a value to another representation results in a *substitution character* being used (not X'FF' or NX'FFFF'). Consider the following example:

```
01 natl-data  PIC NN  Usage National.
01 alph-data  PIC XX.
. . .
    MOVE HIGH-VALUE TO natl-data, alph-data
    IF natl-data = alph-data. . .
```

The IF statement above evaluates as false even though each of its operands was set to HIGH-VALUE. Before an elementary alphanumeric operand is compared to a national operand, the alphanumeric operand is treated as though it were moved to a temporary national data item, and the alphanumeric characters are converted to the corresponding national characters. When X'FF' is converted to UTF-16, however, the UTF-16 item gets a substitution character value and so does not compare equally to NX'FFFF'.

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 133
"Comparing national (UTF-16) data" on page 138

RELATED REFERENCES
Figurative constants (*Enterprise COBOL Language Reference*)
DISPLAY-OF (*Enterprise COBOL Language Reference*)
Support for Unicode: Using Conversion Services

Defining national numeric data items

Define data items with the USAGE NATIONAL clause to hold numeric data that is represented in national characters (UTF-16). You can define national decimal items and national floating-point items.

To define a national decimal item, code a PICTURE clause that contains only the symbols 9, P, S, and V. If the PICTURE clause contains S, the SIGN IS SEPARATE clause must be in effect for that item.

To define a national floating-point item, code a PICTURE clause that defines a floating-point item (for example, +99999.9E-99).

You can use national decimal items in the same way that you use zoned decimal items. You can use national floating-point items in the same way that you use display floating-point items.

RELATED TASKS
"Defining numeric data" on page 45
"Displaying numeric data" on page 47

RELATED REFERENCES
SIGN clause (*Enterprise COBOL Language Reference*)

National groups

National groups, which are specified either explicitly or implicitly with the GROUP-USAGE NATIONAL clause, contain only data items that have USAGE NATIONAL. In most cases, a national group item is processed as though it were redefined as an elementary category national item described as PIC N(*m*), where *m* is the number of national (UTF-16) characters in the group.

For some operations on national groups, however (just as for some operations on alphanumeric groups), group semantics apply. Such operations (for example, MOVE CORRESPONDING and INITIALIZE) recognize or process the elementary items within the national group.

Where possible, use national groups instead of alphanumeric groups that contain USAGE NATIONAL items. National groups provide several advantages for the processing of national data compared to the processing of national data within alphanumeric groups:

- When you move a national group to a longer data item that has USAGE NATIONAL, the receiving item is padded with national characters. By contrast, if you move an alphanumeric group that contains national characters to a longer alphanumeric group that contains national characters, alphanumeric spaces are used for padding. As a result, mishandling of data items could occur.
- When you move a national group to a shorter data item that has USAGE NATIONAL, the national group is truncated at national-character boundaries. By contrast, if you move an alphanumeric group that contains national characters to a shorter alphanumeric group that contains national characters, truncation might occur between the 2 bytes of a national character.
- When you move a national group to a national-edited or numeric-edited item, the content of the group is edited. By contrast, if you move an alphanumeric group to an edited item, no editing takes place.
- When you use a national group as an operand in a STRING, UNSTRING, or INSPECT statement:

- The group content is processed as national characters rather than as single-byte characters.
- TALLYING and POINTER operands operate at the logical level of national characters.
- The national group operand is supported with a mixture of other national operand types.

By contrast, if you use an alphanumeric group that contains national characters in these contexts, the characters are processed byte by byte. As a result, invalid handling or corruption of data could occur.

USAGE NATIONAL groups: A group item can specify the USAGE NATIONAL clause at the group level as a convenient shorthand for the USAGE of each of the elementary data items within the group. Such a group is *not* a national group, however, but an alphanumeric group, and behaves in many operations, such as moves and compares, like an elementary data item of USAGE DISPLAY (except that no editing or conversion of data occurs).

RELATED TASKS
"Assigning values to group data items (MOVE)" on page 35
"Joining data items (STRING)" on page 101
"Splitting data items (UNSTRING)" on page 103
"Tallying and replacing data items (INSPECT)" on page 111
"Using national groups"

RELATED REFERENCES
GROUP-USAGE clause (*Enterprise COBOL Language Reference*)

Using national groups

To define a group data item as a national group, code a GROUP-USAGE NATIONAL clause at the group level for the item. The group can contain only data items that explicitly or implicitly have USAGE NATIONAL.

The following data description entry specifies that a level-01 group and its subordinate groups are national group items:

```
01  Nat-Group-1    GROUP-USAGE NATIONAL.
    02  Group-1.
        04  Month    PIC 99.
        04  DayOf    PIC 99.
        04  Year     PIC 9999.
    02  Group-2    GROUP-USAGE NATIONAL.
        04  Amount   PIC 9(4).99  USAGE NATIONAL.
```

In the example above, Nat-Group-1 is a national group, and its subordinate groups Group-1 and Group-2 are also national groups. A GROUP-USAGE NATIONAL clause is implied for Group-1, and USAGE NATIONAL is implied for the subordinate items in Group-1. Month, DayOf, and Year are national decimal items, and Amount is a numeric-edited item that has USAGE NATIONAL.

You can subordinate national groups within alphanumeric groups as in the following example:

```
01  Alpha-Group-1.
    02  Group-1.
        04  Month   PIC 99.
        04  DayOf   PIC 99.
        04  Year    PIC 9999.
    02  Group-2   GROUP-USAGE NATIONAL.
        04  Amount  PIC 9(4).99.
```

In the example above, Alpha-Group-1 and Group-1 are alphanumeric groups; USAGE DISPLAY is implied for the subordinate items in Group-1. (If Alpha-Group-1 specified USAGE NATIONAL at the group level, USAGE NATIONAL would be implied for each of the subordinate items in Group-1. However, Alpha-Group-1 and Group-1 would be alphanumeric groups, not national groups, and would behave like alphanumeric groups during operations such as moves and compares.) Group-2 is a national group, and USAGE NATIONAL is implied for the numeric-edited item Amount.

You cannot subordinate alphanumeric groups within national groups. All elementary items within a national group must be explicitly or implicitly described as USAGE NATIONAL, and all group items within a national group must be explicitly or implicitly described as GROUP-USAGE NATIONAL.

RELATED CONCEPTS
"National groups" on page 129

RELATED TASKS
"Using national groups as elementary items"
"Using national groups as group items" on page 132

RELATED REFERENCES
GROUP-USAGE clause (*Enterprise COBOL Language Reference*)

Using national groups as elementary items

In most cases, you can use a national group as though it were an elementary data item.

In the following example, a national group item, Group-1, is moved to a national-edited item, Edited-date. Because Group-1 is treated as an elementary data item during the move, editing takes place in the receiving data item. The value in Edited-date after the move is 06/23/2006 in national characters.

```
01  Edited-date  PIC NN/NN/NNNN  USAGE NATIONAL.
01  Group-1    GROUP-USAGE NATIONAL.
    02 Month     PIC 99   VALUE 06.
    02 DayOf     PIC 99   VALUE 23.
    02 Year      PIC 9999 VALUE 2006.
    . . .
    MOVE Group-1 to Edited-date.
```

If Group-1 were instead an alphanumeric group in which each of its subordinate items had USAGE NATIONAL (specified either explicitly with a USAGE NATIONAL clause on each elementary item, or implicitly with a USAGE NATIONAL clause at the group level), a group move, rather than an elementary move, would occur. Neither editing nor conversion would take place during the move. The value in the first eight character positions of Edited-date after the move would be 06232006 in national characters, and the value in the remaining two character positions would be 4 bytes of alphanumeric spaces.

RELATED TASKS
"Assigning values to group data items (MOVE)" on page 35
"Comparing national data and alphanumeric-group operands" on page 140
"Using national groups as group items" on page 132

RELATED REFERENCES
MOVE statement (*Enterprise COBOL Language Reference*)

Using national groups as group items

In some cases when you use a national group, it is handled with group semantics; that is, the elementary items in the group are recognized or processed.

In the following example, an INITIALIZE statement that acts upon national group item Group-OneN causes the value 15 in national characters to be moved to only the numeric items in the group:

```
01  Group-OneN    Group-Usage National.
    05  Trans-codeN     Pic N   Value "A".
    05  Part-numberN    Pic NN  Value "XX".
    05  Trans-quanN     Pic 99  Value 10.
    . . .
    Initialize Group-OneN Replacing Numeric Data By 15
```

Because only Trans-quanN in Group-OneN above is numeric, only Trans-quanN receives the value 15. The other subordinate items are unchanged.

The table below summarizes the cases where national groups are processed with group semantics.

Table 17. **National group items that are processed with group semantics**

Language feature	Uses of national group items	Comment
CORRESPONDING phrase of the ADD, SUBTRACT, or MOVE statement	Specify a national group item for processing as a group in accordance with the rules of the CORRESPONDING phrase.	Elementary items within the national group are processed like elementary items that have USAGE NATIONAL within an alphanumeric group.
Host variable in EXEC SQL statement	Specify a national group item as a host variable.	The national group item is in effect shorthand for the set of host variables that are subordinate to the group item.
INITIALIZE statement	Specify a national group for processing as a group in accordance with the rules of the INITIALIZE statement.	Elementary items within the national group are initialized like elementary items that have USAGE NATIONAL within an alphanumeric group.
Name qualification	Use the name of a national group item to qualify the names of elementary data items and of subordinate group items in the national group.	Follow the same rules for qualification as for an alphanumeric group.
THROUGH phrase of the RENAMES clause	To specify a national group item in the THROUGH phrase, use the same rules as for an alphanumeric group item.	The result is an alphanumeric group item.
FROM phrase of the XML GENERATE statement	Specify a national group item in the FROM phrase for processing as a group in accordance with the rules of the XML GENERATE statement.	Elementary items within the national group are processed like elementary items that have USAGE NATIONAL within an alphanumeric group.

RELATED TASKS
"Initializing a structure (INITIALIZE)" on page 32

"Initializing a table (INITIALIZE)" on page 75
"Assigning values to elementary data items (MOVE)" on page 34
"Assigning values to group data items (MOVE)" on page 35
"Finding the length of data items" on page 118
"Generating XML output" on page 511
"Using national group items in SQL statements" on page 409

RELATED REFERENCES
Qualification (*Enterprise COBOL Language Reference*)
RENAMES clause (*Enterprise COBOL Language Reference*)

Storage of national data

Use the table below to compare alphanumeric (DISPLAY), DBCS (DISPLAY-1), and Unicode (NATIONAL) encoding and to plan storage usage.

Table 18. **Encoding and size of alphanumeric, DBCS, and national data**

Characteristic	DISPLAY	DISPLAY-1	NATIONAL
Character encoding unit	1 byte	2 bytes	2 bytes
Code page[1]	EBCDIC	EBCDIC DBCS	UTF-16BE
Encoding units per graphic character	1	1	1 or 2[2]
Bytes per graphic character	1 byte	2 bytes	2 or 4 bytes

1. Use the CODEPAGE compiler option to specify the EBCDIC code page that is applicable to alphanumeric or DBCS data.
2. Most characters are represented in UTF-16 using one encoding unit. In particular, the following characters are represented using a single UTF-16 encoding unit per character:
   - COBOL characters A-Z, a-z, 0-9, space, + -*/= $,;."()><:'
   - All characters that are converted from an EBCDIC, ASCII, or EUC code page

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 125

Converting to or from national (Unicode) representation

You can implicitly or explicitly convert data items to national (UTF-16) representation.

You can implicitly convert alphabetic, alphanumeric, DBCS, or integer data to national data by using the MOVE statement. Implicit conversions also take place in other COBOL statements, such as IF statements that compare an alphanumeric data item with a data item that has USAGE NATIONAL.

You can explicitly convert to and from national data items by using the intrinsic functions NATIONAL-OF and DISPLAY-OF, respectively. By using these intrinsic functions, you can specify a code page for the conversion that is different from the code page that is in effect with the CODEPAGE compiler option.

RELATED TASKS
"Converting alphanumeric, DBCS, and integer data to national data (MOVE)" on page 134
"Converting alphanumeric and DBCS data to national data (NATIONAL-OF)" on page 135

"Converting national data to alphanumeric data (DISPLAY-OF)" on page 135
"Overriding the default code page" on page 135
"Comparing national (UTF-16) data" on page 138

RELATED REFERENCES
"CODEPAGE" on page 305
"Conversion exceptions" on page 136

Converting alphanumeric, DBCS, and integer data to national data (MOVE)

You can use a MOVE statement to implicitly convert data to national representation.

You can move the following kinds of data to category national or national-edited data items, and thus convert the data to national representation:

- Alphabetic
- Alphanumeric
- Alphanumeric-edited
- DBCS
- Integer of USAGE DISPLAY
- Numeric-edited of USAGE DISPLAY

You can likewise move the following kinds of data to numeric-edited data items that have USAGE NATIONAL:

- Alphanumeric
- Display floating-point (floating-point of USAGE DISPLAY)
- Numeric-edited of USAGE DISPLAY
- Integer of USAGE DISPLAY

For complete rules about moves to national data, see the related reference about the MOVE statement.

For example, the MOVE statement below moves the alphanumeric literal "AB" to the national data item UTF16-Data:

```
01  UTF16-Data  Pic N(2) Usage National.
    . . .
    Move "AB" to UTF16-Data
```

After the MOVE statement above, UTF16-Data contains NX'00410042', the national representation of the alphanumeric characters 'AB'.

If padding is required in a receiving data item that has USAGE NATIONAL, the default UTF-16 space character (NX'0020') is used. If truncation is required, it occurs at the boundary of a national-character position.

RELATED TASKS
"Assigning values to elementary data items (MOVE)" on page 34
"Assigning values to group data items (MOVE)" on page 35
"Displaying numeric data" on page 47
"Coding for use of DBCS support" on page 141

RELATED REFERENCES
MOVE statement (*Enterprise COBOL Language Reference*)

Converting alphanumeric and DBCS data to national data (NATIONAL-OF)

Use the `NATIONAL-OF` intrinsic function to convert alphabetic, alphanumeric, or DBCS data to a national data item. Specify the source code page as the second argument if the source is encoded in a different code page than is in effect with the `CODEPAGE` compiler option.

"Example: converting to and from national data" on page 136

RELATED TASKS
"Processing UTF-8 data" on page 137
"Processing Chinese GB 18030 data" on page 137
"Processing alphanumeric data items that contain DBCS data" on page 143

RELATED REFERENCES
"CODEPAGE" on page 305
NATIONAL-OF (*Enterprise COBOL Language Reference*)

Converting national data to alphanumeric data (DISPLAY-OF)

Use the `DISPLAY-OF` intrinsic function to convert national data to an alphanumeric (`USAGE DISPLAY`) character string that is represented in a code page that you specify as the second argument.

If you omit the second argument, the output code page is the one that was in effect with the `CODEPAGE` compiler option when the source was compiled.

If you specify an EBCDIC or ASCII code page that combines single-byte character set (SBCS) and DBCS characters, the returned string might contain a mixture of SBCS and DBCS characters. The DBCS substrings are delimited by shift-in and shift-out characters if the code page in effect for the function is an EBCDIC code page.

"Example: converting to and from national data" on page 136

RELATED TASKS
"Processing UTF-8 data" on page 137
"Processing Chinese GB 18030 data" on page 137

RELATED REFERENCES
DISPLAY-OF (*Enterprise COBOL Language Reference*)

Overriding the default code page

In some cases, you might need to convert data to or from a code page that differs from the CCSID that is specified as the `CODEPAGE` option value. To do so, convert the item by using a conversion function in which you explicitly specify the code page.

If you specify a code page as an argument to the `DISPLAY-OF` intrinsic function, and the code page differs from the code page that is in effect with the `CODEPAGE` compiler option, do not use the function result in any operations that involve implicit conversion (such as an assignment to, or comparison with, a national data item). Such operations assume the EBCDIC code page that is specified with the `CODEPAGE` compiler option.

RELATED REFERENCES
"CODEPAGE" on page 305

Conversion exceptions

Implicit or explicit conversion between national data and alphanumeric data can fail and generate a severity-3 Language Environment condition.

Failures can occur in any of the following cases:

- Unicode Conversion Services is not installed on your system.
- The code page that you specified implicitly or explicitly is not a valid code page.
- The combination of the CCSID that you specified implicitly or explicitly (such as by using the CODEPAGE compiler option) and the UTF-16 CCSID (1200) is not configured on your system as a valid conversion pair for the Unicode Conversion Services.

A character that does not have a counterpart in the target CCSID does not result in a conversion exception. Such a character is converted to a *substitution character* in the target code page.

RELATED TASKS
Customizing Unicode support for COBOL (*Enterprise COBOL Customization Guide*)

RELATED REFERENCES
"CODEPAGE" on page 305

Example: converting to and from national data

The following example shows the NATIONAL-OF and DISPLAY-OF intrinsic functions and the MOVE statement for converting to and from national (UTF-16) data items. It also demonstrates the need for explicit conversions when you operate on strings that are encoded in multiple code pages.

```
 CBL CODEPAGE(00037)
* . . .
 01  Data-in-Unicode          pic N(100) usage national.
 01  Data-in-Greek            pic X(100).
 01  other-data-in-US-English pic X(12) value "PRICE in $ =".
* . . .
     Read Greek-file into Data-in-Greek
     Move function National-of(Data-in-Greek, 00875)
         to Data-in-Unicode
* . . . process Data-in-Unicode here . . .
     Move function Display-of(Data-in-Unicode, 00875)
         to Data-in-Greek
     Write Greek-record from Data-in-Greek
```

The example above works correctly because the input code page is specified. Data-in-Greek is converted as data represented in CCSID 00875 (Greek). However, the following statement results in an incorrect conversion unless all the characters in the item happen to be among those that have a common representation in both the Greek and the English code pages:

```
Move Data-in-Greek to Data-in-Unicode
```

The MOVE statement above converts Data-in-Greek to Unicode representation based on the CCSID 00037 (U.S. English) to UTF-16 conversion. This conversion does not produce the expected results because Data-in-Greek is encoded in CCSID 00875.

If you can correctly set the CODEPAGE compiler option to CCSID 00875 (that is, the rest of your program also handles EBCDIC data in Greek), you can code the same example correctly as follows:

```
+ CBL CODEPAGE(00875)
* . . .
 01  Data-in-Unicode pic N(100) usage national.
 01  Data-in-Greek   pic X(100).
* . . .
     Read Greek-file into Data-in-Greek
* . . . process Data-in-Greek here ...
* . . . or do the following (if need to process data in Unicode):
     Move Data-in-Greek to Data-in-Unicode
* . . . process Data-in-Unicode
     Move function Display-of(Data-in-Unicode) to Data-in-Greek
     Write Greek-record from Data-in-Greek
```

Processing UTF-8 data

When you need to process UTF-8 data, first convert the data to UTF-16 in a national data item. After processing the national data, convert it back to UTF-8 for output. For the conversions, use the intrinsic functions NATIONAL-OF and DISPLAY-OF, respectively. Use code page 1208 for UTF-8 data.

You need to do two steps to convert ASCII or EBCDIC data to UTF-8:

1. Use the function NATIONAL-OF to convert the ASCII or EBCDIC string to a national (UTF-16) string.
2. Use the function DISPLAY-OF to convert the national string to UTF-8.

The following example converts Greek EBCDIC data to UTF-8:

```
01 Greek-EBCDIC pic X(10) value "αβγδεζηθ".
01 UnicodeString pic N(10).
01 UTF-8-String pic X(20).
    Move function National-of(Greek-EBCDIC, 00875) to UnicodeString
    Move function Display-of(UnicodeString, 01208) to UTF-8-String
```

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 133

Processing Chinese GB 18030 data

GB 18030 is a national-character standard specified by the government of the People's Republic of China.

+ GB 18030 characters can be encoded in either UTF-16 or in code page CCSID 1392.
+ Code page 1392 is an ASCII multibyte code page that uses 1, 2, or 4 bytes per
+ character. A subset of the GB 18030 characters can be encoded in the Chinese ASCII
+ code page, CCSID 1386, or in the Chinese EBCDIC code page, CCSID 1388.

Enterprise COBOL does not have explicit support for GB 18030, but does support the processing of GB 18030 characters in several ways. You can:

- Use DBCS data items to process GB 18030 characters that are represented in CCSID 1388.
- Use national data items to define and process GB 18030 characters that are represented in UTF-16, CCSID 01200.

+ • Process data in any code page (including CCSID 1388 or 1392) by converting the data to UTF-16, processing the UTF-16 data, and then converting the data back to the original code-page representation.

When you need to process Chinese GB 18030 data that requires conversion, first convert the input data to UTF-16 in a national data item. After you process the national data item, convert it back to Chinese GB 18030 for output. For the conversions, use the intrinsic functions NATIONAL-OF and DISPLAY-OF, respectively, and specify code page 1388 or 1392 as the second argument of each function. +

The following example illustrates these conversions:

```
 01 Chinese-EBCDIC pic X(16) value "奥林匹克运动会".
 01 Chinese-GB18030-String pic X(16).
 01 UnicodeString pic N(14).
 . . .
     Move function National-of(Chinese-EBCDIC, 1388) to UnicodeString
*  Process data in Unicode
     Move function Display-of(UnicodeString, 1388) to Chinese-GB18030-String
```

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 133
"Coding for use of DBCS support" on page 141

RELATED REFERENCES
"Storage of national data" on page 133

Comparing national (UTF-16) data

| You can compare national (UTF-16) data, that is, national literals and data items that have USAGE NATIONAL (whether of class national or class numeric), explicitly or implicitly with other kinds of data in relation conditions.

You can code conditional expressions that use national data in the following statements:

- EVALUATE
- IF
- INSPECT
- PERFORM
- SEARCH
- STRING
- UNSTRING

| The following sections provide an overview about comparing national data to other data items. For full details, see the related references.

RELATED TASKS
"Comparing two class national operands" on page 139
"Comparing class national and class numeric operands" on page 139
"Comparing national numeric and other numeric operands" on page 140
"Comparing national character-string and other character-string operands" on page 140
"Comparing national data and alphanumeric-group operands" on page 140

RELATED REFERENCES
Relation conditions (*Enterprise COBOL Language Reference*)
General relation conditions (*Enterprise COBOL Language Reference*)
National comparisons (*Enterprise COBOL Language Reference*)
Group comparisons (*Enterprise COBOL Language Reference*)

Comparing two class national operands

You can compare the character values of two operands of class national.

Either operand (or both) can be any of the following types of items:
- A national group
- An elementary category national or national-edited data item
- A numeric-edited data item that has `USAGE NATIONAL`

One of the operands can instead be a national literal or a national intrinsic function.

When you compare two class national operands that have the same length, they are determined to be equal if all pairs of the corresponding characters are equal. Otherwise, comparison of the binary values of the first pair of unequal characters determines the operand with the larger binary value.

When you compare operands that have unequal lengths, the shorter operand is treated as if it were padded on the right with default UTF-16 space characters (`NX'0020'`) to the length of the longer operand.

The `PROGRAM COLLATING SEQUENCE` clause does not affect the comparison of two class national operands.

RELATED CONCEPTS
"National groups" on page 129

RELATED TASKS
"Using national groups" on page 130

RELATED REFERENCES
National comparisons (*Enterprise COBOL Language Reference*)

Comparing class national and class numeric operands

You can compare national literals or class national data items to integer literals or numeric data items that are defined as integer (that is, national decimal items or zoned decimal items). At most one of the operands can be a literal.

You can also compare national literals or class national data items to floating-point data items (that is, display floating-point or national floating-point items).

Numeric operands are converted to national (UTF-16) representation if they are not already in national representation. A comparison is made of the national character values of the operands.

RELATED REFERENCES
General relation conditions (*Enterprise COBOL Language Reference*)

Comparing national numeric and other numeric operands

National numeric operands (national decimal and national floating-point operands) are data items of class numeric that have `USAGE NATIONAL`.

You can compare the algebraic values of numeric operands regardless of their `USAGE`. Thus you can compare a national decimal item or a national floating-point item with a binary item, an internal-decimal item, a zoned decimal item, a display floating-point item, or any other numeric item.

RELATED TASKS
"Defining national numeric data items" on page 129

RELATED REFERENCES
General relation conditions (*Enterprise COBOL Language Reference*)

Comparing national character-string and other character-string operands

You can compare the character value of a national literal or class national data item with the character value of any of the following other character-string operands: alphabetic, alphanumeric, alphanumeric-edited, DBCS, or numeric-edited of `USAGE DISPLAY`.

These operands are treated as if they were moved to an elementary national data item. The characters are converted to national (UTF-16) representation, and the comparison proceeds with two national character operands.

RELATED TASKS
"Using national-character figurative constants" on page 128

RELATED REFERENCES
National comparisons (*Enterprise COBOL Language Reference*)

Comparing national data and alphanumeric-group operands

You can compare a national literal, a national group item, or any elementary data item that has `USAGE NATIONAL` to an alphanumeric group.

Neither operand is converted. The national operand is treated as if it were moved to an alphanumeric group item of the same size in bytes as the national operand, and the two groups are compared. An alphanumeric comparison is done regardless of the representation of the subordinate items in the alphanumeric group operand.

For example, `Group-XN` is an alphanumeric group that consists of two subordinate items that have `USAGE NATIONAL`:

```
01  Group-XN.
    02 TransCode PIC NN   Value "AB"  Usage National.
    02 Quantity  PIC 999  Value 123   Usage National.
    . . .
    If N"AB123" = Group-XN  Then Display "EQUAL"
    Else Display "NOT EQUAL".
```

When the `IF` statement above is executed, the 10 bytes of the national literal `N"AB123"` are compared byte by byte to the content of `Group-XN`. The items compare equally, and "`EQUAL`" is displayed.

RELATED REFERENCES
Group comparisons (*Enterprise COBOL Language Reference*)

Coding for use of DBCS support

IBM Enterprise COBOL for z/OS supports using applications in any of many national languages, including languages that use double-byte character sets (DBCS).

The following list summarizes the support for DBCS:

- DBCS characters in user-defined words (DBCS names)
- DBCS characters in comments
- DBCS data items (defined with `PICTURE N`, `G`, or `G` and `B`)
- DBCS literals
- `DBCS` compiler option

RELATED TASKS
"Declaring DBCS data"
"Using DBCS literals"
"Testing for valid DBCS characters" on page 142
"Processing alphanumeric data items that contain DBCS data" on page 143
Appendix C, "Converting double-byte character set (DBCS) data," on page 663

RELATED REFERENCES
"DBCS" on page 309

Declaring DBCS data

Use the `PICTURE` and `USAGE` clauses to declare DBCS data items. DBCS data items can use `PICTURE` symbols `G`, `G` and `B`, or `N`. Each DBCS character position is 2 bytes in length.

You can specify a DBCS data item by using the `USAGE DISPLAY-1` clause. When you use `PICTURE` symbol `G`, you must specify `USAGE DISPLAY-1`. When you use `PICTURE` symbol `N` but omit the `USAGE` clause, `USAGE DISPLAY-1` or `USAGE NATIONAL` is implied depending on the setting of the `NSYMBOL` compiler option.

If you use a `VALUE` clause with the `USAGE` clause in the declaration of a DBCS item, you must specify a DBCS literal or the figurative constant `SPACE` or `SPACES`.

For the purpose of handling reference modifications, each character in a DBCS data item is considered to occupy the number of bytes that corresponds to the code-page width (that is, 2).

RELATED REFERENCES
"NSYMBOL" on page 323

Using DBCS literals

You can use the prefix `N` or `G` to represent a DBCS literal.

That is, you can specify a DBCS literal in either of these ways:

- `N`'*dbcs characters*' (provided that the compiler option `NSYMBOL(DBCS)` is in effect)
- `G`'*dbcs characters*'

You can use quotation marks (") or single quotation marks (') as the delimiters of a DBCS literal irrespective of the setting of the `APOST` or `QUOTE` compiler option. You must code the same opening and closing delimiter for a DBCS literal.

The shift-out (SO) control character `X'0E'` must immediately follow the opening delimiter, and the shift-in (SI) control character `X'0F'` must immediately precede the closing delimiter.

In addition to DBCS literals, you can use alphanumeric literals to specify any character in one of the supported code pages. However, any string of DBCS characters that is within an alphanumeric literal must be delimited by the SO and SI characters, and the `DBCS` compiler option must be in effect for the SO and SI characters to be recognized as shift codes.

You cannot continue an alphanumeric literal that contains DBCS characters. The length of a DBCS literal is likewise limited by the available space in Area B on a single source line. The maximum length of a DBCS literal is thus 28 double-byte characters.

An alphanumeric literal that contains DBCS characters is processed byte by byte, that is, with semantics appropriate for single-byte characters, except when it is converted explicitly or implicitly to national data representation, as for example in an assignment to or comparison with a national data item.

RELATED TASKS
"Using figurative constants" on page 28

RELATED REFERENCES
"DBCS" on page 309
"NSYMBOL" on page 323
"QUOTE/APOST" on page 331
DBCS literals (*Enterprise COBOL Language Reference*)

Testing for valid DBCS characters

The Kanji class test tests for valid Japanese graphic characters. This testing includes Katakana, Hiragana, Roman, and Kanji character sets.

The Kanji class test is done by checking characters for the range `X'41'` through `X'7E'` in the first byte and `X'41'` through `X'FE'` in the second byte, plus the space character `X'4040'`.

The DBCS class test tests for valid graphic characters for the code page.

The DBCS class test is done by checking characters for the range `X'41'` through `X'FE'` in both the first and second byte of each character, plus the space character `X'4040'`.

RELATED TASKS
"Coding conditional expressions" on page 94

RELATED REFERENCES
Class condition (*Enterprise COBOL Language Reference*)

Processing alphanumeric data items that contain DBCS data

If you use byte-oriented operations (for example, STRING, UNSTRING, or reference modification) on an alphanumeric data item that contains DBCS characters, results are unpredictable. You should instead convert the item to a national data item before you process it.

That is, do these steps:

1. Convert the item to UTF-16 in a national data item by using a MOVE statement or the NATIONAL-OF intrinsic function.
2. Process the national data item as needed.
3. Convert the result back to an alphanumeric data item by using the DISPLAY-OF intrinsic function.

RELATED TASKS
"Joining data items (STRING)" on page 101
"Splitting data items (UNSTRING)" on page 103
"Referring to substrings of data items" on page 107
"Converting to or from national (Unicode) representation" on page 133

Chapter 8. Processing files

Reading and writing data is an essential part of every program. Your program retrieves information, processes it as you request, and then produces the results.

The source of the information and the target for the results can be one or more of the following items:
- Another program
- Direct-access storage device
- Magnetic tape
- Printer
- Terminal
- Card reader or punch

The information as it exists on an external device is in a physical record or block, a collection of information that is handled as a unit by the system during input or output operations.

Your COBOL program does not directly handle physical records. It processes logical records. A logical record can correspond to a complete physical record, part of a physical record, or to parts or all of one or more physical records. Your COBOL program handles logical records exactly as you have defined them.

In COBOL, a collection of logical records is a file, a sequence of pieces of information that your program can process.

RELATED CONCEPTS
"File organization and input-output devices"

RELATED TASKS
"Choosing file organization and access mode" on page 147
"Allocating files" on page 149
"Checking for input or output errors" on page 150

File organization and input-output devices

Depending on the input-output devices, your file organization can be sequential, line sequential, indexed, or relative. Decide on the file types and devices to be used when you design your program.

You have the following choices of file organization:

Sequential file organization
The chronological order in which records are entered when a file is created establishes the arrangement of the records. Each record except the first has a unique predecessor record, and each record except the last has a unique successor record. Once established, these relationships do not change.

The access (record transmission) mode allowed for sequential files is sequential only.

Line-sequential file organization

Line-sequential files are sequential files that reside on the hierarchical file system (HFS) and that contain only characters as data. Each record ends with a new-line character.

The only access (record transmission) mode allowed for line-sequential files is sequential.

Indexed file organization

Each record in the file contains a special field whose contents form the record key. The position of the key is the same in each record. The index component of the file establishes the logical arrangement of the file, an ordering by record key. The actual physical arrangement of the records in the file is not significant to your COBOL program.

An indexed file can also use alternate indexes in addition to the record key. These keys let you access the file using a different logical ordering of the records.

The access (record transmission) modes allowed for indexed files are sequential, random, or dynamic. When you read or write indexed files sequentially, the sequence is that of the key values.

Relative file organization

Records in the file are identified by their location relative to the beginning of the file. 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 on.

The access (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.

With IBM Enterprise COBOL for z/OS, requests to the operating system for the storage and retrieval of records from input-output devices are handled by the two access methods QSAM and VSAM, and the UNIX file system.

The device type upon which you elect to store your data could affect the choices of file organization available to you. Direct-access storage devices provide greater flexibility in the file organization options. Sequential-only devices limit organization options but have other characteristics, such as the portability of tapes, that might be useful.

Sequential-only devices

Terminals, printers, card readers, and punches are called *unit-record devices* because they process one line at a time. Therefore, you must also process records one at a time sequentially in your program when it reads from or writes to unit-record devices.

On tape, records are ordered sequentially, so your program must process them sequentially. Use QSAM physical sequential files when processing tape files. The records on tape can be fixed length or variable length. The rate of data transfer is faster than it is for cards.

Direct-access storage devices

Direct-access storage devices hold many records. The record arrangement of files stored on these devices determines the ways that your program can process the data. When using direct-access devices, you have greater flexibility within your program, because your can use several types of file organization:

- Sequential (VSAM or QSAM)

- Line sequential (UNIX)
- Indexed (VSAM)
- Relative (VSAM)

RELATED TASKS
"Allocating files" on page 149
Chapter 9, "Processing QSAM files," on page 151
Chapter 10, "Processing VSAM files," on page 179
Chapter 11, "Processing line-sequential files," on page 207
"Choosing file organization and access mode"

Choosing file organization and access mode

There are several guidelines you can use to determine which file organization and access mode to use in an application.

Consider the following guidelines when choosing file organization:
- If an application accesses records (whether fixed-length or variable-length) only sequentially and does not insert records between existing records, a QSAM or VSAM sequential file is the simplest type.
- If you are developing an application for UNIX that sequentially accesses records that contain only printable characters and certain control characters, line-sequential files work best.
- If an application requires both sequential and random access (whether records are fixed length or variable length), a VSAM indexed file is the most flexible type.
- If an application inserts and deletes records randomly, a relative file works well.

Consider the following guidelines when choosing access mode:
- If a large percentage of a file is referenced or updated in an application, sequential access is faster than random or dynamic access.
- If a small percentage of records is processed during each run of an application, use random or dynamic access.

Table 19. **Summary of file organizations, access modes, and record formats of COBOL files**

File organization	Sequential access	Random access	Dynamic access	Fixed length	Variable length
QSAM (physical sequential)	X			X	X
Line sequential	X			X[1]	X
VSAM sequential (ESDS)	X			X	X
VSAM indexed (KSDS)	X	X	X	X	X
VSAM relative (RRDS)	X	X	X	X	X

1. The data itself is in variable format but can be read into and written from COBOL fixed-length records.

RELATED REFERENCES
"Format for coding input and output" on page 148
"Allowable control characters" on page 208

Format for coding input and output

The following code shows the general format of input-output coding. Explanations of the user-supplied information follow the code.

```
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.
```

The user-supplied information in the code above is as follows:

(1) ***filename***
Any legal COBOL name. You must use the same file-name in the SELECT clause and in the FD entry, and on the READ, OPEN, and CLOSE statements. In addition, the file-name is required if you use the START or DELETE statements. This name is not necessarily the actual name of the data set as known to the system. Each file requires its own SELECT clause, FD entry, and input-output statements.

(2) ***assignment-name***
Any name you choose, provided that it follows COBOL and system naming rules. The name can be 1-30 characters long if it is a user-defined word, or 1-160 characters long if it is a literal. You code the *name* part of the *assignment-name* on a DD statement, in an ALLOCATE command (TSO) or as an environment variable (for example, in an export command) (UNIX).

(3) ***org*** The organization can be SEQUENTIAL, LINE SEQUENTIAL, INDEXED, or RELATIVE. This clause is optional for QSAM files.

(4) ***access***
The access mode can be SEQUENTIAL, RANDOM, or DYNAMIC. For sequential file processing, including line-sequential, you can omit this clause.

(5) ***file-status***
| The COBOL file status key. You can specify the file status key as a
| two-character category alphanumeric or category national item, or as a
| two-digit zoned decimal (USAGE DISPLAY) or national decimal (USAGE
| NATIONAL) item.

(6) ***recordname***
The name of the record used in the `WRITE` and `REWRITE` statements.

(7) ***fieldlength***
The logical length of the field.

(8) ***type***
The record format of the file. If you break the record entry beyond the level-01 description, each element should map accurately against the fields in the record.

(9) ***iomode***
The `INPUT` or `OUTPUT` mode. If you are only reading from a file, code `INPUT`. If you are only writing to it, code `OUTPUT` or `EXTEND`. If you are both reading and writing, code `I-O`, except for organization `LINE SEQUENTIAL`.

RELATED TASKS
Chapter 9, "Processing QSAM files," on page 151
Chapter 10, "Processing VSAM files," on page 179
Chapter 11, "Processing line-sequential files," on page 207

Allocating files

For any type of file (sequential, line sequential, indexed, or relative) in your z/OS or UNIX applications, you can define the external name with either a ddname or an environment variable name. The external name is the name in the *assignment-name* of the `ASSIGN` clause.

If the file is in the HFS, you can use either a `DD` definition or an environment variable to define the file by specifying its path name with the `PATH` keyword.

The environment variable name must be uppercase. The allowable attributes for its value depend on the organization of the file being defined.

Because you can define the external name in either of two ways, the COBOL run time goes through the following steps to find the definition of the file:

1. If the ddname is explicitly allocated, it is used. The definition can be from a `DD` statement in JCL, an `ALLOCATE` command from TSO/E, or a user-initiated dynamic allocation.
2. If the ddname is not explicitly allocated and an environment variable of the same name is set, the value of the environment variable is used.

   The file is dynamically allocated using the attributes specified by the environment variable. At a minimum, you must specify either the `PATH()` or `DSN()` option. All options and attributes must be in uppercase, except for the *path-name* suboption of the `PATH` option, which is case sensitive. You cannot specify a temporary data set name in the `DSN()` option.

   File status code 98 results from any of the following:

   - The contents (including a value of null or all blanks) of the environment variable are not valid.
   - The dynamic allocation of the file fails.
   - The dynamic deallocation of the file fails.

   The COBOL run time checks the contents of the environment variable at each `OPEN` statement. If a file with the same external name was dynamically allocated by a previous `OPEN` statement, and the contents of the environment variable have changed since that `OPEN`, the run time dynamically deallocates the previous allocation and reallocates the file using the options currently set in the environment variable. If the contents of the environment variable have not changed, the run time uses the current allocation.

3. If neither a ddname nor an environment variable is defined, the following steps occur:
   a. If the allocation is for a QSAM file and the `CBLQDA` runtime option is in effect, `CBLQDA` dynamic allocation processing takes place for those eligible files. This type of "implicit" dynamic allocation persists for the life of the run unit and cannot be reallocated.
   b. Otherwise, the allocation fails.

The COBOL run time deallocates all dynamic allocations at run unit termination, except for implicit `CBLQDA` allocations.

RELATED TASKS
"Setting and accessing environment variables" on page 424
"Defining and allocating QSAM files" on page 166
"Dynamically creating QSAM files with CBLQDA" on page 162
"Allocating VSAM files" on page 201

Checking for input or output errors

After each input or output statement is performed, the file status key is updated with a value that indicates the success or failure of the operation.

Using a `FILE STATUS` clause, test the file status key after each input or output statement, and call an error-handling procedure if a nonzero file status code is returned. With VSAM files, you can use a second data item in the `FILE STATUS` clause to get additional VSAM status code information.

Another way of handling errors in input and output operations is to code `ERROR` (synonymous with `EXCEPTION`) declaratives.

RELATED TASKS
"Handling errors in input and output operations" on page 235
"Coding ERROR declaratives" on page 238
"Using file status keys" on page 239

Chapter 9. Processing QSAM files

Queued sequential access method (QSAM) files are unkeyed files in which the records are placed one after another, according to entry order.

Your program can process these files only sequentially, retrieving (with the READ statement) records in the same order as they are in the file. Each record is placed after the preceding record. To process QSAM files in your program, use COBOL language statements that:

- Identify and describe the QSAM files in the ENVIRONMENT DIVISION and the DATA DIVISION.
- Process the records in these files in the PROCEDURE DIVISION.

After you have created a record, you cannot change its length or its position in the file, and you cannot delete it. You can, however, update QSAM files on direct-access storage devices (using REWRITE), though not in the HFS.

QSAM files can be on tape, direct-access storage devices (DASDs), unit-record devices, and terminals. QSAM processing is best for tables and intermediate storage.

You can also access byte-stream files in the HFS using QSAM. These files are binary byte-oriented sequential files with no record structure. The record definitions that you code in your COBOL program and the length of the variables that you read into and write from determine the amount of data transferred.

RELATED CONCEPTS
"Labels for QSAM files" on page 173
Access methods (*z/OS DFSMS: Using Data Sets*)

RELATED TASKS
"Defining QSAM files and records in COBOL"
"Coding input and output statements for QSAM files" on page 161
"Handling errors in QSAM files" on page 165
"Working with QSAM files" on page 166
"Processing QSAM ASCII files on tape" on page 176
"Processing ASCII file labels" on page 177

Defining QSAM files and records in COBOL

Use the FILE-CONTROL entry to define the files in a COBOL program as QSAM files, and to associate the files with their external file-names.

An *external file-name* (a ddname or environment variable name) is the name by which a file is known to the operating system. In the following example, COMMUTER-FILE-MST is your program's name for the file; COMMUTR is the external name:

```
FILE-CONTROL.
    SELECT COMMUTER-FILE-MST
    ASSIGN TO S-COMMUTR
    ORGANIZATION IS SEQUENTIAL
    ACCESS MODE IS SEQUENTIAL.
```

The ASSIGN clause *name* can include an S- before the external name to document that the file is a QSAM file. Both the ORGANIZATION and ACCESS MODE clauses are optional.

RELATED TASKS
"Establishing record formats"
"Setting block sizes" on page 159

Establishing record formats

In the FD entry in the DATA DIVISION, code the record format and indication of whether the records are blocked. In the associated record description entry or entries, specify the *record-name* and record length.

You can code a record format of F, V, S, or U in the RECORDING MODE clause. COBOL determines the record format from the RECORD clause or from the record descriptions associated with the FD entry for the file. If you want the records to be blocked, code the BLOCK CONTAINS clause in the FD entry.

The following example shows how the FD entry might look for a file that has fixed-length records:

```
FILE SECTION.
FD  COMMUTER-FILE-MST
    RECORDING MODE IS F
    BLOCK CONTAINS 0 RECORDS
    RECORD CONTAINS 80 CHARACTERS.
01  COMMUTER-RECORD-MST.
    05  COMMUTER-NUMBER        PIC  X(16).
    05  COMMUTER-DESCRIPTION   PIC  X(64).
```

A recording mode of S is not supported for files in the HFS. The above example is appropriate for such a file.

RELATED CONCEPTS
"Logical records"

RELATED TASKS
"Requesting fixed-length format" on page 153
"Requesting variable-length format" on page 154
"Requesting spanned format" on page 156
"Requesting undefined format" on page 158
"Defining QSAM files and records in COBOL" on page 151

RELATED REFERENCES
"FILE SECTION entries" on page 14

Logical records

COBOL uses the term *logical record* in a slightly different way than z/OS QSAM.

For format-V and format-S files, a QSAM logical record includes a 4-byte prefix in front of the user data portion of the record that is not included in the definition of a COBOL logical record.

For format-F and format-U files, and for HFS byte-stream files, the definitions of QSAM logical record and COBOL logical record are identical.

In this information, *QSAM logical record* refers to the QSAM definition, and *logical record* refers to the COBOL definition.

RELATED REFERENCES
"Layout of format-F records"
"Layout of format-V records" on page 155
"Layout of format-S records" on page 157
"Layout of format-U records" on page 159

Requesting fixed-length format

Fixed-length records are in format F. Use `RECORDING MODE F` to explicitly request this format.

You can omit the `RECORDING MODE` clause. The compiler determines the recording mode to be `F` if the length of the largest level-01 record associated with the file is not greater than the block size coded in the `BLOCK CONTAINS` clause, and you take one of the following actions:

| • Use the `RECORD CONTAINS` *integer* clause (format-1 `RECORD` clause) to indicate the
| length of the record in bytes.

  When you use this clause, the file is always fixed format with record length *integer* even if there are multiple level-01 record description entries with different lengths associated with the file.
- Omit the `RECORD CONTAINS` *integer* clause, but code the same fixed size and no `OCCURS DEPENDING ON` clause for all level-01 record description entries associated with the file. This fixed size is the record length.

In an unblocked format-F file, the logical record is the same as the block.

In a blocked format-F file, the number of logical records in a block (the *blocking factor*) is constant for every block in the file except the last block, which might be shorter.

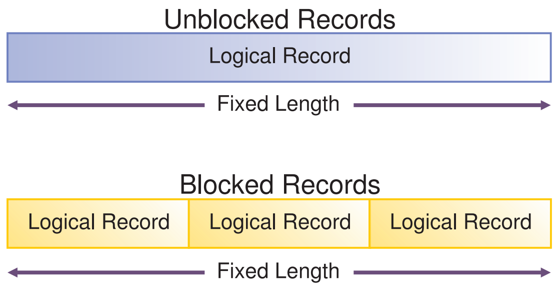
Files in the HFS are never blocked.

RELATED CONCEPTS
"Logical records" on page 152

RELATED TASKS
"Requesting variable-length format" on page 154
"Requesting spanned format" on page 156
"Requesting undefined format" on page 158
"Establishing record formats" on page 152

RELATED REFERENCES
"Layout of format-F records"

Layout of format-F records: The layout of format-F QSAM records is shown below.



RELATED CONCEPTS
"Logical records" on page 152

RELATED TASKS
"Requesting fixed-length format" on page 153
Fixed-length record formats (*z/OS DFSMS: Using Data Sets*)

RELATED REFERENCES
"Layout of format-V records" on page 155
"Layout of format-S records" on page 157
"Layout of format-U records" on page 159

Requesting variable-length format

Variable-length records can be in format V or format D. Format-D records are variable-length records on ASCII tape files. Format-D records are processed in the same way as format-V records.

Use `RECORDING MODE V` for both. You can omit the `RECORDING MODE` clause. The compiler determines the recording mode to be `V` if the largest level-01 record associated with the file is not greater than the block size set in the `BLOCK CONTAINS` clause, and you take one of the following actions:

- Use the `RECORD IS VARYING` clause (format-3 `RECORD` clause).

  If you provide values for *integer-1* and *integer-2* (`RECORD IS VARYING FROM` *integer-1* `TO` *integer-2*), the maximum record length is the value coded for *integer-2* regardless of the lengths coded in the level-01 record description entries associated with the file. The integer sizes indicate the minimum and maximum record lengths in numbers of bytes regardless of the `USAGE` of the data items in the record.

  If you omit *integer-1* and *integer-2*, the maximum record length is determined to be the size of the largest level-01 record description entry associated with the file.
- Use the `RECORD CONTAINS` *integer-1* `TO` *integer-2* clause (format-2 `RECORD` clause). Make *integer-1* and *integer-2* match the minimum length and the maximum length in bytes of the level-01 record description entries associated with the file. The maximum record length is the *integer-2* value.
- Omit the `RECORD` clause, but code multiple level-01 records (associated with the file) that are of different sizes or contain an `OCCURS DEPENDING ON` clause.

  The maximum record length is determined to be the size of the largest level-01 record description entry associated with the file.

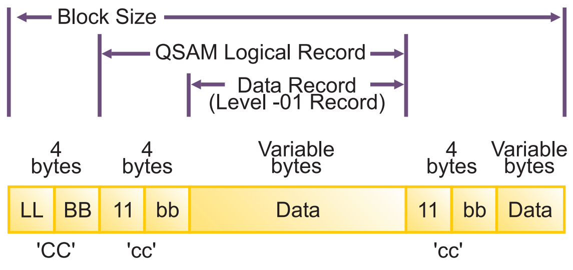
When you specify a `READ INTO` statement for a format-V file, the record size read for that file is used in the `MOVE` statement generated by the compiler. Consequently, you might not get the result you expect if the record just read does not correspond to the level-01 record description. All other rules of the `MOVE` statement apply. For example, when you specify a `MOVE` statement for a format-V record read in by the `READ` statement, the size of the record moved corresponds to its level-01 record description.

When you specify a `READ` statement for a format-V file followed by a `MOVE` of the level-01 record, the actual record length is not used. The program will attempt to move the number of bytes described by the level-01 record description. If this number exceeds the actual record length and extends outside the area addressable by the program, results are unpredictable. If the number of bytes described by the level-01 record description is shorter than the physical record read, truncation of bytes beyond the level-01 description occurs. To find the actual length of a variable-length record, specify *data-name-1* in format 3 of the RECORD clause of the File Definition (FD).

RELATED TASKS
"Requesting fixed-length format" on page 153
"Requesting spanned format" on page 156
"Requesting undefined format" on page 158
"Establishing record formats" on page 152

RELATED REFERENCES
"FILE SECTION entries" on page 14
"Layout of format-V records"
Moving from the VS COBOL II run time (*Enterprise COBOL Compiler and Runtime Migration Guide*)

Layout of format-V records: Format-V QSAM records have control fields that precede the data. The QSAM logical record length is determined by adding 4 bytes (for the control fields) to the record length defined in your program, but you must not include these 4 bytes in the description of the record and record length.



CC The first 4 bytes of each block contain control information.

LL Represents 2 bytes designating the length of the block (including the 'CC' field).

BB Represents 2 bytes reserved for system use.

cc The first 4 bytes of each logical record contain control information.

11 Represents 2 bytes designating the logical record length (including the 'cc' field).

bb Represents 2 bytes reserved for system use.

The block length is determined as follows:

- Unblocked format-V records: CC + cc + the data portion
- Blocked format-V records: CC + the cc of each record + the data portion of each record

The operating system provides the control bytes when the file is written; the control byte fields do not appear in your description of the logical record in the DATA DIVISION of your program. COBOL allocates input and output buffers large enough to accommodate the control bytes. These control fields in the buffer are not available for you to use in your program. When variable-length records are written on unit record devices, control bytes are neither printed nor punched. They appear, however, on other external storage devices, as well as in buffer areas of storage. If you move V-mode records from an input buffer to a WORKING-STORAGE area, they'll be moved without the control bytes.

Files in the HFS are never blocked.

RELATED CONCEPTS
"Logical records" on page 152

RELATED TASKS
"Requesting variable-length format" on page 154

RELATED REFERENCES
"Layout of format-F records" on page 153
"Layout of format-S records" on page 157
"Layout of format-U records" on page 159

Requesting spanned format

Spanned records are in format S. A *spanned record* is a QSAM logical record that can be contained in one or more physical blocks.

You can code RECORDING MODE S for spanned records in QSAM files that are assigned to magnetic tape or to direct access devices. Do not request spanned records for files in the HFS. You can omit the RECORDING MODE clause. The compiler determines the recording mode to be S if the maximum record length (in bytes) plus 4 is greater than the block size set in the BLOCK CONTAINS clause.

For files with format S in your program, the compiler determines the maximum record length with the same rules as are used for format V. The length is based on your usage of the RECORD clause.

When creating files that contain format-S records and a record is larger than the remaining space in a block, COBOL writes a segment of the record to fill the block. The rest of the record is stored in the next block or blocks depending on its length. COBOL supports QSAM spanned records up to 32,760 bytes in length.

When retrieving files that have format-S records, a program can retrieve only complete records.

Benefits of format-S files: You can efficiently use external storage and still organize your files with logical record lengths by defining files with format-S records:

- You can set block lengths to efficiently use track capacities on direct access devices.
- You are not required to adjust the logical record lengths to device-dependent physical block lengths. One logical record can span two or more physical blocks.
- You have greater flexibility when you want to transfer logical records between direct access storage types.

You will, however, have additional overhead in processing format-S files.

Format-S files and READ INTO: When you specify a READ INTO statement for a format-S file, the compiler generates a MOVE statement that uses the size of the record that it just read for that file. If the record just read does not correspond to the level-01 record description, you might not get the result that you expect. All other rules of the MOVE statement apply.

RELATED CONCEPTS
"Logical records" on page 152
"Spanned blocked and unblocked files"

RELATED TASKS
"Requesting fixed-length format" on page 153
"Requesting variable-length format" on page 154
"Requesting undefined format" on page 158
"Establishing record formats" on page 152

RELATED REFERENCES
"FILE SECTION entries" on page 14
"Layout of format-S records"

Spanned blocked and unblocked files: A spanned blocked QSAM file is made up of blocks, each containing one or more logical records or segments of logical records. A spanned unblocked file is made up of physical blocks, each containing one logical record or one segment of a logical record.

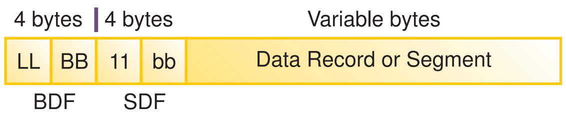
In a spanned blocked file, a logical record can be either fixed or variable in length, and its size can be smaller than, equal to, or larger than the physical block size. There are no required relationships between logical records and physical block sizes.

In a spanned unblocked file, the logical records can be either fixed or variable in length. When the physical block contains one logical record, the block length is determined by the logical record size. When a logical record has to be segmented, the system always writes the largest physical block possible. The system segments the logical record when the entire logical record cannot fit on a track.

RELATED CONCEPTS
"Logical records" on page 152

RELATED TASKS
"Requesting spanned format" on page 156

Layout of format-S records: Spanned records are preceded by control fields, as shown below.



Each block is preceded by a 4-byte block descriptor field ('BDF' in the image). There is only one block descriptor field at the beginning of each physical block.

Each segment of a record in a block, even if the segment is the entire record, is preceded by a 4-byte segment descriptor field ('SDF' in the image). There is one segment descriptor field for each record segment in the block. The segment descriptor field also indicates whether the segment is the first, the last, or an intermediate segment.

You do not describe these fields in the DATA DIVISION of your COBOL program, and the fields are not available for you to use in your program.

RELATED TASKS
"Requesting spanned format" on page 156

RELATED REFERENCES
"Layout of format-F records" on page 153
"Layout of format-V records" on page 155
"Layout of format-U records" on page 159

Requesting undefined format

Format-U records have undefined or unspecified characteristics. With format U, you can process blocks that do not meet format-F or format-V specifications.

When you use format-U files, each block of storage is one logical record. A read of a format-U file returns the entire block as a record. A write to a format-U file writes a record out as a block. The compiler determines the recording mode to be U only if you code RECORDING MODE U.

It is recommended that you not use format U to update or extend a file that was written with a different record format. If you use format U to update a file that was written with a different format, the RECFM value in the data-set label could be changed or the data set could contain records written in different formats.

The record length is determined in your program based on how you use the RECORD clause:

- If you use the RECORD CONTAINS *integer* clause (format-1 RECORD clause), the record length is the *integer* value regardless of the lengths of the level-01 record description entries associated with the file. The integer size indicates the number of bytes in a record regardless of the USAGE of its data items.
- If you use the RECORD IS VARYING clause (format-3 RECORD clause), the record length is determined based on whether you code *integer-1* and *integer-2*.

  If you code *integer-1* and *integer-2* (RECORD IS VARYING FROM *integer-1* TO *integer-2*), the maximum record length is the *integer-2* value regardless of the lengths of the level-01 record description entries associated with the file. The integer sizes indicate the minimum and maximum record lengths in numbers of bytes regardless of the USAGE of the data items in the record.

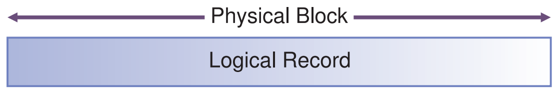
  If you omit *integer-1* and *integer-2*, the maximum record length is determined to be the size of the largest level-01 record description entry associated with the file.
- If you use the RECORD CONTAINS *integer-1* TO *integer-2* clause (format-2 RECORD clause), with *integer-1* and *integer-2* matching the minimum length and the maximum length in bytes of the level-01 record description entries associated with the file, the maximum record length is the *integer-2* value.
- If you omit the RECORD clause, the maximum record length is determined to be the size of the largest level-01 record description entry associated with the file.

Format-U files and READ INTO: When you specify a READ INTO statement for a format-U file, the compiler generates a MOVE statement that uses the size of the record that it just read for that file. If the record just read does not correspond to the level-01 record description, you might not get the result that you expect. All other rules of the MOVE statement apply.

RELATED TASKS
"Requesting fixed-length format" on page 153
"Requesting variable-length format" on page 154
"Requesting spanned format" on page 156
"Establishing record formats" on page 152

RELATED REFERENCES
"FILE SECTION entries" on page 14
"Layout of format-U records"

Layout of format-U records: With format-U, each block of external storage is handled as a logical record. There are no record-length or block-length fields.



RELATED CONCEPTS
"Logical records" on page 152

RELATED TASKS
"Requesting undefined format" on page 158

RELATED REFERENCES
"Layout of format-F records" on page 153
"Layout of format-V records" on page 155
"Layout of format-S records" on page 157

Setting block sizes

In COBOL, you establish the size of a physical record by using the BLOCK CONTAINS clause. If you omit this clause, the compiler assumes that the records are not blocked.

Blocking QSAM files on tape and disk can enhance processing speed and minimize storage requirements. You can block z/OS UNIX System Services files (including those in the HFS), PDSE members, and spooled data sets, but doing so has no effect on how the system stores the data.

If you set the block size explicitly in the BLOCK CONTAINS clause, the size must not be greater than the maximum block size for the device. If you specify the CHARACTERS phrase of the BLOCK CONTAINS clause, size must indicate the number of bytes in a record regardless of the USAGE of the data items in the record. The block size that is set for a format-F file must be an integral multiple of the record length.

If your program uses QSAM files on tape, use a physical block size of at least 12 to 18 bytes. Otherwise, the block will be skipped over when a parity check occurs during one of the following actions:
- Reading a block of records of fewer than 12 bytes
- Writing a block of records of fewer than 18 bytes

Larger blocks generally give you better performance. Blocks of only a few kilobytes are particularly inefficient; you should choose a block size of at least tens of kilobytes. If you specify record blocking and omit the block size, the system will pick a block size that is optimal for device utilization and for data transfer speed.

Letting z/OS determine block size: To maximize performance, do not explicitly set the block size for a blocked file in your COBOL source program. For new blocked data sets, it is simpler to allow z/OS to supply a system-determined block size. To use this feature, follow these guidelines:

- Code `BLOCK CONTAINS 0` in your source program.
- Do not code `RECORD CONTAINS 0` in your source program.
- Do not code a `BLKSIZE` value in the JCL `DD` statement.

Setting block size explicitly: If you prefer to set a block size explicitly, your program will be most flexible if you follow these guidelines:

- Code `BLOCK CONTAINS 0` in your source program.
- Code a `BLKSIZE` value in the ddname definition (the JCL `DD` statement).

For extended-format data sets on z/OS, z/OS DFSMS adds a 32-byte block suffix to the physical record. If you specify a block size explicitly (using JCL or ISPF), do not include the size of this block suffix in the block size. This block suffix is not available for you to use in your program. z/OS DFSMS allocates the space used to read in the block suffix. However, when you calculate how many blocks of an extended-format data set will fit on a track of a direct-access device, you need to include the size of the block suffix in the block size.

If you specify a block size that is larger than 32760 directly in the `BLOCK CONTAINS` clause or indirectly with the use of `BLOCK CONTAINS` *n* `RECORDS`, the `OPEN` of the data set fails with file status code 90 unless you define the data set to be on tape.

For existing blocked data sets, it is simplest to:

- Code `BLOCK CONTAINS 0` in your source program.
- Not code a `BLKSIZE` value in the ddname definition.

When you omit the `BLKSIZE` from the ddname definition, the block size is automatically obtained by the system from the data-set label.

Taking advantage of LBI: You can improve the performance of tape data sets by using the large block interface (LBI) for large block sizes. When the LBI is available, the COBOL run time automatically uses this facility for those tape files for which you use system-determined block size. LBI is also used for those files for which you explicitly define a block size in JCL or a `BLOCK CONTAINS` clause. Use of the LBI allows block sizes to exceed 32760 if the tape device supports it.

The LBI is not used in all cases. An attempt to use a block size greater than 32760 in the following cases is diagnosed at compile time or results in a failure at `OPEN`:

- Spanned records
- `OPEN I-O`

Using a block size that exceeds 32760 might result in your not being able to read the tape on another system. A tape that you create with a block size greater than 32760 can be read only on a system that has a tape device that supports block sizes greater than 32760. If you specify a block size that is too large for the file, the device, or the operating system level, a runtime message is issued.

To limit a system-determined block size to 32760, do not specify `BLKSIZE` anywhere, and set one of the following items to 32760:

- The `BLKSZLIM` keyword on the `DD` statement for the data set

- BLKSZLIM for the data class by using the BLKSZLIM keyword (must be set by your systems programmer)
- A block-size limit for the system in the DEVSUP*xx* member of SYS1.PARMLIB by using the keyword TAPEBLKSZLIM (must be set by your systems programmer)

The block-size limit is the first nonzero value that the compiler finds by checking these items.

If no BLKSIZE or BLKSZLIM value is available from any source, the system limits BLKSIZE to 32760. You can then enable block sizes larger than 32760 in one of two ways:

- Specify a BLKSZLIM value greater than 32760 in the DD statement for the file and use BLOCK CONTAINS 0 in your COBOL source.
- Specify a value greater than 32760 for the BLKSIZE in the DD statement or in the BLOCK CONTAINS clause in your COBOL source.

BLKSZLIM is device-independent.

Block size and the DCB RECFM subparameter: Under z/OS, you can code the S or T option in the DCB RECFM subparameter:

- Use the S (standard) option in the DCB RECFM subparameter for a format-F record with only standard blocks (ones that have no truncated blocks or unfilled tracks in the file, except for the last block of the file). S is also supported for records on tape. It is ignored if the records are not on DASD or tape.

  Using this standard block option might improve input-output performance, especially for direct-access devices.
- The T (track overflow) option for QSAM files is no longer useful.

RELATED TASKS
"Defining QSAM files and records in COBOL" on page 151
z/OS DFSMS: Using Data Sets

RELATED REFERENCES
"FILE SECTION entries" on page 14
BLOCK CONTAINS clause (*Enterprise COBOL Language Reference*)

Coding input and output statements for QSAM files

You can code the following input and output statements to process a QSAM file or a byte-stream file in the HFS using QSAM: OPEN, READ, WRITE, REWRITE, and CLOSE.

OPEN Makes the file available to your program. You can open all QSAM files as INPUT, OUTPUT, or EXTEND (depending on device capabilities).

You can also open QSAM files on direct access storage devices as I-O. You cannot open HFS files as I-O; a file status of 37 results if you attempt to do so.

READ Reads a record from the file. With sequential processing, your program reads one record after another in the same order in which they were entered when the file was created.

WRITE Creates a record in the file. Your program writes new records to the end of the file.

REWRITE
Updates a record. You cannot update a file in the HFS using REWRITE.

`CLOSE` Releases the connection between the file and your program.

RELATED TASKS
"Opening QSAM files"
"Adding records to QSAM files" on page 163
"Updating QSAM files" on page 163
"Writing QSAM files to a printer or spooled data set" on page 164
"Closing QSAM files" on page 164

RELATED REFERENCES
OPEN statement (*Enterprise COBOL Language Reference*)
READ statement (*Enterprise COBOL Language Reference*)
WRITE statement (*Enterprise COBOL Language Reference*)
REWRITE statement (*Enterprise COBOL Language Reference*)
CLOSE statement (*Enterprise COBOL Language Reference*)
File status key (*Enterprise COBOL Language Reference*)

Opening QSAM files

Before your program can use any `READ`, `WRITE`, or `REWRITE` statements to process records in a file, it must first open the file with an `OPEN` statement.

An `OPEN` statement works if both of the following conditions are true:

- The file is available or has been dynamically allocated.
- The *fixed file attributes* coded in the ddname definition or the data-set label for the file match the attributes coded for that file in the `SELECT` clause and `FD` entry.

  Mismatches in the file-organization attributes, code set, maximum record size, or record format (fixed or variable) result in a file status code 39, and the failure of the `OPEN` statement. Mismatches in maximum record size and record format are not errors when opening files in the HFS.

  For fixed-length QSAM files, if you code `RECORD CONTAINS 0` in the `FD` entry, the record size attributes are not in conflict. The record size is taken from the `DD` statement or the data-set label, and the `OPEN` statement is successful.

Code `CLOSE WITH LOCK` so that the file cannot be opened again while the program is running.

Use the `REVERSED` option of the `OPEN` statement to process tape files in reverse order. The file is positioned at the end, and `READ` statements read the data records in reverse order, starting with the last record. The `REVERSED` option is supported only for files that have fixed-length records.

RELATED TASKS
"Dynamically creating QSAM files with CBLQDA"
"Ensuring that file attributes match your program" on page 169

RELATED REFERENCES
OPEN statement (*Enterprise COBOL Language Reference*)

Dynamically creating QSAM files with CBLQDA

Sometimes a QSAM file is unavailable on the operating system, but a COBOL program specifies that the file be created. Under certain circumstances, the file is created for you dynamically.

(A file is considered to be available on z/OS when it has been identified to the operating system using a valid DD statement, an export command for an environment variable, or a TSO ALLOCATE command. Note that a DD statement with a misspelled ddname is equivalent to a missing DD statement, and an environment variable with a value that is not valid is equivalent to an unset variable.)

The QSAM file is implicitly created if you use the runtime option CBLQDA and one of the following circumstances exists:

- An optional file is being opened as EXTEND or I-O.

  Optional files are files that are not necessarily present each time the program is run. You define a file that is being opened in INPUT, I-O, or EXTEND mode as optional by coding the SELECT OPTIONAL clause in the FILE-CONTROL paragraph.
- The file is being opened for OUTPUT, regardless of the OPTIONAL phrase.

The file is allocated with the system default attributes established at your installation and the attributes coded in the SELECT clause and FD entry in your program.

Do not confuse this implicit allocation mechanism with the explicit dynamic allocation of files by means of environment variables. Explicit dynamic allocation requires that a valid environment variable be set. CBLQDA support is used only when the QSAM file is unavailable as defined above, which includes no valid environment variable being set.

Under z/OS, files created using the CBLQDA option are temporary data sets and do not exist after the program has run.

RELATED TASKS
"Opening QSAM files" on page 162

Adding records to QSAM files

To add to a QSAM file, open the file as EXTEND and use the WRITE statement to add records immediately after the last record in the file.

To add records to a file opened as I-O, you must first close the file and open it as EXTEND.

RELATED REFERENCES
READ statement (*Enterprise COBOL Language Reference*)
WRITE statement (*Enterprise COBOL Language Reference*)

Updating QSAM files

You can update QSAM files only if they reside on direct access storage devices. You cannot update files in the HFS.

Replace an existing record with another record of the same length by doing these steps:

1. Open the file as I-O.
2. Use REWRITE to update an existing record. (The last file processing statement before REWRITE must have been a successful READ statement.)

You cannot open as I-O an extended format data set that you allocate in compressed format.

RELATED REFERENCES
REWRITE statement (*Enterprise COBOL Language Reference*)

Writing QSAM files to a printer or spooled data set

COBOL provides language statements to control the size of a printed page and control the vertical positioning of records.

Controlling the page size: Use the `LINAGE` clause of the `FD` entry to control the size of your printed page: the number of lines in the top and bottom margins and in the footing area of the page. When you use the `LINAGE` clause, COBOL handles the file as if you had also requested the `ADV` compiler option.

If you use the `LINAGE` clause in combination with `WRITE BEFORE|AFTER ADVANCING` *nn* `LINES`, be careful about the values you set. With the `ADVANCING` *nn* `LINES` phrase, COBOL first calculates the sum of `LINAGE-COUNTER` plus *nn*. Subsequent actions depend on the size of *nn*. The `END-OF-PAGE` imperative phrase is performed after the `LINAGE-COUNTER` is increased. Consequently, the `LINAGE-COUNTER` could be pointing to the next logical page instead of to the current footing area when the `END-OF-PAGE` phrase is performed.

`AT END-OF-PAGE` or `NOT AT END-OF-PAGE` imperative phrases are performed only if the write operation completes successfully. If the write operation is unsuccessful, control is passed to the end of the `WRITE` statement, and all conditional phrases are omitted.

Controlling the vertical positioning of records: Use the `WRITE ADVANCING` statement to control the vertical positioning of each record you write on a printed page.

`BEFORE ADVANCING` prints the record before the page is advanced. `AFTER ADVANCING` prints the record after the page is advanced.

Specify the number of lines the page is advanced with an integer (or an *identifier* with a *mnemonic-name*) following `ADVANCING`. If you omit the `ADVANCING` phrase from a `WRITE` statement, the effect is as if you had coded:

```
AFTER ADVANCING 1 LINE
```

RELATED REFERENCES
WRITE statement (*Enterprise COBOL Language Reference*)

Closing QSAM files

Use the `CLOSE` statement to disconnect your program from a QSAM file. If you try to close a file that is already closed, you will get a logic error.

If you do not close a QSAM file, the file is automatically closed for you under the following conditions, except for files defined in any OS/VS COBOL programs in the run unit:

- When the run unit ends normally, the run time closes all open files that are defined in any COBOL programs in the run unit.
- If the run unit ends abnormally and the `TRAP(ON)` runtime option is in effect, the run time closes all open files that are defined in any COBOL programs in the run unit.
- When Language Environment condition handling has completed and the application resumes in a routine other than where the condition occurred, the

run time closes all open files that are defined in any COBOL programs in the run unit that might be called again and reentered.

You can change the location where the program resumes running (after a condition is handled) by moving the resume cursor with the Language Environment CEEMRCR callable service or by using language constructs such as a C longjmp.

- When you use CANCEL for a COBOL subprogram, the run time closes any open nonexternal files that are defined in that program.
- When a COBOL subprogram with the INITIAL attribute returns control, the run time closes any open nonexternal files that are defined in that program.
- When a thread of a multithreaded application ends, both external and nonexternal files that you opened from within that same thread are closed.

File status key data items in the DATA DIVISION are set when these implicit CLOSE operations are performed, but your EXCEPTION/ERROR and LABEL declaratives are not invoked.

Errors: If you open a QSAM file in a multithreaded application, you must close it from the same thread of execution from which the file was opened. Attempting to close the file from a different thread results in a close failure with file-status condition 90.

RELATED REFERENCES
CLOSE statement (*Enterprise COBOL Language Reference*)

Handling errors in QSAM files

When an input statement or output statement fails, COBOL does not take corrective action for you. You choose whether your program should continue running after a less-than-severe input or output error occurs.

COBOL provides these ways for you to intercept and handle certain QSAM input and output errors:

- End-of-file phrase (AT END)
- EXCEPTION/ERROR declarative
- FILE STATUS clause
- INVALID KEY phrase

If you do not code a FILE STATUS key or a declarative, serious QSAM processing errors will cause a message to be issued and a Language Environment condition to be signaled, which will cause an abend if you specify the runtime option ABTERMENC(ABEND).

If you use the FILE STATUS clause or the EXCEPTION/ERROR declarative, code EROPT=ACC in the DCB of the DD statement for that file. Otherwise, your COBOL program will not be able to continue processing after some error conditions.

If you use the FILE STATUS clause, be sure to check the key and take appropriate action based on its value. If you do not check the key, your program might continue, but the results will probably not be what you expected.

RELATED TASKS
"Handling errors in input and output operations" on page 235

Working with QSAM files

To work with QSAM files in a COBOL program, you define and allocate them, retrieve them, and ensure that their file attributes match those in your program. You can also use striped extended-format QSAM data sets to help improve performance.

RELATED TASKS
"Defining and allocating QSAM files"
"Retrieving QSAM files" on page 168
"Ensuring that file attributes match your program" on page 169
"Using striped extended-format QSAM data sets" on page 171

RELATED REFERENCES
"Allocation of buffers for QSAM files" on page 172

Defining and allocating QSAM files

You can define a QSAM file or a byte-stream file in the HFS by using either a `DD` statement or an environment variable. Allocation of these files follows the general rules for the allocation of COBOL files.

When you use an environment variable, the name must be in uppercase. Specify the MVS data set in one of these ways:
- `DSN(`*dataset-name*`)`
- `DSN(`*dataset-name*`(`*member-name*`))`

dataset-name must be fully qualified and cannot be a temporary data set (that is, it must not start with `&`).

| **Restriction:** You cannot create a PDS or PDSE by using an environment variable.

You can optionally specify the following attributes in any order after `DSN`:
- A disposition value, one of: `NEW`, `OLD`, `SHR`, or `MOD`
- `TRACKS` or `CYL`
- `SPACE(`*nnn*`,`*mmm*`)`
- `VOL(`*volume-serial*`)`
- `UNIT(`*type*`)`
- `KEEP`, `DELETE`, `CATALOG`, or `UNCATALOG`
- `STORCLAS(`*storage-class*`)`
- `MGMTCLAS(`*management-class*`)`
- `DATACLAS(`*data-class*`)`

You can use either an environment variable or a `DD` definition to define a file in the HFS. To do so, define one of the following items with a name that matches the external name in the `ASSIGN` clause:
- A `DD` allocation that uses `PATH='`*absolute-path-name*`'` and `FILEDATA=BINARY`
- An environment variable with a value `PATH(`*pathname*`)`, where *pathname* is an absolute path name (starting with `/`)

For compatibility with releases of COBOL before COBOL for OS/390 & VM Version 2 Release 2, you can also specify `FILEDATA=TEXT` when using a `DD` allocation for HFS files, but this use is not recommended. To process text files in the HFS, use

LINE SEQUENTIAL organization. If you do use QSAM to process text files in the HFS, you cannot use environment variables to define the files.

When you define a QSAM file, use the parameters as shown below.

Table 20. **QSAM file allocation**

What you want to do	DD parameter to use	EV keyword to use
Name the file.	DSNAME (data set name)	DSN
Select the type and quantity of input-output devices to be allocated for the file.	UNIT	UNIT for type only
Give instructions for the volume in which the file will reside and for volume mounting.	VOLUME (or let the system choose an output volume)	VOL
Allocate the type and amount of space the file needs. (Only for direct-access storage devices.)	SPACE	SPACE for the amount of space (primary and secondary only); TRACKS or CYL for the type of space
Specify the type and some of the contents of the label associated with the file.	LABEL	n/a
Indicate whether you want to catalog, pass, or keep the file after the job step is completed.	DISP	NEW, OLD, SHR, MOD plus KEEP, DELETE, CATALOG, or UNCATALOG
Complete any data control block information that you want to add.	DCB subparameters	n/a

Some of the information about the QSAM file must always be coded in the FILE-CONTROL paragraph, the FD entry, and other COBOL clauses. Other information must be coded in the DD statement or environment variable for output files. For input files, the system can obtain information from the file label (for standard label files). If DCB information is provided in the DD statement for input files, it overrides information on the data-set label. For example, the amount of space allocated for a new direct-access device file can be set in the DD statement by the SPACE parameter.

You cannot express certain characteristics of QSAM files in the COBOL language, but you can code them in the DD statement for the file by using the DCB parameter. Use the subparameters of the DCB parameter to provide information that the system needs for completing the data set definition, including the following:

- Block size (BLKSIZE=), if BLOCK CONTAINS 0 RECORDS was coded at compile time (recommended)
- Options to be executed if an error occurs in reading or writing a record
- TRACK OVERFLOW or standard blocks
- Mode of operation for a card reader or punch

DCB attributes coded for a DD DUMMY do not override those coded in the FD entry of your COBOL program.

"Example: setting and accessing environment variables" on page 426

RELATED TASKS
"Setting block sizes" on page 159
"Defining QSAM files and records in COBOL" on page 151
"Allocating files" on page 149

RELATED REFERENCES
"Parameters for creating QSAM files"
MVS JCL Reference

Parameters for creating QSAM files

The following DD statement parameters are frequently used to create QSAM files.

```
         ┌                            ┐
DSNAME=  │ dataset-name               │
         │ dataset-name(member-name)  │
DSN=     │ &&name                     │
         │ &&name(member-name)        │
         └                            ┘

UNIT=    ( name[,unitcount] )

VOLUME=  ( [PRIVATE] [,RETAIN] [,vol-sequence-num] [,volume-count] ...
VOL=
                         ... ┌ ,SER=(volume-serial[,volume-serial]...) ┐ )
                             └ ,REF=┌dsname                  ┐        ┘
                                    │*.ddname                │
                                    │*.stepname.ddname       │
                                    └*.stepname.procstep.ddname┘

SPACE=   (┌TRK                 ┐,(primary-quantity[,secondary-quantity][,directory-quantity]))
          │CYL                 │
          └average-record-length┘

LABEL=   ( [Data-set-sequence-number,]┌ NL ┐ ┌,EXPDT= ┌ yyddd   ┐┐ )
                                      │ SL │ │        └ yyyy/ddd┘│
                                      └ SUL┘ └,RETPD=xxxx        ┘

DISP=    ( ┌ NEW ┐ ┌,DELETE┐ ┌,DELETE┐ )
           └ MOD ┘ │,KEEP  │ │,KEEP  │
                   │,PASS  │ └,CATLG ┘
                   └,CATLG ┘

DCB=     ( subparameter-list )
```

RELATED TASKS
"Defining and allocating QSAM files" on page 166

Retrieving QSAM files

You retrieve QSAM files, cataloged or not, by using job control statements or environment variables.

Cataloged files
All data set information, such as volume and space, is stored in the catalog and file label. All you have to code are the data set name and a disposition. When you use a DD statement, this is the DSNAME parameter and the DISP parameter. When you use an environment variable, this is the DSN parameter and one of the parameters OLD, SHR, or MOD.

Noncataloged files
Some information is stored in the file label, but you must code the unit and volume information, and the *dsname* and disposition.

If you are using JCL, and you created the file in the current job step or in a previous job step in the current job, you can refer to the previous DD statement for most of the data set information. You do, however, need to code DSNAME and DISP.

RELATED REFERENCES
"Parameters for retrieving QSAM files"

Parameters for retrieving QSAM files

The following DD statement parameters are used to retrieve previously created files.

```
         ┌                            ┐
DSNAME=  │ dataset-name               │
DSN=     │ dataset-name(member-name)  │
         │ *.ddname                   │
         │ *.stepname.ddname          │
         │ &&name                     │
         │ &&name(member-name)        │
         └                            ┘

UNIT=    ( name[,unitcount] )

VOLUME=  ( subparameter-list )
VOL=

LABEL=   ( subparameter-list )


DISP=    ( ┌OLD┐  ┌,DELETE  ┐ ┌,DELETE  ┐ )
           │SHR│  │,KEEP    │ │,KEEP    │
           └MOD┘  │,PASS    │ │,CATLG   │
                  │,CATLG   │ └,UNCATLG ┘
                  └,UNCATLG ┘

DCB=     ( subparameter-list )
```

RELATED TASKS
"Retrieving QSAM files" on page 168

Ensuring that file attributes match your program

When the fixed file attributes in the DD statement or the data-set label and the attributes that are coded for that file in the SELECT clause and FD entry are not consistent, an OPEN statement in your program might not work.

Mismatches in the attributes for file organization, record format (fixed or variable), record length, or the code set result in file status code 39 and the failure of the OPEN statement. An exception exists for files in the HFS: mismatches in record format and record length do not cause an error.

To prevent common file status 39 problems, follow the guidelines for processing existing or new files.

If you have not made a file available with a DD statement or a TSO ALLOCATE command, and your COBOL program specifies that the file be created, Enterprise COBOL dynamically allocates the file. When the file is opened, the file attributes that are coded in your program are used. You do not have to worry about file attribute conflicts.

Remember that information in the JCL or environment variable overrides information in the data-set label.

RELATED TASKS
"Processing existing files"
"Processing new files" on page 171
"Opening QSAM files" on page 162

RELATED REFERENCES
"FILE SECTION entries" on page 14

Processing existing files

When your program processes an existing file, code the description of the file in your COBOL program to be consistent with the file attributes of the data set. Use the guidelines below to define the maximum record length.

Table 21. **Maximum record length of QSAM files**

For this format:	Specify this:
V or S	Exactly 4 bytes less than the length attribute of the data set
F	Same value as the length attribute of the data set
U	Same value as the length attribute of the data set

The easiest way to define variable-length (format-V) records in a program is to use the `RECORD IS VARYING FROM` *integer-1* `TO` *integer-2* clause in the `FD` entry and set an appropriate value for *integer-2*. Express the integer sizes in bytes regardless of the underlying `USAGE` of the data items in the record. For example, assume that you determine that the length attribute of the data set is 104 bytes (`LRECL=104`). Remembering that the maximum record length is determined from the `RECORD IS VARYING` clause and not from the level-01 record descriptions, you could define a format-V file in your program with this code:

```
FILE SECTION.
FD  COMMUTER-FILE-MST
    RECORDING MODE IS V
    RECORD IS VARYING FROM 4 TO 100 CHARACTERS.
01  COMMUTER-RECORD-A   PIC X(4).
01  COMMUTER-RECORD-B   PIC X(75).
```

Assume that the existing file in the previous example was format-U instead of format-V. If the 104 bytes are all user data, you could define the file in your program with this code:

```
FILE SECTION.
FD  COMMUTER-FILE-MST
    RECORDING MODE IS U
    RECORD IS VARYING FROM 4 TO 104 CHARACTERS.
01  COMMUTER-RECORD-A   PIC X(4).
01  COMMUTER-RECORD-B   PIC X(75).
```

To define fixed-length records in your program, either code the `RECORD CONTAINS` *integer* clause, or omit this clause and code all level-01 record descriptions to be the same fixed size. In either case, use a value that equals the value of the length attribute of the data set. If you intend to use the same program to process different files at run time, and those files have differing fixed lengths, avoid record-length conflicts by coding `RECORD CONTAINS 0`.

If the existing file is an ASCII data set (`DCB=(OPTCD=Q)`), you must use the `CODE-SET` clause in the `FD` entry for the file.

RELATED TASKS
"Processing new files"
"Requesting fixed-length format" on page 153
"Requesting variable-length format" on page 154
"Requesting undefined format" on page 158
"Opening QSAM files" on page 162

RELATED REFERENCES
"FILE SECTION entries" on page 14

Processing new files

If your COBOL program writes records to a new file that will be made available before the program runs, ensure that the file attributes in the DD statement, the environment variable, or the allocation do not conflict with the attributes in the program.

Usually you need to code only a minimum of parameters when predefining files. But if you need to explicitly set a length attribute for the data set (for example, you are using an ISPF allocation panel, or your DD statement is for a batch job in which the program uses RECORD CONTAINS 0), follow these guidelines:

- For format-V and format-S files, set a length attribute that is 4 bytes larger than that defined in the program.
- For format-F and format-U files, set a length attribute that is the same as that defined in the program.
- If you open the file as OUTPUT and write it to a printer, the compiler might add 1 byte to the record length to account for the carriage-control character, depending on the ADV compiler option and the language used in your program. In such a case, take the added byte into account when coding the LRECL value.

For example, if your program contains the following code for a file that has variable-length records, the LRECL value in the DD statement or allocation should be 54.

```
FILE SECTION.
FD  COMMUTER-FILE-MST
    RECORDING MODE IS V
    RECORD CONTAINS 10 TO 50 CHARACTERS.
01  COMMUTER-RECORD-A   PIC X(10).
01  COMMUTER-RECORD-B   PIC X(50).
```

RELATED TASKS
"Processing existing files" on page 170
"Requesting fixed-length format" on page 153
"Requesting variable-length format" on page 154
"Requesting undefined format" on page 158
"Opening QSAM files" on page 162
"Dynamically creating QSAM files with CBLQDA" on page 162

RELATED REFERENCES
"FILE SECTION entries" on page 14

Using striped extended-format QSAM data sets

Striped extended-format QSAM data sets can benefit applications that process files that have large amounts of data or in which the time needed for I/O operations significantly affects overall performance.

A striped extended-format QSAM data set is an extended-format QSAM data set that is spread over multiple volumes, thus allowing parallel data access.

For you to gain the maximum benefit from using QSAM striped data sets, z/OS DFSMS needs to be able to allocate the required number of buffers above the 16-MB line. When you develop applications that contain files allocated to QSAM striped data sets, follow these guidelines:

- Avoid using a QSAM striped data set for a file that cannot have buffers allocated above the 16-MB line.
- Omit the `RESERVE` clause in the `FILE-CONTROL` entry for the file. Doing so lets z/OS DFSMS determine the optimum number of buffers for the data set.
- Compile your program with the `DATA(31)` and `RENT` compiler options, and make the load module `AMODE 31`.
- Specify the `ALL31(ON)` runtime option if the file is an `EXTERNAL` file with format-F, format-V, or format-U records.

Note that all striped data sets are extended-format data sets, but not all extended-format data sets are striped.

RELATED TASKS
z/OS DFSMS: Using Data Sets (performance considerations)

RELATED REFERENCES
"Allocation of buffers for QSAM files"

Allocation of buffers for QSAM files

z/OS DFSMS automatically allocates buffers for storing input and output for a QSAM file above or below the 16-MB line as appropriate for the file.

Most QSAM files have buffers allocated above the 16-MB line. Exceptions are:

- Programs running in `AMODE 24`.
- Programs compiled with the `DATA(24)` and `RENT` options.
- Programs compiled with the `NORENT` and `RMODE(24)` options.
- Programs compiled with the `NORENT` and `RMODE(AUTO)` options.
- `EXTERNAL` files when the `ALL31(OFF)` runtime option is specified. To specify the `ALL31(ON)` runtime option, all programs in the run unit must be capable of running in 31-bit addressing mode.
- Files allocated to the TSO terminal.
- A file with format-S (spanned) records, if the file is any of the following:
  - An `EXTERNAL` file (even if `ALL31(ON)` is specified)
  - A file specified in a `SAME RECORD AREA` clause of the `I-O-CONTROL` paragraph
  - A blocked file that is opened `I-O` and updated using the `REWRITE` statement

RELATED CONCEPTS
"Storage and its addressability" on page 41

RELATED TASKS
"Using striped extended-format QSAM data sets" on page 171

Accessing HFS files using QSAM

You can process byte-stream files in the hierarchical file system (HFS) as `ORGANIZATION SEQUENTIAL` files using QSAM. To do this, specify as the *assignment-name* in the `ASSIGN` clause either a ddname or an environment-variable name.

ddname
: A `DD` allocation that identifies the file with the keywords `PATH=` and `FILEDATA=BINARY`

Environment-variable name
: An environment variable that holds the runtime value of the HFS path for the file

Observe the following restrictions:

- Spanned record format is not supported.
- `OPEN I-O` and `REWRITE` are not supported. If you attempt one of these operations, one of the following file-status conditions results:
  - 37 from `OPEN I-O`
  - 47 from `REWRITE` (because you could not have successfully opened the file as `I-O`)

Usage notes

- File status 39 (fixed file attribute conflict) is not enforced for either of the following types of conflicts:
  - Record-length conflict
  - Record-type conflict (fixed as opposed to variable)
- A `READ` returns the number of bytes of the maximum logical record size for the file except for the last record, which might be shorter.

  For example, suppose that a file definition has level-01 record descriptions of 3, 5, and 10 bytes long, and you write the following three records: ′abc′, ′defgh′, and ′ijklmnopqr′, in that order. The first `READ` of this file returns ′abcdefghij′, the second `READ` returns ′klmnopqr ′, and the third `READ` results in the `AT END` condition.

For compatibility with releases of IBM COBOL before COBOL for OS/390 & VM Version 2 Release 2, you can also specify `FILEDATA=TEXT` when using a `DD` allocation for HFS files, but this use is not recommended. To process text files in the HFS, use `LINE SEQUENTIAL` organization. If you use QSAM to process text files in the HFS, you cannot use environment variables to define the files.

RELATED TASKS
"Allocating files" on page 149
"Defining and allocating QSAM files" on page 166
Accessing HFS files using BSAM and QSAM (*z/OS DFSMS: Using Data Sets*)

Labels for QSAM files

You can use labels to identify magnetic tape and direct access volumes and data sets. The operating system uses label-processing routines to identify and verify labels and locate volumes and data sets.

There are two kinds of labels: standard and nonstandard. Enterprise COBOL does not support nonstandard user labels. In addition, standard user labels contain user-specified information about the associated data set.

Standard labels consist of volume labels and groups of data-set labels. Volume labels precede or follow data on the volume, and identify and describe the volume. The data-set labels precede or follow each data set on the volume, and identify and describe the data set.

- The data-set labels that precede the data set are called *header labels*.
- The data-set labels that follow the data set are called *trailer labels*. They are similar to the header labels, except that they also contain a count of blocks in the data set.
- The data-set label groups can optionally include standard user labels.
- The volume label groups can optionally include standard user labels.

RELATED TASKS
"Using trailer and header labels"

RELATED REFERENCES
"Format of standard labels" on page 175

Using trailer and header labels

You can create, examine, or update user labels when the beginning or end of a data set or volume (reel) is reached. End-of-volume or beginning-of-volume exits are allowed. You can also create or examine intermediate trailers and headers.

You can create, examine, or update up to eight header labels and eight trailer labels on each volume of the data set. (QSAM `EXTEND` works in a manner identical to `OUTPUT` except that the beginning-of-file label is not processed.) Labels reside on the initial volume of a multivolume data set. This volume must be mounted as `CLOSE` if trailer labels are to be created, examined, or updated. Trailer labels for files opened as `INPUT` or `I-O` are processed when a `CLOSE` statement is performed for the file that has reached an `AT END` condition.

If you code a header or trailer with the wrong position number, the result is unpredictable. (Data management might force the label to the correct relative position.)

When you use standard label processing, code the label type of the standard and user labels (SUL) on the `DD` statement that describes the data set.

Getting a user-label track: If you use a `LABEL` subparameter of `SUL` for direct access volumes, a separate user-label track is allocated when the data set is created. This additional track is allocated at initial allocation and for sequential data sets at end-of-volume (volume switch). The user-label track (one per volume of a sequential data set) contains both user header and user trailer labels. If a `LABEL` name is referenced outside the user `LABEL` declarative, results are unpredictable.

Handling user labels: The `USE AFTER LABEL` declarative provides procedures for handling user labels on supported files. The `AFTER` option indicates processing of standard user labels.

List the labels as *data-names* in the `LABEL RECORDS` clause in the `FD` entry for the file.

Table 22. **Handling of QSAM user labels**

When the file is opened as:	And:	Result:
INPUT	USE . . . LABEL declarative is coded for the OPEN option or for the file.	The label is read and control is passed to the LABEL declarative.
OUTPUT	USE . . . LABEL declarative is coded for the OPEN option or for the file.	A buffer area for the label is provided, and control is passed to the LABEL declarative.
INPUT or I-O	CLOSE statement is performed for the file that has reached the AT END condition.	Control is passed to the LABEL declarative for processing trailer labels.

You can specify a special exit by using the statement GO TO MORE-LABELS. When this statement results in an exit from a label DECLARATIVE SECTION, the system takes one of the following actions:

- Writes the current beginning or ending label, and then reenters the USE section at its beginning to create more labels. After creating the last label, the system exits by performing the last statement of the section.
- Reads an additional beginning or ending label, and then reenters the USE section at its beginning to check more labels. When processing user labels, the system reenters the section only if there is another user label to check. Hence, a program path that flows through the last statement in the section is not needed.

If a GO TO MORE-LABELS statement is not performed for a user label, the DECLARATIVE SECTION is not reentered to check or create any immediately succeeding user labels.

RELATED CONCEPTS
"Labels for QSAM files" on page 173

Format of standard labels

Standard labels are 80-character records that are recorded in EBCDIC or ASCII. The first four characters are always used to identify the labels.

Table 23. **Identifiers for standard tape labels**

Identifier	Description
VOL1	Volume label
HDR1 or HDR2	Data set header labels
EOV1 or EOV2	Data set trailer labels (end-of-volume)
EOF1 or EOF2	Data set trailer labels (end-of-data-set)
UHL1 to UHL8	User header labels
UTL1 to UTL8	User trailer labels

The format of the label for a direct-access volume is the almost the same as the format of the label group for a tape volume label group. The difference is that a data-set label of the initial DASTO volume label consists of the data set control block (DSCB). The DSCB appears in the volume table of contents (VTOC) and contains the equivalent of the tape data set header and trailer, in addition to control information such as space allocation.

Standard user labels

User labels are optional within the standard label groups. The format for user header labels (UHL1-8) and user trailer labels (UTL1-8) consists of a label 80 characters in length recorded in either:

- EBCDIC on DASD or on IBM standard labeled tapes
- ASCII or ISO/ANSI labeled tapes

The first 3 bytes consist of the characters that identify the label as either:

- UHL for a user header label (at the beginning of a data set)
- UTL for a user trailer label (at the end-of-volume or end-of-data set)

The next byte contains the relative position of this label within a set of labels of the same type; one to eight labels are permitted. The remaining 76 bytes consist of user-specified information.

Standard user labels are not supported for QSAM striped data sets.

RELATED CONCEPTS
"Labels for QSAM files" on page 173

Processing QSAM ASCII files on tape

If your program processes a QSAM ASCII file, you must request the ASCII alphabet, define the record formats, and define the ddname (with JCL).

In addition, if your program processes signed numeric data items from ASCII files, define the numeric data as zoned decimal items with separate signs, that is, as `USAGE DISPLAY` and with the `SEPARATE` phrase of the `SIGN` clause.

The `CODEPAGE` compiler option has no effect on the code page used for conversions between ASCII and EBCDIC for ASCII tape support. See the z/OS DFSMS documentation for information about how CCSIDs used for the ASCII tape support are selected and what the default CCSIDs are.

Requesting the ASCII alphabet: In the `SPECIAL-NAMES` paragraph, code `STANDARD-1` for ASCII:

```
ALPHABET-NAME IS STANDARD-1
```

In the `FD` entry for the file, code:

```
CODE-SET IS ALPHABET-NAME
```

Defining the record formats: Process QSAM ASCII tape files with any of these record formats:

- Fixed length (format F)
- Undefined (format U)
- Variable length (format V)

If you use variable-length records, you cannot explicitly code format D; instead, code `RECORDING MODE V`. The format information is internally converted to D mode. D-mode records have a 4-byte record descriptor for each record.

Defining the ddname: Under z/OS, processing ASCII files requires special JCL coding. Code these subparameters of the `DCB` parameter in the `DD` statement:

BUFOFF=[*L* | *n*]

L	A 4-byte block prefix that contains the block length (including the block prefix)
n	The length of the block prefix: • For input, from 0 through 99 • For output, either 0 or 4 Use this value if you coded `BLOCK CONTAINS 0`.

BLKSIZE=*n*

n	The size of the block, including the length of the block prefix

LABEL=[AL | AUL | NL]

AL	American National Standard (ANS) labels
AUL	ANS and user labels
NL	No labels

OPTCD=Q

Q	This value is required for ASCII files and is the default if the file is created using Enterprise COBOL.

RELATED TASKS
"Processing ASCII file labels"
Converting character data (*z/OS DFSMS: Using Data Sets*)

Processing ASCII file labels

Standard label processing for ASCII files is the same as standard label processing for EBCDIC files. The system translates ASCII code into EBCDIC before processing.

All ANS user labels are optional. ASCII files can have user header labels (`UHL`*n*) and user trailer labels (`UTL`*n*). There is no limit to the number of user labels at the beginning and the end of a file; you can write as many labels as you need. All user labels must be 80 bytes in length.

To create or verify user labels (user label exit), code a `USE AFTER STANDARD LABEL` procedure. You cannot use `USE BEFORE STANDARD LABEL` procedures.

ASCII files on tape can have:
- ANS labels
- ANS and user labels
- No labels

Any labels on an ASCII tape must be in ASCII code only. Tapes that contain a combination of ASCII and EBCDIC cannot be read.

RELATED TASKS
"Processing QSAM ASCII files on tape" on page 176

Chapter 10. Processing VSAM files

Virtual storage access method (VSAM) is an access method for files on direct-access storage devices. With VSAM you can load files, retrieve records from files, update files, and add, replace, and delete records in files.

VSAM processing has these advantages over QSAM:

- Protection of data against unauthorized access
- Compatibility across systems
- Independence of devices (no need to be concerned with block size and other control information)
- Simpler JCL (information needed by the system is provided in integrated catalogs)
- Ability to use indexed file organization or relative file organization

The table below shows how VSAM terms differ from COBOL terms and other terms that you might be familiar with.

Table 24. **Comparison of VSAM, COBOL, and non-VSAM terminology**

VSAM term	COBOL term	Similar non-VSAM term
Data set	File	Data set
Entry-sequenced data set (ESDS)	Sequential file	QSAM data set
Key-sequenced data set (KSDS)	Indexed file	ISAM data set
Relative-record data set (RRDS)	Relative file	BDAM data set
Control interval		Block
Control interval size (CISZ)		Block size
Buffers (BUFNI/BUFND)		BUFNO
Access method control block (ACB)		Data control block (DCB)
Cluster (CL)		Data set
Cluster definition		Data-set allocation
`AMP` parameter of JCL `DD` statement		`DCB` parameter of JCL `DD` statement
Record size		Record length

The term *file* in this VSAM documentation refers to either a COBOL file or a VSAM data set.

If you have complex requirements or frequently use VSAM, review the VSAM publications for your operating system.

RELATED CONCEPTS
"VSAM files" on page 180

RELATED TASKS
"Defining VSAM file organization and records" on page 181
"Coding input and output statements for VSAM files" on page 188
"Handling errors in VSAM files" on page 196
"Protecting VSAM files with a password" on page 196

"Working with VSAM data sets under z/OS and UNIX" on page 197
"Improving VSAM performance" on page 204

RELATED REFERENCES
z/OS DFSMS: Using Data Sets
z/OS DFSMS Macro Instructions for Data Sets
z/OS DFSMS: Access Method Services for Catalogs

VSAM files

The physical organization of VSAM data sets differs considerably from the organizations used by other access methods.

VSAM data sets are held in control intervals (CI) and control areas (CA). The size of the CI and CA is normally determined by the access method, and the way in which they are used is not visible to you.

You can use three types of file organization with VSAM:

VSAM sequential file organization
(Also referred to as VSAM *ESDS (entry-sequenced data set)* organization.) In VSAM sequential file organization, the records are stored in the order in which they were entered.

VSAM entry-sequenced data sets are equivalent to QSAM sequential files. The order of the records is fixed.

VSAM indexed file organization
(Also referred to as VSAM *KSDS (key-sequenced data set)* organization.) In a VSAM indexed file (KSDS), the records are ordered according to the collating sequence of an embedded prime key field, which you define. The prime key consists of one or more consecutive characters in the records. The prime key uniquely identifies the record and determines the sequence in which it is accessed with respect to other records. A prime key for a record might be, for example, an employee number or an invoice number.

VSAM relative file organization
(Also referred to as VSAM fixed-length or variable-length *RRDS (relative-record data set)* organization.) A VSAM relative-record data set (RRDS) contains records ordered by their relative key. The *relative key* is the relative record number, which represents the location of the record relative to where the file begins. The relative record number identifies the fixed- or variable-length record.

In a VSAM fixed-length RRDS, records are placed in a series of fixed-length slots in storage. Each slot is associated with a relative record number. For example, in a fixed-length RRDS containing 10 slots, the first slot has a relative record number of 1, and the tenth slot has a relative record number of 10.

In a VSAM variable-length RRDS, the records are ordered according to their relative record number. Records are stored and retrieved according to the relative record number that you set.

Throughout this documentation, the term *VSAM relative-record data set* (or *RRDS*) is used to mean both relative-record data sets with fixed-length records and with variable-length records, unless they need to be differentiated.

The following table compares the characteristics of the different types of VSAM data sets.

Table 25. **Comparison of VSAM data-set types**

Characteristic	Entry-sequenced data set (ESDS)	Key-sequenced data set (KSDS)	Relative-record data set (RRDS)
Order of records	Order in which they are written	Collating sequence by key field	Order of relative record number
Access	Sequential	By key through an index	By relative record number, which is handled like a key
Alternate indexes	Can have one or more alternate indexes, although not supported in COBOL	Can have one or more alternate indexes	Cannot have alternate indexes
Relative byte address (RBA) and relative record number (RRN) of a record	RBA cannot change.	RBA can change.	RRN cannot change.
Space for adding records	Uses space at the end of the data set	Uses distributed free space for inserting records and changing their lengths in place	For fixed-length RRDS, uses empty slots in the data set For variable-length RRDS, uses distributed free space and changes the lengths of added records in place
Space from deleting records	You cannot delete a record, but you can reuse its space for a record of the same length.	Space from a deleted or shortened record is automatically reclaimed in a control interval.	Space from a deleted record can be reused.
Spanned records	Can have spanned records	Can have spanned records	Cannot have spanned records
Reuse as work file	Can be reused unless it has an alternate index, is associated with key ranges, or exceeds 123 extents per volume	Can be reused unless it has an alternate index, is associated with key ranges, or exceeds 123 extents per volume	Can be reused

RELATED TASKS
"Specifying sequential organization for VSAM files" on page 182
"Specifying indexed organization for VSAM files" on page 182
"Specifying relative organization for VSAM files" on page 184
"Defining VSAM files" on page 198

Defining VSAM file organization and records

Use an entry in the FILE-CONTROL paragraph in the ENVIRONMENT DIVISION to define the file organization and access modes for the VSAM files in your COBOL program.

In the FILE SECTION of the DATA DIVISION, code a file description (FD) entry for the file. In the associated record description entry or entries, define the *record-name* and record length. Code the logical size of the records with the RECORD clause.

Important: You can process VSAM data sets in Enterprise COBOL programs only after you define them with access method services.

Table 26. **VSAM file organization, access mode, and record format**

File organization	Sequential access	Random access	Dynamic access	Fixed length	Variable length
VSAM sequential (ESDS)	Yes	No	No	Yes	Yes
VSAM indexed (KSDS)	Yes	Yes	Yes	Yes	Yes
VSAM relative (RRDS)	Yes	Yes	Yes	Yes	Yes

RELATED TASKS
"Specifying sequential organization for VSAM files"
"Specifying indexed organization for VSAM files"
"Specifying relative organization for VSAM files" on page 184
"Specifying access modes for VSAM files" on page 185
"Defining record lengths for VSAM files" on page 186
"Using file status keys" on page 239
"Using VSAM status codes (VSAM files only)" on page 240
"Defining VSAM files" on page 198

Specifying sequential organization for VSAM files

Identify VSAM ESDS files in a COBOL program with the ORGANIZATION IS SEQUENTIAL clause. You can access (read or write) records in sequential files only sequentially.

After you place a record in 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.

The following example shows typical FILE-CONTROL entries for a VSAM sequential file (ESDS):

```
SELECT S-FILE
    ASSIGN TO SEQUENTIAL-AS-FILE
    ORGANIZATION IS SEQUENTIAL
    ACCESS IS SEQUENTIAL
    FILE STATUS IS FSTAT-CODE VSAM-CODE.
```

RELATED CONCEPTS
"VSAM files" on page 180

Specifying indexed organization for VSAM files

Identify a VSAM KSDS file in a COBOL program by using the ORGANIZATION IS INDEXED clause. Code a prime key for the record by using the RECORD KEY clause. You can also use alternate keys and an alternate index.

```
RECORD KEY IS data-name
```

In the example above, *data-name* is the name of the prime key field as you define it
| in the record description entry in the DATA DIVISION. The prime key data item can
| be class alphabetic, alphanumeric, DBCS, numeric, or national. If it has USAGE
| NATIONAL, the prime key can be category national, or can be a national-edited,
| numeric-edited, national decimal, or national floating-point data item. The collation
of record keys is based on the binary value of the keys regardless of the class or category of the keys.

The following example shows the statements for a VSAM indexed file (KSDS) that is accessed dynamically. In addition to the primary key, COMMUTER-NO, an alternate key, LOCATION-NO, is specified:

```
SELECT I-FILE
    ASSIGN TO INDEXED-FILE
    ORGANIZATION IS INDEXED
    ACCESS IS DYNAMIC
    RECORD KEY IS IFILE-RECORD-KEY
    ALTERNATE RECORD KEY IS IFILE-ALTREC-KEY
    FILE STATUS IS FSTAT-CODE VSAM-CODE.
```

RELATED CONCEPTS
"VSAM files" on page 180

RELATED TASKS
"Using alternate keys"
"Using an alternate index"

RELATED REFERENCES
RECORD KEY clause (*Enterprise COBOL Language Reference*)
Classes and categories of data (*Enterprise COBOL Language Reference*)

Using alternate keys

In addition to the primary key, you can code one or more alternate keys for a VSAM KSDS file. By using alternate keys, you can access an indexed file to read records in some sequence other than the prime-key sequence.

Alternate keys do not need to be unique. More than one record could be accessed if alternate keys are coded to allow duplicates. For example, you could access the file through employee department rather than through employee number.

You define the alternate key in your COBOL program with the ALTERNATE RECORD KEY clause:

```
ALTERNATE RECORD KEY IS data-name
```

In the example above, *data-name* is the name of the alternate key field as you
| define it in the record description entry in the DATA DIVISION. Alternate key data
| items, like prime key data items, can be class alphabetic, alphanumeric, DBCS,
| numeric, or national. The collation of alternate keys is based on the binary value of
the keys regardless of the class or category of the keys.

Using an alternate index

To use an alternate index for a VSAM KSDS file, you need to define a data set called the *alternate index (AIX)* by using access method services.

The AIX contains one record for each value of a given alternate key. The records are in sequential order by alternate-key value. Each record contains the corresponding primary keys of all records in the associated indexed files that contain the alternate-key value.

RELATED TASKS
"Creating alternate indexes" on page 199

Specifying relative organization for VSAM files

Identify VSAM RRDS files in a COBOL program by using the `ORGANIZATION IS RELATIVE` clause. Use the `RELATIVE KEY IS` clause to associate each logical record with its relative record number.

The following example shows a relative-record data set (RRDS) that is accessed randomly by the value in the relative key:

```
SELECT R-FILE
    ASSIGN TO RELATIVE-FILE
    ORGANIZATION IS RELATIVE
    ACCESS IS RANDOM
    RELATIVE KEY IS RFILE-RELATIVE-KEY
    FILE STATUS IS FSTAT-CODE VSAM-CODE.
```

You can use a randomizing routine to associate a key value in each record with the relative record number for that record. Although there are many techniques to convert a record key to a relative record number, the most commonly used is the division/remainder technique. With this technique, you divide the key by a value equal to the number of slots in the data set to produce a quotient and remainder. When you add one to the remainder, the result is a valid relative record number.

Alternate indexes are not supported for VSAM RRDS.

RELATED CONCEPTS
"VSAM files" on page 180
"Fixed-length and variable-length RRDS"

RELATED TASKS
"Using variable-length RRDS"
"Defining VSAM files" on page 198

Fixed-length and variable-length RRDS

In an RRDS that has fixed-length records, each record occupies one slot. You store and retrieve records according to the relative record number of the slot. A variable-length RRDS does not have slots; instead, the free space that you define allows for more efficient record insertions.

When you load an RRDS that has fixed-length records, you have the option of skipping over slots and leaving them empty. When you load an RRDS that has variable-length records, you can skip over relative record numbers.

Using variable-length RRDS

To use relative-record data sets that have variable-length records, use VSAM variable-length RRDS if possible. But if you cannot use VSAM support, use *COBOL simulated variable-length RRDS*. Enterprise COBOL simulates variable-length RRDS using a VSAM KSDS when you use the `SIMVRD` runtime option.

The coding that you use in a COBOL program to identify and describe VSAM variable-length RRDS and COBOL simulated variable-length RRDS is similar. To use a variable-length RRDS, do the steps shown below, depending on whether you want to simulate an RRDS.

Table 27. **Steps for using variable-length RRDS**

Step	VSAM variable-length RRDS	COBOL simulated variable-length RRDS
1	Define the file with the `ORGANIZATION IS RELATIVE` clause.	Same
2	Use `FD` entries to describe the records with variable-length sizes.	Same, but you must also code `RECORD IS VARYING` in the `FD` entry of every COBOL program that accesses the data set.
3	Use the `NOSIMVRD` runtime option.	Use the `SIMVRD` runtime option.
4	Define the VSAM file through access-method services as an RRDS.	Define the VSAM file through access-method services as follows: `DEFINE CLUSTER INDEXED` `KEYS(4,0)` `RECORDSIZE(`*avg*`,`*m*`)` ***avg*** Is the average size of the COBOL records; strictly less than *m*. ***m*** Is greater than or equal to the maximum size COBOL record + 4.

In step 2 for simulated variable-length RRDS, coding other language elements that imply a variable-length record format does not give you COBOL simulated variable-length RRDS. For example, these elements alone do not provide correct file access:

- Multiple `FD` records of different lengths
- `OCCURS . . . DEPENDING ON` in the record definitions
- `RECORD CONTAINS` *integer-1* `TO` *integer-2* `CHARACTERS`

When you define the cluster in step 4 for simulated variable-length RRDS, observe these restrictions:

- Do not define an alternate index.
- Do not specify `KEYRANGE`.
- Do not specify `SPANNED`.

Also, use the `REUSE` parameter when you open for output a file that contains records.

Errors: When you work with simulated variable-length relative data sets and true VSAM RRDS data sets, an `OPEN` file status 39 occurs if the COBOL file definition and the VSAM data-set attributes do not match.

Specifying access modes for VSAM files

You can access records in VSAM sequential files only sequentially. You can access records in VSAM indexed and relative files in three ways: sequentially, randomly, or dynamically.

For sequential access, code `ACCESS IS SEQUENTIAL` in the `FILE-CONTROL` entry. Records in indexed files are then accessed in the order of the key field selected (either primary or alternate). Records in relative files are accessed in the order of the relative record numbers.

For random access, code ACCESS IS RANDOM in the FILE-CONTROL entry. Records in indexed files are then accessed according to the value you place in a key field. Records in relative files are accessed according to the value you place in the relative key.

For 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 write one program to perform both sequential and random processing, accessing some records in sequential order and others by their keys.

"Example: using dynamic access with VSAM files"

RELATED TASKS
"Reading records from a VSAM file" on page 192

Example: using dynamic access with VSAM files

Suppose that you have an indexed file of employee records, and the employee's hourly wage forms the record key.

If your program processes those employees who earn between $15.00 and $20.00 per hour and those who earn $25.00 per hour and above, using dynamic access of VSAM files, the program would:

1. Retrieve the first record randomly (with a random-retrieval READ) based on the key of 1500.
2. Read sequentially (using READ NEXT) until the salary field exceeds 2000.
3. Retrieve the next record randomly, based on a key of 2500.
4. Read sequentially until the end of the file.

RELATED TASKS
"Reading records from a VSAM file" on page 192

Defining record lengths for VSAM files

You can define VSAM records to be fixed or variable in length. COBOL determines the record format from the RECORD clause and the record descriptions that are associated with the FD entry for a file.

Because the concept of blocking has no meaning for VSAM files, you can omit the BLOCK CONTAINS clause. The clause is syntax-checked, but it has no effect on how the program runs.

RELATED TASKS
"Defining fixed-length records"
"Defining variable-length records" on page 187
Enterprise COBOL Compiler and Runtime Migration Guide

RELATED REFERENCES
"FILE SECTION entries" on page 14

Defining fixed-length records

To define VSAM records as fixed length, use one of these coding options.

Table 28. **Definition of VSAM fixed-length records**

`RECORD` clause	Clause format	Record length	Comments
Code `RECORD CONTAINS` *integer*.	1	Fixed in size with a length of *integer-3* bytes	The lengths of the level-01 record description entries associated with the file do not matter.
Omit the `RECORD` clause, but code all level-01 records that are associated with the file as the same size; and code none with an `OCCURS DEPENDING ON` clause.		The fixed size that you coded	

RELATED REFERENCES
RECORD clause (*Enterprise COBOL Language Reference*)

Defining variable-length records

To define VSAM records as variable length, use one of these coding options.

Table 29. **Definition of VSAM variable-length records**

`RECORD` clause	Clause format	Maximum record length	Comments
Code `RECORD IS VARYING FROM` *integer-6* `TO` *integer-7*.	3	*integer-7* bytes	The lengths of the level-01 record description entries associated with the file do not matter.
Code `RECORD IS VARYING`.	3	Size of the largest level-01 record description entry associated with the file	The compiler determines the maximum record length.
Code `RECORD CONTAINS` *integer-4* `TO` *integer-5*.	2	*integer-5* bytes	The minimum record length is *integer-4* bytes.
Omit the `RECORD` clause, but code multiple level-01 records that are associated with the file and are of different sizes or contain an `OCCURS DEPENDING ON` clause.		Size of the largest level-01 record description entry associated with the file	The compiler determines the maximum record length.

When you specify a `READ INTO` statement for a format-V file, the record size that is read for that file is used in the `MOVE` statement generated by the compiler. Consequently, you might not get the result you expect if the record read in does not correspond to the level-01 record description. All other rules of the `MOVE` statement apply. For example, when you specify a `MOVE` statement for a format-V record read in by the `READ` statement, the size of the record corresponds to its level-01 record description.

RELATED REFERENCES
RECORD clause (*Enterprise COBOL Language Reference*)

Coding input and output statements for VSAM files

Use the COBOL statements shown below to process VSAM files.

`OPEN` To connect the VSAM data set to your COBOL program for processing.

`WRITE` To add records to a file or load a file.

`START` To establish the current location in the cluster for a `READ NEXT` statement.

`START` does not retrieve a record; it only sets the current record pointer.

`READ` and `READ NEXT`
To retrieve records from a file.

`REWRITE`
To update records.

`DELETE` To logically remove records from indexed and relative files only.

`CLOSE` To disconnect the VSAM data set from your program.

All of the following factors determine which input and output statements you can use for a given VSAM data set:

- Access mode (sequential, random, or dynamic)
- File organization (ESDS, KSDS, or RRDS)
- Mode of `OPEN` statement (`INPUT`, `OUTPUT`, `I-O`, or `EXTEND`)

The following table shows the possible combinations of statements and open modes for sequential files (ESDS). The X indicates that you can use a statement with the open mode shown at the top of the column.

Table 30. **I/O statements for VSAM sequential files**

Access mode	COBOL statement	`OPEN INPUT`	`OPEN OUTPUT`	`OPEN I-O`	`OPEN EXTEND`
Sequential	`OPEN`	X	X	X	X
	`WRITE`		X		X
	`START`				
	`READ`	X		X	
	`REWRITE`			X	
	`DELETE`				
	`CLOSE`	X	X	X	X

The following table shows the possible combinations of statements and open modes you can use with indexed (KSDS) files and relative (RRDS) files. The X indicates that you can use the statement with the open mode shown at the top of the column.

Table 31. **I/O statements for VSAM relative and indexed 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
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	

The fields that you code in the `FILE STATUS` clause are updated by VSAM after each input-output statement to indicate the success or failure of the operation.

RELATED CONCEPTS
"File position indicator"

RELATED TASKS
"Opening a file (ESDS, KSDS, or RRDS)" on page 190
"Reading records from a VSAM file" on page 192
"Updating records in a VSAM file" on page 193
"Adding records to a VSAM file" on page 194
"Replacing records in a VSAM file" on page 194
"Deleting records from a VSAM file" on page 195
"Closing VSAM files" on page 195

RELATED REFERENCES
File status key (*Enterprise COBOL Language Reference*)

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 in your program. It is set by successful `OPEN`, `START`, `READ`, and `READ NEXT` statements.

Subsequent READ or READ NEXT 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.

RELATED TASKS
"Reading records from a VSAM file" on page 192

Opening a file (ESDS, KSDS, or RRDS)

Before you can use WRITE, START, READ, REWRITE, or DELETE statements to process records in a file, you must first open the file with an OPEN statement.

File availability and creation affect OPEN processing, optional files, and file status codes 05 and 35. For example, if you open a file that is neither optional nor available in EXTEND, I-O, or INPUT mode, you get file status 35 and the OPEN statement fails. If the file is OPTIONAL, the same OPEN statement creates the file and returns file status 05.

An OPEN operation works successfully only when you set fixed file attributes in the DD statement or data-set label for a file and specify consistent attributes for the file in the SELECT clause and FD entries of your COBOL program. Mismatches in the following items result in a file status code 39 and the failure of the OPEN statement:

- Attributes for file organization (sequential, relative, or indexed)
- Prime record key
- Alternate record keys
- Maximum record size
- Record type (fixed or variable)

How you code the OPEN statement for a VSAM file depends on whether the file is empty (a file that has never contained records) or loaded. For either type of file, your program should check the file status key after each OPEN statement.

RELATED TASKS
"Opening an empty file"
"Opening a loaded file (a file with records)" on page 192

RELATED REFERENCES
"Statements to load records into a VSAM file" on page 192

Opening an empty file

To open a file that has never contained records (an empty file), use a form of the OPEN statement.

Depending on the type of file that you are opening, use one of the following statements:

- OPEN OUTPUT for ESDS files.
- OPEN OUTPUT or OPEN EXTEND for KSDS and RRDS files. (Either coding has the same effect.) If you coded the file for random or dynamic access and the file is optional, you can use OPEN I-O.

Optional files are files that are not necessarily present each time a program is run. You can define files opened in INPUT, I-O, or OUTPUT mode as optional by defining them with the SELECT OPTIONAL clause in the FILE-CONTROL paragraph.

Initially loading a file sequentially: Initially loading a file means writing records into the file for the first time. Doing so is not the same as writing records into a file from which all previous records have been deleted. To initially load a VSAM file:

1. Open the file.
2. Use sequential processing (ACCESS IS SEQUENTIAL). (Sequential processing is faster than random or dynamic processing.)
3. Use WRITE to add a record to the file.

Using OPEN OUTPUT to load a VSAM file significantly improves the performance of your program. Using OPEN I-O or OPEN EXTEND has a negative effect on the performance of your program.

When you load VSAM indexed files sequentially, you optimize both loading performance and subsequent processing performance, because sequential processing maintains user-defined free space. Future insertions will be more efficient.

With ACCESS IS SEQUENTIAL, you must write the records in ascending RECORD KEY order.

When you load VSAM relative files sequentially, the records are placed in the file in the ascending order of relative record numbers.

Initially loading a file randomly or dynamically: You can use random or dynamic processing to load a file, but they are not as efficient as sequential processing. Because VSAM does not support random or dynamic processing, COBOL has to perform some extra processing to enable you to use ACCESS IS RANDOM or ACCESS IS DYNAMIC with OPEN OUTPUT or OPEN I-O. These steps prepare the file for use and give it the status of a loaded file because it has been used at least once.

In addition to extra overhead for preparing files for use, random processing does not consider any user-defined free space. As a result, any future insertions might be inefficient. Sequential processing maintains user-defined free space.

| When you are loading an extended-format VSAM data set, file status 30 will occur
| for the OPEN if z/OS DFSMS system-managed buffering sets the buffering to local
| shared resources (LSR). To successfully load the VSAM data set in this case, specify
| ACCBIAS=USER in the DD AMP parameter for the VSAM data set to bypass
| system-managed buffering.

Loading a VSAM data set with access method services: You can load or update a VSAM data set by using the access method services REPRO command. Use REPRO whenever possible.

RELATED TASKS
"Opening a loaded file (a file with records)" on page 192

RELATED REFERENCES
"Statements to load records into a VSAM file" on page 192
REPRO (*z/OS DFSMS: Access Method Services for Catalogs*)

Statements to load records into a VSAM file

Use the statements shown below to load records into a VSAM file.

Table 32. **Statements to load records into a VSAM file**

Division	ESDS	KSDS	RRDS
ENVIRONMENT DIVISION	SELECT ASSIGN FILE STATUS PASSWORD ACCESS MODE	SELECT ASSIGN ORGANIZATION IS INDEXED RECORD KEY ALTERNATE RECORD KEY FILE STATUS PASSWORD ACCESS MODE	SELECT ASSIGN ORGANIZATION IS RELATIVE RELATIVE KEY FILE STATUS PASSWORD ACCESS MODE
DATA DIVISION	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

RELATED TASKS
"Opening an empty file" on page 190
"Updating records in a VSAM file" on page 193

Opening a loaded file (a file with records)

To open a file that already contains records, use OPEN INPUT, OPEN I-O, or OPEN EXTEND.

If you open a VSAM entry-sequenced or relative-record file as EXTEND, the added records are placed after the last existing records in the file.

If you open a VSAM key-sequenced file as EXTEND, each record you add must have a record key higher than the highest record in the file.

RELATED TASKS
"Opening an empty file" on page 190
"Working with VSAM data sets under z/OS and UNIX" on page 197

RELATED REFERENCES
"Statements to load records into a VSAM file"
z/OS DFSMS: Access Method Services for Catalogs

Reading records from a VSAM 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. Your program should check the file status key after each READ.

You can retrieve records in VSAM sequential files only in the sequence in which they were written.

You can retrieve records in VSAM indexed and relative record files in any of the following ways:

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 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 statement to set the file position indicator to point to a particular record. When you code START followed by READ NEXT, the next record is read and the file position indicator is reset to the next record. You can move the file position indicator randomly by using START, but all reading is done sequentially from that point.

```
START file-name KEY IS EQUAL TO ALTERNATE-RECORD-KEY
```

When a direct READ is performed for a VSAM indexed file, based on an alternate index for which duplicates exist, only the first record in the data set (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 code of 02 is returned if there are more records with the same alternate key value to be read; a code of 00 is returned when the last record with that key value has been read.

RELATED CONCEPTS
"File position indicator" on page 189

RELATED TASKS
"Specifying access modes for VSAM files" on page 185

Updating records in a VSAM file

To update a VSAM file, use these PROCEDURE DIVISION statements.

Table 33. **Statements to update records in a VSAM file**

Access method	ESDS	KSDS	RRDS
ACCESS IS SEQUENTIAL	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE 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

Table 33. **Statements to update records in a VSAM file** *(continued)*

Access method	ESDS	KSDS	RRDS

`ACCESS IS RANDOM`	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	`OPEN I-O`
		`READ NEXT`
		`WRITE`
		`REWRITE`
		`START`
		`DELETE`
		`CLOSE`
			`OPEN I-O`
			`READ NEXT`
			`WRITE`
			`REWRITE`
			`START`
			`DELETE`
			`CLOSE`

`ACCESS IS DYNAMIC` (random processing)	Not applicable	`OPEN I-O`
		`READ`
		`WRITE`
		`REWRITE`
		`DELETE`
		`CLOSE`
			`OPEN I-O`
			`READ`
			`WRITE`
			`REWRITE`
			`DELETE`
			`CLOSE`

RELATED REFERENCES
"Statements to load records into a VSAM file" on page 192

Adding records to a VSAM file

Use the COBOL `WRITE` statement to add a record to a file without replacing any existing records. The record to be added must not be larger than the maximum record size that you set when you defined the file. Your program should 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 VSAM file that has been opened with either `OUTPUT` or `EXTEND`.

Sequential files are always written sequentially.

For indexed files, you must write new records in ascending key sequence. If you open the file `EXTEND`, the record keys of the records to be added must be higher than the highest primary record key on the file when you opened the file.

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`, you can write the records in any order.

Replacing records in a VSAM file

To replace a record in a VSAM file, use `REWRITE` on a file that you opened as `I-O`. If the file was not opened as `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 replacement record must be the same as the length of the original record. For indexed files or variable-length relative files, you can change the length of the record you replace.

To replace a record randomly or dynamically, you do not have to first READ the record. Instead, locate the record you want to replace as follows:

- 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 VSAM file

To remove an existing record from an indexed or relative file, open the file I-O and use the DELETE statement. You cannot use DELETE on a sequential file.

When you use ACCESS IS SEQUENTIAL or the file contains spanned records, your program must first read the record to be deleted. The DELETE then removes the record that was 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 you use ACCESS IS RANDOM or ACCESS IS DYNAMIC, your program does not have to first read the record to be deleted. To delete a record, move the key of the record to be deleted to the RECORD KEY data item, and then issue the DELETE. Your program should check the file status key after each DELETE statement.

Closing VSAM files

Use the CLOSE statement to disconnect your program from a VSAM file. If you try to close a file that is already closed, you will get a logic error. Check the file status key after each CLOSE statement.

If you do not close a VSAM file, the file is automatically closed for you under the following conditions, except for files defined in any OS/VS COBOL programs in the run unit:

- When the run unit ends normally, all open files defined in any COBOL programs in the run unit are closed.
- When the run unit ends abnormally, if the TRAP(ON) runtime option has been set, all open files defined in any COBOL programs in the run unit are closed.
- When Language Environment condition handling has completed and the application resumes in a routine other than where the condition occurred, open files defined in any COBOL programs in the run unit that might be called again and reentered are closed.

  You can change the location where a program resumes after a condition is handled. To make this change, you can, for example, move the resume cursor with the CEEMRCR callable service or use language constructs such as a C longjmp statement.
- When you issue CANCEL for a COBOL subprogram, any open nonexternal files defined in that program are closed.
- When a COBOL subprogram with the INITIAL attribute returns control, any open nonexternal files defined in that program are closed.
- When a thread of a multithreaded application ends, both external and nonexternal files that were opened from within that same thread are closed.

File status key data items in the DATA DIVISION are set when these implicit CLOSE operations are performed, but your EXCEPTION/ERROR and LABEL declaratives are not invoked.

Errors: If you open a VSAM file in a multithreaded application, you must close it from the same thread of execution. Attempting to close the file from a different thread results in a close failure with file-status condition 90.

Handling errors in VSAM files

When an input or output statement operation fails, COBOL does not perform corrective action for you.

All OPEN and CLOSE errors with a VSAM file, whether logical errors in your program or input/output errors on the external storage media, return control to your COBOL program even if you coded no DECLARATIVE and no FILE STATUS clause.

If any other input or output statement operation fails, you choose whether your program will continue running after a less-than-severe error.

COBOL provides these ways for you to intercept and handle certain VSAM input and output errors:
- End-of-file phrase (AT END)
- EXCEPTION/ERROR declarative
- FILE STATUS clause (file status key and VSAM status code)
- INVALID KEY phrase

You should define a status key for each VSAM file that you define in your program. Check the status key value after each input or output request, especially OPEN and CLOSE.

If you do not code a file status key or a declarative, serious VSAM processing errors will cause a message to be issued and a Language Environment condition to be signaled, which will cause an abend if you specify the runtime option ABTERMENC(ABEND).

RELATED TASKS
"Handling errors in input and output operations" on page 235
"Using VSAM status codes (VSAM files only)" on page 240

RELATED REFERENCES
VSAM macro return and reason codes (*z/OS DFSMS Macro Instructions for Data Sets*)

Protecting VSAM files with a password

Although the preferred security mechanism on a z/OS system is RACF(R), Enterprise COBOL also supports using explicit passwords on VSAM files to prevent unauthorized access and update.

To use explicit passwords, code the PASSWORD clause in the FILE-CONTROL paragraph. Use this clause only if the catalog entry for the files includes a read or an update password:

- If the catalog entry includes a read password, you cannot open and access the file in a COBOL program unless you use the PASSWORD clause in the FILE-CONTROL paragraph and describe it in the DATA DIVISION. The *data-name* referred to must contain a valid password when the file is opened.
- If the catalog entry includes an update password, you can open and access it, but not update it, unless you code the PASSWORD clause in the FILE-CONTROL paragraph and describe it in the DATA DIVISION.
- If the catalog entry includes both a read password and an update password, specify the update password to both read and update the file in your program.

If your program only retrieves records and does not update them, you need only the read password. If your program loads files or updates them, you need to specify the update password that was cataloged.

For indexed files, the PASSWORD data item for the RECORD KEY must contain the valid password before the file can be successfully opened.

If you password-protect a VSAM indexed file, you must also password-protect each alternate index in order to be fully password protected. Where you place the PASSWORD clause is important because each alternate index has its own password. The PASSWORD clause must directly follow the key clause to which it applies.

"Example: password protection for a VSAM indexed file"

Example: password protection for a VSAM indexed file

The following example shows the COBOL code used for a VSAM indexed file that has password protection.

```
. . .
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT LIBFILE
      ASSIGN TO PAYMAST
      ORGANIZATION IS INDEXED
      RECORD KEY IS EMPL-NUM
         PASSWORD IS BASE-PASS
      ALTERNATE RECORD KEY IS EMPL-PHONE
         PASSWORD IS PATH1-PASS
. . .
WORKING-STORAGE SECTION.
01  BASE-PASS         PIC X(8) VALUE "25BSREAD".
01  PATH1-PASS        PIC X(8) VALUE "25ATREAD".
```

Working with VSAM data sets under z/OS and UNIX

There are some special coding considerations for VSAM files under z/OS and UNIX for access method services (IDCAMS) commands, environment variables, and JCL.

A VSAM file is *available* if all of the following conditions are true:

- You define it using access method services.
- You define it for your program by providing a DD statement, an environment variable, or an ALLOCATE command.
- It has previously contained a record.

A VSAM file is *unavailable* if it has never contained a record, even if you have defined it.

You always get a return code of zero on completion of the OPEN statement for a VSAM sequential file.

Use the access method services REPRO command to empty a file. Deleting records in this manner resets the high-use relative byte address (RBA) of the file to zero. The file is effectively empty and appears to COBOL as if it never contained a record.

RELATED TASKS
"Defining files to the operating system" on page 10
"Defining VSAM files"
"Creating alternate indexes" on page 199
"Allocating VSAM files" on page 201
"Sharing VSAM files through RLS" on page 202

Defining VSAM files

You can process VSAM entry-sequenced, key-sequenced, and relative-record data sets in Enterprise COBOL only after you define them through access method services (IDCAMS).

A VSAM *cluster* is a logical definition for a VSAM data set and has one or two components:

- The data component of a VSAM cluster contains the data records.
- The index component of a VSAM key-sequenced cluster consists of the index records.

Use the DEFINE CLUSTER access-method services command to define VSAM data sets (clusters). This process includes creating an entry in an integrated catalog without any data transfer. Define the following information about the cluster:

- Name of the entry
- Name of the catalog to contain this definition and its password (can use default name)
- Organization (sequential, indexed, or relative)
- Device and volumes that the data set will occupy
- Space required for the data set
- Record size and control interval sizes (CISIZE)
- Passwords (if any) required for future access

Depending on what kind of data set is in the cluster, also define the following information for each cluster:

- For VSAM indexed data sets (KSDS), specify length and position of the prime key in the records.
- For VSAM fixed-length relative-record data sets (RRDS), specify the record size as greater than or equal to the maximum size COBOL record:

  ```
  DEFINE CLUSTER NUMBERED
  RECORDSIZE(n,n)
  ```

  When you define a data set in this way, all records are padded to the fixed slot size *n*. If you use the RECORD IS VARYING ON *data-name* form of the RECORD clause, a WRITE or REWRITE uses the length specified in DEPENDING ON *data-name* as the length of the record to be transferred by VSAM. This data is then padded to the fixed slot size. READ statements always return the fixed slot size in the DEPENDING ON *data-name*.

- For VSAM variable-length relative-record data sets (RRDS), specify the average size COBOL record expected and the maximum size COBOL record expected:

  ```
  DEFINE CLUSTER NUMBERED
  RECORDSIZE(avg,m)
  ```

  The average size COBOL record expected must be less than the maximum size COBOL record expected.
- For COBOL simulated variable-length relative-record data sets, specify the average size of the COBOL records and a size that is greater than or equal to the maximum size COBOL record plus 4:

  ```
  DEFINE CLUSTER INDEXED
  KEYS(4,0)
  RECORDSIZE(avg,m)
  ```

  The average size COBOL record expected must be less than the maximum size COBOL record expected.

RELATED TASKS
"Creating alternate indexes"
"Allocating VSAM files" on page 201
"Specifying relative organization for VSAM files" on page 184

RELATED REFERENCES
z/OS DFSMS: Access Method Services for Catalogs

Creating alternate indexes

An alternate index provides access to the records in a data set that uses more than one key. It accesses records in the same way as the prime index key of an indexed data set (KSDS).

When planning to use an alternate index, you must know:
- The type of data set (base cluster) with which the index will be associated
- Whether the keys will be unique or not unique
- Whether the index is to be password protected
- Some of the performance aspects of using alternate indexes

Because an alternate index is, in practice, a VSAM data set that contains pointers to the keys of a VSAM data set, you must define the alternate index and the alternate index path (the entity that establishes the relationship between the alternate index and the prime index). After you define an alternate index, make a catalog entry to establish the relationship (or path) between the alternate index and its base cluster. This path allows you to access the records of the base cluster through the alternate keys.

To use an alternate index, do these steps:
1. Define the alternate index by using the `DEFINE ALTERNATEINDEX` command. In it, specify these items:
   - Name of the alternate index
   - Name of its related VSAM indexed data set
   - Location in the record of any alternate indexes and whether they are unique
   - Whether alternate indexes are to be updated when the data set is changed
   - Name of the catalog to contain this definition and its password (can use default name)

In your COBOL program, the alternate index is identified solely by the ALTERNATE RECORD KEY clause in the FILE-CONTROL paragraph. The ALTERNATE RECORD KEY definitions must match the definitions in the catalog entry. Any password entries that you cataloged should be coded directly after the ALTERNATE RECORD KEY phrase.

2. Relate the alternate index to the base cluster (the data set to which the alternate index gives you access) by using the DEFINE PATH command. In it, specify these items:
   - Name of the path
   - Alternate index to which the path is related
   - Name of the catalog that contains the alternate index

   The base cluster and alternate index are described by entries in the same catalog.
3. Load the VSAM indexed data set.
4. Build the alternate index by using (typically) the BLDINDEX command. Identify the input file as the indexed data set (base cluster) and the output file as the alternate index or its path. BLDINDEX reads all the records in the VSAM indexed data set (or base cluster) and extracts the data needed to build the alternate index.

   Alternatively, you can use the runtime option AIXBLD to build the alternate index at run time. However, this option might adversely affect performance.

"Example: entries for alternate indexes"

RELATED TASKS
"Using an alternate index" on page 183

RELATED REFERENCES
AIXBLD (COBOL only) (*Language Environment Programming Reference*)

Example: entries for alternate indexes

The following example maps the relationships between the COBOL FILE-CONTROL entry and the DD statements or environment variables for a VSAM indexed file that has two alternate indexes.

Using JCL:

```
//MASTERA     DD   DSNAME=clustername,DISP=OLD      (1)
//MASTERA1    DD   DSNAME=path1,DISP=OLD            (2)
//MASTERA2    DD   DSNAME=path2,DISP=OLD            (3)
```

Using environment variables:

```
export MASTERA=DSN(clustername),OLD                 (1)
export MASTERA=DSN(path1),OLD                       (2)
export MASTERA=DSN(path2),OLD                       (3)
. . .
FILE-CONTROL.
  SELECT MASTER-FILE ASSIGN TO MASTERA              (4)
      RECORD KEY IS EM-NAME
      PASSWORD IS PW-BASE                           (5)
      ALTERNATE RECORD KEY IS EM-PHONE              (6)
          PASSWORD IS PW-PATH1
      ALTERNATE RECORD KEY IS EM-CITY               (7)
          PASSWORD IS PW-PATH2.
```

(1) The base cluster name is *clustername*.

(2) The name of the first alternate index path is *path1*.

(3) The name of the second alternate index path is *path2*.

(4) The ddname or environment variable name for the base cluster is specified with the ASSIGN clause.

(5) Passwords immediately follow their indexes.

(6) The key EM-PHONE relates to the first alternate index.

(7) The key EM-CITY relates to the second alternate index.

RELATED TASKS
"Creating alternate indexes" on page 199

Allocating VSAM files

You must predefine and catalog all VSAM data sets through the access method services DEFINE command. Most of the information about a VSAM data set is in the catalog, so you need to specify only minimal DD or environment variable information.

Allocation of VSAM files (indexed, relative, and sequential) follows the general rules for the allocation of COBOL files.

When you use an environment variable to allocate a VSAM file, the variable name must be in uppercase. Usually the input and data buffers are the only variables that you are concerned about. You must specify these options in the order shown, but no others:

1. DSN(*dsname*), where *dsname* is the name of the base cluster
2. OLD or SHR

The basic DD statement that you need for VSAM files and the corresponding export command are these:

```
//ddname   DD   DSN=dsname,DISP=SHR,AMP=AMORG
export  evname="DSN(dsname),SHR"
```

In either case, *dsname* must be the same as the name used in the access method services DEFINE CLUSTER or DEFINE PATH command. DISP must be OLD or SHR because the data set is already cataloged. If you specify MOD when using JCL, the data set is treated as OLD.

AMP is a VSAM JCL parameter that supplements the information that the program supplies about the data set. AMP takes effect when your program opens the VSAM file. Any information that you set through the AMP parameter takes precedence over the information that is in the catalog or that the program supplies. The AMP parameter is required only under the following circumstances:

- You use a dummy VSAM data set. For example,

```
//ddname  DD   DUMMY,AMP=AMORG
```

- You request additional index or data buffers. For example,

```
//ddname   DD  DSN=VSAM.dsname,DISP=SHR,
//             AMP=('BUFNI=4,BUFND=8')
```

You cannot specify AMP if you allocate a VSAM data set with an environment variable.

For a VSAM base cluster, specify the same system-name (ddname or environment variable name) that you specify in the ASSIGN clause after the SELECT clause.

When you use alternate indexes in your COBOL program, you must specify not only a system-name (using a DD statement or environment variable) for the base cluster, but also a system-name for each alternate index path. No language mechanism exists to explicitly declare system-names for alternate index paths within the program. Therefore, you must adhere to the following guidelines for forming the system-name (ddname or environment variable name) for each alternate index path:

- Concatenate the base cluster name with an integer.
- Begin with 1 for the path associated with the first alternate record defined for the file in your program (ALTERNATE RECORD KEY clause of the FILE-CONTROL paragraph).
- Increment by 1 for the path associated with each successive alternate record definition for that file.

For example, if the system-name of a base cluster is ABCD, the system-name for the first alternate index path defined for the file in your program is ABCD1, the system-name for the second alternate index path is ABCD2, and so on.

If the length of the base cluster system-name together with the sequence number exceeds eight characters, the base cluster portion of the system-name is truncated on the right to reduce the concatenated result to eight characters. For example, if the system-name of a base cluster is ABCDEFGH, the system name of the first alternate index path is ABCDEFG1, the tenth is ABCDEF10, and so on.

RELATED TASKS
"Allocating files" on page 149

RELATED REFERENCES
MVS JCL Reference

Sharing VSAM files through RLS

By using the VSAM JCL parameter RLS, you can specify record-level sharing with VSAM. Specifying RLS is the only way to request the RLS mode when running COBOL programs.

Use RLS=CR when consistent read protocols are required, and RLS=NRI when no read integrity protocols are required. You cannot specify RLS if you allocate your VSAM data set with an environment variable

RELATED TASKS
"Preventing update problems with VSAM files in RLS mode"
"Handling errors in VSAM files in RLS mode" on page 203

RELATED REFERENCES
"Restrictions when using RLS" on page 203

Preventing update problems with VSAM files in RLS mode

When you open a VSAM data set in RLS mode for I-O (updates), the first READ causes an exclusive lock of the record regardless of the value of RLS (RLS=CR or RLS=NRI) that you specify.

If the COBOL file is defined as ACCESS RANDOM, VSAM releases the exclusive lock on the record after a WRITE or REWRITE statement is issued or a READ statement is issued for another record. When a WRITE or REWRITE is done, VSAM writes the record immediately.

However, if the COBOL file is defined as ACCESS DYNAMIC, VSAM does not release the exclusive lock on the record after a WRITE or REWRITE statement, nor after a READ statement, unless the I-O statement causes VSAM to move to another control interval (CI). As a result, if a WRITE or REWRITE was done, VSAM does not write the record until processing is moved to another CI and the lock is released. When you use ACCESS DYNAMIC, one way to cause the record to be written immediately, to release the exclusive lock immediately, or both, is to define the VSAM data set to allow only one record per CI.

Specifying RLS=CR locks a record and prevents an update to it until another READ is requested for another record. While a lock on the record being read is in effect, other users can request a READ for the same record, but they cannot update the record until the read lock is released. When you specify RLS=NRI, no lock will be in effect when a READ for input is issued. Another user might update the record.

The locking rules for RLS=CR can cause the application to wait for availability of a record lock. This wait might slow down the READ for input. You might need to modify your application logic to use RLS=CR. Do not use the RLS parameter for batch jobs that update nonrecoverable spheres until you are sure that the application functions correctly in a multiple-updater environment.

When you open a VSAM data set in RLS mode for INPUT or I-O processing, it is good to issue an OPEN or START *immediately* before a READ. If there is a delay between the OPEN or START and the READ, another user might add records before the record on which the application is positioned after the OPEN or START. The COBOL run time points explicitly to the beginning of the VSAM data set at the time when OPEN was requested, but another user might add records that would alter the true beginning of the VSAM data set if the READ is delayed.

Restrictions when using RLS

When you use RLS mode, several restrictions apply to VSAM cluster attributes and to runtime options.

Be aware of these restrictions:

- The VSAM cluster attributes KEYRANGE and IMBED are not supported when you open a VSAM file.
- The VSAM cluster attribute REPLICATE is not recommended because the benefits are negated by the system-wide buffer pool and potentially large CF cache structure in the storage hierarchy.
- The AIXBLD runtime option is not supported when you open a VSAM file because VSAM does not allow an empty path to be opened. If you need the AIXBLD runtime option to build the alternate index data set, open the VSAM data set in non-RLS mode.
- The SIMVRD runtime option is not supported for VSAM files.
- Temporary data sets are not allowed.

Handling errors in VSAM files in RLS mode

If your application accesses a VSAM data set in RLS mode, be sure to check the file status and VSAM feedback codes after *each* request.

If your application encounters “SMSVSAM server not available” while processing input or output, explicitly close the VSAM file before you try to open it again. VSAM generates return code 16 for such failures, and there is no feedback code. You can have COBOL programs check the first 2 bytes of the second file status area for VSAM return code 16. The COBOL run time generates message IGZ0205W and automatically closes the file if the error occurs during OPEN processing.

All other RLS mode errors return a VSAM return code of 4, 8, or 12.

RELATED TASKS
"Using VSAM status codes (VSAM files only)" on page 240

Improving VSAM performance

Your system programmer is most likely responsible for tuning the performance of COBOL and VSAM. As an application programmer, you can control the aspects of VSAM that are listed below.

Table 34. **Methods for improving VSAM performance**

Aspect of VSAM	What you can do	Rationale and comments
Invoking access methods service	Build your alternate indexes in advance, using IDCAMS.	
Buffering	For sequential access, request more data buffers; for random access, request more index buffers. Specify both BUFND and BUFNI when ACCESS IS DYNAMIC. Avoid coding additional buffers unless your application will run interactively; then code buffers only when response-time problems arise that might be caused by delays in input and output.	The default is one index (BUFNI) and two data buffers (BUFND).
Loading records, using access methods services	Use the access methods service REPRO command when: • The target indexed data set already contains records. • The input sequential data set contains records to be updated or inserted into the indexed data set. If you use a COBOL program to load the file, use OPEN OUTPUT and ACCESS SEQUENTIAL.	The REPRO command can update an indexed data set as fast or faster than any COBOL program under these conditions.
File access modes	For best performance, access records sequentially.	Dynamic access is less efficient than sequential access, but more efficient than random access. Random access results in increased EXCPs because VSAM must access the index for each request.
Key design	Design the key in the records so that the high-order portion is relatively constant and the low-order portion changes often.	This method compresses the key best.

Table 34. **Methods for improving VSAM performance** *(continued)*

Aspect of VSAM	What you can do	Rationale and comments
Multiple alternate indexes	Avoid using multiple alternate indexes.	Updates must be applied through the primary paths and are reflected through multiple alternate paths, perhaps slowing performance.
Relative file organization	Use VSAM fixed-length relative data sets rather than VSAM variable-length relative data sets.	Although not as space efficient, VSAM fixed-length relative data sets are more runtime efficient than VSAM variable-length relative data sets, which have performance characteristics comparable to COBOL simulated relative data sets.
Control interval sizes (`CISZ`)	Provide your system programmer with information about the data access and future growth of your VSAM data sets. From this information, your system programmer can determine the best control interval size (`CISZ`) and FREESPACE size (`FSPC`). Choose proper values for `CISZ` and `FSPC` to minimize control area (CA) splits. You can diagnose the current number of CA splits by issuing the `LISTCAT ALL` command on the cluster, and then compress (using `EXPORT`, `IMPORT`, or `REPRO`) the cluster to omit all CA splits periodically.	VSAM calculates `CISZ` to best fit the direct-access storage device (DASD) usage algorithm, which might not, however, be efficient for your application. An average `CISZ` of 4K is suitable for most applications. A smaller `CISZ` means faster retrieval for random processing at the expense of inserts (that is, more `CISZ` splits and therefore more space in the data set). A larger `CISZ` results in the transfer of more data across the channel for each `READ`. This is more efficient for sequential processing, similar to a large `OS BLKSIZE`. Many control area (CA) splits are unfavorable for VSAM performance. The FREESPACE value can affect CA splits, depending on how the file is used.

RELATED TASKS
"Specifying access modes for VSAM files" on page 185
Deciding the size of a virtual resource pool (*z/OS DFSMS: Using Data Sets*)
Selecting the optimal percentage of free space (*z/OS DFSMS: Using Data Sets*)

RELATED REFERENCES
z/OS DFSMS: Access Method Services for Catalogs

Chapter 11. Processing line-sequential files

Line-sequential files reside in the hierarchical file system (HFS) and contain only printable characters and certain control characters as data. Each record ends with an EBCDIC new-line character (X'15'), which is not included in the record length.

Because these are sequential files, records are placed one after another according to entry order. Your program can process these files only sequentially, retrieving (with the READ statement) records in the same order as they are in the file. A new record is placed after the preceding record.

To process line-sequential files in your program, use COBOL language statements that:

- Identify and describe the files in the ENVIRONMENT DIVISION and the DATA DIVISION
- Process the records in the files in the PROCEDURE DIVISION

After you have created a record, you cannot change its length or its position in the file, and you cannot delete it.

RELATED TASKS
"Defining line-sequential files and records in COBOL"
"Describing the structure of a line-sequential file" on page 208
"Coding input-output statements for line-sequential files" on page 209
"Handling errors in line-sequential files" on page 212
"Defining and allocating line-sequential files" on page 209
UNIX System Services User's Guide

RELATED REFERENCES
"Allowable control characters" on page 208

Defining line-sequential files and records in COBOL

Use the FILE-CONTROL paragraph in the ENVIRONMENT DIVISION to define the files in a COBOL program as line-sequential files, and to associate the files with the corresponding external file-names (ddnames or environment variable names).

An external file-name is the name by which a file is known to the operating system. In the following example, COMMUTER-FILE is the name that your program uses for the file; COMMUTR is the external name:

```
FILE-CONTROL.
    SELECT COMMUTER-FILE
    ASSIGN TO COMMUTR
    ORGANIZATION IS LINE SEQUENTIAL
    ACCESS MODE IS SEQUENTIAL
    FILE STATUS IS ECODE.
```

The ASSIGN *assignment-name* clause must not include an organization field (S- or AS-) before the external name. The ACCESS phrase and the FILE STATUS clause are optional.

RELATED TASKS
"Describing the structure of a line-sequential file" on page 208

"Coding input-output statements for line-sequential files" on page 209
"Defining and allocating line-sequential files" on page 209

RELATED REFERENCES
"Allowable control characters"

Allowable control characters

The control characters shown in the table below are the only characters other than printable characters that line-sequential files can contain. The hexadecimal values are in EBCDIC.

Hexadecimal value	Control character
X'05'	Horizontal tab
X'0B'	Vertical tab
X'0C'	Form feed
X'0D'	Carriage return
X'0E'	DBCS shift-out
X'0F'	DBCS shift-in
X'15'	New-line
X'16'	Backspace
X'2F'	Alarm

The new-line character is treated as a record delimiter. The other control characters are treated as data and are part of the record.

RELATED TASKS
"Defining line-sequential files and records in COBOL" on page 207

Describing the structure of a line-sequential file

In the FILE SECTION, code a file description (FD) entry for the file. In the associated record description entry or entries, define the *record-name* and record length.

Code the logical size in bytes of the records by using the RECORD clause. Line-sequential files are stream files. Because of their character-oriented nature, the physical records are of variable length.

The following examples show how the FD entry might look for a line-sequential file:

With fixed-length records:

```
FILE SECTION.
FD  COMMUTER-FILE
    RECORD CONTAINS 80 CHARACTERS.
01  COMMUTER-RECORD.
    05  COMMUTER-NUMBER        PIC  X(16).
    05  COMMUTER-DESCRIPTION   PIC  X(64).
```

With variable-length records:

```
FILE SECTION.
FD  COMMUTER-FILE
    RECORD VARYING FROM 16 TO 80 CHARACTERS.
```

```
01  COMMUTER-RECORD.
    05  COMMUTER-NUMBER        PIC  X(16).
    05  COMMUTER-DESCRIPTION   PIC  X(64).
```

If you code the same fixed size and no `OCCURS DEPENDING ON` clause for any level-01 record description entries associated with the file, that fixed size is the logical record length. However, because blanks at the end of a record are not written to the file, the physical records might be of varying lengths.

RELATED TASKS
"Defining line-sequential files and records in COBOL" on page 207
"Coding input-output statements for line-sequential files"
"Defining and allocating line-sequential files"

RELATED REFERENCES
Data division—file description entries (*Enterprise COBOL Language Reference*)

Defining and allocating line-sequential files

You can define a line-sequential file in the HFS by using either a `DD` statement or an environment variable. Allocation of these files follows the general rules for allocating COBOL files.

To define a line-sequential file, code a `DD` allocation or an environment variable with a name that matches the external name in the `ASSIGN` clause:

- A `DD` allocation:
  - A `DD` statement that specifies `PATH=`'*absolute-path-name*'
  - A TSO allocation that specifies `PATH(`'*absolute-path-name*'`)`

  You can optionally also specify these options:
  - `PATHOPTS`
  - `PATHMODE`
  - `PATHDISP`
- An environment variable with a value of `PATH(`*absolute-path-name*`)`. No other values can be specified.

  For example, to have your program use HFS file /u/myfiles/commuterfile for a COBOL file that has an *assignment-name* of `COMMUTR`, you could use the following command:

  ```
  export COMMUTR="PATH(/u/myfiles/commuterfile)"
  ```

RELATED TASKS
"Allocating files" on page 149
"Defining line-sequential files and records in COBOL" on page 207

RELATED REFERENCES
MVS JCL Reference

Coding input-output statements for line-sequential files

Code the input and output statements shown below to process a line-sequential file.

`OPEN` To make a file available to your program.

You can open a line-sequential file as `INPUT`, `OUTPUT`, or `EXTEND`. You cannot open a line-sequential file as `I-O`.

`READ` To read a record from a file.

With sequential processing, a program reads one record after another in the same order in which the records were entered when the file was created.

`WRITE` To create a record in a file.

A program writes new records to the end of the file.

`CLOSE` To release the connection between a file and the program.

RELATED TASKS
"Defining line-sequential files and records in COBOL" on page 207
"Describing the structure of a line-sequential file" on page 208
"Opening line-sequential files"
"Reading records from line-sequential files"
"Adding records to line-sequential files" on page 211
"Closing line-sequential files" on page 211
"Handling errors in line-sequential files" on page 212

RELATED REFERENCES
OPEN statement (*Enterprise COBOL Language Reference*)
READ statement (*Enterprise COBOL Language Reference*)
WRITE statement (*Enterprise COBOL Language Reference*)
CLOSE statement (*Enterprise COBOL Language Reference*)

Opening line-sequential files

Before your program can use any `READ` or `WRITE` statements to process records in a file, it must first open the file with an `OPEN` statement. An `OPEN` statement works if the file is available or has been dynamically allocated.

Code `CLOSE WITH LOCK` so that the file cannot be opened again while the program is running.

RELATED TASKS
"Reading records from line-sequential files"
"Adding records to line-sequential files" on page 211
"Closing line-sequential files" on page 211
"Defining and allocating line-sequential files" on page 209

RELATED REFERENCES
OPEN statement (*Enterprise COBOL Language Reference*)
CLOSE statement (*Enterprise COBOL Language Reference*)

Reading records from line-sequential files

To read from a line-sequential file, open the file and use the `READ` statement. Your program reads one record after another in the same order in which the records were entered when the file was created.

Characters in the file record are read one at a time into the record area until one of the following conditions occurs:

- The record delimiter (the EBCDIC new-line character) is encountered.

The delimiter is discarded and the remainder of the record area is filled with spaces. (Record area is longer than the file record.)

- The entire record area is filled with characters.

  If the next unread character is the record delimiter, it is discarded. The next READ reads from the first character of the next record. (Record area is the same length as the file record.)

  Otherwise the next unread character is the first character to be read by the next READ. (Record area is shorter than the file record.)

+ - End-of-file is encountered.

+ The remainder of the record area is filled with spaces. (Record area is longer
+ than the file record.)

RELATED TASKS
"Opening line-sequential files" on page 210
"Adding records to line-sequential files"
"Closing line-sequential files"
"Defining and allocating line-sequential files" on page 209

RELATED REFERENCES
OPEN statement (*Enterprise COBOL Language Reference*)
WRITE statement (*Enterprise COBOL Language Reference*)

Adding records to line-sequential files

To add to a line-sequential file, open the file as EXTEND and use the WRITE statement to add records immediately after the last record in the file.

Blanks at the end of the record area are removed, and the record delimiter is added. The characters in the record area from the first character up to and including the added record delimiter are written to the file as one record.

Records written to line-sequential files must contain only USAGE DISPLAY and
| DISPLAY-1 items. Zoned decimal data items must be unsigned or declared with the SEPARATE phrase of the SIGN clause if signed.

RELATED TASKS
"Opening line-sequential files" on page 210
"Reading records from line-sequential files" on page 210
"Closing line-sequential files"
"Defining and allocating line-sequential files" on page 209

RELATED REFERENCES
OPEN statement (*Enterprise COBOL Language Reference*)
WRITE statement (*Enterprise COBOL Language Reference*)

Closing line-sequential files

Use the CLOSE statement to disconnect your program from a line-sequential file. If you try to close a file that is already closed, you will get a logic error.

If you do not close a line-sequential file, the file is automatically closed for you under the following conditions:

- When the run unit ends normally.
- When the run unit ends abnormally, if the TRAP(ON) runtime option is set.

- When Language Environment condition handling is completed and the application resumes in a routine other than where the condition occurred, open files defined in any COBOL programs in the run unit that might be called again and reentered are closed.

  You can change the location where the program resumes (after a condition is handled) by moving the resume cursor with the Language Environment CEEMRCR callable service or using HLL language constructs such as a C longjmp call.

File status codes are set when these implicit `CLOSE` operations are performed, but `EXCEPTION/ERROR` declaratives are not invoked.

RELATED TASKS
"Opening line-sequential files" on page 210
"Reading records from line-sequential files" on page 210
"Adding records to line-sequential files" on page 211
"Defining and allocating line-sequential files" on page 209

RELATED REFERENCES
CLOSE statement (*Enterprise COBOL Language Reference*)

Handling errors in line-sequential files

When an input or output statement fails, COBOL does not take corrective action for you. You choose whether your program should continue running after an input or output statement fails.

COBOL provides these language elements for intercepting and handling certain line-sequential input and output errors:
- End-of-file phrase (`AT END`)
- `EXCEPTION/ERROR` declarative
- `FILE STATUS` clause

If you do not use one of these techniques, an error in processing input or output raises a Language Environment condition.

If you use the `FILE STATUS` clause, be sure to check the key and take appropriate action based on its value. If you do not check the key, your program might continue, but the results will probably not be what you expected.

RELATED TASKS
"Coding input-output statements for line-sequential files" on page 209
"Handling errors in input and output operations" on page 235

Chapter 12. Sorting and merging files

You can arrange records in a particular sequence by using a SORT or MERGE statement. You can mix SORT and MERGE statements in the same COBOL program.

SORT statement
: Accepts input (from a file or an internal procedure) that is not in sequence, and produces output (to a file or an internal procedure) in a requested sequence. You can add, delete, or change records before or after they are sorted.

MERGE statement
: Compares records from two or more sequenced files and combines them in order. You can add, delete, or change records after they are merged.

A program can contain any number of sort and merge operations. They can be the same operation performed many times or different operations. However, one operation must finish before another begins.

With Enterprise COBOL, your IBM licensed program for sorting and merging must be DFSORT$^{(TM)}$ or an equivalent. Where DFSORT is mentioned, you can use any equivalent sort or merge product.

COBOL programs that contain SORT or MERGE statements can reside above or below the 16-MB line.

The steps you take to sort or merge are generally as follows:

1. Describe the sort or merge file to be used for sorting or merging.
2. Describe the input to be sorted or merged. If you want to process the records before you sort them, code an input procedure.
3. Describe the output from sorting or merging. If you want to process the records after you sort or merge them, code an output procedure.
4. Request the sort or merge.
5. Determine whether the sort or merge operation was successful.

Restrictions:

- You cannot run a COBOL program that contains SORT or MERGE statements under UNIX. This restriction includes BPXBATCH.
- You cannot use SORT or MERGE statements in programs compiled with the THREAD option. This includes programs that use object-oriented syntax and multithreaded applications, both of which require the THREAD option.

RELATED CONCEPTS
"Sort and merge process" on page 214

RELATED TASKS
"Describing the sort or merge file" on page 214
"Describing the input to sorting or merging" on page 215
"Describing the output from sorting or merging" on page 217
"Requesting the sort or merge" on page 220
"Determining whether the sort or merge was successful" on page 224
"Stopping a sort or merge operation prematurely" on page 225

"Improving sort performance with FASTSRT" on page 225
"Controlling sort behavior" on page 228
DFSORT Application Programming Guide

RELATED REFERENCES
"CICS SORT application restrictions" on page 231
SORT statement (*Enterprise COBOL Language Reference*)
MERGE statement (*Enterprise COBOL Language Reference*)

Sort and merge process

During the sorting of a file, all of the records in the file are ordered according to the contents of one or more fields (*keys*) in each record. You can sort the records in either ascending or descending order of each key.

If there are multiple keys, the records are first sorted according to the content of the first (or primary) key, then according to the content of the second key, and so on.

To sort a file, use the COBOL `SORT` statement.

During the merging of two or more files (which must already be sorted), the records are combined and ordered according to the contents of one or more keys in each record. You can order the records in either ascending or descending order of each key. As with sorting, the records are first ordered according to the content of the primary key, then according to the content of the second key, and so on.

Use `MERGE . . . USING` to name the files that you want to combine into one sequenced file. 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 that you name in the `GIVING` phrase.

RELATED TASKS
"Setting sort or merge criteria" on page 221

RELATED REFERENCES
SORT statement (*Enterprise COBOL Language Reference*)
MERGE statement (*Enterprise COBOL Language Reference*)

Describing the sort or merge file

Describe the sort file to be used for sorting or merging. You need `SELECT` clauses and `SD` entries even if you are sorting or merging data items only from `WORKING-STORAGE` or `LOCAL-STORAGE`.

Code as follows:

1. Write one or more `SELECT` clauses in the `FILE-CONTROL` paragraph of the `ENVIRONMENT DIVISION` to name a sort file. For example:

   ```
   ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
       SELECT Sort-Work-1 ASSIGN TO SortFile.
   ```

   Sort-Work-1 is the name of the file in your program. Use this name to refer to the file.

2. Describe the sort file in an SD entry in the FILE SECTION of the DATA DIVISION. Every SD entry must contain a record description. For example:

```
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).
```

The file described in an SD entry is the working file used for a sort or merge operation. You cannot perform any input or output operations on this file and you do not need to provide a ddname definition for it.

RELATED REFERENCES
"FILE SECTION entries" on page 14

Describing the input to sorting or merging

Describe the input file or files for sorting or merging by following the procedure below.

1. Write one or more SELECT clauses in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION to name the input files. For example:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT Input-File ASSIGN TO InFile.
```

   Input-File is the name of the file in your program. Use this name to refer to the file.
2. Describe the input file (or files when merging) in an FD entry in the FILE SECTION of the DATA DIVISION. For example:

```
DATA DIVISION.
FILE SECTION.
FD  Input-File
    LABEL RECORDS ARE STANDARD
    BLOCK CONTAINS 0 CHARACTERS
    RECORDING MODE IS F
    RECORD CONTAINS 100 CHARACTERS.
01  Input-Record   PIC X(100).
```

RELATED TASKS
"Coding the input procedure" on page 216
"Requesting the sort or merge" on page 220

RELATED REFERENCES
"FILE SECTION entries" on page 14

Example: describing sort and input files for SORT

The following example shows the ENVIRONMENT DIVISION and DATA DIVISION entries needed to describe sort work files and an input file.

```
 ID Division.
 Program-ID. SmplSort.
 Environment Division.
 Input-Output Section.
 File-Control.
*
* Assign name for a working 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
     Label Records Are Standard
     Block Contains 0 Characters
     Recording Mode is F
     Record Contains 100 Characters.
 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).
     . . .
```

RELATED TASKS
"Requesting the sort or merge" on page 220

Coding the input procedure

To process the records in an input file before they are released to the sort program, use the INPUT PROCEDURE phrase of the SORT statement.

You can use an input procedure to:
- Release data items to the sort file from WORKING-STORAGE or LOCAL-STORAGE.
- Release records that have already been read elsewhere in the program.
- Read records from an input file, select or process them, and release them to the sort file.

Each input procedure must be contained in either paragraphs or sections. For example, to release records from a table in WORKING-STORAGE or LOCAL-STORAGE to the sort file SORT-WORK-2, you could code as follows:

```
    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.
```

To transfer records to the sort program, all input procedures must contain at least one RELEASE or RELEASE FROM statement. To release A from X, for example, you can code:

```
MOVE X TO A.
RELEASE A.
```

Alternatively, you can code:

```
RELEASE A FROM X.
```

The following table compares the RELEASE and RELEASE FROM statements.

RELEASE	**RELEASE FROM**

`MOVE EXT-RECORD`
`  TO SORT-EXT-RECORD`
`PERFORM RELEASE-SORT-RECORD`
`. . .`
`RELEASE-SORT-RECORD.`
`  RELEASE SORT-RECORD`
	`PERFORM RELEASE-SORT-RECORD`
	`. . .`
	`RELEASE-SORT-RECORD.`
	`  RELEASE SORT-RECORD`
	`    FROM SORT-EXT-RECORD`

RELATED REFERENCES
"Restrictions on input and output procedures" on page 219
RELEASE statement (*Enterprise COBOL Language Reference*)

Describing the output from sorting or merging

If the output from sorting or merging is a file, describe the file by following the procedure below.

1. Write a SELECT clause in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION to name the output file. For example:

   ```
   ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
       SELECT Output-File ASSIGN TO OutFile.
   ```

   Output-File is the name of the file in your program. Use this name to refer to the file.
2. Describe the output file (or files when merging) in an FD entry in the FILE SECTION of the DATA DIVISION. For example:

   ```
   DATA DIVISION.
   FILE SECTION.
   FD  Output-File
       LABEL RECORDS ARE STANDARD
       BLOCK CONTAINS 0 CHARACTERS
       RECORDING MODE IS F
       RECORD CONTAINS 100 CHARACTERS.
   01  Output-Record   PIC X(100).
   ```

RELATED TASKS
"Coding the output procedure" on page 218
"Requesting the sort or merge" on page 220

RELATED REFERENCES
"FILE SECTION entries" on page 14

Coding the output procedure

To select, edit, or otherwise change sorted records before writing them from the sort work file into another file, use the OUTPUT PROCEDURE phrase of the SORT statement.

Each output procedure must be contained in either a section or a paragraph. An output procedure must include both of the following elements:

- At least one RETURN statement or one RETURN statement with the INTO phrase
- Any statements necessary to process the records that are made available, one at a time, by the RETURN statement

The RETURN statement makes each sorted record available to the output procedure. (The RETURN statement for a sort file is similar to a READ statement for an input file.)

You can use the AT END and END-RETURN phrases with the RETURN statement. The imperative statements in the AT END phrase are performed after all the records have been returned from the sort file. The END-RETURN explicit scope terminator delimits the scope of the RETURN statement.

If you use RETURN INTO instead of RETURN, the records will be returned to WORKING-STORAGE, LOCAL-STORAGE, or to an output area.

DFSORT coding: When you use DFSORT and a RETURN statement does not encounter an AT END condition before a COBOL program finishes running, the SORT statement could end abnormally with DFSORT message IEC025A. To avoid this situation, be sure to code the RETURN statement with the AT END phrase. In addition, ensure that the RETURN statement is executed until the AT END condition is encountered. The AT END condition occurs after the last record is returned to the program from the sort work file and a subsequent RETURN statement is executed.

"Example: coding the output procedure when using DFSORT"

RELATED REFERENCES
"Restrictions on input and output procedures" on page 219
RETURN statement (*Enterprise COBOL Language Reference*)

Example: coding the output procedure when using DFSORT

The following example shows a coding technique that ensures that the RETURN statement encounters the AT END condition before the program finishes running. The RETURN statement, coded with the AT END phrase, is executed until the AT END condition occurs.

```
IDENTIFICATION DIVISION.
DATA DIVISION.
FILE SECTION.
SD  OUR-FILE.
01  OUR-SORT-REC.
    03  SORT-KEY                PIC X(10).
    03  FILLER                  PIC X(70).
. . .
WORKING-STORAGE SECTION.
01  WS-SORT-REC                 PIC X(80).
01  END-OF-SORT-FILE-INDICATOR  PIC X VALUE 'N'.
    88  NO-MORE-SORT-RECORDS          VALUE 'Y'.
. . .
PROCEDURE DIVISION.
A-CONTROL SECTION.
```

```
    SORT OUR-FILE ON ASCENDING KEY SORT-KEY
      INPUT PROCEDURE IS B-INPUT
      OUTPUT PROCEDURE IS C-OUTPUT.
    . . .
B-INPUT SECTION.
    MOVE . . .. . .. TO WS-SORT-REC.
    RELEASE OUR-SORT-REC FROM WS-SORT-REC.
    . . .
C-OUTPUT SECTION.
    DISPLAY 'STARTING READS OF SORTED RECORDS: '.
    RETURN OUR-FILE
      AT END
        SET NO-MORE-SORT-RECORDS TO TRUE.
    PERFORM WITH TEST BEFORE UNTIL NO-MORE-SORT-RECORDS
      IF SORT-RETURN = 0 THEN
        DISPLAY 'OUR-SORT-REC = ' OUR-SORT-REC
        RETURN OUR-FILE
          AT END
            SET NO-MORE-SORT-RECORDS TO TRUE
      END-IF
    END-PERFORM.
```

Restrictions on input and output procedures

The restrictions listed below apply to each input or output procedure called by SORT and to each output procedure called by MERGE.

- The procedure must not contain any SORT or MERGE statements.
- You can use ALTER, GO TO, and PERFORM statements in the procedure to refer to procedure-names outside the input or output procedure. However, control must return to the input or 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 or output procedure (with the exception of the return of control from a declarative section).
- In an input or output procedure, you can call a program that follows standard linkage conventions. However, the called program cannot issue a SORT or MERGE statement.
- During a SORT or MERGE operation, the SD data item is used. You must not use it in the output procedure before the first RETURN executes. If you move data into this record area before the first RETURN statement, the first record to be returned will be overwritten.
- Language Environment condition handling does not allow user-written condition handlers to be established in an input or output procedure.

RELATED TASKS
"Coding the input procedure" on page 216
"Coding the output procedure" on page 218
Preparing to link-edit and run (*Language Environment Programming Guide*)

Defining sort and merge data sets

To use DFSORT under z/OS, code DD statements in the runtime JCL to describe the necessary data sets that are listed below.

Sort or merge work
Define a minimum of three data sets: SORTWK01, SORTWK02, SORTWK03, . . ., SORTWK*nn* (where *nn* is 99 or less). These data sets cannot be in the HFS.

SYSOUT Define for sort diagnostic messages, unless you change the data-set name.

(Change the name using either the MSGDDN keyword of the OPTION control statement in the SORT-CONTROL data set, or using the SORT-MESSAGE special register.)

SORTCKPT
Define if the sort or merge is to take checkpoints.

Input and output
Define input and output data sets, if any.

SORTLIB (DFSORT library)
Define the library that contains the sort modules, for example, SYS1.SORTLIB.

RELATED TASKS
"Controlling sort behavior" on page 228
"Using checkpoint/restart with DFSORT" on page 231

Sorting variable-length records

Your sort work file will be variable length only if you define it to be variable length, even if the input file to the sort contains variable-length records.

The compiler determines that the sort work file is variable length if you code one of the following elements in the SD entry:
- A RECORD IS VARYING clause
- Two or more record descriptions that define records that have different sizes, or records that contain an OCCURS DEPENDING ON clause

You cannot use RECORDING MODE V for the sort work file because the SD entry does not allow the RECORDING MODE clause.

Performance consideration: To improve sort performance of variable-length files, specify the most frequently occurring record length of the input file (the modal length) on the SMS= control card or in the SORT-MODE-SIZE special register.

RELATED TASKS
"Changing DFSORT defaults with control statements" on page 229
"Controlling sort behavior" on page 228

Requesting the sort or merge

To read records from an input file (files for MERGE) without preliminary processing, use SORT . . . USING or MERGE . . . USING and the name of the input file (files) that you declared in a SELECT clause.

To transfer sorted or merged records from the sort or merge program to another file without any further processing, use SORT . . . GIVING or MERGE . . . GIVING and the name of the output file that you declared in a SELECT clause. For example:

```
SORT Sort-Work-1
    ON ASCENDING KEY Sort-Key-1
    USING Input-File
    GIVING Output-File.
```

For SORT . . . USING or MERGE . . . USING, the compiler generates an input procedure to open the file (files), read the records, release the records to the sort or merge program, and close the file (files). The file (files) must not be open when the

SORT or MERGE statement begins execution. For SORT . . . GIVING or MERGE . . . GIVING, the compiler generates an output procedure to open the file, return the records, write the records, and close the file. The file must not be open when the SORT or MERGE statement begins execution.

The USING or GIVING files in a SORT or MERGE statement can be sequential files residing in the HFS.

"Example: describing sort and input files for SORT" on page 215

If you want an input procedure to be performed on the sort records before they are sorted, use SORT . . . INPUT PROCEDURE. If you want an output procedure to be performed on the sorted records, use SORT . . . OUTPUT PROCEDURE. For example:

```
SORT Sort-Work-1
    ON ASCENDING KEY Sort-Key-1
    INPUT PROCEDURE EditInputRecords
    OUTPUT PROCEDURE FormatData.
```

"Example: sorting with input and output procedures" on page 222

Restriction: You cannot use an input procedure with the MERGE statement. The source of input to the merge operation must be a collection of already sorted files. However, if you want an output procedure to be performed on the merged records, use MERGE . . . OUTPUT PROCEDURE. For example:

```
MERGE Merge-Work
    ON ASCENDING KEY Merge-Key
    USING Input-File-1 Input-File-2 Input-File-3
    OUTPUT PROCEDURE ProcessOutput.
```

In the FILE SECTION, you must define *Merge-Work* in an SD entry, and the input files in FD entries.

RELATED TASKS
"Defining sort and merge data sets" on page 219

RELATED REFERENCES
SORT statement (*Enterprise COBOL Language Reference*)
MERGE statement (*Enterprise COBOL Language Reference*)

Setting sort or merge criteria

To set sort or merge criteria, define the keys on which the operation is to be performed.

Do these steps:

1. In the record description of the files to be sorted or merged, define the key or keys.

   There is no maximum number of keys, but the keys must be located in the first 4092 bytes of the record description. The total length of the keys cannot exceed 4092 bytes unless the EQUALS keyword is coded in the DFSORT OPTION control statement, in which case the total length of the keys must not exceed 4088 bytes.

   Restriction: A key cannot be variably located.
2. In the SORT or MERGE statement, specify the key fields to be used for sequencing by coding the ASCENDING or DESCENDING KEY phrase, or both. When you code more than one key, some can be ascending, and some descending.

Specify the names of the keys in decreasing order of significance. The leftmost key is the primary key. The next key is the secondary key, and so on.

SORT and MERGE keys can be of class alphabetic, alphanumeric, national, or numeric (but not numeric of USAGE NATIONAL). If it has USAGE NATIONAL, a key can be of category national or can be a national-edited or numeric-edited data item. A key cannot be a national decimal data item or a national floating-point data item.

The collation order for national keys is determined by the binary order of the keys. If you specify a national data item as a key, any COLLATING SEQUENCE phrase in the SORT or MERGE statement does not apply to that key.

You can mix SORT and MERGE statements in the same COBOL program. A program can perform any number of sort or merge operations. However, one operation must end before another can begin.

RELATED REFERENCES
SORT control statement (*DFSORT Application Programming Guide*)
SORT statement (*Enterprise COBOL Language Reference*)
MERGE statement (*Enterprise COBOL Language Reference*)

Example: sorting with input and output procedures

The following example shows the use of an input and an output procedure in a SORT statement. The example also shows how you can define a primary key (SORT-GRID-LOCATION) and a secondary key (SORT-SHIFT) before using them in the SORT statement.

```
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).
. . .
WORKING-STORAGE SECTION.
01  TAB1.
    05 TAB-ENTRY OCCURS 10 TIMES
           INDEXED BY TAB-INDX.
        10  WS-SHIFT               PIC X(1).
        10  WS-GRID-LOCATION       PIC X(2).
        10  WS-REPORT              PIC X(3).
        10  WS-EXT-EMPLOYEE-NUM    PIC X(6).
        10  WS-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.
    . . .
600-SORT3-INPUT.
    PERFORM VARYING TAB-INDX FROM 1 BY 1 UNTIL TAB-INDX > 10
```

```
        RELEASE SORT-RECORD FROM TAB-ENTRY(TAB-INDX)
    END-PERFORM.
. . .
700-SORT3-OUTPUT.
    PERFORM VARYING TAB-INDX FROM 1 BY 1 UNTIL TAB-INDX > 10
        RETURN SORT-FILE INTO TAB-ENTRY(TAB-INDX)
            AT END DISPLAY 'Out Of Records In SORT File'
        END-RETURN
    END-PERFORM.
```

RELATED TASKS
"Requesting the sort or merge" on page 220

Choosing alternate collating sequences

You can sort or merge records on the EBCDIC or ASCII collating sequence, or on another collating sequence. The default collating sequence is EBCDIC unless you code the `PROGRAM COLLATING SEQUENCE` clause in the `OBJECT-COMPUTER` paragraph.

To override the default sequence, use the `COLLATING SEQUENCE` phrase of the `SORT` or `MERGE` statement. You can use different collating sequences for each `SORT` or `MERGE` statement in your program.

The `PROGRAM COLLATING SEQUENCE` clause and the `COLLATING SEQUENCE` phrase apply only to keys of class alphabetic or alphanumeric.

When you sort or merge an ASCII file, you have to request the ASCII collating sequence. To do so, code the `COLLATING SEQUENCE` phrase of the `SORT` or `MERGE` statement, and define the *alphabet-name* as `STANDARD-1` in the `SPECIAL-NAMES` paragraph.

RELATED TASKS
"Specifying the collating sequence" on page 8
"Setting sort or merge criteria" on page 221

RELATED REFERENCES
OBJECT-COMPUTER paragraph (*Enterprise COBOL Language Reference*)
SORT statement (*Enterprise COBOL Language Reference*)
Classes and categories of data (*Enterprise COBOL Language Reference*)

Sorting on windowed date fields

You can specify windowed date fields as sort keys if your version of DFSORT supports the `Y2PAST` option. If so, DFSORT can sort or merge on the windowed date sequence.

To sort on a windowed date field, use the `DATE FORMAT` clause to define a windowed date field; then use the field as the sort key. DFSORT will use the same century window as that used by the compilation unit. Specify the century window with the `YEARWINDOW` compiler option.

DFSORT supports year-last windowed date fields, although the compiler itself does not provide automatic windowing for year-last windowed date fields in statements other than `MERGE` or `SORT`.

RELATED CONCEPTS
"Millennium language extensions (MLE)" on page 598

RELATED TASKS
"Sorting and merging by date" on page 612

RELATED REFERENCES
"YEARWINDOW" on page 348
DATE FORMAT clause (*Enterprise COBOL Language Reference*)
OPTION control statement: Y2PAST (*DFSORT Application Programming Guide*)

Preserving the original sequence of records with equal keys

You can preserve the order of identical collating records from input to output.

Use one of these techniques:

- Install DFSORT with the EQUALS option as the default.
- Provide, at run time, an OPTION card that has the EQUALS keyword in the IGZSRTCD data set.
- Use the WITH DUPLICATES IN ORDER phrase in the SORT statement. Doing so adds the EQUALS keyword to the OPTION card in the IGZSRTCD data set.

  Do not use both the NOEQUALS keyword on the OPTION card *and* the DUPLICATES phrase, or the run unit will end.

RELATED REFERENCES
OPTION control statement (*DFSORT Application Programming Guide*)

Determining whether the sort or merge was successful

The DFSORT program returns a completion code of either 0 (successful completion) or 16 (unsuccessful completion) after each sort or merge has finished. The completion code is stored in the SORT-RETURN special register.

You should test for successful completion after each SORT or MERGE statement. For example:

```
    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.
    . . .
```

If you do not reference SORT-RETURN anywhere in your program, the COBOL run time tests the completion code. If it is 16, COBOL issues a runtime diagnostic message.

By default, DFSORT diagnostic messages are sent to the SYSOUT data set. If you want to change this default, use the MSGDDN parameter of the DFSORT OPTION control card or use the SORT-MESSAGE special register.

If you test SORT-RETURN for one or more (but not necessarily all) SORT or MERGE statements, the COBOL run time does not check the completion code.

RELATED TASKS
"Checking for sort errors with NOFASTSRT" on page 227
"Controlling sort behavior" on page 228

RELATED REFERENCES
DFSORT messages and return codes (*DFSORT Application Programming Guide*)

Stopping a sort or merge operation prematurely

To stop a sort or merge operation, move the integer 16 into the SORT-RETURN special register.

Move 16 into the register in either of the following ways:

- Use MOVE in an input or output procedure.

  Sort or merge processing will be stopped immediately after the next RELEASE or RETURN statement is performed.
- Reset the register in a declarative section entered during processing of a USING or GIVING file.

  Sort or merge processing will be stopped immediately after the next implicit RELEASE or RETURN is performed, which will occur after a record has been read from or written to the USING or GIVING file.

Control then returns to the statement following the SORT or MERGE statement.

Improving sort performance with FASTSRT

Using the FASTSRT compiler option improves the performance of most sort operations. With FASTSRT, the DFSORT product (instead of Enterprise COBOL) performs the I/O on the input and output files you name in the SORT . . . USING and SORT . . . GIVING statements.

The compiler issues informational messages to point out statements in which FASTSRT can improve performance.

Usage notes

- You cannot use the DFSORT options SORTIN or SORTOUT if you use FASTSRT. The FASTSRT compiler option does not apply to line-sequential files you use as USING or GIVING files.
- If you specify file status and use FASTSRT, file status is ignored during the sort.

RELATED REFERENCES
"FASTSRT" on page 314
"FASTSRT requirements for JCL"
"FASTSRT requirements for sort input and output files" on page 226

FASTSRT requirements for JCL

In the runtime JCL, you must assign the sort work files (SORTWK*nn*) to a direct-access device, not to tape data sets.

For the input and output files, the DCB parameter of the DD statement must match the FD description.

FASTSRT requirements for sort input and output files

If you specify FASTSRT but your code does not meet FASTSRT requirements, the compiler issues a message and the COBOL run time performs the I/O instead. Your program will not experience the performance improvements that are otherwise possible.

To use FASTSRT, you must describe and process the input files to the sort and the output files from the sort in these ways:

- You can name only one input file in the USING phrase. You can name only one output file in the GIVING phrase.
- You cannot use an input procedure on an input file nor an output procedure on an output file.

  Instead of using input or output procedures, you might be able to use these DFSORT control statements:

  - INREC
  - OUTFILE
  - OUTREC
  - INCLUDE
  - OMIT
  - STOPAFT
  - SKIPREC
  - SUM

  Many DFSORT functions perform the same operations that are common in input or output procedures. Code the appropriate DFSORT control statements instead, and place them either in the IGZSRTCD or SORTCNTL data set.
- Do not code the LINAGE clause for the output FD entry.
- Do not code any INPUT declarative (for input files), OUTPUT declarative (for output files), or file-specific declaratives (for either input or output files) to apply to any FDs used in the sort.
- Do not use a variable relative file as the input or output file.
- Do not use a line-sequential file as the input or output file.
- For either an input or an output file, the record descriptions of the SD and FD entry must define the same format (fixed or variable), and the largest records of the SD and FD entry must define the same record length.

If you code a RELATIVE KEY clause for an output file, it will not be set by the sort.

Performance tip: If you block your input and output records, the sort performance could be significantly improved.

QSAM requirements

- QSAM files must have a record format of fixed, variable, or spanned.
- A QSAM input file can be empty.
- To use the same QSAM file for both input and output, you must describe the file using two different DD statements. For example, in the FILE-CONTROL SECTION you might code this:

  ```
  SELECT FILE-IN ASSIGN INPUTF.
  SELECT FILE-OUT ASSIGN OUTPUTF.
  ```

  In the DATA DIVISION, you would have an FD entry for both FILE-IN and FILE-OUT, where FILE-IN and FILE-OUT are identical except for their names.

In the PROCEDURE DIVISION, your SORT statement could look like this:

```
SORT file-name
  ASCENDING KEY data-name-1
  USING FILE-IN GIVING FILE-OUT
```

Then in your JCL, assuming that data set INOUT has been cataloged, you would code:

```
//INPUTF  DD DSN=INOUT,DISP=SHR
//OUTPUTF DD DSN=INOUT,DISP=SHR
```

On the other hand, if you code the same file-name in the USING and GIVING phrases, or assign the input and output files the same ddname, then the file can be accepted for FASTSRT either for input or output, but not both. If no other conditions disqualify the file from being eligible for FASTSRT on input, then the file will be accepted for FASTSRT on input, but not on output. If the file was found to be ineligible for FASTSRT on input, it might be eligible for FASTSRT on output.

A QSAM file that qualifies for FASTSRT can be accessed by the COBOL program while the SORT statement is being performed. For example, if the file is used for FASTSRT on input, you can access it in an output procedure; if it is used for FASTSRT on output, you can access it in an input procedure.

VSAM requirements

- A VSAM input file must not be empty.
- VSAM files cannot be password-protected.
- You cannot name the same VSAM file in both the USING and GIVING phrases.
- A VSAM file that qualifies for FASTSRT cannot be accessed by the COBOL program until the SORT statement processing is completed. For example, if the file qualifies for FASTSRT on input, you cannot access it in an output procedure and vice versa. (If you do so, OPEN fails.)

RELATED TASKS
DFSORT Application Programming Guide

Checking for sort errors with NOFASTSRT

When you compile with the NOFASTSRT option, the sort process does not check for errors in open, close, or input or output operations for files that you reference in the USING or GIVING phrase of the SORT statement. Therefore, you might need to check whether SORT completed successfully.

The code required depends on whether you code a FILE STATUS clause or an ERROR declarative for the files referenced in the USING and GIVING phrases, as shown in the table below.

Table 35. **Methods for checking for sort errors with NOFASTSRT**

FILE STATUS clause?	ERROR declarative?	Then do:
No	No	No special coding. Any failure during the sort process causes the program to end abnormally.
Yes	No	Test the SORT-RETURN special register after the SORT statement, and test the file status key. (Not recommended if you want complete file-status checking, because the file status code is set but COBOL cannot check it.)

Table 35. **Methods for checking for sort errors with NOFASTSRT** *(continued)*

FILE STATUS clause?	ERROR declarative?	Then do:
Maybe	Yes	In the ERROR declarative, set the SORT-RETURN special register to 16 to stop the sort process and indicate that it was not successful. Test the SORT-RETURN special register after the SORT statement.

RELATED TASKS
"Determining whether the sort or merge was successful" on page 224
"Using file status keys" on page 239
"Coding ERROR declaratives" on page 238
"Stopping a sort or merge operation prematurely" on page 225

Controlling sort behavior

You can control several aspects of sort behavior by inserting values in special registers before the sort or by using compiler options. You might also have a choice of control statements and keywords.

You can verify sort behavior by examining the contents of special registers after the sort.

The table below lists those aspects of sort behavior that you can affect by using special registers or compiler options, and the equivalent sort control statement keywords if any are available.

Table 36. **Methods for controlling sort behavior**

To set or test	Use this special register or compiler option	Or this control statement (and keyword if applicable)
Amount of main storage to be reserved	SORT-CORE-SIZE special register	OPTION (keyword RESINV)
Amount of main storage to be used	SORT-CORE-SIZE special register	OPTION (keywords MAINSIZE or MAINSIZE=MAX)
Modal length of records in a file with variable-length records	SORT-MODE-SIZE special register	SMS=*nnnnn*
Name of sort control statement data set (default IGZSRTCD)	SORT-CONTROL special register	None
Name of sort message file (default SYSOUT)	SORT-MESSAGE special register	OPTION (keyword MSGDDN)
Number of sort records	SORT-FILE-SIZE special register	OPTION (keyword FILSZ)
Sort completion code	SORT-RETURN special register	None
Century window for sorting or merging on date fields	YEARWINDOW compiler option	OPTION (keyword Y2PAST)
Format of windowed date fields used as sort or merge keys	(Derived from PICTURE, USAGE, and DATE FORMAT clauses)	SORT (keyword FORMAT=Y2*x*)

Sort special registers: SORT-CONTROL is an eight-character COBOL special register that contains the ddname of the sort control statement file. If you do not want to use the default ddname IGZSRTCD, assign to `SORT-CONTROL` the ddname of the data set that contains your sort control statements.

The `SORT-CORE-SIZE`, `SORT-FILE-SIZE`, `SORT-MESSAGE`, and `SORT-MODE-SIZE` special registers are used in the `SORT` interface if you assign them nondefault values. At run time, however, any parameters in control statements in the sort control statement data set override corresponding settings in the special registers, and a message to that effect is issued.

You can use the `SORT-RETURN` special register to determine whether the sort or merge was successful and to stop a sort or merge operation prematurely.

A compiler warning message (W-level) is issued for each sort special register that you set in a program.

RELATED TASKS
"Determining whether the sort or merge was successful" on page 224
"Stopping a sort or merge operation prematurely" on page 225
"Changing DFSORT defaults with control statements"
"Allocating space for sort files" on page 230
Using DFSORT program control statements (*DFSORT Application Programming Guide*)

RELATED REFERENCES
"Default characteristics of the IGZSRTCD data set"

Changing DFSORT defaults with control statements

If you want to change DFSORT system defaults to improve sort performance, pass information to DFSORT through control statements in the runtime data set IGZSRTCD.

The control statements that you can include in IGZSRTCD (in the order listed) are:

1. `SMS=`*nnnnn*, where *nnnnn* is the length in bytes of the most frequently occurring record size. (Use only if the `SD` file is variable length.)
2. `OPTION` (except keywords `SORTIN` or `SORTOUT`).
3. Other DFSORT control statements (except `SORT`, `MERGE`, `RECORD`, or `END`).

Code control statements between columns 2 and 71. You can continue a control statement record by ending the line with a comma and starting the next line with a new keyword. You cannot use labels or comments on a record, and a record itself cannot be a DFSORT comment statement.

RELATED TASKS
"Controlling sort behavior" on page 228
Using DFSORT program control statements (*DFSORT Application Programming Guide*)

RELATED REFERENCES
"Default characteristics of the IGZSRTCD data set"

Default characteristics of the IGZSRTCD data set

The IGZSRTCD data set is optional. Its defaults are `LRECL=80`, `BLKSIZE=400`, and ddname IGZSRTCD.

You can use a different ddname by coding it in the `SORT-CONTROL` special register. If you defined a ddname for the `SORT-CONTROL` data set and you receive the message IGZ0027W, an OPEN failure occurred that you should investigate.

RELATED TASKS
"Controlling sort behavior" on page 228

Allocating storage for sort or merge operations

Certain parameters set during the installation of DFSORT determine the amount of storage that DFSORT uses. In general, the more storage DFSORT has available, the faster the sort or merge operations in your program will be.

DFSORT installation should not allocate all the free space in the region for its COBOL operation, however. When your program is running, storage must be available for:

- COBOL programs that are dynamically called from an input or output procedure
- Language Environment runtime library modules
- Data management modules that can be loaded into the region for use by an input or output procedure
- Any storage obtained by these modules

For a specific sort or merge operation, you can override the DFSORT storage values set at installation. To do so, code the `MAINSIZE` and `RESINV` keywords on the `OPTION` control statement in the sort control statement data set, or use the `SORT-CORE-SIZE` special register.

Be careful not to override the storage allocation to the extent that all the free space in the region is used for sort operations for your COBOL program.

RELATED TASKS
"Controlling sort behavior" on page 228
DFSORT Installation and Customization

RELATED REFERENCES
OPTION control statement (*DFSORT Application Programming Guide*)

Allocating space for sort files

If you use `NOFASTSRT` or an input procedure, DFSORT does not know the size of the file that you are sorting. This can lead to an out-of-space condition when you sort large files or to overallocation of resources when you sort small files.

If this occurs, you can use the `SORT-FILE-SIZE` special register to help DFSORT determine the amount of resource (for example, workspace or *hiperspace*) needed for the sort. Set `SORT-FILE-SIZE` to a reasonable estimate of the number of input records. This value is passed to DFSORT as its `FILSZ=E`*n* value.

RELATED TASKS
"Controlling sort behavior" on page 228
"Coding the input procedure" on page 216
DFSORT Application Programming Guide

Using checkpoint/restart with DFSORT

You cannot use checkpoints taken while DFSORT is running under z/OS to restart, unless the checkpoints are taken by DFSORT. Checkpoints taken by a COBOL program while `SORT` or `MERGE` statements execute are invalid; such restarts are detected and canceled.

To take a checkpoint during a sort or merge operation, do these steps:

1. Add a `DD` statement for `SORTCKPT` in the JCL.
2. Code the `RERUN` clause in the `I-O-CONTROL` paragraph:

   `RERUN ON` *`assignment-name`*
3. Code the `CKPT` (or `CHKPT`) keyword on an `OPTION` control statement in the sort control statement data set (default ddname IGZSRTCD).

RELATED CONCEPTS
Chapter 32, "Interrupts and checkpoint/restart," on page 587

RELATED TASKS
"Changing DFSORT defaults with control statements" on page 229
"Setting checkpoints" on page 587

Sorting under CICS

There is no IBM sort product that is supported under CICS. However, you can use the `SORT` statement with a sort program you write that runs under CICS to sort small amounts of data.

You must have both an input and an output procedure for the `SORT` statement. In the input procedure, use the `RELEASE` statement to transfer records from the COBOL program to the sort program before the sort is performed. In the output procedure, use the `RETURN` statement to transfer records from the sort program to the COBOL program after the sort is performed.

RELATED TASKS
"Coding the input procedure" on page 216
"Coding the output procedure" on page 218
"Coding COBOL programs to run under CICS" on page 393

RELATED REFERENCES
"CICS SORT application restrictions"
"CICS reserved-word table" on page 401

CICS SORT application restrictions

Several restrictions apply to COBOL applications that run under CICS and use the `SORT` statement.

The restrictions are:

- `SORT` statements that include the `USING` or `GIVING` phrase are not supported.
- Sort control data sets are not supported. Data in the `SORT-CONTROL` special register is ignored.
- These CICS commands in the input or output procedures can cause unpredictable results:
  - `CICS LINK`

- CICS XCTL
- CICS RETURN
- CICS HANDLE
- CICS IGNORE
- CICS PUSH
- CICS POP

You can use CICS commands other than these if you use the NOHANDLE or RESP option. Unpredictable results can occur if you do not use NOHANDLE or RESP.

RELATED REFERENCES
"CICS reserved-word table" on page 401

Chapter 13. Handling errors

Put code in your programs that anticipates possible system or runtime problems. If you do not include such code, output data or files could be corrupted, and the user might not even be aware that there is a problem.

The error-handling code can take actions such as handling the situation, issuing a message, or halting the program. You might for example create error-detection routines for data-entry errors or for errors as your installation defines them. In any event, coding a warning message is a good idea.

Enterprise COBOL contains special elements to help you anticipate and correct error conditions:

- User-requested dumps
- `ON OVERFLOW` in `STRING` and `UNSTRING` operations
- `ON SIZE ERROR` in arithmetic operations
- Elements for handling input or output errors
- `ON EXCEPTION` or `ON OVERFLOW` in `CALL` statements
- User-written routines for handling errors

RELATED TASKS
"Handling errors in joining and splitting strings" on page 234
"Handling errors in arithmetic operations" on page 234
"Handling errors in input and output operations" on page 235
"Handling errors when calling programs" on page 244
"Writing routines for handling errors" on page 244

Requesting dumps

You can cause a formatted dump of the Language Environment runtime environment and the member language libraries at any prespecified point in your program by coding a call to the Language Environment callable service CEE3DMP.

```
77  Title-1         Pic x(80)   Display.
77  Options         Pic x(255)  Display.
01  Feedback-code   Pic x(12)   Display.
. . .
    Call "CEE3DMP" Using Title-1, Options, Feedback-code
```

To have symbolic variables included in the formatted dump, compile with the `SYM` suboption of the `TEST` compiler option and use the `VARIABLES` subparameter of CEE3DMP. You can also request, through runtime options, that a dump be produced for error conditions of your choosing.

You can cause a system dump at any prespecified point in your program. Request an abend without cleanup by calling the Language Environment service CEE3ABD with a cleanup value of zero. This callable service stops the run unit immediately, and a system dump is requested when the abend is issued.

RELATED REFERENCES
Language Environment Debugging Guide
CEE3DMP—generate dump (*Language Environment Programming Reference*)

Handling errors in joining and splitting strings

During the joining or splitting of strings, the pointer used by STRING or UNSTRING might fall outside the range of the receiving field. A potential overflow condition exists, but COBOL does not let the overflow happen.

Instead, the STRING or UNSTRING operation is not completed, the receiving field remains unchanged, and control passes to the next sequential statement. If you do not code the ON OVERFLOW phrase of the STRING or UNSTRING statement, 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
```

These are the 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)	*bbbbbbbb*[1]	*bbbbbbbb*[1]
String-ptr	9(2)	0	0
1. The symbol *b* represents a blank space.			

Because String-ptr has a value (0) that falls short of the receiving field, an overflow condition occurs and the STRING operation is not completed. (Overflow would also occur if String-ptr were greater than 9.) If ON OVERFLOW had not been specified, you would not be notified that the contents of Item-4 remained unchanged.

Handling errors in arithmetic operations

The results of arithmetic operations might be larger than the fixed-point field that is to hold them, or you might have tried dividing by zero. 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) runtime option.

The imperative statement of the ON SIZE ERROR clause will be performed and the result field will not change in these cases:

- Fixed-point overflow
- Division by zero
- Zero raised to the zero power
- Zero raised to a negative number
- Negative number raised to a fractional power

Floating-point exponent overflow occurs when the value of a floating-point computation cannot be represented in the zSeries floating-point operand format. This type of overflow does not cause `SIZE ERROR`; an abend occurs instead. You could code a user-written condition handler to intercept the abend and provide your own error recovery logic.

"Example: checking for division by zero"

Example: checking for division by zero

The following example shows how you can code an `ON SIZE ERROR` imperative statement so that the program issues an informative message if division by zero occurs.

```
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.
```

If division by zero occurs, the program writes a message and halts program execution.

Handling errors in input and output operations

When an input or output operation fails, COBOL does not automatically take corrective action. You choose whether your program will continue running after a less-than-severe input or output error.

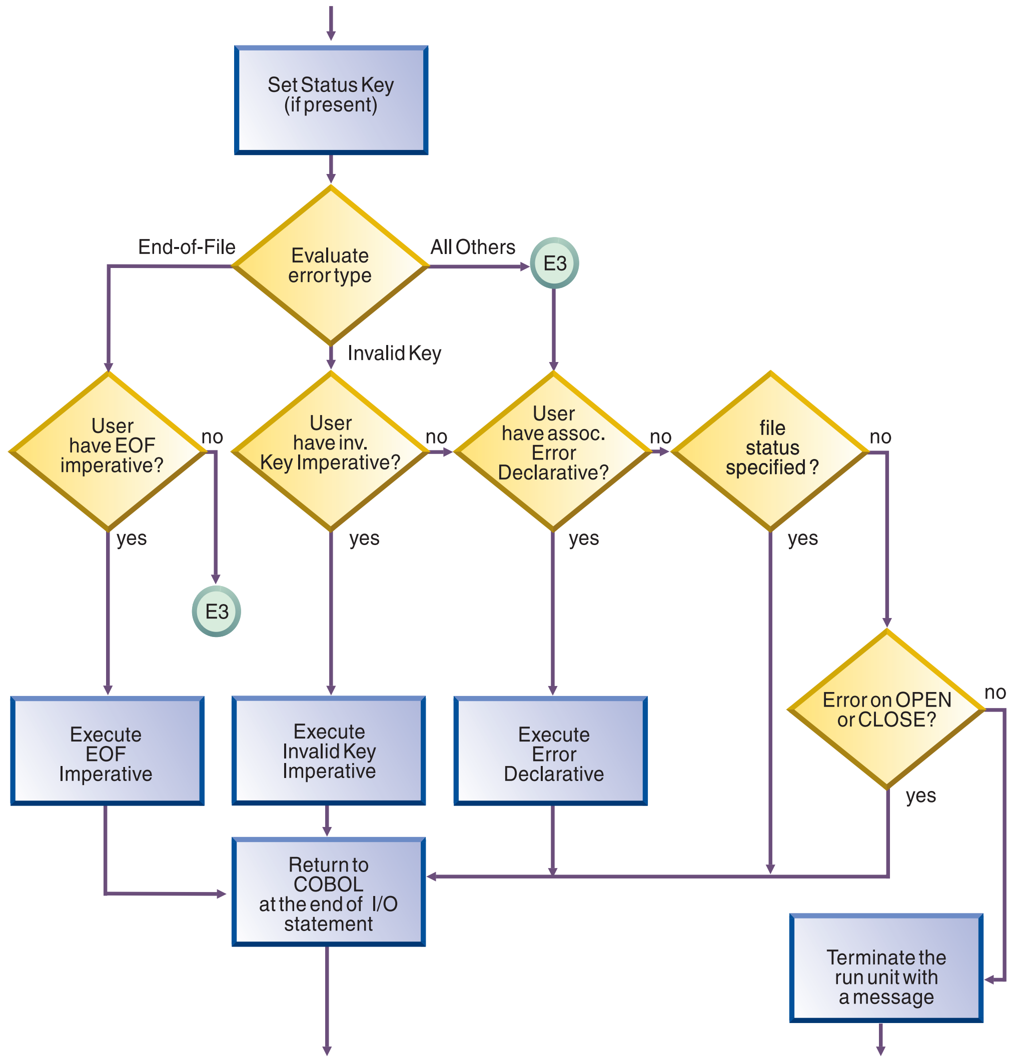
You can use any of the following techniques for intercepting and handling certain input or output conditions or errors:

- End-of-file condition (`AT END`)
- `ERROR` declaratives
- `FILE STATUS` clause and file status key
- File system status code
- Imperative-statement phrases on `READ` or `WRITE` statements

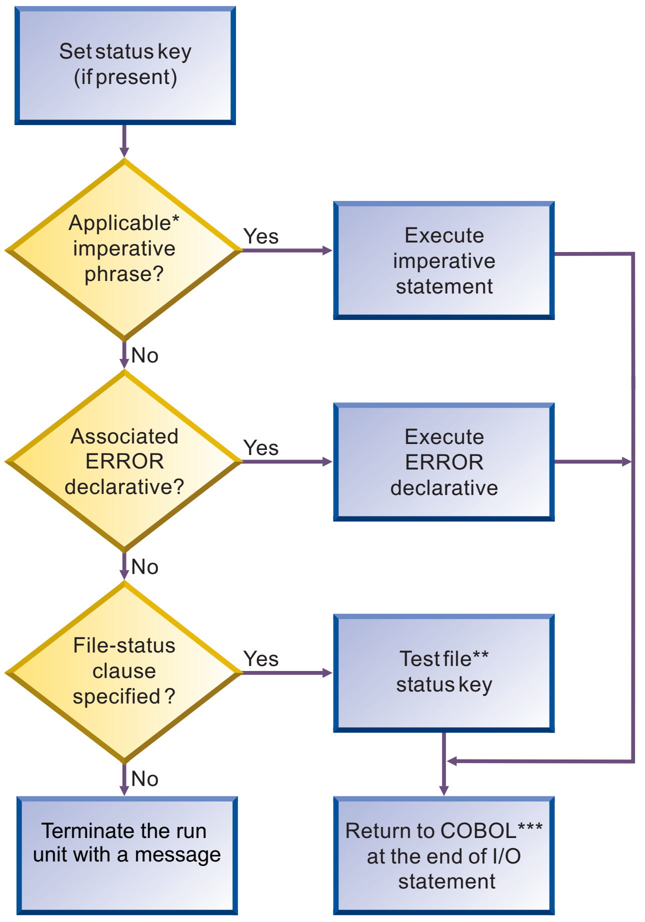
  For VSAM files, if you specify a `FILE STATUS` clause, you can also test the VSAM status code to direct your program to error-handling logic.
- `INVALID KEY` phrase

To have your program continue, you must code the appropriate error-recovery procedure. You might code, for example, a procedure to check the value of the file status key. If you do not handle an input or output error in any of these ways, a severity-3 Language Environment condition is signaled, which causes the run unit to end if the condition is not handled.

The following figure shows the flow of logic after a VSAM input or output error:


Set Status Key (if present)
Evaluate error type
End-of-File
All Others
E3
Invalid Key
User have EOF imperative?
no
yes
E3
User have inv. Key Imperative?
no
yes
User have assoc. Error Declarative?
no
yes
file status specified ?
no
yes
Error on OPEN or CLOSE?
no
yes
Execute EOF Imperative
Execute Invalid Key Imperative
Execute Error Declarative
Return to COBOL at the end of I/O statement
Terminate the run unit with a message

The following figure shows the flow of logic after an input or output error with QSAM or line-sequential files. The error can be from a READ statement, a WRITE statement, or a CLOSE statement with a REEL/UNIT clause (QSAM only).


Set status key (if present)
Applicable* imperative phrase?
Yes
Execute imperative statement
No
Associated ERROR declarative?
Yes
Execute ERROR declarative
No
File-status clause specified?
Yes
Test file** status key
No
Terminate the run unit with a message
Return to COBOL*** at the end of I/O statement

*Possible phrases for QSAM are `AT END`, `AT END-OF-PAGE`, and `INVALID KEY`; for line sequential, `AT END`.

**You need to write the code to test the file status key.

***Execution of your COBOL program continues after the input or output statement that caused the error.

RELATED TASKS
"Using the end-of-file condition (AT END)"
"Coding ERROR declaratives" on page 238
"Using file status keys" on page 239
"Handling errors in QSAM files" on page 165
"Using VSAM status codes (VSAM files only)" on page 240
"Handling errors in line-sequential files" on page 212
"Coding INVALID KEY phrases" on page 243

RELATED REFERENCES
File status key (*Enterprise COBOL Language Reference*)

Using the end-of-file condition (AT END)

You code the `AT END` phrase of the `READ` statement to handle errors or normal conditions, according to your program design. At end-of-file, the `AT END` phrase is performed. If you do not code an `AT END` phrase, the associated `ERROR` declarative is performed.

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 that contains 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
```

Any NOT AT END phrase is performed only if the READ statement completes successfully. If the READ operation fails because of a 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 any associated declarative procedure is performed.

You might choose not to code either an AT END phrase or an EXCEPTION declarative procedure, but to code a status key clause for the file. In that case, control passes to the next sequential instruction after the input or output statement that detected the end-of-file condition. At that place, you should have some code that takes appropriate action.

RELATED REFERENCES
AT END phrases (*Enterprise COBOL Language Reference*)

Coding ERROR declaratives

You can code one or more ERROR declarative procedures that will be given control if an input or output error occurs during the execution of your program. If you do not code such procedures, your job could be canceled or abnormally terminated after an input or output error occurs.

Place each such procedure in the declaratives section of the PROCEDURE DIVISION. You can code:

- A single, common procedure for the entire program
- Procedures for each file open mode (whether INPUT, OUTPUT, I-O, or EXTEND)
- Individual procedures for each file

In an ERROR declarative procedure, you can code corrective action, retry the operation, continue, or end execution. (If you continue processing a blocked file, though, you might lose the remaining records in a block after the record that caused the error.) You can use the ERROR declaratives procedure in combination with the file status key if you want a further analysis of the error.

Multithreading: Avoid deadlocks when coding I/O declaratives in multithreaded applications. When an I/O operation results in a transfer of control to an I/O declarative, the automatic serialization lock associated with the file is held during the execution of the statements within the declarative. If you code I/O operations within your declaratives, your logic might result in a deadlock as illustrated by the following sample:

```
Declaratives.
D1 section.
Use after standard error procedure on F1
    Read F2.
    . . .
```

```
D2 section.
Use after standard error procedure on F2
    Read F1.
    . . .
End declaratives.
    . . .
    Rewrite R1.
    Rewrite R2.
```

When this program is running on two threads, the following sequence of events could occur:

1. Thread 1: Rewrite R1 acquires lock on F1 and encounters I/O error.
2. Thread 1: Enter declarative D1, holding lock on F1.
3. Thread 2: Rewrite R2 acquires lock on F2 and encounters I/O error.
4. Thread 2: Enter declarative D2.
5. Thread 1: Read F2 from declarative D1; wait on F2 lock held by thread 2.
6. Thread 2: Read F1 from declarative D2; wait on F1 lock held by thread 1.
7. Deadlock.

RELATED REFERENCES
EXCEPTION/ERROR declarative (*Enterprise COBOL Language Reference*)

Using file status keys

After each input or output statement is performed on a file, the system updates values in the two digit positions of the file status key. In general, a zero in the first position indicates a successful operation, and a zero in both positions means that nothing abnormal occurred.

Establish a file status key by coding:

- The FILE STATUS clause in the FILE-CONTROL paragraph:

  ```
  FILE STATUS IS data-name-1
  ```
- Data definitions in the DATA DIVISION (WORKING-STORAGE, LOCAL-STORAGE, or LINKAGE SECTION), for example:

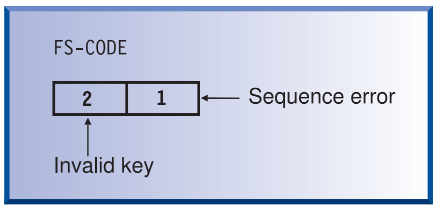
  ```
  WORKING-STORAGE SECTION.
  01  data-name-1  PIC 9(2)  USAGE NATIONAL.
  ```

Specify the file status key *data-name-1* as a two-character category alphanumeric or category national item, or as a two-digit zoned decimal or national decimal item. This *data-name-1* cannot be variably located.

Your program can check the file status key to discover whether an error occurred, and, if so, what type of error occurred. For example, suppose that 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.

  Doing so means that you can determine the cause of a file input or output exception, such as an application logic error or a disk error.
- Check the file status key after each input or output request.

  If the file status key contains a value other than 0, your program can issue an error message or can take action based on that value.

  You do not have to reset the file status key code, because it is set after each input or output attempt.

For VSAM files, you can additionally code a second identifier in the `FILE STATUS` clause to get more detailed information about VSAM input or output requests.

You can use the file status key alone or in conjunction with the `INVALID KEY` option, or to supplement the `EXCEPTION` or `ERROR` declarative. Using the file status key in this way gives you precise information about the results of each input or output operation.

"Example: file status key"

RELATED TASKS
"Using VSAM status codes (VSAM files only)"

RELATED REFERENCES
FILE STATUS clause (*Enterprise COBOL Language Reference*)
File status key (*Enterprise COBOL Language Reference*)

Example: file status key

The following example shows how you can perform a simple check of the file 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 "Nonzero file status returned from OPEN " MASTER-CHECK-KEY
    . . .
```

Using VSAM status codes (VSAM files only)

Often the COBOL file status code is too general to pinpoint the disposition of a request. You can get more detailed information about VSAM input or output requests by coding a second data item in the `FILE STATUS` clause.

`FILE STATUS IS` *`data-name-1 data-name-8`*

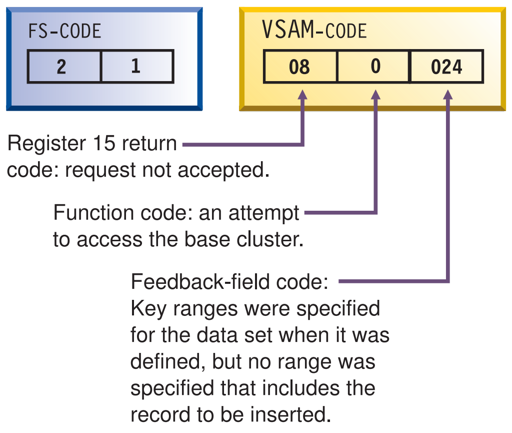
The data item *data-name-1* shown above specifies the COBOL file status key, which you define as a two-character alphanumeric or national data item, or as a two-digit zoned decimal or national decimal item.

The data item *data-name-8* specifies the VSAM status code, which you define as a 6-byte alphanumeric group data item that has three subordinate 2-byte binary fields. The VSAM status code contains meaningful values when the COBOL file status key is not 0.

You can define *data-name-8* in the `WORKING-STORAGE SECTION`, as in `VSAM-CODE` below.

```
01 RETURN-STATUS.
    05 FS-CODE              PIC X(2).
    05 VSAM-CODE.
       10 VSAM-R15-RETURN   PIC S9(4) Usage Comp-5.
       10 VSAM-FUNCTION     PIC S9(4) Usage Comp-5.
       10 VSAM-FEEDBACK     PIC S9(4) Usage Comp-5.
```

Enterprise COBOL uses *data-name-8* to pass information supplied by VSAM. In the following example, `FS-CODE` corresponds to *data-name-1* and `VSAM-CODE` corresponds to *data-name-8*:


FS-CODE
2
1
VSAM-CODE
08
0
024
Register 15 return code: request not accepted.
Function code: an attempt to access the base cluster.
Feedback-field code: Key ranges were specified for the data set when it was defined, but no range was specified that includes the record to be inserted.

"Example: checking VSAM status codes"

RELATED REFERENCES
FILE STATUS clause (*Enterprise COBOL Language Reference*)
File status key (*Enterprise COBOL Language Reference*)
VSAM macro return and reason codes (*z/OS DFSMS Macro Instructions for Data Sets*)

Example: checking VSAM status codes

The following example reads an indexed file (starting at the fifth record), checks the file status key after each input or output request, and displays the VSAM status codes when the file status key is not zero.

This example also illustrates how output from this program might look if the file being processed contained six records.

```
IDENTIFICATION DIVISION
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT VSAMFILE ASSIGN TO VSAMFILE
    ORGANIZATION IS INDEXED
    ACCESS DYNAMIC
    RECORD KEY IS VSAMFILE-KEY
    FILE STATUS IS FS-CODE VSAM-CODE.
DATA DIVISION.
FILE SECTION.
FD  VSAMFILE
    RECORD  30.
01  VSAMFILE-REC.
    10 VSAMFILE-KEY            PIC X(6).
    10 FILLER                  PIC X(24).
WORKING-STORAGE SECTION.
01  RETURN-STATUS.
    05 FS-CODE                 PIC XX.
    05 VSAM-CODE.
       10 VSAM-RETURN-CODE     PIC S9(2) Usage Binary.
       10 VSAM-COMPONENT-CODE  PIC S9(1) Usage Binary.
       10 VSAM-REASON-CODE     PIC S9(3) Usage Binary.
PROCEDURE DIVISION.
    OPEN  INPUT VSAMFILE.
    DISPLAY "OPEN INPUT VSAMFILE FS-CODE: " FS-CODE.

    IF FS-CODE NOT = "00"
       PERFORM VSAM-CODE-DISPLAY
       STOP RUN
    END-IF.

    MOVE "000005" TO VSAMFILE-KEY.
    START VSAMFILE KEY IS EQUAL TO VSAMFILE-KEY.
    DISPLAY "START VSAMFILE KEY="  VSAMFILE-KEY
            " FS-CODE: "  FS-CODE.
    IF FS-CODE NOT = "00"
       PERFORM VSAM-CODE-DISPLAY
    END-IF.

    IF FS-CODE = "00"
       PERFORM READ-NEXT UNTIL FS-CODE NOT = "00"
    END-IF.

    CLOSE VSAMFILE.
    STOP RUN.

READ-NEXT.
    READ VSAMFILE NEXT.
    DISPLAY "READ NEXT VSAMFILE FS-CODE: " FS-CODE.
    IF FS-CODE NOT = "00"
       PERFORM VSAM-CODE-DISPLAY
    END-IF.
    DISPLAY VSAMFILE-REC.

VSAM-CODE-DISPLAY.
    DISPLAY "VSAM-CODE ==>"
            " RETURN: "  VSAM-RETURN-CODE,
            " COMPONENT: "  VSAM-COMPONENT-CODE,
            " REASON: "  VSAM-REASON-CODE.
```

Below is a sample of the output from the example program that checks VSAM status-code information:

```
OPEN INPUT VSAMFILE FS-CODE: 00
START VSAMFILE KEY=000005 FS-CODE: 00
READ NEXT VSAMFILE FS-CODE: 00
000005 THIS IS RECORD NUMBER 5
READ NEXT VSAMFILE FS-CODE: 00
000006 THIS IS RECORD NUMBER 6
READ NEXT VSAMFILE FS-CODE: 10
VSAM-CODE ==> RETURN: 08 COMPONENT: 2 REASON: 004
```

Coding INVALID KEY phrases

You can include an INVALID KEY phrase on READ, START, WRITE, REWRITE, and DELETE statements for VSAM indexed and relative files. The INVALID KEY phrase is given control if an input or output error occurs because of a faulty index key.

You can also include the INVALID KEY phrase in WRITE requests for QSAM files, but the phrase has limited meaning for QSAM files. It is used only if you try to write to a disk that is full.

Use the FILE STATUS clause with the INVALID KEY phrase to evaluate the status key and determine the specific INVALID KEY condition.

INVALID KEY phrases differ from ERROR declaratives in several ways. INVALID KEY phrases:

- Operate for only limited types of errors. ERROR declaratives encompass all forms.
- Are coded directly in the input or output verb. ERROR declaratives are coded separately.
- Are specific for a single input or output operation. 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. Any ERROR declaratives that you coded are not performed.

If you code a NOT INVALID KEY phrase, it is performed only if the statement completes successfully. If the operation fails because of a condition other than INVALID KEY, neither the INVALID KEY nor the NOT INVALID KEY phrase is performed. Instead, after the program performs any associated ERROR declaratives, control passes to the end of the statement.

"Example: FILE STATUS and INVALID KEY"

Example: FILE STATUS and INVALID KEY

The following example shows how you can use the file status code and the INVALID KEY phrase to determine more specifically why an input or output statement failed.

Assume that you have a file that contains master customer records and you need to update some of these records with information from a transaction update file. The program reads each transaction record, finds the corresponding record in the master file, and makes the necessary updates. The records in both files 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 that define indexed organization, random access, MASTER-CUSTOMER-NUMBER as the prime record key, and CUSTOMER-FILE-STATUS as the file status key.

```
.
.  (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
```

Handling errors when calling programs

When a program dynamically calls a separately compiled program, the called program might be unavailable. For example, the system might be out of storage or unable to locate the load module. If the CALL statement does not have an ON EXCEPTION or ON OVERFLOW phrase, your application might abend.

Use the ON EXCEPTION phrase to perform a series of statements and to perform your own error handling. For example, in the code fragment below, if program REPORTA is unavailable, control passes to the ON EXCEPTION phrase.

```
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
```

The ON EXCEPTION phrase applies only to the availability of the called program. If an error occurs while the called program is running, the ON EXCEPTION phrase is not performed.

RELATED TASKS
Enterprise COBOL Compiler and Runtime Migration Guide

Writing routines for handling errors

You can handle most error conditions that might occur while your program is running by using the ON EXCEPTION phrase, ON SIZE ERROR phrase, or other language constructs. But if an extraordinary condition such as a machine check occurs, usually your application is abnormally terminated.

Enterprise COBOL and Language Environment provide a way for a user-written program to gain control when such conditions occur. Using Language Environment condition handling, you can write your own error-handling routines in COBOL. They can report, analyze, or even fix up a program and enable it to resume running.

To have Language Environment pass control to a user-written error program, you must first identify and register its entry point to Language Environment. PROCEDURE-POINTER data items enable you to pass the entry address of procedure entry points to Language Environment services.

RELATED TASKS
"Using procedure and function pointers" on page 447

Part 2. Compiling and debugging your program

Chapter 14. Compiling under z/OS

You can compile Enterprise COBOL programs under z/OS using job control language (JCL), TSO commands, CLISTs, or ISPF panels.

For compiling with JCL, IBM provides a set of cataloged procedures, which can reduce the amount of JCL coding that you need to write. If the cataloged procedures do not meet your needs, you can write your own JCL. Using JCL, you can compile a single program or compile several programs as part of a batch job.

When compiling under TSO, you can use TSO commands, CLISTs, or ISPF panels.

You can also compile in a z/OS UNIX shell by using the `cob2` command.

You might instead want to start the Enterprise COBOL compiler from an assembler program, for example, if your shop has developed a tool or interface that calls the Enterprise COBOL compiler.

As part of the compilation step, you need to define the data sets needed for the compilation and specify any compiler options necessary for your program and the desired output.

The compiler translates your COBOL program into language that the computer can process (object code). The compiler also lists errors in your source statements and provides supplementary information to help you debug and tune your program. Use compiler-directing statements and compiler options to control your compilation.

After compiling your program, you need to review the results of the compilation and correct any compiler-detected errors.

RELATED TASKS
"Compiling with JCL"
"Compiling under TSO" on page 259
Chapter 15, "Compiling under UNIX," on page 279
"Starting the compiler from an assembler program" on page 260
"Defining compiler input and output" on page 262
"Specifying compiler options under z/OS" on page 267
"Compiling multiple programs (batch compilation)" on page 270
"Correcting errors in your source program" on page 274

RELATED REFERENCES
Chapter 18, "Compiler-directing statements," on page 351
"Data sets used by the compiler under z/OS" on page 262
"Compiler options and compiler output under z/OS" on page 269

Compiling with JCL

Include the following information in the JCL for compilation: job description, statement to invoke the compiler, and definitions of the needed data sets (including the directory paths of HFS files, if any).

The simplest way to compile your program under z/OS is to code JCL that uses a cataloged procedure. A *cataloged procedure* is a set of job control statements in a partitioned data set called the *procedure library* (SYS1.PROCLIB).

The following JCL shows the general format for a cataloged procedure.

```
//jobname  JOB  parameters
//stepname EXEC [PROC=]procname[,{PARM=|PARM.stepname=}'options']
//SYSIN    DD   data-set parameters
. . .           (source program to be compiled)
/*
//
```

Additional considerations apply when you use cataloged procedures to compile object-oriented programs.

“Example: sample JCL for a procedural DLL application” on page 468

RELATED TASKS
“Using a cataloged procedure”
“Writing JCL to compile programs” on page 257
“Specifying compiler options under z/OS” on page 267
“Specifying compiler options in a batch compilation” on page 272
“Compiling programs to create DLLs” on page 466

RELATED REFERENCES
“Data sets used by the compiler under z/OS” on page 262

Using a cataloged procedure

Specify a cataloged procedure in an EXEC statement in your JCL.

For example, the following JCL calls the IBM-supplied cataloged procedure IGYWC for compiling an Enterprise COBOL program and defining the required data sets:

```
//JOB1      JOB1
//STEPA     EXEC PROC=IGYWC
//COBOL.SYSIN DD *
000100 IDENTIFICATION DIVISION
      * (the source code)
. . .
/*
```

You can omit /* after the source code. If your source code is stored in a data set, replace SYSIN DD * with appropriate parameters that describe the data set.

You can use these procedures with any of the job schedulers that are part of z/OS. When a scheduler encounters parameters that it does not require, the scheduler either ignores them or substitutes alternative parameters.

If the compiler options are not explicitly supplied with the procedure, default options established at the installation apply. You can override these default options by using an EXEC statement that includes the desired options.

You can specify data sets to be in the hierarchical file system by overriding the corresponding DD statement. However, the compiler utility files (SYSUT*x*) and copy libraries (SYSLIB) you specify must be MVS data sets.

Additional details about invoking cataloged procedures, overriding and adding to EXEC statements, and overriding and adding to DD statements are in the Language Environment information.

RELATED TASKS
Language Environment Programming Guide

RELATED REFERENCES
"Compile procedure (IGYWC)"
"Compile and link-edit procedure (IGYWCL)" on page 250
"Compile, link-edit, and run procedure (IGYWCLG)" on page 251
"Compile, load, and run procedure (IGYWCG)" on page 252
"Compile, prelink, and link-edit procedure (IGYWCPL)" on page 253
"Compile, prelink, link-edit, and run procedure (IGYWCPLG)" on page 254
"Prelink and link-edit procedure (IGYWPL)" on page 255
"Compile, prelink, load, and run procedure (IGYWCPG)" on page 256
MVS Program Management: User's Guide and Reference

Compile procedure (IGYWC)

IGYWC is a single-step cataloged procedure for compiling a program. It produces an object module. The compile steps in all other cataloged procedures that invoke the compiler are similar.

You must supply the following DD statement, indicating the location of the source program, in the input stream:

```
//COBOL.SYSIN DD  *       (or appropriate parameters)
```

If you use copybooks in the program that you are compiling, you must also supply a DD statement for SYSLIB or other libraries that you specify in COPY statements. For example:

```
//COBOL.SYSLIB  DD  DISP=SHR,DSN=DEPT88.BOBS.COBLIB
```

```
//IGYWC  PROC  LNGPRFX='IGY.V3R4M0',SYSLBLK=3200
//*
//*  COMPILE A COBOL PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   LNGPRFX   IGY.V3R4M0       PREFIX FOR LANGUAGE DATA SET NAMES
//*
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,          (1)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))         (2)
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
```

(1) STEPLIB can be installation-dependent.

(2) SYSUT5 is needed only if the LIB option is used.

"Example: JCL for compiling using HFS"

Example: JCL for compiling using HFS: The following job uses procedure IGYWC to compile a COBOL program demo.cbl that is located in the hierarchical file system (HFS). It writes the generated compiler listing demo.lst, object file demo.o, and SYSADATA file demo.adt to the HFS.

```
//HFSDEMO JOB ,
// TIME=(1),MSGLEVEL=(1,1),MSGCLASS=H,CLASS=A,REGION=50M,
// NOTIFY=&SYSUID,USER=&SYSUID
//COMPILE EXEC IGYWC,
// PARM.COBOL='LIST,MAP,RENT,FLAG(I,I),XREF,ADATA'
//SYSPRINT DD PATH='/u/userid/cobol/demo.lst',    (1)
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),              (2)
// PATHMODE=SIRWXU,                               (3)
// FILEDATA=TEXT                                  (4)
//SYSLIN DD PATH='/u/userid/cobol/demo.o',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
//SYSADATA DD PATH='/u/userid/cobol/demo.adt',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
//SYSIN DD PATH='/u/userid/cobol/demo.cbl',
// PATHOPTS=ORDONLY,
// FILEDATA=TEXT,
// RECFM=F
```

(1) PATH specifies the path name for an HFS file.

(2) PATHOPTS indicates the access for the file (such as read or read-write) and sets the status for the file (such as append, create, or truncate).

(3) PATHMODE indicates the permissions, or file access attributes, to be set when a file is created.

(4) FILEDATA specifies whether the data is to be treated as text or binary.

You can use a mixture of HFS (PATH='*hfs-directory-path*') and MVS data sets (DSN=*traditional-data-set-name*) on the compilation DD statements shown in this example as overrides. However, the compiler utility files (DD statements SYSUT*x*) and COPY libraries (DD statements SYSLIB) must be MVS data sets.

RELATED REFERENCES
UNIX System Services Command Reference
MVS JCL Reference
"Data sets used by the compiler under z/OS" on page 262

Compile and link-edit procedure (IGYWCL)

IGYWCL is a two-step cataloged procedure to compile and link-edit a program.

The COBOL job step produces an object module that is input to the linkage editor or binder. You can add other object modules. You must supply the following DD statement, indicating the location of the source program, in the input stream:

```
//COBOL.SYSIN DD  *       (or appropriate parameters)
```

If the program uses copybooks, you must also supply a DD statement for SYSLIB or other libraries that you specify in COPY statements. For example:

```
//COBOL.SYSLIB  DD  DISP=SHR,DSN=DEPT88.BOBS.COBLIB
```

```
//IGYWCL PROC  LNGPRFX='IGY.V3R4M0',SYSLBLK=3200,
//             LIBPRFX='CEE',
//             PGMLIB='&&GOSET',GOPGM=GO
//*
```

```
//*  COMPILE AND LINK EDIT A COBOL PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IGY.V3R4M0       PREFIX FOR LANGUAGE DATA SET NAMES
//*   SYSLBLK   3200             BLOCK SIZE FOR OBJECT DATA SET
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   PGMLIB    &&GOSET          DATA SET NAME FOR LOAD MODULE
//*   GOPGM     GO               MEMBER NAME FOR LOAD MODULE
//*
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,              (1)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))            (2)
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//LKED   EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,              (3)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=&PGMLIB(&GOPGM),
//             SPACE=(TRK,(10,10,1)),
//             UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
```

(1) STEPLIB can be installation-dependent.

(2) SYSUT5 is needed only if the LIB option is used.

(3) SYSLIB can be installation-dependent.

Compile, link-edit, and run procedure (IGYWCLG)

IGYWCLG is a three-step cataloged procedure to compile, link-edit, and run a program.

The COBOL job step produces an object module that is input to the linkage editor or binder. You can add other object modules. If the COBOL program refers to any data sets, you must also supply DD statements that define these data sets. You must supply the following DD statement, indicating the location of the source program, in the input stream:

```
//COBOL.SYSIN DD  *       (or appropriate parameters)
```

If the program uses copybooks, you must also supply a DD statement for SYSLIB or other libraries that you specify in COPY statements. For example:

```
//COBOL.SYSLIB  DD  DISP=SHR,DSN=DEPT88.BOBS.COBLIB
```

```
//IGYWCLG PROC LNGPRFX='IGY.V3R4M0',SYSLBLK=3200,
//             LIBPRFX='CEE',GOPGM=GO
//*
//*  COMPILE, LINK EDIT AND RUN A COBOL PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IGY.V3R4M0       PREFIX FOR LANGUAGE DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   GOPGM     GO               MEMBER NAME FOR LOAD MODULE
//*
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,                     (1)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))                   (2)
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//LKED   EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,                     (3)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=&&GOSET(&GOPGM),SPACE=(TRK,(10,10,1)),
//             UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
//GO     EXEC PGM=*.LKED.SYSLMOD,COND=((8,LT,COBOL),(4,LT,LKED)),
//             REGION=2048K
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,                      (1)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

(1) STEPLIB can be installation-dependent.

(2) SYSUT5 is needed only if the LIB option is used.

(3) SYSLIB can be installation-dependent.

Compile, load, and run procedure (IGYWCG)

IGYWCG is a two-step cataloged procedure to compile, load, and run a program.

The COBOL job step produces an object module that is input to the loader. If the COBOL program refers to any data sets, you must supply the DD statements that define these data sets. You must supply the following DD statement, indicating the location of the source program, in the input stream:

```
//COBOL.SYSIN DD  *       (or appropriate parameters)
```

If the program uses copybooks, you must also supply a DD statement for SYSLIB or other libraries that you specify in COPY statements. For example:

```
//COBOL.SYSLIB  DD  DISP=SHR,DSN=DEPT88.BOBS.COBLIB
```

```
//IGYWCG PROC  LNGPRFX='IGY.V3R4M0',SYSLBLK=3200,
//             LIBPRFX='CEE'
//*
//*  COMPILE, LOAD AND RUN A COBOL PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IGY.V3R4M0       PREFIX FOR LANGUAGE DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*
```

```
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,                 (1)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,               (2)
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))               (3)
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//GO     EXEC PGM=LOADER,COND=(8,LT,COBOL),REGION=2048K
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,                 (4)
//             DISP=SHR
//SYSLOUT  DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,                  (1)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

(1) STEPLIB can be installation-dependent.

(2) SYSLIN can reside in the HFS.

(3) SYSUT5 is needed only if the LIB option is used.

(4) SYSLIB can be installation-dependent.

Compile, prelink, and link-edit procedure (IGYWCPL)

IGYWCPL is a three-step cataloged procedure for compiling, prelinking, and link-editing a program.

You must supply the following DD statement, indicating the location of the source program, in the input stream:

```
SYSIN DD  *       (or appropriate parameters)
```

If the program uses copybooks, you must also supply a DD statement for SYSLIB or other libraries that you specify in COPY statements. For example:

```
//COBOL.SYSLIB  DD  DISP=SHR,DSN=DEPT88.BOBS.COBLIB
```

```
//IGYWCPL PROC LNGPRFX='IGY.V3R4M0',SYSLBLK=3200,
//             LIBPRFX='CEE',PLANG=EDCPMSGE,
//             PGMLIB='&&GOSET',GOPGM=GO
//*
//*  COMPILE, PRELINK AND LINK EDIT A COBOL PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IGY.V3R4M0       PREFIX FOR LANGUAGE DATA SET NAMES
//*   SYSLBLK   3200             BLOCK SIZE FOR OBJECT DATA SET
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   PLANG     EDCPMSGE         PRELINKER MESSAGES MODULE
//*   PGMLIB    &&GOSET          DATA SET NAME FOR LOAD MODULE
//*   GOPGM     GO               MEMBER NAME FOR LOAD MODULE
//*
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,               (1)
//             DISP=SHR
```

```
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))            (2)
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//PLKED   EXEC PGM=EDCPRLK,PARM='',COND=(8,LT,COBOL),
//             REGION=2048K
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,
//             DISP=SHR
//SYSMSGS  DD  DSNAME=&LIBPRFX..SCEEMSGP(&PLANG),
//             DISP=SHR
//SYSLIB   DD  DUMMY
//SYSIN    DD  DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSMOD   DD  DSNAME=&&PLKSET,UNIT=SYSDA,DISP=(NEW,PASS),
//             SPACE=(32000,(100,50)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSDEFSD DD  DUMMY
//SYSOUT   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//*
//LKED   EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,              (3)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&PLKSET,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=&PGMLIB(&GOPGM),
//             SPACE=(TRK,(10,10,1)),
//             UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
```

(1) STEPLIB can be installation-dependent.

(2) SYSUT5 is needed only if the LIB option is used.

(3) SYSLIB can be installation-dependent.

Compile, prelink, link-edit, and run procedure (IGYWCPLG)

IGYWCPLG is a four-step cataloged procedure for compiling, prelinking, link-editing, and running a program.

You must supply the following DD statement, indicating the location of the source program, in the input stream:

```
SYSIN DD  *      (or appropriate parameters)
```

If the program uses copybooks, you must also supply a DD statement for SYSLIB or other libraries that you specify in COPY statements. For example:

```
//COBOL.SYSLIB  DD  DISP=SHR,DSN=DEPT88.BOBS.COBLIB
```

```
//IGYWCPLG PROC LNGPRFX='IGY.V3R4M0',SYSLBLK=3200,
//             PLANG=EDCPMSGE,
//             LIBPRFX='CEE',GOPGM=GO
//*
//*  COMPILE, PRELINK, LINK EDIT, AND RUN A COBOL PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IGY.V3R4M0       PREFIX FOR LANGUAGE DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   PLANG     EDCPMSGE         PRELINKER MESSAGES MODULE
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   GOPGM     GO               MEMBER NAME FOR LOAD MODULE
//*
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,                    (1)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))                  (2)
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//PLKED   EXEC PGM=EDCPRLK,PARM='',COND=(8,LT,COBOL),
//             REGION=2048K
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,
//             DISP=SHR
//SYSMSGS  DD  DSNAME=&LIBPRFX..SCEEMSGP(&PLANG),
//             DISP=SHR
//SYSLIB   DD  DUMMY
//SYSIN    DD  DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSMOD   DD  DSNAME=&&PLKSET,UNIT=SYSDA,DISP=(NEW,PASS),
//             SPACE=(32000,(100,50)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSDEFSD DD  DUMMY
//SYSOUT   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//*
//LKED   EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,                    (3)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&PLKSET,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=&&GOSET(&GOPGM),SPACE=(TRK,(10,10,1)),
//             UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
//GO     EXEC PGM=*.LKED.SYSLMOD,COND=((8,LT,COBOL),(4,LT,LKED)),
//             REGION=2048K
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

(1) STEPLIB can be installation-dependent.

(2) SYSUT5 is needed only if the `LIB` option is used.

(3) SYSLIB can be installation-dependent.

Prelink and link-edit procedure (IGYWPL)

The IGYWPL cataloged procedure is a two-step procedure for prelinking and link-editing a program.

```
//IGYWPL PROC  PLANG=EDCPMSGE,SYSLBLK=3200,
//             LIBPRFX='CEE',
//             PGMLIB='&&GOSET',GOPGM=GO
//*
//*  PRELINK AND LINK EDIT A COBOL PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   PLANG     EDCPMSGE         PRELINK MESSAGES MEMBER NAME
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   PGMLIB    &&GOSET          DATA SET NAME FOR LOAD MODULE
//*   GOPGM     GO               MEMBER NAME FOR LOAD MODULE
//*
//*  CALLER MUST SUPPLY //PLKED.SYSIN DD . . .
//*
//PLKED   EXEC PGM=EDCPRLK,PARM='',
//             REGION=2048K
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,               (1)
//             DISP=SHR
//SYSMSGS  DD  DSNAME=&LIBPRFX..SCEEMSGP(&PLANG),
//             DISP=SHR
//SYSLIB   DD  DUMMY
//SYSMOD   DD  DSNAME=&&PLKSET,UNIT=SYSDA,DISP=(NEW,PASS),
//             SPACE=(32000,(100,50)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=&SYSLBLK)
//SYSDEFSD DD  DUMMY
//SYSOUT   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//*
//LKED   EXEC PGM=HEWL,COND=(4,LT,PLKED),REGION=1024K
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,              (2)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=&PGMLIB(&GOPGM),SPACE=(TRK,(10,10,1)),
//             UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSIN    DD  DUMMY
```

(1) STEPLIB can be installation-dependent.

(2) SYSLIB can be installation-dependent.

Compile, prelink, load, and run procedure (IGYWCPG)

IGYWCPG is a four-step cataloged procedure for compiling, prelinking, loading, and running a program.

You must supply the following DD statement, indicating the location of the source program, in the input stream:

```
//COBOL.SYSIN DD  *       (or appropriate parameters)
```

If the program uses copybooks, you must also supply a DD statement for SYSLIB or other libraries that you specify in COPY statements. For example:

```
//COBOL.SYSLIB  DD  DISP=SHR,DSN=DEPT88.BOBS.COBLIB
```

```
//IGYWCPG PROC LNGPRFX='IGY.V3R4M0',SYSLBLK=3200,
//             PLANG=EDCPMSGE,
//             LIBPRFX='CEE'
//*
//*  COMPILE, PRELINK, LOAD, AND RUN A COBOL PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IGY.V3R4M0       PREFIX FOR LANGUAGE DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   PLANG     EDCPMSGE         PRELINKER MESSAGES MODULE
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//COBOL   EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,                 (1)
//             DISP=SHR
```

```
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))                (2)
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//PLKED   EXEC PGM=EDCPRLK,PARM='',COND=(8,LT,COBOL),
//             REGION=2048K
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,
//             DISP=SHR
//SYSMSGS  DD  DSNAME=&LIBPRFX..SCEEMSGP(&PLANG),
//             DISP=SHR
//SYSLIB   DD  DUMMY
//SYSIN    DD  DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSMOD   DD  DSNAME=&&PLKSET,UNIT=SYSDA,DISP=(NEW,PASS), (3)
//             SPACE=(32000,(100,50)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSDEFSD DD  DUMMY
//SYSOUT   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//*
//GO      EXEC PGM=LOADER,COND=(8,LT,COBOL),REGION=2048K
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,                  (4)
//             DISP=SHR
//SYSLOUT  DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&PLKSET,DISP=(OLD,DELETE)
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

(1) STEPLIB can be installation-dependent.

(2) SYSUT5 is needed only if the LIB option is used.

(3) SYSMOD can reside in the HFS.

(4) SYSLIB can be installation-dependent.

Writing JCL to compile programs

If the cataloged procedures do not give you the flexibility you need for more complex programs, write your own job control statements. The following example shows the general format of JCL used to compile a program.

```
//jobname  JOB  acctno,name,MSGCLASS=1                (1)
//stepname EXEC PGM=IGYCRCTL,PARM=(options)           (2)
//STEPLIB  DD   DSNAME=IGY.V3R4M0.SIGYCOMP,DISP=SHR   (3)
//SYSUT1   DD   UNIT=SYSDA,SPACE=(subparms)           (4)
//SYSUT2   DD   UNIT=SYSDA,SPACE=(subparms)
//SYSUT3   DD   UNIT=SYSDA,SPACE=(subparms)
//SYSUT4   DD   UNIT=SYSDA,SPACE=(subparms)
//SYSUT5   DD   UNIT=SYSDA,SPACE=(subparms)
//SYSUT6   DD   UNIT=SYSDA,SPACE=(subparms)
//SYSUT7   DD   UNIT=SYSDA,SPACE=(subparms)
//SYSPRINT DD   SYSOUT=A                              (5)
//SYSLIN   DD   DSNAME=MYPROG,UNIT=SYSDA,             (6)
//              DISP=(MOD,PASS),SPACE=(subparms)
//SYSIN    DD   DSNAME=dsname,UNIT=device,            (7)
                VOLUME=(subparms),DISP=SHR
```

(1) The JOB statement indicates the beginning of a job.

(2) The EXEC statement specifies that the Enterprise COBOL compiler (IGYCRCTL) is to be invoked.

(3) This DD statement defines the data set where the Enterprise COBOL compiler resides.

(4) The SYSUT DD statements define the utility data sets that the compiler will use to process the source program. All SYSUT files must be on direct-access storage devices.

(5) The SYSPRINT DD statement defines the data set that receives output from options such as LIST and MAP. SYSOUT=A is the standard designation for data sets whose destination is the system output device.

(6) The SYSLIN DD statement defines the data set that receives output from the OBJECT option (the object module).

(7) The SYSIN DD statement defines the data set to be used as input to the job step (source code).

You can use a mixture of HFS (PATH='*hfs-directory-path*') and MVS data sets (DSN=*traditional-data-set-name*) in the compilation DD statements for the following data sets:

- Sources files
- Object files
- Listings
- ADATA files
- Debug files
- Executable modules

However, the compiler utility files (DD statements SYSUT*x*) and COPY libraries (DD statement SYSLIB) must be MVS data sets.

"Example: user-written JCL for compiling"
"Example: sample JCL for a procedural DLL application" on page 468

RELATED REFERENCES
MVS JCL Reference

Example: user-written JCL for compiling

The following example shows a few possibilities for adapting the basic JCL.

```
//JOB1     JOB                                 (1)
//STEP1    EXEC PGM=IGYCRCTL,PARM='OBJECT'     (2)
//STEPLIB  DD   DSNAME=IGY.V3R4M0.SIGYCOMP,DISP=SHR
//SYSUT1   DD   UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD   UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD   UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD   UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD   UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD   UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD   UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSPRINT DD   SYSOUT=A
//SYSLIN   DD   DSNAME=MYPROG,UNIT=SYSDA,
//              DISP=(MOD,PASS),SPACE=(TRK,(3,3))
//SYSIN    DD   *                              (3)
000100 IDENTIFICATION DIVISION.
. . .
/*                                             (4)
```

(1) JOB1 is the name of the job.

(2) STEP1 is the name of the sole job step in the job. The EXEC statement also specifies that the generated object code should be placed on disk or tape (to be used as input to the link step).

(3) The asterisk indicates that the input data set follows in the input stream.

(4) The delimiter statement /* separates data from subsequent control statements in the input stream.

Compiling under TSO

Under TSO, you can use TSO commands, command lists (CLISTs), REXX execs, or ISPF to compile programs using traditional MVS data sets. You can use TSO commands or REXX execs to compile programs using HFS files.

With each method, you need to allocate the data sets and request the compilation:

1. Use the ALLOCATE command to allocate data sets.

   For any compilation, allocate the work data sets (SYSUT*n*) and the SYSIN and SYSPRINT data sets. If you specify certain compiler options, you must allocate other data sets:

   - OBJECT: Allocate the SYSLIN data set to produce an object module.
   - TERMINAL: Allocate the SYSTERM data set to receive compiler messages at your terminal.
   - LIB: Allocate SYSUT5. If you used COPY or REPLACE statements in your Enterprise COBOL program, also allocate the SYSLIB data set. (These must be traditional MVS data sets, not HFS paths.)

   You can allocate data sets in any order. However, you must allocate all needed data sets before you start to compile.
2. Use the CALL command at the READY prompt to request compilation:

   ```
   CALL 'IGY.V3R4M0.SIGYCOMP(IGYCRCTL)'
   ```

You can specify the ALLOCATE and CALL commands on the TSO command line, or, if you are not using HFS files, you can include them in a CLIST.

You can allocate HFS files for all the compiler data sets except the SYSUT*x* utility data sets and the SYSLIB libraries. ALLOCATE statements have the following form:

```
Allocate File(SYSIN) Path('/u/myu/myap/std/prog2.cbl')
Pathopts(ORDONLY) Filedata(TEXT)
```

"Example: ALLOCATE and CALL for compiling under TSO"
"Example: CLIST for compiling under TSO" on page 260

Example: ALLOCATE and CALL for compiling under TSO

The following example shows how to specify ALLOCATE and CALL commands when you are compiling under TSO.

```
[READY]
ALLOCATE FILE(SYSUT1) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSUT2) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSUT3) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSUT4) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSUT5) CYLINDERS SPACE(1 1)
[READY]
```

```
ALLOCATE FILE(SYSUT6) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSUT7) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSPRINT) SYSOUT
[READY]
ALLOCATE FILE(SYSTERM) DATASET(*)
[READY]
ALLOCATE FILE(SYSLIN) DATASET(PROG2.OBJ) NEW TRACKS SPACE(3,3)
[READY]
ALLOCATE FILE(SYSIN) DATASET(PROG2.COBOL) SHR
[READY]
CALL 'IGY.V3R4M0.SIGYCOMP(IGYCRCTL)' 'LIST,NOCOMPILE(S),OBJECT,FLAG(E,E),TERMINAL'
  .
   (COBOL listings and messages)
  .
[READY]
FREE FILE(SYSUT1,SYSUT2,SYSUT3,SYSUT4,SYSUT5,SYSUT6,SYSUT7,SYSPRINT,SYSTERM,+
SYSIN,SYSLIN)
[READY]
```

Example: CLIST for compiling under TSO

The following example shows a CLIST for compiling under TSO. The FREE commands are not required. However, good programming practice dictates that you free files before you allocate them.

```
PROC 1 MEM
CONTROL LIST
FREE (SYSUT1)
FREE (SYSUT2)
FREE (SYSUT3)
FREE (SYSUT4)
FREE (SYSUT5)
FREE (SYSUT6)
FREE (SYSUT7)
FREE (SYSPRINT)
FREE (SYSIN)
FREE (SYSLIN)
ALLOC F(SYSPRINT) SYSOUT
ALLOC F(SYSIN) DA(COBOL.SOURCE(&MEM))  SHR REUSE
ALLOC F(SYSLIN) DA(COBOL.OBJECT(&MEM)) OLD REUSE
ALLOC F(SYSUT1) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
ALLOC F(SYSUT2) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
ALLOC F(SYSUT3) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
ALLOC F(SYSUT4) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
ALLOC F(SYSUT5) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
ALLOC F(SYSUT6) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
ALLOC F(SYSUT7) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
CALL 'IGY.V3R4M0.SIGYCOMP(IGYCRCTL)'
```

Starting the compiler from an assembler program

You can start the Enterprise COBOL compiler from within an assembler program by using the ATTACH or the LINK macro by dynamic invocation. You must identify the compiler options and the ddnames of the data sets to be used during processing.

For example:

symbol {LINK|ATTACH} EP=IGYCRCTL,PARAM=(*optionlist*[,*ddnamelist*]),VL=1

EP
Specifies the symbolic name of the compiler. The control program (from the library directory entry) determines the entry point at which the program should begin running.

PARAM Specifies, as a sublist, address parameters to be passed from the assembler program to the compiler.

The first fullword in the address parameter list contains the address of the COBOL *optionlist*. The second fullword contains the address of the *ddnamelist*. The third and fourth fullwords contain the addresses of null parameters, or zero.

optionlist
Specifies the address of a variable-length list that contains the COBOL options specified for compilation. This address must be written even if no list is provided.

The *optionlist* must begin on a halfword boundary. The 2 high-order bytes contain a count of the number of bytes in the remainder of the list. If no options are specified, the count must be zero. The *optionlist* is freeform, with each field separated from the next by a comma. No blanks or zeros should appear. The compiler recognizes only the first 100 characters.

ddnamelist
Specifies the address of a variable-length list that contains alternative ddnames for the data sets used during compiler processing. If standard ddnames are used, the *ddnamelist* can be omitted.

The *ddnamelist* must begin on a halfword boundary. The 2 high-order bytes contain a count of the number of bytes in the remainder of the list. Each name of less than 8 bytes must be left justified and padded with blanks. If an alternate ddname is omitted from the list, the standard name is assumed. If the name is omitted, the 8-byte entry must contain binary zeros. You can omit names from the end by shortening the list.

All SYSUT*n* data sets specified must be on direct-access storage devices and have physical sequential organization. They must not reside in the HFS.

The following table shows the sequence of the 8-byte entries in the *ddnamelist*.

ddname 8-byte entry	Name for which substituted
1	SYSLIN
2	Not applicable
3	Not applicable
4	SYSLIB
5	SYSIN
6	SYSPRINT
7	SYSPUNCH
8	SYSUT1
9	SYSUT2
10	SYSUT3
11	SYSUT4
12	SYSTERM
13	SYSUT5
14	SYSUT6
15	SYSUT7
16	SYSADATA
17	SYSJAVA
18	SYSDEBUG
19	SYSMDECK

VL Specifies that the sign bit is to be set to 1 in the last fullword of the address parameter list.

When the compiler completes processing, it puts a return code in register 15.

RELATED TASKS
"Defining compiler input and output"

RELATED REFERENCES
"Data sets used by the compiler under z/OS"
"Compiler options and compiler output under z/OS" on page 269

Defining compiler input and output

You need to define several kinds of data sets that the compiler uses to do its work. The compiler takes input data sets and libraries and produces various types of output, including object code, listings, and messages. The compiler also uses utility data sets during compilation.

RELATED TASKS
"Defining the source code data set (SYSIN)" on page 264
"Specifying source libraries (SYSLIB)" on page 265
"Defining the output data set (SYSPRINT)" on page 265
"Directing compiler messages to your terminal (SYSTERM)" on page 265
"Creating object code (SYSLIN or SYSPUNCH)" on page 265
"Defining an associated-data file (SYSADATA)" on page 266
"Defining the Java-source output file (SYSJAVA)" on page 266
"Defining the debug data set (SYSDEBUG)" on page 266
"Defining the library-processing output file (SYSMDECK)" on page 267

RELATED REFERENCES
"Data sets used by the compiler under z/OS"
"Compiler options and compiler output under z/OS" on page 269

Data sets used by the compiler under z/OS

The following table lists the function, device requirements, and allowable device classes for each data set that the compiler uses.

Table 37. **Compiler data sets**

Type	ddname	Function	Required?	Device requirements	Allowable device classes	Reside in HFS?
Input	SYSIN[1]	Reads source program	Yes	Card reader; intermediate storage	Any	Yes
	SYSLIB or other copy libraries[1]	Reads user source libraries (PDSs)	If program has COPY or BASIS statements (LIB is required)	Direct access	SYSDA	No

Table 37. **Compiler data sets** *(continued)*

Type	ddname	Function	Required?	Device requirements	Allowable device classes	Reside in HFS?
Utility	SYSUT1, SYSUT2, SYSUT3, SYSUT4, SYSUT6[2]	Work data set used by compiler during compilation	Yes	Direct access	SYSDA	No
	SYSUT5[2]	Work data set used by compiler during compilation	If program has COPY, REPLACE, or BASIS statements (LIB is required)	Direct access	SYSDA	No
	SYSUT7[2]	Work data set used by compiler to create listing	Yes	Direct access	SYSDA	No
Output	SYSPRINT[1]	Writes storage map, listings, and messages	Yes	Printer; intermediate storage	SYSSQ, SYSDA, standard output class A	Yes
	SYSTERM	Writes progress and diagnostic messages	If TERM is in effect	Output device; TSO terminal		Yes
	SYSPUNCH	Creates object code	If DECK is in effect	Card punch; direct access	SYSSQ, SYSDA	Yes
	SYSLIN	Creates object module data set as output from compiler and input to linkage editor or binder	If OBJECT is in effect	Direct access	SYSSQ, SYSDA	Yes
	SYSADATA	Writes associated data file records	If ADATA or EVENTS is in effect	Output device		Yes
	SYSJAVA	Creates generated Java source file for a class definition	If compiling a class definition	(Must be an HFS file)		Yes
	SYSUDUMP, SYSABEND, or SYSMDUMP	Writes dump	If DUMP is in effect (should be rarely used)	Direct access	SYSDA	Yes
	SYSDEBUG	Writes symbolic debug information tables to a data set separate from the object module	If TEST(. . .,SYM,SEPARATE) is in effect	Direct access	SYSDA	Yes
	SYSMDECK	Writes expansion of COPY, BASIS, REPLACE, and EXEC SQL INCLUDE statements	If MDECK is in effect	Direct access	SYSDA	Yes

1. You can use the EXIT option to provide user exits from these data sets.
2. These data sets must be single volume.

RELATED REFERENCES
"Logical record length and block size" on page 264
"EXIT" on page 313

Logical record length and block size

For compiler data sets other than the work data sets (SYSUT*n*) and HFS files, you can set the block size using the BLKSIZE subparameter of the DCB parameter. The value must be permissible for the device on which the data set resides. The values you set depend on whether the data sets are fixed length or variable length.

For fixed-length records (RECFM=F or RECFM=FB), LRECL is the logical record length; and BLKSIZE equals LRECL multiplied by *n* where *n* is equal to the blocking factor.

The following table shows the defined values for the fixed-length data sets. In general, you should not change these values, but you can change the value for:

- SYSDEBUG: You can specify any LRECL in the listed range, with 1024 recommended.
- SYSPRINT, SYSDEBUG: You can specify BLKSIZE=0, which results in a system-determined block size.

Table 38. **Block size of fixed-length compiler data sets**

Data set	RECFM	LRECL (bytes)	BLKSIZE[1]
SYSDEBUG[2]	F or FB	80 to 1024	LRECL x *n*
SYSIN	F or FB	80	80 x *n*
SYSLIB or other copy libraries	F or FB	80	80 x *n*
SYSLIN	F or FB	80	80 x *n*
SYSMDECK	F or FB	80	80 x *n*
SYSPRINT[2]	F or FB	133	133 x *n*
SYSPUNCH	F or FB	80	80 x *n*
SYSTERM	F or FB	80	80 x *n*

1. *n* = blocking factor
2. If you specify BLKSIZE=0, the system will determine the block size.

For variable-length records (RECFM=V), LRECL is the logical record length, and BLKSIZE equals LRECL plus 4.

Table 39. **Block size of variable-length compiler data sets**

Data set	RECFM	LRECL (bytes)	BLKSIZE (bytes) minimum acceptable value
SYSADATA	VB	1020	1024

Defining the source code data set (SYSIN)

Define the data set that contains your source code by using the SYSIN DD statement.

```
//SYSIN  DD  DSNAME=dsname,UNIT=SYSSQ,
//  VOLUME=(subparms),DISP=SHR
```

You can place your source code or BASIS statement directly in the input stream. To do so, use this SYSIN DD statement:

```
//SYSIN  DD  *
```

The source code or BASIS statement must follow the DD * statement. If another job step follows the compilation, the EXEC statement for that step must follow the /* statement or the last source statement.

Specifying source libraries (SYSLIB)

Use SYSLIB DD statements if your program contains COPY or BASIS statements. These DD statements define the libraries (partitioned data sets) that contain the data requested by COPY statements in the source code or by BASIS statements in the input stream.

```
//SYSLIB  DD  DSNAME=copylibname,DISP=SHR
```

Concatenate multiple DD statements if you have multiple copy or basis libraries:

```
//SYSLIB  DD  DSNAME=PROJECT.USERLIB,DISP=SHR
//        DD  DSNAME=SYSTEM.COPYX,DISP=SHR
```

Libraries are on direct-access storage devices. They cannot be in the HFS when you compile with JCL or under TSO.

You do not need the SYSLIB DD statement if the NOLIB option is in effect.

Defining the output data set (SYSPRINT)

You can use ddname SYSPRINT to produce a listing. The listing includes the results of the default or requested options of the PARM parameter (that is, diagnostic messages and the object-code listing).

You can direct the output to a SYSOUT data set, a printer, a direct-access storage device, or a magnetic-tape device. For example:

```
//SYSPRINT DD SYSOUT=A
```

Specify the following items for the data set that you define:

- The name of a sequential data set, the name of a PDS or PDSE member, or an HFS path.
- A data set LRECL of 133.
- A data set RECFM of F or FB.
- A block size, using the BLKSIZE subparameter of the DCB parameter. Otherwise, let the system set a system-determined default block size.

Directing compiler messages to your terminal (SYSTERM)

If you are compiling under TSO, you can define the SYSTERM data set to send compiler messages to your terminal.

```
ALLOC F(SYSTERM) DA(*)
```

You can define SYSTERM in various other ways, for example to a SYSOUT data set, a data set on disk, a file in the HFS, or to another print class.

Creating object code (SYSLIN or SYSPUNCH)

When using the OBJECT compiler option, you can store the object code on disk as a traditional MVS data set or an HFS file, or on tape. The compiler uses the file that you define in the SYSLIN or SYSPUNCH DD statement.

```
//SYSLIN   DD  DSNAME=dsname,UNIT=SYSDA,
//             SPACE=(subparms),DISP=(MOD,PASS)
```

Use the DISP parameter of the SYSLIN DD statement to indicate whether the object code data set is to be:

- Passed to the linkage editor or binder
- Cataloged
- Kept
- Added to an existing cataloged library

In the example above, the data is created and passed to another job step, the linkage editor or binder job step.

Your installation might use the DECK option and the SYSPUNCH DD statement. B is the standard output class for punch data sets:

```
//SYSPUNCH DD  SYSOUT=B
```

You do not need the SYSLIN DD statement if the NOOBJECT option is in effect. You do not need the SYSPUNCH DD statement if the NODECK option is in effect.

RELATED REFERENCES
"OBJECT" on page 326
"DECK" on page 310

Defining an associated-data file (SYSADATA)

Define a SYSADATA file if you use the ADATA compiler option.

```
//SYSADATA DD DSNAME=dsname,UNIT=SYSDA
```

The SYSADATA file will be a sequential file that contains specific record types that have information about the program that is collected during compilation. The file can be a traditional MVS data set or an HFS file.

RELATED REFERENCES
"ADATA" on page 301

Defining the Java-source output file (SYSJAVA)

Add the SYSJAVA DD statement if you are compiling an OO program. The generated Java source file is written to the SYSJAVA ddname.

```
//SYSJAVA   DD PATH='/u/userid/java/Classname.java',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU,
// FILEDATA=TEXT
```

The SYSJAVA file must be in the HFS.

RELATED TASKS
"Compiling OO applications using JCL or TSO/E" on page 291

Defining the debug data set (SYSDEBUG)

When you compile from JCL or from TSO and specify the TEST(. . .,SYM,SEPARATE) compiler option, the symbolic debug information tables are written to the data set that you specify in the SYSDEBUG DD statement.

```
//SYSDEBUG DD  DSNAME=dsname,UNIT=SYSDA
```

Specify the following items for the SYSDEBUG data set:

- The name of a sequential data set, the name of a PDS or PDSE member, or an HFS path.
- A data set LRECL that is greater than or equal to 80 and less than or equal to 1024. The default LRECL for SYSDEBUG is 1024.
- A data set RECFM of F or FB.
- A block size, using the BLKSIZE subparameter of the DCB parameter; or let the system set the system-determined default block size.

The data set name that you provide in ddname SYSDEBUG is used by Language Environment dump services. You cannot change the name of that data set at run time. If you want the name of this data set to follow the naming conventions for your production data sets, use that name when you compile the program. You can direct Debug Tool to a renamed data set.

RELATED REFERENCES
"TEST" on page 338

Defining the library-processing output file (SYSMDECK)

Define a SYSMDECK file if you use the MDECK compiler option.

```
//SYSMDECK DD DSNAME=dsname,UNIT=SYSDA
```

The SYSMDECK file will contain the output from library processing, that is, the expansion of COPY, BASIS, REPLACE, and EXEC SQL INCLUDE statements. The file can be a traditional MVS data set or an HFS file.

RELATED REFERENCES
"MDECK" on page 321

Specifying compiler options under z/OS

The compiler is installed with default compiler options. While installing the compiler, the system programmer can fix compiler option settings to, for example, ensure better performance or maintain certain standards. You cannot override any compiler options that are fixed.

For options that are not fixed, you can override the default settings by specifying compiler options in either of these ways:
- Code them on the PROCESS or CBL statement in COBOL source.
- Include them when you start the compiler, either on the PARM parameter on the EXEC statement in the JCL or on the command line under TSO.

The compiler recognizes the options in the following order of precedence from highest to lowest:
1. Installation defaults that are fixed by your site
2. Values of the BUFSIZE, LIB, OUTDD, SIZE, and SQL compiler options in effect for the first program in a batch
3. Options specified on PROCESS (or CBL) statements, preceding the IDENTIFICATION DIVISION
4. Options specified on the compiler invocation (JCL PARM parameter or the TSO CALL command)
5. Installation defaults that are not fixed

This order of precedence also determines which options are in effect when conflicting or mutually exclusive options are specified.

Most of the options come in pairs; you select one or the other. For example, the option pair for a cross-reference listing is `XREF|NOXREF`. If you want a cross-reference listing, specify `XREF`; if you do not, specify `NOXREF`.

Some options have subparameters. For example, if you want 44 lines per page on your listings, specify `LINECOUNT(44)`.

"Example: specifying compiler options using JCL" on page 269
"Example: specifying compiler options under TSO" on page 269

RELATED TASKS
"Specifying compiler options with the PROCESS (CBL) statement"
"Specifying compiler options in a batch compilation" on page 272

RELATED REFERENCES
"Compiler options and compiler output under z/OS" on page 269
"Conflicting compiler options" on page 300
Chapter 17, "Compiler options," on page 297

Specifying compiler options with the PROCESS (CBL) statement

You can code compiler options in the `PROCESS` statement in COBOL programs. Code it before the `IDENTIFICATION DIVISION` header and before any comment lines or compiler-directing statements.

CBL/PROCESS statement syntax

```
►►──┬─CBL─────┬──┬──────────────┬──────────────────────────────►◄
    └─PROCESS─┘  └─options-list─┘
```

You can start the `PROCESS` statement in column 1 through 66 if you don't code a sequence field. A sequence field is allowed in columns 1 through 6; if used, the sequence field must contain six characters, and the first character must be numeric. When used with a sequence field, `PROCESS` can start in column 8 through 66.

You can use `CBL` as a synonym for `PROCESS`. `CBL` can start in column 1 through 70. When used with a sequence field, `CBL` can start in column 8 through 70.

Use one or more blanks to separate `PROCESS` from the first option in *options-list*. Separate options with a comma or a blank. Do not insert spaces between individual options and their suboptions.

You can use more than one `PROCESS` statement. If you do so, the `PROCESS` statements must follow each other with no intervening statements. You cannot continue options across multiple `PROCESS` statements.

Your programming organization can inhibit the use of PROCESS statements by using the default options module of the COBOL compiler. When PROCESS statements are found in a COBOL program but are not allowed by the organization, the COBOL compiler generates error diagnostics.

RELATED REFERENCES
CBL (PROCESS) statement (*Enterprise COBOL Language Reference*)

Example: specifying compiler options using JCL

The following example shows how to specify compiler options under z/OS using JCL.

```
. . .
//STEP1     EXEC PGM=IGYCRCTL,
//               PARM='LIST,NOCOMPILE(S),OBJECT,FLAG(E,E)'
```

Example: specifying compiler options under TSO

The following example shows how to specify compiler options under TSO.

```
. . .
[READY]
CALL 'SYS1.LINKLIB(IGYCRCTL)' 'LIST,NOCOMPILE(S),OBJECT,FLAG(E,E)'
```

Compiler options and compiler output under z/OS

When the compiler finishes processing your source program, it will have produced one or more outputs, depending on the compiler options that were in effect.

Table 40. **Types of compiler output under z/OS**

Compiler option	Compiler output	Type of output
ADATA or EVENTS	Information about the program being compiled	Associated-data file
DLL	Object module that is enabled for DLL support	Object
DUMP	System dump, if compilation ended with abnormal termination (requires SYSUDUMP, SYSABEND, or SYSMDUMP DD statement); should be used rarely	Listing
EXPORTALL	Exported symbols for a DLL	Object
FLAG	List of errors that the compiler found in your program	Listing
LIST	Listing of object code in machine and assembler language	Listing
MAP	Map of the data items in your program	Listing
MDECK	Expansion of library-processing statements in your program	Library-processing side file
NUMBER	User-supplied line numbers shown in listing	Listing
OBJECT or DECK with COMPILE	Your object code	Object
OFFSET	Map of the relative addresses in your object code	Listing
OPTIMIZE	Optimized object code if OBJECT in effect	Object
RENT	Reentrant object code if OBJECT in effect	Object
SOURCE	Listing of your source program	Listing
SQL	SQL statements and host variable information for DB2 bind process	Database request module
SSRANGE	Extra code for checking references within tables	In object
TERMINAL	Progress and diagnostic messages sent to terminal	Terminal

Table 40. **Types of compiler output under z/OS** *(continued)*

Compiler option	Compiler output	Type of output
TEST(*hook location suboption*)	Compiled-in hooks for Debug Tool	Extra code in object
TEST(SYM,NOSEP)	Information tables for Debug Tool and for formatted dumps	Object
TEST(SYM,SEP)	Information tables for Debug Tool and for formatted dumps	Separate debug file
VBREF	Cross-reference listing of verbs in your source program	Listing
XREF	Sorted cross-reference listing of names of procedures, programs, and data	Listing

Listing output from compilation will be in the data set defined by SYSPRINT; object output will be in SYSLIN or SYSPUNCH. Progress and diagnostic messages can be directed to the SYSTERM data set as well as included in the SYSPRINT data set. The database request module (DBRM) is the data set defined in DBRMLIB. The separate debug file is the data set defined in SYSDEBUG.

Save the listings you produced during compilation. You can use them during the testing of your work if you need to debug or tune.

After compilation, you next fix any errors that the compiler found in your program. If no errors were detected, you can go to the next step in the process: link-editing, or binding, your program. (If you used compiler options to suppress object code generation, you must recompile to obtain object code.)

RELATED REFERENCES
"Messages and listings for compiler-detected errors" on page 275
Chapter 17, "Compiler options," on page 297

RELATED TASKS
Preparing to link-edit and run under Language Environment (*Language Environment Programming Guide*)

Compiling multiple programs (batch compilation)

You can compile a sequence of separate COBOL programs by using a single invocation of the compiler. You can link the object program produced from this compilation into one load module or separate load modules, controlled by the NAME compiler option.

When you compile several programs as part of a batch job, you need to:

- Determine whether you want to create one or more load modules.
- Terminate each program in the sequence.
- Specify compiler options, with an awareness of the effect of compiler options specified in programs within the batch job.

To create separate load modules, precede each set of modules with the NAME compiler option. When the compiler encounters the NAME option, the first program in the sequence and all subsequent programs until the next NAME compiler option is encountered are link-edited into a single load module. Then each successive program that is compiled with the NAME option is included in a separate load module.

Use the END PROGRAM marker to terminate each program in the sequence except the last program in the batch (for which the END PROGRAM marker is optional). Alternatively, you can precede each program in the sequence with a CBL or PROCESS statement.

If you omit the END PROGRAM marker from a program (other than the last program in a sequence of separate programs), the next program in the sequence will be nested in the preceding program. An error can occur in either of the following situations:

- A PROCESS statement is in a program that is now nested.
- A CBL statement is not coded entirely in the sequence number area (columns 1 through 6).

  If a CBL statement is coded entirely in the sequence number area (columns 1 through 6), no error message is issued for the CBL statement because it is considered a label for the source statement line.

"Example: batch compilation"

RELATED TASKS
"Specifying compiler options in a batch compilation" on page 272

RELATED REFERENCES
"NAME" on page 323

Example: batch compilation

The following example shows a batch compilation for three programs (PROG1, PROG2, and PROG3) and the creation of two load modules using one invocation of the IGYWCL cataloged procedure.

The following steps occur:

- PROG1 and PROG2 are link-edited together to form one load module that has the name PROG2. The entry point of this load module defaults to the first program in the load module, PROG1.
- PROG3 is link-edited by itself into a load module that has the name PROG3. Because it is the only program in the load module, the entry point is also PROG3.

```
//jobname  JOB  acctno,name,MSGLEVEL=1
//stepname EXEC IGYWCL
//COBOL.SYSIN  DD *
010100 IDENTIFICATION DIVISION.
010200  PROGRAM-ID PROG1.
        . . .
019000  END PROGRAM PROG1.
020100 IDENTIFICATION DIVISION.
020200  PROGRAM-ID PROG2.
        . . .
029000  END PROGRAM PROG2.
 CBL NAME
030100 IDENTIFICATION DIVISION.
030200  PROGRAM-ID PROG3.
        . . .
039000  END PROGRAM PROG3.
/*
//LKED.SYSLMOD DD DSN=&&GOSET                  (1)
```

```
/*
//P2       EXEC PGM=PROG2
//STEPLIB  DD   DSN=&&GOSET,DISP=(SHR,PASS)     (2)
. . .                                           (3)
/*
//P3       EXEC PGM=PROG3
//STEPLIB  DD   DSN=&&GOSET,DISP=(SHR,PASS)     (2)
. . .                                           (4)
/*
//
```

(1) The data set name for the LKED step SYSLMOD is changed to the temporary name &&GOSET, without any member name.

(2) The temporary data set &&GOSET is used as the STEPLIB for steps P2 and P3 to run the compiled programs. If the Language Environment library does not reside in shared storage, you must also add the library data set as a DD statement for STEPLIB.

(3) Other DD statements and input that are required to run PROG1 and PROG2 must be added.

(4) Other DD statements and input that are required to run PROG3 must be added.

RELATED REFERENCES
IBM-supplied cataloged procedures (*Language Environment Programming Guide*)

Specifying compiler options in a batch compilation

You can specify compiler options for each program in the batch sequence either with a CBL or PROCESS statement that precedes the program, or upon invocation of the compiler.

If a CBL or PROCESS statement is specified in the current program, the compiler resolves the CBL or PROCESS statements together with the options in effect before the first program. If the current program does not contain CBL or PROCESS statements, the compiler uses the settings of options in effect for the previous program.

You should be aware of the effect of certain compiler options on the precedence of compiler option settings for each program in the sequence:

1. Installation defaults that are fixed at your site
2. Values of the BUFSIZE, LIB, OUTDD, SIZE, and SQL compiler options in effect for the first program in the batch
3. Options on CBL or PROCESS statements, if any, for the current program
4. Options specified in the compiler invocation (JCL PARM or TSO CALL)
5. Installation defaults that are not fixed

If any program in the batch sequence requires the BUF, LIB, OUTDD, SIZE, or SQL option, that option must be in effect for the first program in the batch sequence. (When processing BASIS, COPY, or REPLACE statements, the compiler handles all programs in the batch as a single input file.)

If you specify the LIB option for the batch, you cannot change the NUMBER and SEQUENCE options during the batch compilation. The compiler treats all programs in the batch as a single input file during NUMBER and SEQUENCE processing under the LIB option; therefore, the sequence numbers of the entire input file must be in ascending order.

If the compiler diagnoses the LANGUAGE option on the CBL or PROCESS statement as an error, the language selection reverts to what was in effect before the compiler encountered the first CBL or PROCESS statement. The language in effect during a batch compilation conforms to the rules of processing CBL or PROCESS statements in that environment.

"Example: precedence of options in a batch compilation"
"Example: LANGUAGE option in a batch compilation"

Example: precedence of options in a batch compilation

The following example listing shows the precedence of compiler options for batch compilation.

```
PP 5655-G53 IBM Enterprise COBOL for z/OS  3.4.0  Date 06/30/2005. . .

INVOCATION PARAMETERS:
NOTERM

PROCESS(CBL) statements:
        CBL  CURRENCY,FLAG(I,I)

Options in effect:  All options are installation defaults unless otherwise noted:
    NOADATA
     ADV
      QUOTE
    NOAWO
      BUFSIZE(4096)
      CURRENCY      Process option PROGRAM 1
      . . .
      FLAG(I,I)     Process option PROGRAM 1
      . . .
    NOTERM          INVOCATION option
      . . .
      End of compilation for program 1
      . . .
PP 5655-G53 IBM Enterprise COBOL for z/OS  3.4.0  Date 06/30/2005. . .
PROCESS(CBL) statements:
       CBL APOST
Options in effect:
    NOADATA
      ADV
    APOST           Process option PROGRAM 2
    NOAWO
      BUFSIZE(4096)
    NOCURRENCY      Installation default option for PROGRAM 2
      . . .
      FLAG(I)       Installation default option
      . . .
    NOTERM          INVOCATION option remains in effect
      . . .
End of compilation for program 2
```

Example: LANGUAGE option in a batch compilation

The following example shows the behavior of the LANGUAGE compiler option in a batch environment. The default installation option is ENGLISH (abbreviated EN), and the invocation option is XX, a nonexistent language.

```
CBL LANG(JP),FLAG(I,I),APOST,SIZE(MAX)  (1)
    IDENTIFICATION DIVISION.            (2)
    PROGRAM-ID.  COMPILE1.
    .  .  .
    END PROGRAM  COMPILE1.
CBL LANGUAGE(YY)                        (3)
CBL SIZE(2048K),LANGUAGE(JP),LANG(!!)   (4)
```

```
    IDENTIFICATION DIVISION.            (2)
    PROGRAM-ID.  COMPILE2.
    .  .  .
    END PROGRAM  COMPILE2.
    IDENTIFICATION DIVISION.
    PROGRAM-ID.  COMPILE3.
    .  .  .
    END PROGRAM  COMPILE3.
CBL LANGUAGE(JP),LANGUAGE(YY)           (5)
    .  .  .
```

(1) The installation default is EN. The invocation option was XX, a nonexistent language. EN is the language in effect.

(2) After the CBL statement is scanned, JP is the language in effect.

(3) CBL resets the language to EN. YY is ignored because it is superseded by JP.

(4) !! is not alphanumeric and is discarded.

(5) CBL resets the language to EN. YY supersedes JP but is nonexistent.

For the program COMPILE1, the default language English (EN) is in effect when the compiler scans the invocation options. A diagnostic message is issued in mixed-case English because XX is a nonexistent language identifier. The default EN remains in effect when the compiler scans the CBL statement. The unrecognized option APOST in the CBL statement is diagnosed in mixed-case English because the CBL statement has not completed processing and EN was the last valid language option. After the compiler processes the CBL options, the language in effect becomes Japanese (JP).

In the program COMPILE2, the compiler diagnoses CBL statement errors in mixed-case English because English is the language in effect before the first program is used. If more than one LANGUAGE option is specified, only the last valid language specified is used. In this example, the last valid language is Japanese (JP). Therefore Japanese becomes the language in effect when the compiler finishes processing the CBL options. If you want diagnostics in Japanese for the options in the CBL and PROCESS statements, the language in effect before COMPILE1 must be Japanese.

The program COMPILE3 has no CBL statement. It inherits the language in effect, Japanese (JP), from the previous compilation.

After compiling COMPILE3, the compiler resets the language in effect to English (EN) because of the CBL statement. The language option in the CBL statement resolves the last-specified two-character alphanumeric language identifier, YY. Because YY is nonexistent, the language in effect remains English.

Correcting errors in your source program

Messages about source-code errors indicate where the error occurred (LINEID). The text of a message tells you what the problem is. With this information, you can correct the source program.

Although you should try to correct errors, it is not necessary to fix all of them. You can leave a warning-level or informational-level message in a program without much risk, and you might decide that the recoding and compilation that are needed to remove the error are not worth the effort. Severe-level and error-level errors, however, indicate probable program failure and should be corrected.

In contrast with the four lower levels of errors, an unrecoverable (U-level) error might not result from a mistake in your source program. It could come from a flaw in the compiler itself or in the operating system. In any case, the problem must be resolved, because the compiler is forced to end early and does not produce complete object code or listing. If the message occurs for a program that has many S-level syntax errors, correct those errors and compile the program again. You can also resolve job set-up problems (problems such as missing data-set definitions or insufficient storage for compiler processing) by making changes to the compile job. If your compile job setup is correct and you have corrected the S-level syntax errors, you need to contact IBM to investigate other U-level errors.

After correcting the errors in your source program, recompile the program. If this second compilation is successful, proceed to the link-editing step. If the compiler still finds problems, repeat the above procedure until only informational messages are returned.

RELATED TASKS
"Generating a list of compiler error messages"

RELATED REFERENCES
"Messages and listings for compiler-detected errors"

Generating a list of compiler error messages

You can generate a complete listing of compiler diagnostic messages with their explanations by compiling a program that has the program-name `ERRMSG`.

You can code just the `PROGRAM-ID` paragraph, as shown below. Omit the rest of the program.

```
Identification Division.
Program-ID. ErrMsg.
```

RELATED REFERENCES
"Messages and listings for compiler-detected errors"
"Format of compiler error messages" on page 276

Messages and listings for compiler-detected errors

As the compiler processes your source program, it checks for COBOL language errors. For each error found, the compiler issues a message. These messages are collated in the compiler listing (subject to the `FLAG` option).

Each message in the listing provides the following information:
- Nature of the error
- Compiler phase that detected the error
- Severity level of the error

Wherever possible, the message provides specific instructions for correcting the error.

The messages for errors found during processing of compiler options, `CBL` and `PROCESS` statements, and `BASIS`, `COPY`, or `REPLACE` statements are displayed near the top of the 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 the listing.

RELATED TASKS
"Correcting errors in your source program" on page 274
"Generating a list of compiler error messages" on page 275

RELATED REFERENCES
"Format of compiler error messages"
"Severity codes for compiler error messages"
"FLAG" on page 314

Format of compiler error messages

Each message issued by the compiler has a source line number, a message identifier, and message text.

Each message has the following form:

```
nnnnnn IGYppxxxx-l message-text
```

nnnnnn
: The number of the source statement of the last line that 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 or subphase 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 four-digit number that identifies the error message.

l
: A character that indicates the severity level of the error: I, W, E, S, or U.

message-text
: The message text, which in the case of an error message is a short explanation of the condition that caused the error.

Tip: If you used the `FLAG` option to suppress messages, there might be additional errors in your program.

RELATED REFERENCES
"Severity codes for compiler error messages"
"FLAG" on page 314

Severity codes for compiler error messages

Errors that the compiler can detect fall into five categories of severity.

Table 41. **Severity codes for compiler error messages**

Level of message	Return code	Purpose
Informational (I)	0	To inform you. No action is required and the program runs correctly.

Table 41. **Severity codes for compiler error messages** *(continued)*

Level of message	Return code	Purpose
Warning (W)	4	To indicate a possible error. The program probably runs correctly as written.
Error (E)	8	To indicate a condition that is definitely an error. The compiler attempted to correct the error, but the results of program execution might not be what you expect. You should correct the error.
Severe (S)	12	To indicate a condition that is a serious error. The compiler was unable to correct the error. The program does not run correctly, and execution should not be attempted. Object code might not be created.
Unrecoverable (U)	16	To indicate an error condition of such magnitude that the compilation was terminated.

Chapter 15. Compiling under UNIX

Compile Enterprise COBOL programs under UNIX by using the `cob2` command. Under UNIX, you can compile any COBOL program that you can compile under z/OS. The object code generated under UNIX by the COBOL compiler can run under z/OS.

As part of the compilation step, you define the files needed for the compilation, and specify any compiler options or compiler-directing statements that are necessary for your program and for the output that you want.

The main job of the compiler is to translate COBOL programs into language that the computer can process (object code). The compiler also lists errors in source statements and provides supplementary information to help you debug and tune programs.

RELATED TASKS
"Setting environment variables under UNIX"
"Specifying compiler options under UNIX" on page 280
"Compiling and linking with the cob2 command" on page 281
"Compiling using scripts" on page 285
"Compiling, linking, and running OO applications under UNIX" on page 287

RELATED REFERENCES
"Data sets used by the compiler under z/OS" on page 262
"Compiler options and compiler output under z/OS" on page 269

Setting environment variables under UNIX

An *environment variable* is a name that is associated with a string of characters and that defines some variable aspect of the program environment. You use environment variables to set values that programs, including the compiler, need.

Set the environment variables for the compiler by using the `export` command. For example, to set the SYSLIB variable, issue the `export` command from the shell or from a script file:

```
export SYSLIB=/u/mystuff/copybooks
```

The value that you assign to an environment variable can include other environment variables or the variable itself. The values of these variables apply only when you compile from the shell where you issue the `export` command. If you do not set an environment variable, either a default value is applied or the variable is not defined. The environment-variable names must be uppercase.

The environment variables that you can set for use by the compiler are as follows:

COBOPT
Specify compiler options separated by blanks or commas. Separate suboptions with commas. Blanks at the beginning or the end of the variable value are ignored. Delimit the list of options with quotation marks if it contains blanks or characters that are significant to the UNIX shell. For example:

```
export COBOPT="TRUNC(OPT) XREF"
```

SYSLIB
Specify paths to directories to be used in searching for COBOL copybooks if you do not specify an explicit library-name in the `COPY` statement. Separate multiple paths with a colon. Paths are evaluated in order from the first path to the last in the `export` command. If you set the variable with multiple files of the same name, the first located copy of the file is used.

For `COPY` statements in which you have not coded an explicit library-name, the compiler searches for copybooks in this order:

1. In the current directory
2. In the paths you specify with the `-I` `cob2` option
3. In the paths you specify in the SYSLIB environment variable

library-name
Specify the directory path from which to copy when you specify an explicit library-name in the `COPY` statement. The environment-variable name is identical to the *library-name* in your program. You must set an environment variable for each library; an error will occur otherwise. The environment-variable name *library-name* must be uppercase.

text-name
Specify the name of the file from which to copy text. The environment-variable name is identical to the *text-name* in your program. The environment-variable name *text-name* must be uppercase.

RELATED TASKS
"Specifying compiler options under UNIX"
"Compiling and linking with the cob2 command" on page 281
"Setting and accessing environment variables" on page 424

RELATED REFERENCES
Chapter 18, "Compiler-directing statements," on page 351
Chapter 17, "Compiler options," on page 297
COPY statement (*Enterprise COBOL Language Reference*)

Specifying compiler options under UNIX

The compiler is installed and set up with default compiler options. While installing the compiler, a system programmer can fix compiler option settings to ensure better performance or maintain certain standards. You cannot override any compiler options that your site has fixed.

For options that are not fixed, you can override the default settings by specifying compiler options in any of three ways:

- Code them on the `PROCESS` or `CBL` statement in your COBOL source.
- Specify the `-q` option of the `cob2` command.
- Set the COBOPT environment variable.

The compiler recognizes the options in the above order of precedence, from highest to lowest. The order of precedence also determines which options are in effect when conflicting or mutually exclusive options are specified. When you compile using the `cob2` command, compiler options are recognized in the following order of precedence, from highest to lowest:

- Installation defaults fixed as nonoverridable

- The values of BUFSIZE, LIB, SQL, OUTDD, and SIZE options in effect for the first program in a batch compilation
- The values that you specify on PROCESS or CBL statements in COBOL source programs
- The values that you specify in the cob2 command's -q option string
- The values that you specify in the COBOPT environment variable
- Installation defaults that are not fixed

Restriction: Do not use the SQL option under UNIX. Neither the separate SQL precompiler nor the integrated SQL coprocessor run under UNIX.

RELATED TASKS
"Specifying compiler options with the PROCESS (CBL) statement" on page 268
"Setting environment variables under UNIX" on page 279
"Compiling and linking with the cob2 command"

RELATED REFERENCES
"Conflicting compiler options" on page 300
Chapter 17, "Compiler options," on page 297

Compiling and linking with the cob2 command

Use the cob2 command to compile and link COBOL programs from the UNIX shell. You can specify the options and input file-names in any order, using spaces to separate options and names. Any options that you specify apply to all files on the command line.

To compile multiple files (batch compilation), specify multiple source-file names.

When you compile COBOL programs for UNIX, the RENT option is required. The cob2 command automatically includes the COBOL compiler options RENT and TERM.

The cob2 command invokes the COBOL compiler that is found through the standard MVS search order. If the COBOL compiler is not installed in the LNKLST, or if more than one level of IBM COBOL compiler is installed on your system, you can specify in the STEPLIB environment variable the compiler PDS that you want to use. For example, the following statement specifies IGY.V3R4M0 as the compiler PDS:

```
export STEPLIB=IGY.V3R4M0.SIGYCOMP
```

The cob2 command implicitly uses the UNIX shell command c89 for the link step. c89 is the shell interface to the linker (the z/OS program management binder).

The default location for compiler input and output is the current directory.

Only files with the suffix .cbl are passed to the compiler; cob2 passes all other files to the linker.

The listing output that you request from the compilation of a COBOL source program *file*.cbl is written to *file*.lst. The listing output that you request from the linker is written to stdout.

The linker causes execution to begin at the first main program.

RELATED TASKS
"Creating a DLL under UNIX"
"Preparing OO applications under UNIX" on page 288
UNIX System Services User's Guide

RELATED REFERENCES
"cob2 syntax and options" on page 283
"cob2 input and output files" on page 284
UNIX System Services Command Reference

Creating a DLL under UNIX

To create a DLL from the UNIX shell, you must specify the `cob2` option `-bdll`.

```
cob2 -o mydll -bdll mysub.cbl
```

When you specify `cob2 -bdll`:

- The COBOL compiler uses the compiler options `DLL`, `EXPORTALL`, and `RENT`, which are required for DLLs.
- The link step produces a DLL definition side file that contains `IMPORT` control statements for each of the names exported by the DLL.

The name of the DLL definition side file is based on the output file-name. If the output name has a suffix, that suffix is replaced with *x* to form the side-file name. For example, if the output file-name is foo.dll, the side-file name is foo.x.

To use the DLL definition side file later when you create a module that calls that DLL, specify the side file with any other object files (*file*.o) that you need to link. For example, the following command compiles myappl.cbl, uses the `DLL` option to enable myappl.o to reference DLLs, and links to produce the module myappl:

```
cob2 -o myappl -qdll myappl.cbl mydll.x
```

"Example: using cob2 to compile and link under UNIX"

RELATED TASKS
Chapter 26, "Creating a DLL or a DLL application," on page 465
"Compiling programs to create DLLs" on page 466

RELATED REFERENCES
"cob2 syntax and options" on page 283
"cob2 input and output files" on page 284

Example: using cob2 to compile and link under UNIX

The following examples illustrate the use of `cob2`.

- To compile one file called alpha.cbl, enter:
  ```
  cob2 -c alpha.cbl
  ```
  The compiled file is named alpha.o.
- To compile two files called alpha.cbl and beta.cbl, enter:
  ```
  cob2 -c alpha.cbl beta.cbl
  ```
  The compiled files are named alpha.o and beta.o.
- To link two files, compile them without the `-c` option. For example, to compile and link alpha.cbl and beta.cbl and generate gamma, enter:
  ```
  cob2 alpha.cbl beta.cbl -o gamma
  ```

This command creates alpha.o and beta.o, then links alpha.o, beta.o, and the COBOL libraries. If the link step is successful, it produces an executable program named gamma.

- To compile alpha.cbl with the `LIST` and `NODATA` options, enter:

```
cob2 -qlist,noadata alpha.cbl
```

cob2 syntax and options

cob2 command syntax

```
►►──cob2──┬─────────┬──filenames──────────────────────────────────────►◄
          └─options─┘
```

You can use the options listed below with the `cob2` command. (Do not capitalize `cob2`.)

-b*xxx* Passes the string *xxx* to the linker as parameters. *xxx* is a list of linker options in *name*=*value* format, separated by commas. You must spell out both the name and the value in full (except for the special cases noted below). The name and value are case insensitive. Do not use any spaces between `-b` and *xxx*.

If you do not specify a value for an option, a default value of `YES` is used except for the following options, which have the indicated default values:

- `LIST=NOIMPORT`
- `ALIASES=ALL`
- `COMPAT=CURRENT`
- `DYNAM=DLL`

One special value for *xxx* is `dll`, which specifies that the executable module is to be a DLL. This string is not passed to the linker.

-c Compiles programs but does not link them.

-comprc_ok=*n*

Controls `cob2` behavior on the return code from the compiler. If the return code 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. When the `c89` command is implicitly invoked by `cob2` for the link step, the exit value from the `c89` command is used as the return code from the `cob2` command.

The default is `-comprc_ok=4`.

-e *xxx*

Specifies the name of a program to be used as the entry point of the module. If you do not specify `-e`, the default entry point is the first program (*file*.cbl) or object file (*file*.o) that you specify as a file name on the `cob2` command invocation.

-g Prepares the program for debugging. Equivalent to specifying the `TEST` option with no suboptions.

-I*xxx* Adds a path *xxx* to the directories to be searched for copybooks for which you do not specify a *library-name*.

To specify multiple paths, either use multiple `-I` options, or use a colon to separate multiple path names within a single `-I` option value.

For `COPY` statements in which you have not coded an explicit library-name, the compiler searches for copybooks in this order:

1. In the current directory
2. In the paths you specify with the `-I` `cob2` option
3. In the paths you specify in the SYSLIB environment variable

If you use the `COPY` statement, you must ensure that the `LIB` compiler option is in effect.

`-L` *xxx*
: Specifies the directory paths to be used to search for archive libraries specified by the `-l` operand.

-l *xxx*
: Specifies the name of an archive library for the linker. The `cob2` command searches for the name lib*xxx*.a in the directories specified in the `-L` option, then in the usual search order. (This option is lowercase “el,” not uppercase “eye.”)

`-o` *xxx*
: Names the object module *xxx*. If the `-o` option is not used, the name of the object module is a.out.

`-q`*xxx*
: Passes *xxx* to the compiler, where *xxx* is a list of compiler options separated by blanks or commas.

Enclose *xxx* in quotation marks if a parenthesis is part of the option or suboption, or if you use blanks to separate options. Do not insert spaces between `-q` and *xxx*.

`-v`
: Displays the generated commands that are issued by `cob2` for the compile and link steps, including the options being passed, and executes them. This is sample output:

```
cob2 -v -o mini -qssrange mini.cbl
compiler: ATTCRCTL PARM=RENT,TERM,SSRANGE /u/userid/cobol/mini.cbl
PP 5655-G53 IBM Enterprise COBOL for z/OS  3.4.0  in progress ...
End of compilation 1,  program mini,  no statements flagged.
linker: /bin/c89 -o mini -e // mini.o
```

`-#`
: Displays compile and link steps, but does not execute them.

RELATED TASKS
“Compiling and linking with the cob2 command” on page 281
“Creating a DLL under UNIX” on page 282
“Setting environment variables under UNIX” on page 279

cob2 input and output files

You can specify the following files as input file-names when you use the `cob2` command.

Table 42. **Input files to the cob2 command**

File name	Description	Comments
file.cbl	COBOL source file to be compiled and linked	Will not be linked if you specify the `cob2` option `-c`
file.a	Archive file	Produced by the `ar` command, to be used during the link-edit phase

Table 42. **Input files to the cob2 command** *(continued)*

File name	Description	Comments
file.o	Object file to be link-edited	Can be produced by the COBOL compiler, the C/C++ compiler, or the assembler
file.x	DLL definition side file	Used during the link-edit phase of an application that references the dynamic link library (DLL)

When you use the `cob2` command, the following files are created in the current directory.

Table 43. **Output files from the cob2 command**

File name	Description	Comments
file	Executable module or DLL	Created by the linker if you specify the `cob2` option `-o` *file*
a.out	Executable module or DLL	Created by the linker if you do not specify the `cob2` option `-o`
file.adt	Associated data (ADATA) file corresponding to input COBOL source program *file*.cbl	Created by the compiler if you specify the compiler option `ADATA`
file.dbg	Symbolic information tables for Debug Tool corresponding to input COBOL source program *file*.cbl	Created by the compiler if you specify the compiler option `TEST(`*`hook`*`,SYM,SEPARATE)`
file.dek	Extended COBOL source output from library processing	Created by the compiler if you specify the compiler option `MDECK`
file.lst	Listing file corresponding to input COBOL source program *file*.cbl	Created by the compiler
file.o	Object file corresponding to input COBOL source program *file*.cbl	Created by the compiler
file.x	DLL definition side file	Created during the `cob2` linking phase when creating a DLL named *file*.dll
class.java	Java class definition (source)	Created when you compile a class definition

RELATED TASKS
"Compiling and linking with the cob2 command" on page 281

RELATED REFERENCES
"ADATA" on page 301
"MDECK" on page 321
"TEST" on page 338
UNIX System Services Command Reference

Compiling using scripts

If you use a shell script to automate `cob2` tasks, you must code option syntax carefully to prevent the shell from passing invalid strings to `cob2`.

Code option strings in scripts as follows:

- Use an equal sign and colon rather than a left and right parenthesis, respectively, to specify compiler suboptions. For example, code `-qOPT=FULL:,XREF` instead of `-qOPT(FULL),XREF`.
- Use an underscore rather than a single quotation mark where a compiler option requires single quotation marks for delimiting a suboption.
- Do not use blanks in the option string.

Chapter 16. Compiling, linking, and running OO applications

It is recommended that you compile, link, and run object-oriented (OO) applications in the z/OS UNIX environment. However, with certain limitations discussed in the related tasks, it is possible to compile, link, and run OO COBOL applications by using standard batch JCL or TSO/E commands.

RELATED TASKS
"Compiling, linking, and running OO applications under UNIX"
"Compiling, linking, and running OO applications using JCL or TSO/E" on page 291
"Using IBM SDK for z/OS, Java 2 Technology Edition, V1.4" on page 295

Compiling, linking, and running OO applications under UNIX

When you compile, link, and run OO applications in a z/OS UNIX environment, application components reside in the HFS. You compile and link them by using UNIX shell commands, and run them at a UNIX shell command prompt or with the BPXBATCH utility from JCL or TSO/E.

RELATED TASKS
"Compiling OO applications under UNIX"
"Preparing OO applications under UNIX" on page 288
"Running OO applications under UNIX" on page 289

Compiling OO applications under UNIX

When you compile OO applications in a UNIX shell, use the `cob2` command to compile COBOL client programs and class definitions, and the `javac` command to compile Java class definitions to produce *bytecode* (suffix .class).

To compile COBOL source code that contains OO syntax such as `INVOKE` statements or class definitions, or that uses Java services, you must use these compiler options: `RENT`, `DLL`, `THREAD`, and `DBCS`. (The `RENT` and `DBCS` options are defaults.)

A COBOL source file that contains a class definition must not contain any other class or program definitions.

When you compile a COBOL class definition, two output files are generated:

- The object file (.o) for the class definition.
- A Java source program (.java) that contains a class definition that corresponds to the COBOL class definition. Do not edit this generated Java class definition in any way. If you change the COBOL class definition, you must regenerate both the object file and the Java class definition by recompiling the updated COBOL class definition.

If a COBOL client program or class definition includes the file JNI.cpy by using a `COPY` statement, specify the `include` subdirectory of the COBOL install directory (typically `/usr/lpp/cobol/include`) in the search order for copybooks. You can specify the `include` subdirectory by using the `-I` option of the `cob2` command or by setting the SYSLIB environment variable.

RELATED TASKS
Chapter 15, "Compiling under UNIX," on page 279
"Preparing OO applications under UNIX"
"Running OO applications under UNIX" on page 289
"Setting and accessing environment variables" on page 424
"Accessing JNI services" on page 569

RELATED REFERENCES
"cob2 syntax and options" on page 283
"DBCS" on page 309
"DLL" on page 311
"RENT" on page 331
"THREAD" on page 342

Preparing OO applications under UNIX

Use the `cob2` command to link OO COBOL applications.

To prepare an OO COBOL client program for execution, link the object file with the following two DLL side files to create an executable module:

- libjvm.x, which is provided with your IBM Java 2 Software Development Kit.
- igzcjava.x, which is provided in the `lib` subdirectory of the `cobol` directory in the HFS. The typical complete path is `/usr/lpp/cobol/lib/igzcjava.x`. This DLL side file is also available as the member IGZCJAVA in the SCEELIB PDS (part of Language Environment).

To prepare a COBOL class definition for execution:

1. Link the object file using the two DLL side files discussed above to create an executable DLL module.

   You must name the resulting DLL module lib*Classname*.so, where *Classname* is the external class-name. If the class is part of a package and thus there are periods in the external class-name, you must change the periods to underscores in the DLL module name. For example, if class Account is part of the com.acme package, the external class-name (as defined in the `REPOSITORY` paragraph entry for the class) must be com.acme.Account, and the DLL module for the class must be libcom_acme_Account.so.
2. Compile the generated Java source with the Java compiler to create a class file (.class).

For a COBOL source file *Classname*.cbl that contains the class definition for *Classname*, you would use the following commands to compile and link the components of the application:

Table 44. **Commands for compiling and linking a class definition**

Command	Input	Output
`cob2 -c -qdll,thread` *`Classname`*`.cbl`	*Classname*.cbl	*Classname*.o, *Classname*.java
`cob2 -bdll -o lib`*`Classname`*`.so` *`Classname`*`.o /usr/lpp/java/IBM/J1.3/bin/classic/libjvm.x /usr/lpp/cobol/lib/igzcjava.x`	*Classname*.o	lib*Classname*.so
`javac` *`Classname`*`.java`	*Classname*.java	*Classname*.class

After you issue the `cob2` and `javac` commands successfully, you have the executable components for the program: the executable DLL module lib*Classname*.so and the class file *Classname*.class. All files from these commands are generated in the current working directory.

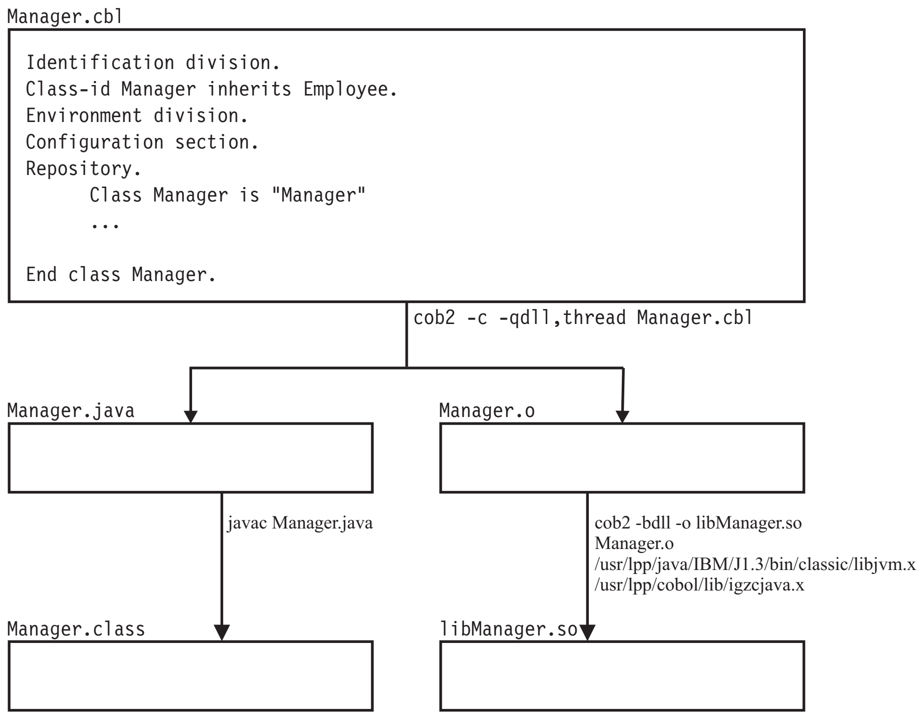
"Example: compiling and linking a COBOL class definition under UNIX"

RELATED TASKS
Chapter 15, "Compiling under UNIX," on page 279
"REPOSITORY paragraph for defining a class" on page 530

RELATED REFERENCES
"cob2 syntax and options" on page 283

Example: compiling and linking a COBOL class definition under UNIX

This example illustrates the commands that you use and the files that are produced when you compile and link a COBOL class definition, Manager.cbl, using UNIX shell commands.



```
Identification division.
Class-id Manager inherits Employee.
Environment division.
Configuration section.
Repository.
      Class Manager is "Manager"
      ...

End class Manager.
```

```
cob2 -c -qdll,thread Manager.cbl
```

```
javac Manager.java
```

```
cob2 -bdll -o libManager.so
Manager.o
/usr/lpp/java/IBM/J1.3/bin/classic/libjvm.x
/usr/lpp/cobol/lib/igzcjava.x
```

The class file Manager.class and the DLL module libManager.so are the executable components of the application, and are generated in the current working directory.

Running OO applications under UNIX

It is recommended that you run object-oriented COBOL applications as UNIX applications. You must do so if an application begins with a Java program or the `main` factory method of a COBOL class.

Specify the directory that contains the DLLs for the COBOL classes in the LIBPATH environment variable. Specify the directory paths for the Java class files that are associated with the COBOL classes in the CLASSPATH environment variable as follows:

- For classes that are not part of a package, end the class path with the directory that contains the .class files.
- For classes that are part of a package, end the class path with the directory that contains the "root" package (the first package in the full package name).
- For a .jar file that contains .class files, end the class path with the name of the .jar file.

Separate multiple path entries with colons.

RELATED TASKS
"Running OO applications that start with a main method"
"Running OO applications that start with a COBOL program" on page 291
"Running J2EE COBOL clients" on page 291
Chapter 23, "Running COBOL programs under UNIX," on page 423
"Setting and accessing environment variables" on page 424
Chapter 30, "Writing object-oriented programs," on page 525
"Structuring OO applications" on page 566

Running OO applications that start with a main method

If the first routine of a mixed COBOL and Java application is the `main` method of a Java class or the `main` factory method of a COBOL class, run the application by using the `java` command and by specifying the name of the class that contains the `main` method.

The `java` command initializes the Java virtual machine (JVM). To customize the initialization of the JVM, specify options on the `java` command as in the following examples:

Table 45. **java command options for customizing the JVM**

Purpose	Option
To set a system property	`-D`*`name`*`=`*`value`*
To request that the JVM generate verbose messages about garbage collection	`-verbose:gc`
To request that the JVM generate verbose messages about class loading	`-verbose:class`
To request that the JVM generate verbose messages about native methods and other Java Native Interface activity	`-verbose:jni`
To set the initial Java heap size to *value* bytes	`-Xms`*`value`*
To set the maximum Java heap size to *value* bytes	`-Xmx`*`value`*

See the output from the `java -h` command or the related references for details about the options that the JVM supports.

RELATED REFERENCES
JVM options (*Persistent Reusable Java Virtual Machine User's Guide*)
Java Naming and Directory Interface (JNDI) (*WebSphere for z/OS: Applications*)

Running OO applications that start with a COBOL program

If the first routine of a mixed COBOL and Java application is a COBOL program, run the application by specifying the program name at the command prompt. If a JVM is not already running in the process of the COBOL program, the COBOL run time automatically initializes a JVM.

To customize the initialization of the JVM, specify options by setting the COBJVMINITOPTIONS environment variable. Use blanks to separate options. For example:

```
export COBJVMINITOPTIONS="-Xms10000000 -Xmx20000000 -verbose:gc"
```

RELATED TASKS
"Using IBM SDK for z/OS, Java 2 Technology Edition, V1.4" on page 295
Chapter 23, "Running COBOL programs under UNIX," on page 423
"Setting and accessing environment variables" on page 424

RELATED REFERENCES
JVM options (*Persistent Reusable Java Virtual Machine User's Guide*)
Java Naming and Directory Interface (JNDI) (*WebSphere for z/OS: Applications*)

Running J2EE COBOL clients: You can use OO syntax in a COBOL program to implement a Java 2 Platform, Enterprise Edition (J2EE) client. You can, for example, invoke methods on enterprise beans that run in the WebSphere(R) for z/OS environment.

Before you run a COBOL J2EE client, you must set the Java system property java.naming.factory.initial to access WebSphere naming services. For example:

```
export COBJVMINITOPTIONS
="-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory"
```

"Example: J2EE client written in COBOL" on page 580

Compiling, linking, and running OO applications using JCL or TSO/E

It is recommended that you compile, link, and run applications that use OO syntax in the z/OS UNIX environment.

However, in limited circumstances it is possible to compile, prepare, and run OO applications by using standard batch JCL or TSO/E commands. To do so, you must follow the guidelines that are in the related tasks. For example, you might follow this approach for applications that consist of a COBOL main program and subprograms that:

- Access objects that are all implemented in Java
- Access enterprise beans that run in a WebSphere server

RELATED TASKS
"Compiling OO applications using JCL or TSO/E"
"Preparing and running OO applications using JCL or TSO/E" on page 292
"Compiling, linking, and running OO applications under UNIX" on page 287

Compiling OO applications using JCL or TSO/E

If you use batch JCL or TSO/E to compile an OO COBOL program or class definition, the generated object file is written, as usual, to the data set that has ddname `SYSLIN` or `SYSPUNCH`. You must use compiler options `RENT`, `DLL`, `THREAD`, and `DBCS`. (`RENT` and `DBCS` are defaults.)

If the COBOL program or class definition uses the JNI environment structure to access JNI callable services, copy the file JNI.cpy from the HFS to a PDS or PDSE member called JNI, identify that library with a `SYSLIB DD` statement, and use a `COPY` statement of the form `COPY JNI` in the COBOL source.

A COBOL source file that contains a class definition must not contain any other class or program definitions.

When you compile a COBOL class definition, a Java source program that contains a class definition that corresponds to the COBOL class definition is generated in addition to the object file. Use the `SYSJAVA` ddname to write the generated Java source file to a file in the HFS. For example:

```
//SYSJAVA DD PATH='/u/userid/java/Classname.java',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU,
// FILEDATA=TEXT
```

Do not edit this generated Java class definition in any way. If you change the COBOL class definition, you must regenerate both the object file and the Java class definition by recompiling the updated COBOL class definition.

Compile Java class definitions by using the `javac` command from a UNIX shell command prompt, or by using the BPXBATCH utility.

"Example: compiling, linking, and running an OO application using JCL" on page 294

RELATED TASKS
"Compiling with JCL" on page 247
"Compiling under TSO" on page 259
"Specifying source libraries (SYSLIB)" on page 265
"Defining the Java-source output file (SYSJAVA)" on page 266
"Accessing JNI services" on page 569
"Compiling OO applications under UNIX" on page 287
"Preparing OO applications under UNIX" on page 288

RELATED REFERENCES
"DBCS" on page 309
"DLL" on page 311
"RENT" on page 331
"THREAD" on page 342
Appendix F, "JNI.cpy," on page 693
The BPXBATCH utility (*UNIX System Services User's Guide*)

Preparing and running OO applications using JCL or TSO/E

It is recommended that you run OO applications in a z/OS UNIX environment. To run OO applications from batch JCL or TSO/E, you should therefore use the BPXBATCH utility.

In limited circumstances, however, you can run an OO application by using standard batch JCL (`EXEC PGM=`*`COBPROG`*) or the TSO/E `CALL` command. To do so, follow these requirements when preparing the application:

- Structure the application to start with a COBOL program. (If an application starts with a Java program or with the `main` factory method of a COBOL class, you must run the application under UNIX, and the application components must reside in the HFS.)

- **Link-edit considerations:** Link the load module for the COBOL program into a
+ PDSE. COBOL programs that contain object-oriented syntax must be link-edited
+ with `AMODE 31`.
- Ensure that the class files and DLLs associated with the COBOL or Java classes that are used by the application reside in the HFS. You must name the class files and DLLs as described in the related task about preparing OO applications under UNIX.
- Specify `INCLUDE` control statements for the DLL side files libjvm.x and igzcjava.x when you bind the object deck for the main program. For example:

  ```
  INCLUDE '/usr/lpp/java/IBM/J1.3/bin/classic/libjvm.x'
  INCLUDE '/usr/lpp/cobol/lib/igzcjava.x'
  ```
- Create a file that contains the environment variable settings that are required for Java. For example, a file /u/*userid*/javaenv might contain the following three lines to set the PATH, LIBPATH, and CLASSPATH environment variables (the LIBPATH setting is shown on two lines because of document length limitations, but you must specify the setting on one unbroken line that has no internal blanks):

  ```
  PATH=/bin:/usr/lpp/java/IBM/J1.3/bin
  LIBPATH=/lib:/usr/lib:/usr/lpp/java/IBM/J1.3/bin:
     /usr/lpp/java/IBM/J1.3/bin/classic:/u/userid/applications
  CLASSPATH=/u/userid/applications
  ```

  To customize the initialization of the JVM that will be used by the application, you can set the COBJVMINITOPTIONS environment variable in the same file. For example, to access enterprise beans that run in a WebSphere server, you must set the Java system property java.naming.factory.initial. For details, see the related task about running OO applications under UNIX.

When you run an OO application that starts with a COBOL program by using standard batch JCL or the TSO/E `CALL` command, follow these guidelines:

- Use the _CEE_ENVFILE environment variable to indicate the location of the file that contains the environment variable settings required by Java. Set _CEE_ENVFILE by using the `ENVAR` runtime option.
- Specify the `POSIX(ON)` runtime option.
- Use `DD` statements to specify files in the HFS for the standard input, output, and error streams for Java:
  - `JAVAIN DD` for the input from statements such as `c=System.in.read();`
  - `JAVAOUT DD` for the output from statements such as `System.out.println(`*string*`);`
  - `JAVAERR DD` for the output from statements such as `System.err.println(`*string*`);`
- Ensure that the SCEERUN2 and SCEERUN load libraries are available in the system library search order, for example, by using a `STEPLIB DD` statement.

"Example: compiling, linking, and running an OO application using JCL" on page 294

RELATED TASKS
"Preparing OO applications under UNIX" on page 288
"Running OO applications under UNIX" on page 289
"Structuring OO applications" on page 566
The BPXBATCH utility (*UNIX System Services User's Guide*)
Running an application under batch (*Language Environment Programming Guide*)

RELATED REFERENCES
_CEE_ENVFILE (*C/C++ Programming Guide*)
ENVAR (*Language Environment Programming Reference*)

Example: compiling, linking, and running an OO application using JCL

This example shows the JCL that you could use to compile, link, and run a COBOL client that invokes a Java method.

The example shows:

- The JCL to compile, link, and run an OO COBOL program, TSTHELLO
- A Java class definition, HelloJ, that contains a method that the COBOL program invokes
- An HFS file, ENV, that contains the environment variable settings that Java requires

JCL for program TSTHELLO

```
//TSTHELLO JOB ,
//  TIME=(1),MSGLEVEL=(1,1),MSGCLASS=H,CLASS=A,REGION=100M,
//  NOTIFY=&SYSUID,USER=&SYSUID
//*
// SET COBPRFX='IGY.V3R4M0'
// SET LIBPRFX='CEE'
//*
//COMPILE EXEC PGM=IGYCRCTL,
// PARM='SIZE(5000K)'
//SYSLIN   DD DSNAME=&&OBJECT(TSTHELLO),UNIT=VIO,DISP=(NEW,PASS),
//            SPACE=(CYL,(1,1,1))
//SYSPRINT DD SYSOUT=*
//STEPLIB  DD DSN=&COBPRFX..SIGYCOMP,DISP=SHR
//         DD DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSUT1   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT2   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT3   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT4   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT5   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT6   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT7   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSIN    DD *
       cbl dll,thread
       Identification division.
       Program-id. "TSTHELLO" recursive.
       Environment division.
       Configuration section.
       Repository.
           Class HelloJ is "HelloJ".
       Data Division.
       Procedure division.
           Display "COBOL program TSTHELLO entered"
           Invoke HelloJ "sayHello"
           Display "Returned from java sayHello to TSTHELLO"
           Goback.
       End program "TSTHELLO".
/*
//LKED EXEC PGM=IEWL,PARM='RENT,LIST,LET,DYNAM(DLL),CASE(MIXED)'
//SYSLIB   DD DSN=&LIBPRFX..SCEELKED,DISP=SHR
//         DD DSN=&LIBPRFX..SCEELKEX,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//SYSLMOD  DD DSN=&&GOSET(TSTHELLO),DISP=(MOD,PASS),UNIT=VIO,
//            SPACE=(CYL,(1,1,1)),DSNTYPE=LIBRARY
//SYSDEFSD DD DUMMY
```

```
//OBJMOD   DD DSN=&&OBJECT,DISP=(OLD,DELETE)
//SYSLIN   DD *
  INCLUDE OBJMOD(TSTHELLO)
  INCLUDE '/usr/lpp/java/IBM/J1.3/bin/classic/libjvm.x'
  INCLUDE '/usr/lpp/cobol/lib/igzcjava.x'
/*
//GO EXEC PGM=TSTHELLO,COND=(4,LT,LKED),
//        PARM='/ENVAR("_CEE_ENVFILE=/u/userid/ootest/tsthello/ENV")
//            POSIX(ON)'
//STEPLIB  DD DSN=*.LKED.SYSLMOD,DISP=SHR
//         DD DSN=&LIBPRFX..SCEERUN2,DISP=SHR
//         DD DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSOUT   DD SYSOUT=*
//CEEDUMP  DD SYSOUT=*
//SYSUDUMP DD DUMMY
//JAVAOUT  DD PATH='/u/userid/ootest/tsthello/javaout',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=(SIRUSR,SIWUSR,SIRGRP)
```

Definition of class HelloJ

```
class HelloJ {
   public static void sayHello() {
     System.out.println("Hello World, from Java!");
   }
}
```

HelloJ.java is compiled with the `javac` command. The resulting .class file resides in the HFS directory u/*userid*/ootest/tsthello, which is specified in the CLASSPATH environment variable in the environment variable settings file.

Environment variable settings file, ENV

```
PATH=/bin:/usr/lpp/java/IBM/J1.3/bin:.
LIBPATH=/lib:/usr/lib:/usr/lpp/java/IBM/J1.3/bin:
   /usr/lpp/java/IBM/J1.3/bin/classic:/u/userid/ootest/tsthello
CLASSPATH=/u/userid/ootest/tsthello
```

(The LIBPATH setting is shown above on two lines because of document length limitations, but you must specify the setting on one unbroken line that has no internal blanks.)

The environment variable settings file also resides in directory u/*userid*/ootest/tsthello, as specified in the _CEE_ENVFILE environment variable in the JCL.

Using IBM SDK for z/OS, Java 2 Technology Edition, V1.4

| The IBM SDK for z/OS, Java 2 Technology Edition, V1.4 is based on the XPLINK linkage convention defined by Language Environment.

| In most cases, using the SDK 1.4 for OO COBOL applications is identical to using previous SDKs:

- | Building COBOL and Java applications is unchanged.
- | Running COBOL and Java applications is unchanged if the application starts with a Java program or the `main` factory method of a COBOL class.

| If the application starts with a Java program or the `main` factory method of a COBOL class, the XPLINK environment is automatically started by the `java` command that starts the JVM and runs the application.

If an application starts with a COBOL program that invokes methods on COBOL or Java classes, you must specify the `XPLINK(ON)` runtime option so that the XPLINK environment is initialized. `XPLINK(ON)` is not recommended as a default setting, however; you should use `XPLINK(ON)` only for applications that specifically require it.

When you are running an application under z/OS UNIX, you can set the `XPLINK(ON)` option by using the _CEE_RUNOPTS environment variable as follows:

```
_CEE_RUNOPTS="XPLINK(ON)"
```

Exporting `_CEE_RUNOPTS="XPLINK(ON)"` so that it is in effect for the entire z/OS UNIX shell session is not recommended, however. Suppose for example that an OO COBOL application starts with a COBOL program called App1Driver. One way to limit the effect of the `XPLINK` option to the execution of the App1Driver application is to set the _CEE_RUNOPTS variable on the command-line invocation of App1Driver as follows:

```
_CEE_RUNOPTS="XPLINK(ON)" App1Driver
```

RELATED TASKS
"Running OO applications under UNIX" on page 289
"Setting and accessing environment variables" on page 424

RELATED REFERENCES
"Runtime environment variables" on page 425
XPLINK (*Language Environment Programming Reference*)
_CEE_RUNOPTS (*C/C++ Programming Guide*)

Chapter 17. Compiler options

You can direct and control your compilation by using compiler options or by using compiler-directing statements (compiler directives).

Compiler options affect the aspects of your program that are listed in the table below. The linked-to information for each option provides the syntax for specifying the option and describes the option, its parameters, and its interaction with other parameters.

Table 46. **Compiler options**

Aspect of your program	Compiler option	Default	Abbreviations
Source language	"ARITH" on page 302	`ARITH(COMPAT)`	`AR(C|E)`
	"CICS" on page 303	`NOCICS`	None
	"CODEPAGE" on page 305	`CODEPAGE(01140)`	`CP(`*ccsid*`)`
	"CURRENCY" on page 306	`NOCURRENCY`	`CURR|NOCURR`
	"DBCS" on page 309	`DBCS`	None
	"LIB" on page 319	`LIB`	None
	"NSYMBOL" on page 323	`NSYMBOL(NATIONAL)`	`NS(DBCS|NAT)`
	"NUMBER" on page 324	`NONUMBER`	`NUM|NONUM`
	"QUOTE/APOST" on page 331	`QUOTE`	`Q|APOST`
	"SEQUENCE" on page 333	`SEQUENCE`	`SEQ|NOSEQ`
	"SQL" on page 335	`NOSQL`	None
+	"SQLCCSID" on page 337	`SQLCCSID`	`SQLC|NOSQLC`
	"WORD" on page 346	`NOWORD`	`WD|NOWD`
Date processing	"DATEPROC" on page 308	`NODATEPROC`, or `DATEPROC(FLAG,NOTRIG)` if only `DATEPROC` is specified	`DP|NODP`
	"INTDATE" on page 317	`INTDATE(ANSI)`	None
	"YEARWINDOW" on page 348	`YEARWINDOW(1900)`	`YW`
Maps and listings	"LANGUAGE" on page 318	`LANGUAGE(ENGLISH)`	`LANG(EN|UE|JA|JP)`
	"LINECOUNT" on page 319	`LINECOUNT(60)`	`LC`
	"LIST" on page 319	`NOLIST`	None
	"MAP" on page 320	`NOMAP`	None
	"OFFSET" on page 326	`NOOFFSET`	`OFF|NOOFF`
	"SOURCE" on page 334	`SOURCE`	`S|NOS`
	"SPACE" on page 335	`SPACE(1)`	None
	"TERMINAL" on page 338	`NOTERMINAL`	`TERM|NOTERM`
	"VBREF" on page 346	`NOVBREF`	None
	"XREF" on page 347	`XREF(FULL)`	`X|NOX`

Table 46. **Compiler options** *(continued)*

Aspect of your program	Compiler option	Default	Abbreviations
Object deck generation	"COMPILE" on page 305	NOCOMPILE(S)	C|NOC
	"DECK" on page 310	NODECK	D|NOD
	"NAME" on page 323	NONAME, or NAME(NOALIAS) if only NAME is specified	None
	"OBJECT" on page 326	OBJECT	OBJ|NOOBJ
	"PGMNAME" on page 329	PGMNAME(COMPAT)	PGMN(CO|LU|LM)
Object code control	"ADV" on page 301	ADV	None
	"AWO" on page 302	NOAWO	None
	"DLL" on page 311	NODLL	None
	"EXPORTALL" on page 313	NOEXPORTALL	EXP|NOEXP
	"FASTSRT" on page 314	NOFASTSRT	FSRT|NOFSRT
	"NUMPROC" on page 325	NUMPROC(NOPFD)	None
	"OPTIMIZE" on page 327	NOOPTIMIZE	OPT|NOOPT
	"OUTDD" on page 328	OUTDD(SYSOUT)	OUT
	"TRUNC" on page 343	TRUNC(STD)	None
	"ZWB" on page 349	ZWB	None
Virtual storage usage	"BUFSIZE" on page 303	4096	BUF
	"SIZE" on page 334	SIZE(MAX)	SZ
	"DATA" on page 307	DATA(31)	None
	"DYNAM" on page 313	NODYNAM	DYN|NODYN
	"RENT" on page 331	RENT	None
	"RMODE" on page 332	AUTO	None
Debugging and diagnostics	"DIAGTRUNC" on page 310	NODIAGTRUNC	DTR|NODTR
	"DUMP" on page 311	NODUMP	DU|NODU
	"FLAG" on page 314	FLAG(I,I)	F|NOF
	"FLAGSTD" on page 316	NOFLAGSTD	None
	"TEST" on page 338	NOTEST	None
	"SSRANGE" on page 337	NOSSRANGE	SSR|NOSSR
Other	"ADATA" on page 301	NOADATA	None
	"EXIT" on page 313	NOEXIT	EX(INX,LIBX,PRTX,ADX)
|	"MDECK" on page 321	NOMDECK	NOMD|MD|MD(C)|MD(NOC)
	"THREAD" on page 342	NOTHREAD	None

Installation defaults: 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. (In some installations, certain compiler options are set up as fixed so that you cannot override them. If you have problems, see your system administrator.) 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.

Nonoverridable 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 `DYNAM`, `FASTSRT`, `NUMPROC`, `OPTIMIZE`, `RENT`, `SSRANGE`, `TEST`, and `TRUNC` compiler options can all affect runtime performance.

RELATED REFERENCES
"Conflicting compiler options" on page 300
Chapter 18, "Compiler-directing statements," on page 351
"Option settings for Standard COBOL 85 conformance"

RELATED TASKS
Chapter 14, "Compiling under z/OS," on page 247
"Compiling under TSO" on page 259
Chapter 15, "Compiling under UNIX," on page 279
Chapter 34, "Tuning your program," on page 623

Option settings for Standard COBOL 85 conformance

Compiler options and runtime options are required for conformance with Standard COBOL 85.

The following compiler options are required:
- `ADV`
- `NOCICS`
- `NODATEPROC`
- `NODLL`
- `DYNAM`
- `NOEXPORTALL`
- `NOFASTSRT`
- `LIB`
- `NAME(ALIAS)` or `NAME(NOALIAS)`
- `NUMPROC(NOPFD)` or `NUMPROC(MIG)`
- `PGMNAME(COMPAT)` or `PGMNAME(LONGUPPER)`
- `QUOTE`
- `NOTHREAD`
- `TRUNC(STD)`
- `NOWORD`
- `ZWB`

The following runtime options are required:
- `AIXBLD`
- `CBLQDA(ON)`
- `TRAP(ON)`

RELATED REFERENCES
Language Environment Programming Reference

Conflicting compiler options

The Enterprise COBOL compiler can encounter conflicting compiler options in either of two ways: both the positive and negative form of an option are specified at the same level in the hierarchy of precedence, or mutually exclusive options are specified at the same level in the hierarchy.

When conflicting options are specified at the same level in the hierarchy (such as specifying both DECK and NODECK in a PROCESS or CBL statement), the option specified last takes effect.

If you specify mutually exclusive compiler options at the same level, the compiler generates an error message and forces one of the options to a nonconflicting value. For example, if you specify both OFFSET and LIST in a PROCESS statement in any order, OFFSET takes effect and LIST is ignored.

However, options coded at a higher level of precedence override any options specified at a lower level of precedence. For example, if you code OFFSET in a JCL statement but LIST in a PROCESS statement, LIST takes effect because the options coded in the PROCESS statement and any options forced on by an option coded in the PROCESS statement have higher precedence.

Table 47. **Mutually exclusive compiler options**

Specified	Ignored[1]	Forced on[1]
CICS	NOLIB	LIB
	DYNAM	NODYNAM
	NORENT	RENT
DLL	DYNAM	NODYNAM
	NORENT	RENT
EXIT	DUMP	NODUMP
EXPORTALL	NODLL	DLL
	DYNAM	NODYNAM
	NORENT	RENT
MDECK	NOLIB	LIB
NSYMBOL(NATIONAL)	NODBCS	DBCS
OFFSET	LIST	NOLIST
SQL	NOLIB	LIB

TEST
TEST(ALL)
TEST(STMT)
TEST(PATH)
TEST(BLOCK)
	NOOBJECT	OBJECT

	OPT(STD) or OPT(FULL)	NOOPTIMIZE
THREAD	NORENT	RENT
WORD	FLAGSTD	NOFLAGSTD
1. Unless in conflict with a fixed installation default option.		

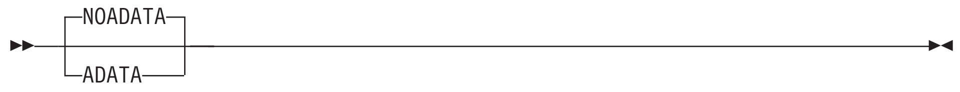
RELATED TASKS
"Specifying compiler options under z/OS" on page 267

"Specifying compiler options in a batch compilation" on page 272
"Specifying compiler options under UNIX" on page 280

ADATA

Use ADATA when you want the compiler to create a SYSADATA file that contains records of additional compilation information.

ADATA option syntax



Default is: NOADATA

Abbreviations are: None

ADATA is required for remote compilation using an IBM Windows COBOL compiler. On z/OS, the SYSADATA file is file is written to ddname SYSADATA. The size of the SYSADATA file generally grows with the size of the associated program.

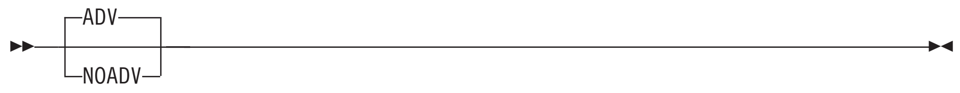
You cannot specify ADATA in a PROCESS (CBL) statement. You can specify it only in one of the following ways:
- In the PARM parameter of JCL
- As a cob2 command option
- As an installation default
- In the COBOPT environment variable

RELATED REFERENCES
Appendix G, "COBOL SYSADATA file contents," on page 699
"Setting environment variables under UNIX" on page 279
"cob2 syntax and options" on page 283

ADV

ADV has meaning only if you use WRITE . . . ADVANCING in your source code. With ADV in effect, the compiler adds 1 byte to the record length to account for the printer control character.

ADV option syntax



Default is: ADV

Abbreviations are: None

Use NOADV if you already adjusted record length to include 1 byte for the printer control character.

ARITH

ARITH affects the maximum number of digits that you can code for integers, and the number of digits used in fixed-point intermediate results.

ARITH option syntax

```
           ┌─COMPAT─┐
►►──ARITH(─┴─EXTEND─┴─)──────────────────────────────────────────►◄
```

Default is: ARITH(COMPAT)

Abbreviations are: AR(C), AR(E)

When you specify ARITH(EXTEND):

- The maximum number of digit positions that you can specify in the PICTURE clause for packed-decimal, external-decimal, and numeric-edited data items is raised from 18 to 31.
- The maximum number of digits that you can specify in a fixed-point numeric literal is raised from 18 to 31. You can use numeric literals with large precision anywhere that numeric literals are currently allowed, including:
  - Operands of PROCEDURE DIVISION statements
  - VALUE clauses (for numeric data items with large-precision PICTURE)
  - Condition-name values (on numeric data items with large-precision PICTURE)
- The maximum number of digits that you can specify in the arguments to NUMVAL and NUMVAL-C is raised from 18 to 31.
- The maximum value of the integer argument to the FACTORIAL function is 29.
- Intermediate results in arithmetic statements use *extended mode*.

When you specify ARITH(COMPAT):

- The maximum number of digit positions in the PICTURE clause for packed-decimal, external-decimal, and numeric-edited data items is 18.
- The maximum number of digits in a fixed-point numeric literal is 18.
- The maximum number of digits in the arguments to NUMVAL and NUMVAL-C is 18.
- The maximum value of the integer argument to the FACTORIAL function is 28.
- Intermediate results in arithmetic statements use *compatibility mode*.

RELATED CONCEPTS
Appendix A, “Intermediate results and arithmetic precision,” on page 647

AWO

If you specify AWO, an implicit APPLY WRITE-ONLY clause is activated for all files in the program that are eligible for this clause. To be eligible, a file must have physical sequential organization and blocked variable-length records.

AWO option syntax

```
   ┌─NOAWO─┐
►►─┴─AWO───┴──────────────────────────────────────────────────────────►◄
```

Default is: NOAWO

Abbreviations are: None

RELATED TASKS
"Optimizing buffer and device space" on page 12

BUFSIZE

Use BUFSIZE to allocate an amount of main storage to the buffer for each compiler work data set. Usually, a large buffer size improves the performance of the compiler.

BUFSIZE option syntax

```
              ┌─nnnnn─┐
►►──BUFSIZE(──┴─nnnK──┴──)────────────────────────────────────────────►◄
```

Default is: 4096

Abbreviations are: BUF

nnnnn specifies a decimal number that must be at least 256.

*nnn*K specifies a decimal number in 1-KB increments, where 1 KB = 1024 bytes.

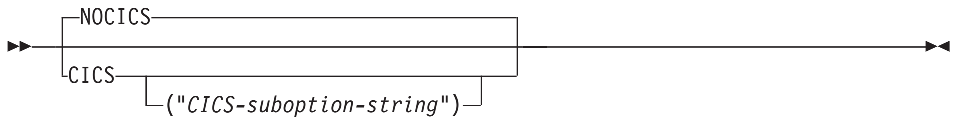
If you use both BUFSIZE and SIZE, the amount allocated to buffers is included in the amount of main storage available for compilation via the SIZE option.

BUFSIZE cannot exceed the track capacity for the device used, nor can it exceed the maximum allowed by data management services.

CICS

The CICS compiler option enables the integrated CICS translator and allows specification of CICS suboptions. You must use the CICS option if your COBOL source program contains EXEC CICS or EXEC DLI statements and the program has not been processed by the separate CICS translator.

CICS option syntax



Default is: NOCICS

Abbreviations are: None

Use the CICS option to compile CICS programs only. Programs compiled with the CICS option will not run in a non-CICS environment.

If you specify the CICS option, the compiler needs access to CICS Transaction Server Version 2 or later.

If you specify the NOCICS option, any CICS statements found in the source program are diagnosed and discarded.

Use either quotation marks or single quotation marks to delimit the string of CICS suboptions.

You can partition a long suboption string into multiple suboption strings on multiple CBL statements. The CICS suboptions are concatenated in the order of their appearance. For example:

```
//STEP1 EXEC IGYWC, . . .
// PARM.COBOL='CICS("string1")'
//COBOL.SYSIN DD *
       CBL CICS('string2')
       CBL CICS("string3")
       IDENTIFICATION DIVISION.
       PROGRAM-ID. DRIVER1.
       . . .
```

The compiler passes the following suboption string to the integrated CICS translator:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces as shown. If multiple instances of the same CICS option are found, the last specification of each option prevails. The compiler limits the length of the concatenated CICS suboptions string to 4 KB.

RELATED CONCEPTS
"Integrated CICS translator" on page 399

RELATED TASKS
"Compiling with the CICS option" on page 397
"Separating CICS suboptions" on page 399

RELATED REFERENCES
"Conflicting compiler options" on page 300

CODEPAGE

`CODEPAGE` specifies the code page to be used for encoding the contents of alphanumeric and DBCS data items at run time, and for encoding alphanumeric, national, and DBCS literals in the COBOL source program. `CODEPAGE` also specifies the default code page for parsing alphanumeric XML documents.

(The runtime code page used for national literals and data items that have `USAGE NATIONAL` is UTF-16BE (big endian), CCSID 1200.)

CODEPAGE option syntax

```
►►──CODEPAGE(ccsid)──────────────────────────────────────────────►◄
```

Default is: `CODEPAGE(1140)`

Abbreviations are: `CP(`*ccsid*`)`

ccsid must be an integer that represents a valid coded character set identifier (CCSID) for an EBCDIC code page.

The default CCSID of 1140 is an equivalent of CCSID 37 (EBCDIC Latin-1, USA), but includes the euro symbol.

Important: The conversion image that was configured as part of the Unicode Conversion Services installation must include support for conversions of the specified CCSID to and from CCSID 1200. For applications that use object-oriented syntax, conversions from the specified CCSID to CCSID 1208 must also be included.

RELATED CONCEPTS
"COBOL and DB2 CCSID determination" on page 411

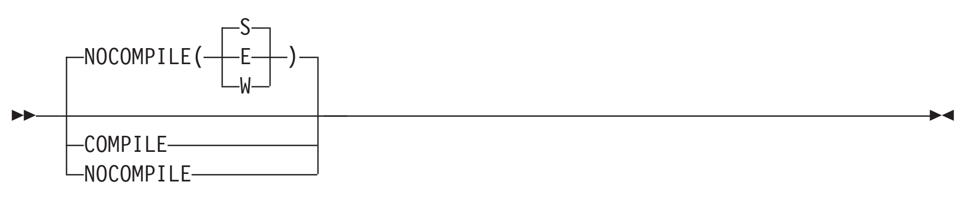
RELATED TASKS
Customizing Unicode support for COBOL (*Enterprise COBOL Customization Guide*)

RELATED REFERENCES
"SQLCCSID" on page 337

COMPILE

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 if the compilation resulted in serious errors: the results could be unpredictable or an abnormal termination could occur.

COMPILE option syntax



Default is: `NOCOMPILE(S)`

Abbreviations are: `C|NOC`

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. 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.

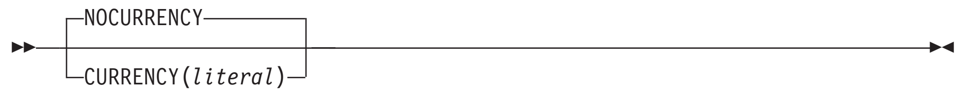
If you request an unconditional `NOCOMPILE`, the following options have no effect because no object code will be produced:
- `LIST`
- `SSRANGE`
- `OPTIMIZE`
- `OBJECT`
- `TEST`

RELATED REFERENCES
"Messages and listings for compiler-detected errors" on page 275

CURRENCY

You can use the `CURRENCY` option to provide an alternate default currency symbol to be used for a COBOL program. (The default currency symbol is the dollar sign ($).)

CURRENCY option syntax



Default is: `NOCURRENCY`

Abbreviations are: `CURR|NOCURR`

`NOCURRENCY` specifies that no alternate default currency symbol will be used.

To change the default currency symbol, specify CURRENCY(*literal*), where *literal* is a valid COBOL alphanumeric literal (optionally a hexadecimal literal) that represents a single character. The literal must not be from the following list:

- Digits zero (0) through nine (9)
- Uppercase alphabetic characters A B C D E G N P R S V X Z or their lowercase equivalents
- The space
- Special characters * + - / , . ; ( ) " = '
- A figurative constant
- A null-terminated literal
- A DBCS literal
- A national literal

If your program processes only one currency type, you can use the CURRENCY option as an alternative to the CURRENCY SIGN clause for indicating the currency symbol you will use in the PICTURE clause of your program. If your program processes more than one currency type, you should use the CURRENCY SIGN clause with the WITH PICTURE SYMBOL phrase to specify the different currency sign types.

If you use both the CURRENCY option and the CURRENCY SIGN clause in a program, the CURRENCY option is ignored. Currency symbols specified in the CURRENCY SIGN clause or clauses can be used in PICTURE clauses.

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: You can delimit the CURRENCY option literal with either quotation marks or single quotation marks, regardless of the QUOTE|APOST compiler option setting.

DATA

The DATA option affects whether storage for dynamic data areas and other dynamic runtime storage is obtained from above or below the 16-MB line.

DATA option syntax

```
              ┌─31─┐
►►── DATA(──┴─24─┴──)─────────────────────────────────────────►◄
```

Default is: DATA(31)

Abbreviations are: None

For reentrant programs, the DATA compiler option and the HEAP runtime option control whether storage for dynamic data areas (such as WORKING-STORAGE and FD record areas) is obtained from below the 16-MB line (DATA(24)) or from unrestricted storage (DATA(31)). (DATA does not affect the location of LOCAL-STORAGE data; the STACK runtime option controls that location instead, along with the AMODE of the program.)

When you specify the runtime option `HEAP(,,BELOW)`, the `DATA` compiler option has no effect; the storage for all dynamic data areas is allocated from below the 16-MB line. However, if `HEAP(,,ANYWHERE)` is in effect, storage for dynamic data areas is allocated from below the line if you compiled the program with `DATA(24)` or from unrestricted storage if you compiled with `DATA(31)`.

Specify `DATA(24)` for programs that run in 31-bit addressing mode and that pass data arguments to programs in 24-bit addressing mode. Doing so ensures that the data will be addressable by the called program.

External data and QSAM buffers: The `DATA` option interacts with other compiler options and runtime options that affect storage and its addressability. See the related information for details.

RELATED CONCEPTS
"Storage and its addressability" on page 41

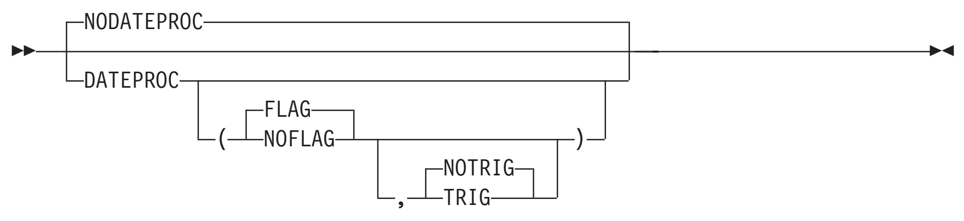
RELATED TASKS
Using run-time options (*Language Environment Programming Guide)*

RELATED REFERENCES
"Allocation of buffers for QSAM files" on page 172

DATEPROC

Use the `DATEPROC` option to enable the millennium language extensions of the COBOL compiler.

DATEPROC option syntax



Default is: `NODATEPROC`, or `DATEPROC(FLAG,NOTRIG)` if only `DATEPROC` is specified

Abbreviations are: `DP|NODP`

`DATEPROC(FLAG)`
With `DATEPROC(FLAG)`, the millennium language extensions are enabled, and the compiler produces a diagnostic message wherever a language element uses or is affected by the extensions. The message is usually an information-level or warning-level message that identifies statements that involve date-sensitive processing. Additional messages that identify errors or possible inconsistencies in the date constructs might be generated.

Production of diagnostic messages, 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 does not produce any related messages unless there are errors or inconsistencies in the COBOL source.

DATEPROC(TRIG)
With DATEPROC(TRIG), the millennium language extensions are enabled, and the automatic windowing that the compiler applies to operations on windowed date fields is sensitive to specific trigger or limit values in the date fields and in other nondate fields that are stored into or compared with the windowed date fields. These special values represent invalid dates that can be tested for or used as upper or lower limits.

Performance considerations: The DATEPROC(TRIG) option results in slower-performing code for windowed date comparisons.

DATEPROC(NOTRIG)
With DATEPROC(NOTRIG), the millennium language extensions are enabled, and the automatic windowing that the compiler applies to operations on windowed dates does not recognize any special trigger values in the operands. Only the value of the year part of dates is relevant to automatic windowing.

Performance considerations: The DATEPROC(NOTRIG) option is a performance option that assumes valid date values in windowed date fields.

NODATEPROC
NODATEPROC indicates that the extensions are not enabled for this compilation unit. This option 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.

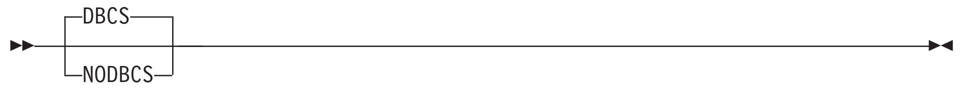
Usage note: You can specify the FLAG|NOFLAG and TRIG|NOTRIG suboptions in any order. If you omit either suboption, it defaults to the current setting.

RELATED REFERENCES
"FLAG" on page 314

DBCS

Using DBCS causes the compiler to recognize X'0E' (SO) and X'0F' (SI) as shift codes for the double-byte portion of an alphanumeric literal.

DBCS option syntax



Default is: DBCS

Abbreviations are: None

With DBCS in effect, the double-byte portion of the literal is syntax-checked and the literal remains category alphanumeric.

RELATED REFERENCES
"Conflicting compiler options" on page 300

DECK

Use DECK to produce object code in the form of 80-column card images. If you use the DECK option, be certain that SYSPUNCH is defined in your JCL for compilation.

DECK option syntax

```
      ┌─NODECK─┐
►►────┴─DECK───┴──────────────────────────────────────────────►◄
```

Default is: NODECK

Abbreviations are: D|NOD

DIAGTRUNC

DIAGTRUNC causes the compiler to issue a severity-4 (Warning) diagnostic message for MOVE statements with numeric receivers when the receiving data item has fewer integer positions than the sending data item or literal. In statements with multiple receivers, the message is issued separately for each receiver that could be truncated.

DIAGTRUNC option syntax

```
      ┌─NODIAGTRUNC─┐
►►────┴─DIAGTRUNC───┴─────────────────────────────────────────►◄
```

Default is: NODIAGTRUNC

Abbreviations are: DTR, NODTR

The diagnostic message is also issued for implicit moves associated with statements such as these:
- INITIALIZE
- READ . . . INTO
- RELEASE . . . FROM
- RETURN . . . INTO
- REWRITE . . . FROM
- WRITE . . . FROM

The diagnostic is also issued for moves to numeric receivers from alphanumeric data-names or literal senders, except when the sending field is reference modified.

There is no diagnostic for `COMP-5` receivers, nor for binary receivers when you specify the `TRUNC(BIN)` option.

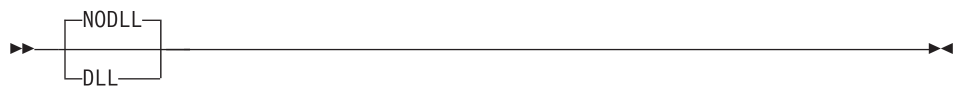
RELATED CONCEPTS
"Formats for numeric data" on page 49
"Reference modifiers" on page 109

RELATED REFERENCES
"TRUNC" on page 343

DLL

Use `DLL` to instruct the compiler to generate an object module that is enabled for dynamic link library (DLL) support. DLL enablement is required if the program will be part of a DLL, will reference DLLs, or if the program contains object-oriented COBOL syntax such as `INVOKE` statements or class definitions.

DLL option syntax



Default is: `NODLL`

Abbreviations are: None

Link-edit considerations: COBOL programs that are compiled with the `DLL` option must be link-edited with the `RENT` and `AMODE(31)` link-edit options.

`NODLL` instructs the compiler to generate an object module that is not enabled for DLL usage.

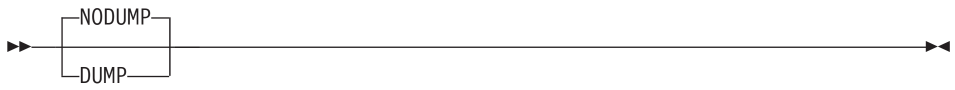
RELATED TASKS
"Making dynamic calls" on page 437

RELATED REFERENCES
"Conflicting compiler options" on page 300

DUMP

Use `DUMP` to produce a system dump at compile time for an internal compiler error.

DUMP option syntax



Default is: NODUMP

Abbreviations are: DU|NODU

Not for general use: The DUMP option should be used only at the request of an IBM representative.

The dump, which consists of a listing of the compiler's registers and a storage dump, is intended primarily for diagnostic personnel for determining errors in the compiler.

If you use the DUMP option, include a DD statement at compile time to define SYSABEND, SYSUDUMP, or SYSMDUMP.

With DUMP, the compiler will not issue a diagnostic message before abnormal termination processing. Instead, a user abend will be issued with an IGY*ppnnnn* message. In general, a message IGY*ppnnnn* corresponds to a compile-time user abend *nnnn*. However, both IGY*pp*5*nnn* and IGY*pp*1*nnn* messages produce a user abend of 1*nnn*. You can usually distinguish whether the message is really a 5*nnn* or a 1*nnn* by recompiling with the NODUMP option.

Use NODUMP if you want normal termination processing, including:
- Diagnostic messages produced so far in compilation.
- A description of the error.
- The name of the compiler phase currently executing.
- The line number of the COBOL statement being processed when the error was found. (If you compiled with OPTIMIZE, the line number might not always be correct; for some errors, it will be the last line in the program.)
- The contents of the general purpose registers.

Using the DUMP and OPTIMIZE compiler options together could cause the compiler to produce a system dump instead of the following optimizer message:

```
"IGYOP3124-W  This statement may cause a program exception at
              execution time."
```

This situation is not a compiler error. Using the NODUMP option will allow the compiler to issue message IGYOP3124-W and continue processing.

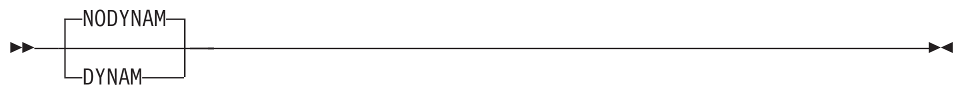
RELATED TASKS
Understanding abend codes (*Language Environment Debugging Guide*)

RELATED REFERENCES
"Conflicting compiler options" on page 300

DYNAM

Use DYNAM to cause nonnested, 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 runtime load of the target program and are not affected by this option.)

DYNAM option syntax



Default is: NODYNAM

Abbreviations are: DYN|NODYN

Restriction: The DYNAM compiler option must not be used in the following cases:
- COBOL programs that are processed by the CICS translator or the CICS compiler option
- COBOL programs that have EXEC SQL statements and are run under CICS or DB2 call attach facility (CAF)

If your COBOL program calls programs that have been linked as dynamic link libraries (DLLs), you must not use the DYNAM option. You must instead compile the program with the NODYNAM and DLL options.

RELATED TASKS
"Making both static and dynamic calls" on page 442
"Choosing the DYNAM or NODYNAM compiler option" on page 415

RELATED REFERENCES
"Conflicting compiler options" on page 300

EXIT

For information about the EXIT compiler option, see the first related reference below.

RELATED REFERENCES
Appendix E, "EXIT compiler option," on page 679
"Conflicting compiler options" on page 300

EXPORTALL

Use EXPORTALL to instruct the compiler to automatically export the PROGRAM-ID name and each alternate entry-point name from each program definition when the object deck is link-edited to form a DLL.

EXPORTALL option syntax

```
      ┌─NOEXPORTALL─┐
►►────┤             ├──────────────────────────────────────────────────►◄
      └─EXPORTALL───┘
```

Default is: `NOEXPORTALL`

Abbreviations are: `EXP|NOEXP`

With these symbols exported from the DLL, the exported program and entry-point names can be called from programs in the root load module or in other DLL load modules in the application, as well as from programs that are linked into the same DLL.

Specification of the `EXPORTALL` option requires that the `RENT` linker option also be used.

`NOEXPORTALL` instructs the compiler to not export any symbols. In this case the programs are accessible only from other routines that are link-edited into the same load module together with this COBOL program definition.

RELATED REFERENCES
"Conflicting compiler options" on page 300

FASTSRT

`FASTSRT` allows IBM DFSORT, or its equivalent, to perform the input and output instead of COBOL.

FASTSRT option syntax

```
      ┌─NOFASTSRT─┐
►►────┤           ├────────────────────────────────────────────────────►◄
      └─FASTSRT───┘
```

Default is: `NOFASTSRT`

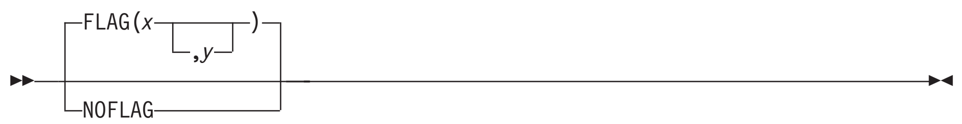
Abbreviations are: `FSRT|NOFSRT`

RELATED TASKS
"Improving sort performance with FASTSRT" on page 225

FLAG

Use `FLAG(`*x*`)` to produce diagnostic messages at the end of the source listing for errors of a severity level *x* or above.

FLAG option syntax



Default is: `FLAG(I,I)`

Abbreviations are: `F|NOF`

x and *y* can be either `I`, `W`, `E`, `S`, or `U`.

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 the embedding of the statement number in an arrow that points to the message code. The message code is 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.  . . .
```

When `FLAG(x,y)` is in effect, messages of severity *y* and above are embedded in the listing following the line that caused the message. (See the related reference below for information about messages for exceptions.)

Use `NOFLAG` to suppress error flagging. `NOFLAG` does not suppress error messages for compiler options.

Embedded messages

- Embedding level-U messages is not recommended. The specification of embedded level-U messages is accepted, but does not produce any messages in the source.
- The `FLAG` option does not affect diagnostic messages that are produced before the compiler options are processed.
- Diagnostic messages that are produced during processing of compiler options, `CBL` or `PROCESS` statements, or `BASIS`, `COPY`, or `REPLACE` statements are not embedded in the source listing. All such messages appear at the beginning of the compiler output.
- Messages that are produced during processing of the `*CONTROL` or `*CBL` statement are not embedded in the source listing.

RELATED REFERENCES
"Messages and listings for compiler-detected errors" on page 275

FLAGSTD

Use FLAGSTD to specify the level or subset of Standard COBOL 85 to be regarded as conforming, and to get informational messages about Standard COBOL 85 elements that are 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 that FLAGSTD is specified, and identified as "nonconforming nonstandard")

FLAGSTD option syntax

```
      ┌─NOFLAGSTD──────────────────┐
►►────┼────────────────────────────┼──────────────────────────────────────►◄
      └─FLAGSTD(x──┬────┬──┬────┬──)─┘
                   └─yy─┘  └─,O─┘
```

Default is: NOFLAGSTD

Abbreviations are: None

x specifies the subset of Standard COBOL 85 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. Elements that are IBM extensions will be flagged as "nonconforming 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 "nonconforming standard."

N Elements from segmentation module level 1 are *not* flagged as "nonconforming standard."

S Elements from segmentation module level 2 are *not* flagged as "nonconforming standard."

If S is specified, N is included (N is a subset of S).

O specifies that obsolete language elements are flagged as "obsolete."

The informational messages appear in the source program listing, and identify:

- The element as "obsolete," "nonconforming standard," or "nonconforming nonstandard" (a language element that is both obsolete and nonconforming is flagged as obsolete only)
- The clause, statement, or header that contains the element
- The source program line and beginning location of the clause, statement, or header that contains the element
- 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
```

RELATED REFERENCES
"Conflicting compiler options" on page 300

INTDATE

INTDATE(ANSI) instructs the compiler to use the Standard COBOL 85 starting date for integer dates used with date intrinsic functions. Day 1 is Jan 1, 1601. INTDATE(LILIAN) instructs the compiler to use the Language Environment Lilian starting date for integer dates used with date intrinsic functions. Day 1 is Oct 15, 1582.

INTDATE option syntax

```
            ┌─ANSI───┐
►►──INTDATE(─┴─LILIAN─┴─)──────────────────────────────────────►◄
```

Default is: INTDATE(ANSI)

Abbreviations are: None

With INTDATE(LILIAN), the date intrinsic functions return results that are compatible with the Language Environment date callable services.

Usage note: When `INTDATE(LILIAN)` is in effect, CEECBLDY is not usable because you have no way to turn an ANSI integer into a meaningful date by using either intrinsic functions or callable services. If you code a `CALL` *literal* statement with CEECBLDY as the target of the call when `INTDATE(LILIAN)` in effect, the compiler diagnoses this and converts the call target to CEEDAYS.

RELATED TASKS
"Using date callable services" on page 62

LANGUAGE

Use the `LANGUAGE` option to select the language in which compiler output will be printed. The information that will be printed in the selected language includes diagnostic messages, source listing page and scale headers, FIPS message headers, message summary headers, compilation summary, and headers and notations that result from the selection of certain compiler options (`MAP`, `XREF`, `VBREF`, and `FLAGSTD`).

LANGUAGE option syntax

```
►►──LANGUAGE(name)──────────────────────────────────────────────►◄
```

Default is: `LANGUAGE(ENGLISH)`

Abbreviations are: `LANG(EN|UE|JA|JP)`

name specifies the language for compiler output messages. Possible values for the `LANGUAGE` option are shown in the table.

Table 48. **Values of the LANGUAGE compiler option**

Name	Abbreviation[1]	Output language
`ENGLISH`	`EN`	Mixed-case English (the default)
`JAPANESE`	`JA`, `JP`	Japanese, using the Japanese character set
`UENGLISH`[2]	`UE`	Uppercase English

1. If your installation's system programmer has provided a language other than those described, you must specify at least the first two characters of this other language's name.
2. To specify a language other than `UENGLISH`, the appropriate language feature must be installed.

If the `LANGUAGE` option is changed at compile time (using `CBL` or `PROCESS` statements), some initial text will be printed using the language that was in effect at the time the compiler was started.

`NATLANG`: The `NATLANG` runtime option allows you to control the national language to be used for the runtime environment, including error messages, month names, and day-of-the-week names. The `LANGUAGE` compiler option and the `NATLANG` runtime option act independently of each other. You can use them together with neither taking precedence over the other.

LIB

If your program uses COPY, BASIS, or REPLACE statements, the LIB compiler option must be in effect.

LIB option syntax

```
     ┌─NOLIB─┐
►►───┼───────┼────────────────────────────────────────────►◄
     └─LIB───┘
```

Default is: NOLIB

Abbreviations are: None

For COPY and BASIS statements, you need additionally to define the library or libraries from which the compiler can take the copied code. Define the libraries by using DD statements, ALLOCATE commands, or environment variables, as appropriate for your environment. When using JCL, also include a DD statement to allocate SYSUT5.

RELATED REFERENCES
Chapter 18, "Compiler-directing statements," on page 351
"Conflicting compiler options" on page 300

LINECOUNT

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.

LINECOUNT option syntax

```
►►──LINECOUNT(nnn)────────────────────────────────────────►◄
```

Default is: LINECOUNT(60)

Abbreviations are: LC

nnn must be an integer between 10 and 255, or 0.

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

Use the LIST compiler option to produce a listing of the assembler-language expansion of your source code.

LIST option syntax

```
      ┌─NOLIST─┐
►►────┤        ├──────────────────────────────────────────────────────►◄
      └─LIST───┘
```

Default is: `NOLIST`

Abbreviations are: None

These items will also be written to the output listing:
- Global tables
- Literal pools
- Information about `WORKING-STORAGE` and `LOCAL-STORAGE`
- Size of the program's `WORKING-STORAGE` and `LOCAL-STORAGE` and its location in the object code if the program is compiled with the `NORENT` option

The output is generated if:
- You specify the `COMPILE` option, or the `NOCOMPILE`(*x*) option is in effect and an error of level *x* or higher does not occur.
- You do not specify the `OFFSET` option.

If you want to limit the assembler listing output, use `*CONTROL` (or `*CBL`) `LIST` or `NOLIST` statements in the `PROCEDURE DIVISION`. Source statements that follow a `*CONTROL NOLIST` statement are not included in the listing until a subsequent `*CONTROL LIST` statement switches the output back to normal `LIST` format.

RELATED TASKS
"Getting listings" on page 365

RELATED REFERENCES
"Conflicting compiler options" on page 300
*CONTROL (*CBL) statement (*Enterprise COBOL Language Reference*)

MAP

Use `MAP` to produce a listing of the items defined in the `DATA DIVISION`.

MAP option syntax

```
      ┌─NOMAP─┐
►►────┤       ├───────────────────────────────────────────────────────►◄
      └─MAP───┘
```

Default is: `NOMAP`

Abbreviations are: None

The output includes the following items:

- DATA DIVISION map
- Global tables
- Literal pools
- Nested program structure map, and program attributes
- Size of the program's WORKING-STORAGE and LOCAL-STORAGE and its location in the object code if the program is compiled with the NORENT option

If you want to limit the MAP output, use *CONTROL MAP or NOMAP statements in the DATA DIVISION. Source statements that follow *CONTROL NOMAP are not included in the listing until a *CONTROL MAP statement switches the output back to normal MAP format. For example:

```
*CONTROL NOMAP            *CBL NOMAP
    01  A                     01  A
    02  B                     02  B
*CONTROL MAP              *CBL MAP
```

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, LOCAL-STORAGE SECTION, and LINKAGE SECTION of the DATA DIVISION. When both XREF data and an embedded MAP summary are on the same line, the embedded summary is printed first.

"Example: MAP output" on page 370

RELATED CONCEPTS
Chapter 19, "Debugging," on page 355

RELATED TASKS
"Getting listings" on page 365

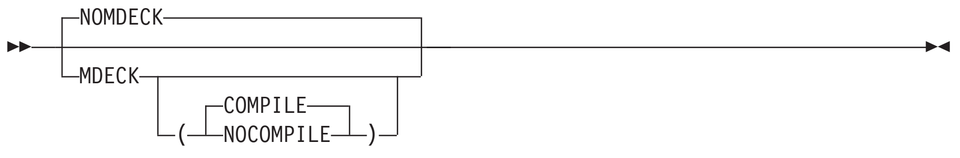
RELATED REFERENCES
*CONTROL (*CBL) statement (*Enterprise COBOL Language Reference*)

MDECK

The MDECK compiler option specifies that output from library processing (that is, expansion of COPY, BASIS, REPLACE, or EXEC SQL INCLUDE statements) is written to a file.

When Enterprise COBOL is running under z/OS UNIX, the MDECK output is written in the current directory to a file that has the same name as the COBOL source file and a suffix of .dek. For Enterprise COBOL running under TSO or batch, the MDECK output is written to the data set defined by the SYSMDECK DD statement, which must specify an MVS data set that has RECFM F or FB and an LRECL of 80 bytes.

MDECK option syntax



Default is: NOMDECK

Abbreviations are: NOMD, MD, MD(C), MD(NOC)

Suboptions:

- When MDECK(COMPILE) is in effect, compilation continues normally after library processing and generation of the MDECK output file have completed, subject to the settings of the COMPILE|NOCOMPILE, DECK|NODECK, and OBJECT|NOOBJECT compiler options.
- When MDECK(NOCOMPILE) is in effect, compilation is terminated after library processing has completed and the expanded source program file has been written. The compiler does no further syntax checking or code generation regardless of the settings of the COMPILE, DECK, and OBJECT compiler options.

When you specify MDECK with no suboption, MDECK(COMPILE) is implied.

Option specification:

You cannot specify MDECK in a PROCESS or CBL statement. You can specify the option only by using:

- The PARM parameter in JCL
- A cob2 command option
- An installation default
- The COBOPT environment variable

Contents of the MDECK output file:

When you use the MDECK option with the CICS compiler option (integrated CICS translator) or the SQL compiler option (DB2 coprocessor), in general, EXEC CICS or EXEC SQL statements in the COBOL source program are included in the MDECK output as is. However, EXEC SQL INCLUDE statements are expanded in the MDECK output in the same manner as COPY statements.

CBL, PROCESS, *CONTROL, and *CBL card images are passed to the MDECK output file in the proper locations.

For a batch compilation (multiple COBOL source programs in a single input file), a single MDECK output file that contains the complete expanded source is created.

Any SEQUENCE compiler-option processing is reflected in the MDECK file.

COPY statements are included in the MDECK file as comments.

RELATED TASKS
"Starting the compiler from an assembler program" on page 260
"Defining the library-processing output file (SYSMDECK)" on page 267

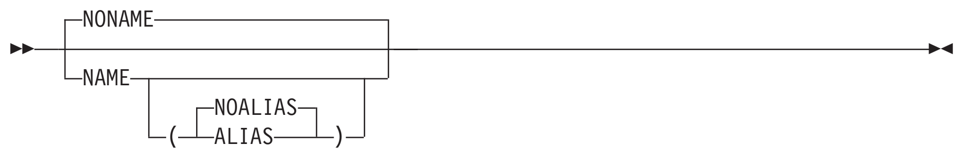
RELATED REFERENCES
"Conflicting compiler options" on page 300
Chapter 18, "Compiler-directing statements," on page 351

NAME

Use NAME to generate a link-edit NAME card for each object module. You can also use NAME to generate names for each load module when you are doing batch compilations.

When NAME is specified, a NAME card is appended to each object module that is created. Load module names are formed using the rules for forming module names from PROGRAM-ID statements.

NAME option syntax



Default is: NONAME, or NAME(NOALIAS) if only NAME is specified

Abbreviations are: None

If you specify NAME(ALIAS), and your program contains ENTRY statements, a link-edit ALIAS card is generated for each ENTRY statement.

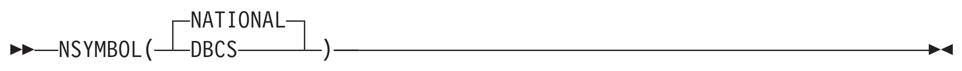
The NAME or NAME(ALIAS) option cannot be used for compiling programs that will be prelinked with the Language Environment prelinker.

RELATED REFERENCES
PROGRAM-ID paragraph (*Enterprise COBOL Language Reference*)

NSYMBOL

The NSYMBOL option controls the interpretation of the N symbol used in literals and PICTURE clauses, indicating whether national or DBCS processing is assumed.

NSYMBOL option syntax



Default is: NSYMBOL(NATIONAL)

Abbreviations are: NS(NAT|DBCS)

With NSYMBOL(NATIONAL):

- Data items defined with a PICTURE clause that consists only of the symbol N without the USAGE clause are treated as if the USAGE NATIONAL clause is specified.
- Literals of the form N". . ." or N'. . .' are treated as national literals.

With NSYMBOL(DBCS):

- Data items defined with a PICTURE clause that consists only of the symbol N without the USAGE clause are treated as if the USAGE DISPLAY-1 clause is specified.
- Literals of the form N". . ." or N'. . .' are treated as DBCS literals.

The NSYMBOL(DBCS) option provides compatibility with previous releases of IBM COBOL, and the NSYMBOL(NATIONAL) option makes the handling of the above language elements consistent with Standard COBOL 2002 in this regard.

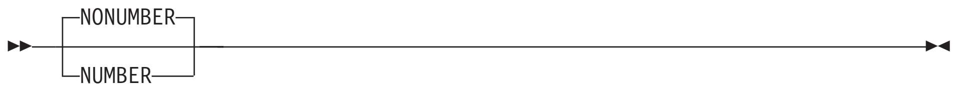
NSYMBOL(NATIONAL) is recommended for applications that use Unicode data or object-oriented syntax for Java interoperability.

RELATED REFERENCES
"Conflicting compiler options" on page 300

NUMBER

Use the NUMBER compiler option if you have line numbers in your source code and want those numbers to be used in error messages and SOURCE, MAP, LIST, and XREF listings.

NUMBER option syntax



Default is: NONUMBER

Abbreviations are: NUM|NONUM

If you request NUMBER, the compiler checks columns 1 through 6 to make sure that they contain only numbers and that the numbers are in numeric collating sequence. (In contrast, SEQUENCE checks the characters in these columns according to EBCDIC 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 higher than the line number of the preceding statement. The compiler flags the new value with two asterisks and includes in the listing a message indicating an out-of-sequence error. 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 copybook line numbers are coordinated.

If you are doing a batch compilation and LIB and NUMBER are in effect, all programs in the batch compile will be treated as a single input file. The sequence numbers of the entire input file must be in ascending order.

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.

NUMPROC

Use NUMPROC(NOPFD) whenever your numeric internal decimal and zoned decimal data might use nonpreferred signs.

NUMPROC option syntax

```
          ┌─NOPFD─┐
►►──NUMPROC(──┼─PFD───┼──)──────────────────────────────────────►◄
          └─MIG───┘
```

Default is: NUMPROC(NOPFD)

Abbreviations are: None

The compiler accepts any valid sign configuration: X'A', X'B', X'C', X'D', X'E', or X'F'. NUMPROC(NOPFD) is the recommended option in most cases.

NUMPROC(PFD) improves the performance of processing numeric internal decimal and zoned decimal data. Use this option *only* if your program data agrees exactly with the following IBM system standards:

Zoned decimal, unsigned: High-order 4 bits of the sign byte contain X'F'.

Zoned decimal, signed overpunch: High-order 4 bits of the sign byte contain X'C' if the number is positive or 0, and X'D' if it is not.

Zoned decimal, separate sign: Separate sign contains the character '+' if the number is positive or 0, and '-' if it is not.

Internal decimal, unsigned: Low-order 4 bits of the low-order byte contain X'F'.

Internal decimal, signed: Low-order 4 bits of the low-order byte contain X'C' if the number is positive or 0, and X'D' if it is not.

Data produced by COBOL arithmetic statements conforms to the above IBM system standards. However, using REDEFINES and group moves could change data so that it no longer conforms. If you use NUMPROC(PFD), use the INITIALIZE statement to initialize data fields, rather than using group moves.

Using NUMPROC(PFD) can affect class tests for numeric data. You should use NUMPROC(NOPFD) or NUMPROC(MIG) if a COBOL program calls programs written in PL/I or FORTRAN.

Sign representation is affected not only by the NUMPROC option, but also by the installation-time option NUMCLS.

Use NUMPROC(MIG) to aid in migrating OS/VS COBOL programs to Enterprise COBOL. When NUMPROC(MIG) is in effect, the following processing occurs:
- Preferred signs are created only on the output of MOVE statements and arithmetic operations.
- No explicit sign repair is done on input.
- Some implicit sign repair might occur during conversion.
- Numeric comparisons are performed by a decimal comparison, not a logical comparison.

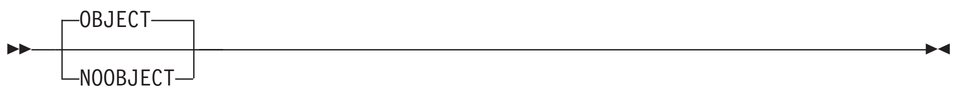
RELATED TASKS
"Checking for incompatible data (numeric class test)" on page 56

RELATED REFERENCES
"Sign representation of zoned and packed-decimal data" on page 55

OBJECT

Use OBJECT to place the generated object code on disk or tape to be later used as input for the linkage editor or binder.

OBJECT option syntax



Default is: OBJECT

Abbreviations are: OBJ|NOOBJ

If you specify OBJECT, include a SYSLIN DD statement in your JCL for compilation.

The only difference between DECK and OBJECT is in the routing of the data sets:
- DECK output goes to the data set associated with ddname SYSPUNCH.
- OBJECT output goes to the data set associated with ddname SYSLIN.

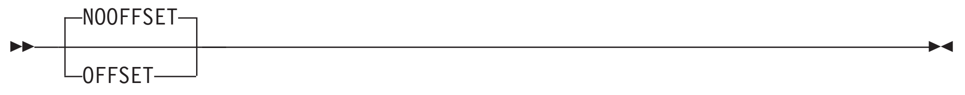
Use the option that your installation guidelines recommend.

RELATED REFERENCES
"Conflicting compiler options" on page 300

OFFSET

Use OFFSET to produce a condensed PROCEDURE DIVISION listing.

OFFSET option syntax



Default is: `NOOFFSET`

Abbreviations are: `OFF|NOOFF`

With `OFFSET`, the condensed `PROCEDURE DIVISION` listing will contain line numbers, statement references, and the location of the first instruction generated for each statement. In addition, the listing also shows:

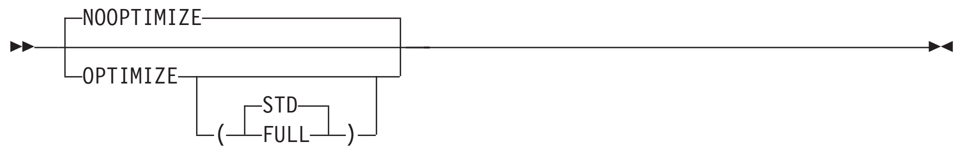
- Global tables
- Literal pools
- Size of the program's `WORKING-STORAGE`, and its location in the object code if the program is compiled with the `NORENT` option

RELATED REFERENCES
"Conflicting compiler options" on page 300

OPTIMIZE

Use `OPTIMIZE` to reduce the run time of your object program. Optimization might also reduce the amount of storage your object program uses. Because `OPTIMIZE` increases compile time and can change the order of statements in your program, you should not use it when debugging.

OPTIMIZE option syntax



Default is: `NOOPTIMIZE`

Abbreviations are: `OPT|NOOPT`

If `OPTIMIZE` is specified without any suboptions, `OPTIMIZE(STD)` will be in effect.

The `FULL` suboption requests that, in addition to the optimizations performed with `OPT(STD)`, the compiler discard unreferenced data items from the `DATA DIVISION` and suppress generation of code to initialize these data items to the values in their `VALUE` clauses. When `OPT(FULL)` is in effect, all unreferenced level-77 items and elementary level-01 items are discarded. In addition, level-01 group items are discarded if none of their subordinate items are referenced. The deleted items are shown in the listing. If the MAP option is in effect, a BL number of XXXXX in the data map information indicates that the data item was discarded.

Unused data items: Do not use OPT(FULL) if your programs depend on making use of unused data items. In the past, this was commonly done in two ways:

- A technique sometimes used in old OS/VS COBOL programs was 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 use this technique, use the SSRANGE compiler option with the CHECK(ON) runtime option. To work around this problem, use the ability of newer COBOL to code large tables and use just one table.
- Place eye-catcher 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 the 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 used and will not delete them.

The OPTIMIZE option is turned off in the case of a severe-level error or higher.

RELATED CONCEPTS
"Optimization" on page 631

RELATED REFERENCES
"Conflicting compiler options" on page 300

OUTDD

Use OUTDD to specify that you want DISPLAY output that is directed to the system logical output device to go to a specific ddname. You can specify a file in the hierarchical file system with the ddname named in OUTDD. See the related task about displaying data for the behavior when this ddname is not allocated.

OUTDD option syntax

```
►►──OUTDD(ddname)──────────────────────────────────────────►◄
```

Default is: OUTDD(SYSOUT)

Abbreviations are: OUT

The MSGFILE runtime option allows you to specify the ddname of the file to which all runtime diagnostics and reports generated by the RPTOPTS and RPTSTG runtime options are written. The IBM-supplied default is MSGFILE(SYSOUT). If the OUTDD compiler option and the MSGFILE runtime option both specify the same ddname, the error message information and DISPLAY output directed to the system logical output device are routed to the same destination.

+ **Restriction:** The OUTDD option has no effect under CICS.

RELATED TASKS
"Displaying data on the system logical output device" on page 38
"Coding COBOL programs to run under CICS" on page 393

RELATED REFERENCES
MSGFILE (*Language Environment Programming Reference*)

PGMNAME

The PGMNAME option controls the handling of program-names and entry-point names.

PGMNAME option syntax

```
              ┌─COMPAT─────┐
►►──PGMNAME(──┼─LONGMIXED──┼──)──────────────────────────────────►◄
              └─LONGUPPER──┘
```

Default is: PGMNAME(COMPAT)

Abbreviations are: PGMN(LM|LU|CO)

LONGUPPER can be abbreviated as UPPER, LU, or U. LONGMIXED can be abbreviated as MIXED, LM, or M.

PGMNAME controls the handling of names used in the following contexts:
- Program-names defined in the PROGRAM-ID paragraph
- Program entry-point names in the ENTRY statement
- Program-name references in:
  - Calls to nested programs
  - Static calls to separately compiled programs
  - Static SET *procedure-pointer* TO ENTRY *literal* statement
  - Static SET *function-pointer* TO ENTRY *literal* statement
  - CANCEL of a nested program

PGMNAME(COMPAT)

With PGMNAME(COMPAT), program-names are handled in a manner compatible with older versions of COBOL compilers:
- 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, except that if the program-name is entered in the literal format and is in the outermost program, then the literal can also contain the extension characters @, #, and $.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last character.

External program-names are processed by the compiler as follows:
- They are folded to uppercase.
- They are truncated to eight characters.
- Hyphens are translated to zero (0).
- If the first character is not alphabetic, it is converted as follows:
  - 1-9 are translated to A-I.

- Anything else is translated to J.

PGMNAME(LONGUPPER)

With `PGMNAME(LONGUPPER)`, 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-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 by the compiler as follows:

- They are folded to uppercase.
- Hyphens are translated to zero (0).
- If the first character is not alphabetic, it is converted as follows:
  - 1-9 are translated to A-I.
  - Anything else is translated to J.

Names of nested programs are folded to uppercase by the compiler but otherwise are processed as is, without truncation or translation.

PGMNAME(LONGMIXED)

With `PGMNAME(LONGMIXED)`, program-names are processed as is, without truncation, translation, or folding to uppercase.

With `PGMNAME(LONGMIXED)`, all program-name definitions must be specified using the literal format of the program-name in the `PROGRAM-ID` paragraph or `ENTRY` statement.

The literal used for a program-name (in any of the contexts listed above as affected by the `PGMNAME` option) can contain any character in the range `X’41’-X’FE’`.

Usage notes

- The following elements are not affected by the `PGMNAME` option:
  - Class-names and method-names.
  - System-names (assignment-names in `SELECT . . . ASSIGN`, and text-names or library-names in `COPY` statements).
  - Dynamic calls. Dynamic calls are resolved with the target program-name truncated to eight characters, folded to uppercase, and translation of embedded hyphens or a leading digit.
  - `CANCEL` of nonnested programs. Name resolution uses the same mechanism as for a dynamic call.

- The PGMNAME option does affect nested-program calls and static calls to programs that are linked together with the caller.
- **Link-edit considerations:** COBOL programs that are compiled with the PGMNAME(LONGUPPER) or PGMNAME(LONGMIXED) option must be link-edited with AMODE 31.
- Dynamic calls are not permitted to COBOL programs compiled with the PGMNAME(LONGMIXED) or PGMNAME(LONGUPPER) options unless the program-name is less than or equal to 8 bytes and all uppercase. In addition, the name of the program must be identical to the name of the module that contains it.
- When using the extended character set supported by PGMNAME(LONGMIXED), be sure to use names that conform to the linkage-editor, binder, prelinker, or system conventions that apply, depending on the mechanism used to resolve the names.

  Using characters such as commas or parentheses is not recommended, because these characters are used in the syntax of linkage-editor and binder control statements.

QUOTE/APOST

Use QUOTE if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more quotation mark (“) characters. Use APOST if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more single quotation mark (’) characters.

QUOTE/APOST option syntax

```
    ┌─QUOTE─┐
►►──┴─APOST─┴──────────────────────────────────────────────────►◄
```

Default is: QUOTE

Abbreviations are: Q|APOST

Delimiters: You can use either quotation marks or single quotation marks 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.

RENT

A program compiled as RENT is generated as a reentrant object program. A program compiled as NORENT is generated as a nonreentrant object program. Either can be invoked as a main program or subprogram.

RENT option syntax

```
    ┌─RENT───┐
►►──┴─NORENT─┴─────────────────────────────────────────────────►◄
```

Default is: RENT

Abbreviations are: None

DATA and RMODE settings: The RENT option interacts with other compiler options that affect storage and its addressability. When a reentrant program is to be run with extended addressing, you can use the DATA(24|31) option to control whether dynamic data areas are allocated in unrestricted storage or in storage obtained from below 16 MB. Compile programs with RENT or RMODE(ANY) if they will be run with extended addressing in virtual storage addresses above 16 MB.

RENT also affects the RMODE (residency mode) of your generated object program. All Enterprise COBOL programs are AMODE ANY.

DATA: The setting of the DATA option does not affect programs compiled with NORENT.

CICS: You must use RENT for programs to be run under CICS.

UNIX: You must use RENT for programs to be run in the UNIX environment.

DB2: RENT is recommended for DB2 stored procedures, especially programs designated as main.

Link-edit considerations: If all programs in a load module are compiled with RENT, it is recommended that the load module be link-edited with the RENT linkage-editor or binder option. (Use the REUS linkage-editor or binder option instead if the load module will also contain any non-COBOL programs that are serially reusable.)

If any program in a load module is compiled with NORENT, the load module must not be link-edited with the RENT or REUS link-edit attributes. The NOREUS linkage-editor or binder option is needed to ensure that the CANCEL statement will guarantee a fresh copy of the program on a subsequent CALL.

RELATED CONCEPTS
"Storage and its addressability" on page 41

RELATED TASKS
Using reentrant code (*DB2 UDB Application Programming and SQL Guide*)

RELATED REFERENCES
"Conflicting compiler options" on page 300

RMODE

The RMODE option setting influences the RMODE (residency mode) of your generated object program.

RMODE option syntax

```
          ┌─AUTO─┐
►►──RMODE(─┼─24───┼─)──────────────────────────────────────►◄
          └─ANY──┘
```

Default is: AUTO

Abbreviations are: None

A program compiled with the RMODE(AUTO) option will have RMODE 24 if NORENT is specified, and RMODE ANY if RENT is specified. RMODE(AUTO) is compatible with older compilers such as VS COBOL II, which produced RMODE 24 for programs compiled with NORENT and RMODE ANY for programs compiled with RENT.

A program compiled with the RMODE(24) option will have RMODE 24 whether NORENT or RENT is specified.

A program compiled with the RMODE(ANY) option will have RMODE ANY whether NORENT or RENT is specified.

DATA and RENT: The RMODE option interacts with other compiler options and runtime options that affect storage and its addressability. See the related concepts for information about passing data between programs with different modes.

Link-edit considerations: When the object code that COBOL generates has an attribute of RMODE 24, you must link-edit it with RMODE 24. When the object code that COBOL generates has an attribute of RMODE ANY, you can link-edit it with RMODE ANY or RMODE 24.

RELATED CONCEPTS
"Storage and its addressability" on page 41

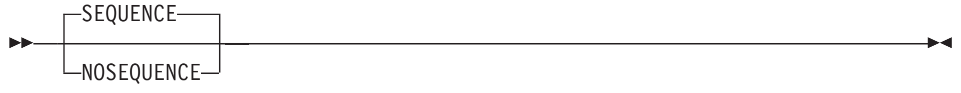
RELATED REFERENCES
"Allocation of buffers for QSAM files" on page 172
"Conflicting compiler options" on page 300

SEQUENCE

When you use SEQUENCE, the compiler examines columns 1 through 6 to check that the source statements are arranged in ascending order according to their EBCDIC collating sequence. The compiler issues a diagnostic message if any statements are not in ascending order.

Source statements with blanks in columns 1 through 6 do not participate in this sequence check and do not result in messages.

SEQUENCE option syntax



Default is: SEQUENCE

Abbreviations are: SEQ|NOSEQ

If you use COPY statements and SEQUENCE is in effect, be sure that your source program sequence fields and the copybook sequence fields are coordinated.

If you use NUMBER and SEQUENCE, the sequence is checked according to numeric, rather than EBCDIC, collating sequence.

If you are doing a batch compilation and LIB and SEQUENCE are in effect, all programs in the batch compilation are treated as a single input file. The sequence numbers of the entire input file must be in ascending order.

Use NOSEQUENCE to suppress this checking and the diagnostic messages.

RELATED TASKS
"Finding line sequence problems" on page 361

SIZE

Use SIZE to indicate the amount of main storage to be made available for compilation.

SIZE option syntax

```
          ┌─MAX───┐
►►──SIZE(─┼─nnnnn─┼─)──────────────────────────────────────────►◄
          └─nnnK──┘
```

Default is: SIZE(MAX)

Abbreviations are: SZ

nnnnn specifies a decimal number, which must be at least 851968.

*nnn*K specifies a decimal number in 1-KB increments, where 1 KB = 1024 bytes. The minimum acceptable value is 832K.

MAX requests the largest available block of storage in the user region.

Do not use SIZE(MAX) if you require that the compiler leave a specific amount of unused storage available in the user region. For example, if you are using the CICS or SQL compiler option, use a value such as SIZE(4000K). (This value should work for most programs.) If you compile in 31-bit mode and specify SIZE(MAX), the compiler uses storage as follows:

- Above the 16-MB line: all the storage in the user region
- Below the 16-MB line: storage for:
  - Work-file buffers
  - Compiler modules that must be loaded below the line

SOURCE

Use SOURCE to get a listing of your source program. This listing will include any statements embedded by PROCESS or COPY statements.

SOURCE option syntax

```
     ┌─SOURCE───┐
►►───┤          ├──────────────────────────────────────────►◄
     └─NOSOURCE─┘
```

Default is: SOURCE

Abbreviations are: S|NOS

You must specify SOURCE 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. Source statements that follow a *CONTROL NOSOURCE statement are not included in the listing until a subsequent *CONTROL SOURCE statement switches the output back to normal SOURCE format.

"Example: MAP output" on page 370

RELATED REFERENCES
*CONTROL (*CBL) statement (*Enterprise COBOL Language Reference*)

SPACE

Use SPACE to select single-, double-, or triple-spacing in your source code listing.

SPACE option syntax

```
            ┌─1─┐
►►──SPACE(──┼─2─┼──)───────────────────────────────────────►◄
            └─3─┘
```

Default is: SPACE(1)

Abbreviations are: None

SPACE has meaning only when the SOURCE compiler option is in effect.

RELATED REFERENCES
"SOURCE" on page 334

SQL

Use the SQL compiler option to enable the DB2 coprocessor capability and to specify DB2 suboptions. You must specify the SQL option if a COBOL source program contains SQL statements and it has not been processed by the DB2 precompiler.

SQL option syntax

```
     ┌─NOSQL─────────────────────────┐
►►───┼───────────────────────────────┼──────────────────────────────────────────►◄
     └─SQL─┬───────────────────────┬─┘
           └─("DB2-suboption-string")─┘
```

Default is: NOSQL

Abbreviations are: None

When you use the SQL option, the compiler writes the database request module (DBRM) to ddname DBRMLIB. Note that the compiler needs access to DB2 for z/OS or OS/390 Version 7 or later.

If you specify the NOSQL option, any SQL statements found in the source program are diagnosed and discarded.

Use either quotation marks or single quotation marks to delimit the string of DB2 suboptions.

You can partition a long suboption string into multiple suboption strings in multiple CBL statements. The DB2 suboptions are concatenated in the order of their appearance. For example:

```
//STEP1 EXEC IGYWC, . . .
// PARM.COBOL='SQL("string1")'
//COBOL.SYSIN DD *
       CBL SQL("string2")
       CBL SQL('string3')
       IDENTIFICATION DIVISION.
       PROGRAM-ID. DRIVER1.
       . . .
```

The compiler passes the following suboption string to the DB2 coprocessor:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces as shown. If multiple instances of the same DB2 option are found, the last specification of each option prevails. The compiler limits the length of the concatenated DB2 suboptions string to 4 KB.

RELATED CONCEPTS
"DB2 coprocessor" on page 405
"COBOL and DB2 CCSID determination" on page 411

RELATED TASKS
"Compiling with the SQL option" on page 409
"Separating DB2 suboptions" on page 411

RELATED REFERENCES
"Conflicting compiler options" on page 300

SQLCCSID

Use the SQLCCSID compiler option to control whether the CODEPAGE compiler option will influence the processing of SQL statements in your COBOL programs.

SQLCCSID option syntax

```
        ┌─SQLCCSID───┐
▶▶──────┤            ├──────────────────────────────────────────────▶◀
        └─NOSQLCCSID─┘
```

Default is: SQLCCSID

Abbreviations are: SQLC|NOSQLC

The SQLCCSID option has an effect only when you use the integrated DB2 coprocessor (SQL compiler option).

RELATED CONCEPTS
"DB2 coprocessor" on page 405
"COBOL and DB2 CCSID determination" on page 411

RELATED TASKS
"Programming with the SQLCCSID or NOSQLCCSID option" on page 412

RELATED REFERENCES
"Code-page determination for string host variables in SQL statements" on page 412
"CODEPAGE" on page 305
"SQL" on page 335

SSRANGE

Use SSRANGE to generate code that checks whether 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 are also checked to ensure that the reference is within their maximum defined length.

SSRANGE option syntax

```
        ┌─NOSSRANGE─┐
▶▶──────┤           ├───────────────────────────────────────────────▶◀
        └─SSRANGE───┘
```

Default is: NOSSRANGE

Abbreviations are: SSR|NOSSR

Reference modification expressions are checked to ensure that:

- The starting position is greater than or equal to 1.
- The starting position is not greater than the current length of the subject data item.
- The length value (if specified) is greater than or equal to 1.
- The 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, range-checking code is generated. You can inhibit range checking by specifying the CHECK(OFF) runtime option. Doing so leaves range-checking code dormant in the object code. Optionally, the range-checking code can be used to aid in resolving unexpected errors without recompilation.

If an out-of-range condition is detected, an error message is generated and the program is terminated.

Remember: Range checking is done only if you compile a program with the SSRANGE option and run it with the CHECK(ON) option.

RELATED CONCEPTS
"Reference modifiers" on page 109

RELATED TASKS
"Checking for valid ranges" on page 361

TERMINAL

Use TERMINAL to send progress and diagnostic messages to the SYSTERM ddname.

TERMINAL option syntax

```
   ┌─NOTERMINAL─┐
►►─┼────────────┼──────────────────────────────────────────────►◄
   └─TERMINAL───┘
```

Default is: NOTERMINAL

Abbreviations are: TERM|NOTERM

Use NOTERMINAL if you do not want this additional output.

TEST

Use TEST to produce object code that enables Debug Tool to perform batch and interactive debugging.

TEST option syntax

```
     ┌─NOTEST──────────────────────────────────────────────────────┐
►►───┼─────────────────────────────────────────────────────────────┼────────────►◄
     └─TEST──┬─────────────────────────────────────────────────────┤
             │      ┌─ALL───┐        ┌─,SYM─┐    ┌─,NOSEPARATE─┐    │
             └─(─┬──┼─NONE──┼───┬────┴──────┴────┼─────────────┼──┬─)─┘
                    ├─BLOCK─┤   │                └─,SEPARATE───┘  │
                    ├─PATH──┤   └─,NOSYM──┬─────────────────┬─────┘
                    └─STMT──┘             └─,NOSEPARATE─────┘
```

Default is: NOTEST

Abbreviations are: SEP for SEPARATE and NOSEP for NOSEPARATE.

The amount of debugging support available depends on which TEST suboptions you use. The TEST option also allows you to request that symbolic variables be included in the formatted dump produced by Language Environment.

Use NOTEST if you do not want to generate object code that has debugging information and do not want the formatted dump to include symbolic variables.

TEST has three suboptions. You can specify any combination of suboptions (one, two, or all), but you can specify SEPARATE only when SYM is in effect.

If you specify TEST without any suboptions, TEST(ALL,SYM,NOSEP) will be in effect.

Hook location suboptions

NONE No hooks will be generated.

BLOCK Hooks will be generated at all program entry and exit points.

PATH Hooks will be generated at all program entry and exit points and at all path points. A *path point* is any location in a program where the logic flow is not necessarily sequential or can change. Some examples of path points are IF-THEN-ELSE constructs, PERFORM loops, ON SIZE ERROR phrases, and CALL statements.

STMT Hooks will be generated at every statement and label, and at all program entry and exit points. In addition, if the DATEPROC option is in effect, hooks will be generated at all date-processing statements.

ALL Hooks will be generated at all statements, all path points, and at all program entry and exit points (both in outermost and in contained programs). In addition, if the DATEPROC option is in effect, hooks will be generated at all date-processing statements.

Symbolic information suboptions

SYM When you specify SYM, the compiler generates symbol information tables. These tables are required if you:

- Refer to COBOL identifiers, such as data-names, file-names, and special registers, in Debug Tool commands such as `LIST`, `MOVE`, `SET`, `IF`, `PERFORM`, and `EVALUATE`
- Use the automonitor and the playback (saving the application data value history) features of Debug Tool
- Specify that the Language Environment dump of the program include a formatted dump of the data items that are declared in the program

NOSYM The compiler does not generate symbol information tables.

Symbolic information location suboptions

SEPARATE
Specify the `SEPARATE` suboption to control module size while retaining debugging capability. Symbolic information is written to the SYSDEBUG file instead of to the object module. See the additional information below about controlling module size while retaining debug capability.

NOSEPARATE
When you do not specify the `SEPARATE` suboption, the symbolic information is included in the object module.

Controlling module size while retaining debugging capability: For symbolic debugging, compile your programs with `TEST(SYM)`. This option causes the compiler to generate debugging information tables that Debug Tool uses to resolve data-names, paragraph-names, and the like. This information can take a lot of storage. You can choose to either compile the information into the object program or write it to the separate SYSDEBUG file. For smaller load modules, use the `SEPARATE` suboption and keep the separate debugging files to use for Debug Tool sessions. To avoid the management of separate debugging files, you can compile with the `NOSEPARATE` suboption, but this option will result in larger load modules.

When you invoke the COBOL compiler from JCL or from TSO and you specify `TEST(. . .,SYM,SEPARATE)`, the symbolic debug information tables are written to the data set that is specified in the `SYSDEBUG DD` statement. The `SYSDEBUG DD` statement must specify the name of a sequential data set, the name of a PDS or PDSE member, or an HFS path. The data set `LRECL` must be greater than or equal to 80 and less than or equal to 1024. The default `LRECL` for `SYSDEBUG` is 1024. The data set `RECFM` can be `F` or `FB`. You can set the block size by using the `BLKSIZE` subparameter of the `DCB` parameter, or leave it to the system to set the system-determined default block size.

When you invoke the COBOL compiler from the UNIX shell and you specify `TEST(. . .,SYM,SEPARATE)`, the symbolic debug information tables are written to *file*.dbg in the current directory, where *file* is the name of the COBOL source file.

Performance versus debugging capability: You can control the amount of debugging capability you get and so also the performance, as follows:

- For high performance and some restrictions on debugging, specify `OPT` and `TEST(NONE,SYM)`.

  When you use the dynamic debug facility of Debug Tool (`SET DYNDEBUG ON`), you can interactively debug your program even if the program has no compiled-in

debug hooks. To use this function, compile with TEST(NONE,SYM). With this option, you can also compile with OPTIMIZE (either OPTIMIZE(STD) or OPTIMIZE(FULL)) for a more efficient program, but with some restrictions on debugging:
  - The GOTO Debug Tool command is not supported.
  - Except for the DESCRIBE ATTRIBUTES command, Debug Tool commands cannot refer to any data item that was discarded from your program by the OPTIMIZE(FULL) option.
  - The AT CALL *entry-name* Debug Tool command is not supported.
- For medium performance and fewer restrictions on debugging, specify NOOPT and TEST(NONE,SYM).

  This combination does not run as fast as optimized code, but it provides increased debugging capability. All Debug Tool commands are supported except the AT CALL *entry-name* command.
- For slow performance but maximum debugging capabilities, specify TEST(ALL).

  The TEST(ALL) option causes the compiler to put compiled-in hooks at every statement, resulting in much slower code, but all Debug Tool commands are supported.

Language Environment: The TEST option can improve your formatted dumps from Language Environment in two ways:
- Use the TEST option (with any suboptions) to have line numbers in the dump that indicate the failing statement rather than just an offset.
- Use the SYM suboption of TEST to have the values of the program variables listed in the dump.

With NOTEST, the dump will not have program variables and will not have a line number for the failing statement.

Enterprise COBOL uses the Language Environment-provided dump services to produce dumps that are consistent in content and format with those that are produced by other Language Environment-conforming member languages. Whether Language Environment produces a dump for unhandled conditions depends on the setting of the runtime option TERMTHDACT. If you specify TERMTHDACT(DUMP), a dump is generated when a condition of severity 2 or greater goes unhandled.

SEPARATE suboption and Language Environment: For programs that are compiled with the SEPARATE suboption of TEST, Language Environment gets the data-set name for the separate file (which is stored in DD SYSDEBUG by the compiler) from
| the object program. To change the name of the separate file that is used by
| Language Environment, use the Language Environment COBOL debug file exit.

RELATED CONCEPTS
Considerations for setting TERMTHDACT options (*Language Environment Debugging Guide*)

RELATED TASKS
"Defining the debug data set (SYSDEBUG)" on page 266
Generating a dump (*Language Environment Debugging Guide*)
Starting Debug Tool by using the TEST runtime option (*Debug Tool User's Guide*)

RELATED REFERENCES
"Conflicting compiler options" on page 300

"OPTIMIZE" on page 327
TEST | NOTEST (*Language Environment Programming Reference*)

THREAD

THREAD indicates that a COBOL program is to be enabled for execution in a Language Environment enclave that has multiple POSIX threads or PL/I tasks.

THREAD option syntax

```
       ┌─NOTHREAD─┐
►►─────┤          ├──────────────────────────────────────────────────────►◄
       └─THREAD───┘
```

Default is: NOTHREAD

Abbreviations are: None

A program that has been compiled with the THREAD option can also be used in a nonthreaded application. However, if a COBOL program is to be run in a threaded application, all the COBOL programs in the Language Environment enclave must be compiled with the THREAD option.

NOTHREAD indicates that the COBOL program is not to be enabled for execution in an enclave that has multiple POSIX threads or PL/I tasks.

Programs compiled using compilers earlier than Enterprise COBOL are treated as if compiled with NOTHREAD.

When the THREAD option is in effect, the following items are not supported. If encountered, they are diagnosed as errors:

- ALTER statement
- DEBUG-ITEM special register
- GO TO statement without procedure-name
- INITIAL phrase in PROGRAM-ID clause
- Nested programs
- RERUN
- Segmentation module
- SORT or MERGE statements
- STOP *literal* statement
- USE FOR DEBUGGING statement

Additionally, some language constructs have different semantics than in the nonthreaded case.

Although threaded applications are subject to a number of programming and environment restrictions, the use of a program in nonthreaded applications is not so restricted. For example, a program compiled with the THREAD option can run in the CICS and IMS environments, can run AMODE 24, and can call and be called by other programs that are not enabled for multithreading, as long as the application does not contain multiple POSIX threads or PL/I tasks at run time.

Programs compiled with the THREAD option are supported in a reusable environment that is created by calling the Language Environment preinitialization routine CEEPIPI. But a reusable environment created by calling IGZERRE or ILBOSTP0 or by using the RTEREUS runtime option is not supported for programs compiled with the THREAD option.

Performance consideration: When using the THREAD option, you can expect some runtime performance degradation due to the overhead of serialization logic that is automatically generated.

RELATED TASKS
Chapter 27, "Preparing COBOL programs for multithreading," on page 477

RELATED REFERENCES
"Conflicting compiler options" on page 300

TRUNC

TRUNC affects the way that binary data is truncated during moves and arithmetic operations.

TRUNC option syntax

```
              ┌─STD─┐
►►──TRUNC(────┼─OPT─┼──)──────────────────────────────────────────►◄
              └─BIN─┘
```

Default is: TRUNC(STD)

Abbreviations are: None

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 suboption specified.

TRUNC(STD)
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 in effect, the compiler assumes that data conforms to PICTURE specifications in 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).

Tips:

- 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 could occur. This truncation is performed in the most efficient manner possible; therefore, the results are 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.

- There are some cases when programs compiled with the TRUNC(OPT) option under Enterprise COBOL could give different results than the same programs compiled under OS/VS COBOL with NOTRUNC. You must actually lose nonzero high-order digits for this difference to appear. For statements where loss of high-order digits might cause a difference in results between Enterprise COBOL and OS/VS COBOL, Enterprise COBOL issues a diagnostic message. If you receive this message, make sure that either the sending items will not contain large numbers or that the receivers are defined with enough digits in the PICTURE clause to handle the largest sending data items.

TRUNC(BIN)

The TRUNC(BIN) option applies to all COBOL language that processes USAGE BINARY data. When TRUNC(BIN) is in effect, all binary items (USAGE COMP, COMP-4, or BINARY) are handled as native hardware binary items, that is, as if they were each individually declared USAGE COMP-5:

- 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 fields is significant.
- DISPLAY will convert the entire content of binary fields with no truncation.

Recommendations: TRUNC(BIN) is the recommended option for programs that use binary values set by other products. Other products, such as IMS, DB2, 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 items. You can use TRUNC(OPT) with CICS programs as long as your data conforms to the PICTURE clause for your BINARY data items.

USAGE COMP-5 has the effect of applying TRUNC(BIN) behavior to individual data items. Therefore, you can avoid the performance overhead of using TRUNC(BIN) for every binary data item by specifying COMP-5 on only some of the binary data items, such as those data items that are passed to non-COBOL programs or other products and subsystems. The use of COMP-5 is not affected by the TRUNC suboption in effect.

Large literals in VALUE clauses: When you use the compiler option TRUNC(BIN), numeric literals specified in VALUE clauses for binary data items (COMP, COMP-4, or BINARY) can generally contain a value of magnitude up to the capacity of the native binary representation (2, 4, or 8 bytes) rather than being limited to the value implied by the number of 9s in the PICTURE clause.

TRUNC example 1

```
+              01  BIN-VAR     PIC S99 USAGE BINARY.
               . . .
                   MOVE 123451 to BIN-VAR
```

The following table shows values of the data items after the MOVE:

Data item	Decimal	Hex	Display
Sender	123451	00|01|E2|3B	123451
Receiver `TRUNC(STD)`	51	00|33	51
Receiver `TRUNC(OPT)`	-7621	E2|3B	2J
Receiver `TRUNC(BIN)`	-7621	E2|3B	762J

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 you compile the program with `TRUNC(BIN)`, the result of the `MOVE` statement is -7621. The reason for the unusual result is that nonzero high-order digits are truncated. Here, the generated code sequence would merely move the lower halfword quantity X'E23B' to the receiver. Because the new truncated value overflows into the sign bit of the binary halfword, the value becomes a negative number.

It is better not to compile this `MOVE` statement with `TRUNC(OPT)`, because 123451 has greater precision than the `PICTURE` clause for `BIN-VAR`. With `TRUNC(OPT)`, the results are again -7621. This is because the best performance was gained by not doing a decimal truncation.

TRUNC example 2

```
01  BIN-VAR     PIC 9(6)  USAGE BINARY
. . .
    MOVE 1234567891 to BIN-VAR
```

The following table shows values of the data items after the `MOVE`:

Data item	Decimal	Hex	Display
Sender	1234567891	49|96|02|D3	1234567891
Receiver `TRUNC(STD)`	567891	00|08|AA|53	567891
Receiver `TRUNC(OPT)`	567891	53|AA|08|00	567891
Receiver `TRUNC(BIN)`	1234567891	49|96|02|D3	1234567891

When you specify `TRUNC(STD)`, the sending data is truncated to six integer digits to conform to the `PICTURE` clause of the `BINARY` receiver.

When you specify `TRUNC(OPT)`, 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 is truncation as if `TRUNC(STD)` were in effect.

When you specify `TRUNC(BIN)`, no truncation occurs because all of the sending data fits into the binary fullword allocated for `BIN-VAR`.

RELATED CONCEPTS
"Formats for numeric data" on page 49

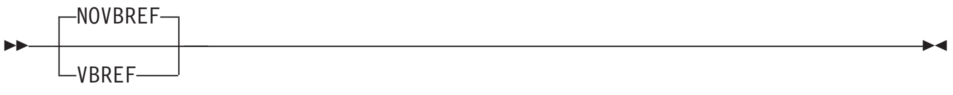
RELATED TASKS
"Compiling with the CICS option" on page 397

RELATED REFERENCES
VALUE clause (*Enterprise COBOL Language Reference*)

VBREF

Use VBREF to get a cross-reference among all verb 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.

VBREF option syntax



Default is: NOVBREF

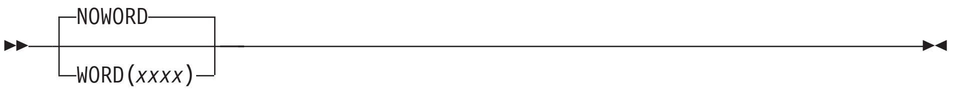
Abbreviations are: None

Use NOVBREF for more efficient compilation.

WORD

Use WORD(*xxxx*) to specify that an alternate reserved-word table is to be used during compilation.

WORD option syntax



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 four standard characters of the name, and *xxxx* can be one to four characters in length.

Alternate reserved-word tables provide changes to the IBM-supplied default reserved-word table. Your systems programmer might have created one or more alternate reserved-word tables for your site. See your systems programmer for the names of alternate reserved-word tables.

Enterprise COBOL provides an alternate reserved-word table (IGYCCICS) specifically for CICS applications. It is set up to flag COBOL words not supported under CICS with an error message. If you want to use this CICS reserved-word table during your compilation, specify the compiler option WORD(CICS).

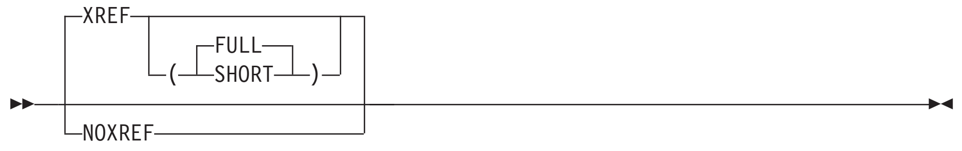
RELATED TASKS
"Compiling with the CICS option" on page 397

RELATED REFERENCES
"Conflicting compiler options" on page 300
"CICS reserved-word table" on page 401

XREF

Use XREF to get a sorted cross-reference listing.

XREF option syntax



Default is: XREF(FULL)

Abbreviations are: X|NOX

You can choose XREF, XREF(FULL), or XREF(SHORT). If you specify XREF without any suboptions, XREF(FULL) is in effect.

EBCDIC data-names and procedure-names are listed in alphanumeric order. DBCS data-names and procedure-names are listed based on their physical order in the program, and appear before the EBCDIC data-names and procedure-names unless the DBCSXREF installation option is selected with a DBCS ordering program. In this case, DBCS data-names and procedure-names are ordered as specified by the DBCS ordering program.

Also included is a section that lists all the program-names that are referenced in your program and the line numbers where they are defined. External program-names are identified.

If you use XREF and SOURCE, cross-reference information is printed on the same line as the original source. Line-number references or other information appears on the right-hand side of the listing page. On the right of source lines that reference an intrinsic function, the letters IFN are printed with the line number of the locations where the function arguments are defined. Information included in the embedded references lets you know if an identifier is undefined (UND) or defined more than once (DUP), if items are implicitly defined (IMP) (such as special registers or figurative constants), or if a program-name is external (EXT).

If you use XREF and NOSOURCE, you get only the sorted cross-reference listing.

XREF(SHORT) prints only the explicitly referenced data items in the cross-reference listing. XREF(SHORT) applies to DBCS data-names and procedure-names as well as to single-byte names.

NOXREF suppresses this listing.

Usage notes

- Group names used in a MOVE CORRESPONDING statement are in the XREF listing. The elementary names in those groups are also listed.
- In the data-name XREF listing, line numbers that are preceded by the letter M indicate that the data item is explicitly modified by a statement on that line.
- XREF listings take additional storage.

RELATED CONCEPTS
Chapter 19, "Debugging," on page 355

RELATED TASKS
"Getting listings" on page 365

RELATED REFERENCES
COBOL compiler options (*Language Environment Debugging Guide*)

YEARWINDOW

Use YEARWINDOW 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.

YEARWINDOW option syntax

```
►►──YEARWINDOW(base-year)──────────────────────────────────────────►◄
```

Default is: YEARWINDOW(1900)

Abbreviations are: YW

base-year represents the first year of the 100-year window. You must specify it with one of the following values:

- 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. The first year of the window is calculated by adding the negative integer to the current year. For example, YEARWINDOW(-80) indicates that the first year of the century window is 80 years before the year at the time the program is run.

Usage notes

- The YEARWINDOW option has no effect unless the DATEPROC option is also in effect.
- 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, if the current year is 2006, the DATEPROC option is in effect, and you use the YEARWINDOW(1900) option, the program will terminate with an error message.

ZWB

If you compile with ZWB, the compiler removes the sign from a signed zoned decimal (DISPLAY) field before comparing this field to an alphanumeric elementary field during execution.

ZWB option syntax

```
          ┌─ZWB───┐
►►────────┤       ├─────────────────────────────────────────────►◄
          └─NOZWB─┘
```

Default is: ZWB

Abbreviations are: None

If the zoned decimal item is a scaled item (that is, it contains the symbol P in its PICTURE string), its use in comparisons is not affected by ZWB. Such items always have their sign removed before the comparison is made to an alphanumeric field.

ZWB affects how a program runs. The same COBOL source program can give different results, depending on this option setting.

Use NOZWB if you want to test input numeric fields for SPACES.

Chapter 18. Compiler-directing statements

Several statements help you to direct the compilation of your program.

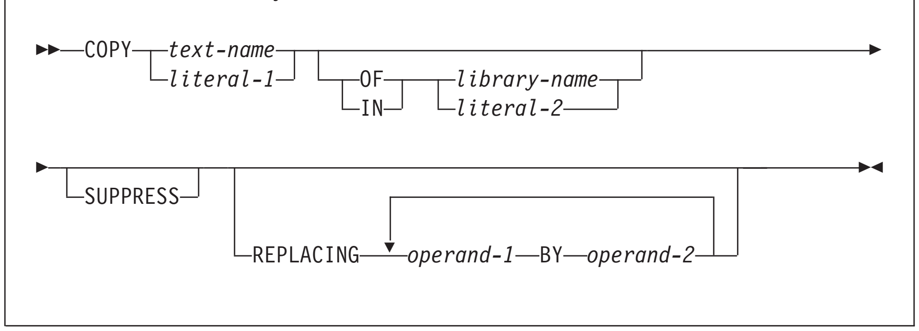
These are the compiler-directing statements:

`BASIS` statement
: This extended source program library statement provides a complete COBOL program as the source for a compilation. For rules of formation and processing, see the description of *text-name* for the `COPY` statement.

`*CONTROL (*CBL)` statement
: This compiler-directing statement selectively suppresses or allows output to be produced. The names `*CONTROL` and `*CBL` are synonymous.

`COPY` statement

COPY statement syntax



This library statement places prewritten text into a COBOL program. A user-defined word can be the same as a *text-name* or a *library-name*. The uniqueness of *text-name* and *library-name* is determined after the formation and conversion rules for a system-dependent name have been applied. If *library-name* is omitted, SYSLIB is assumed.

When compiling with JCL:

text-name, *library-name*, and *literal* are processed as follows:

- The name (which can be one to 30 characters long) is truncated to eight characters. Only the first eight characters of *text-name* and *library-name* are used as the identifying name. These eight characters must be unique within one COBOL library.
- The name is folded to uppercase.
- Hyphens that are not the first or last character are translated to zero (0), and a warning message is given.
- If the first character is numeric, then the characters 1-9 are translated to A-I, zero (0) is converted to J, and a warning message is produced.

For example:

```
COPY INVOICES1Q
COPY "Company-#Employees" IN Personellib
```

In the IN/OF phrase, *library-name* is the ddname that identifies the partitioned data set to be copied from. Use a DD statement such as in the following example to define *library-name*:

```
//COPYLIB DD DSNAME=ABC.COB,VOLUME=SER=111111,
//           DISP=SHR,UNIT=3380
```

To specify more than one copy library, use either JCL or a combination of JCL and the IN/OF phrase. Using just JCL, concatenate data sets on your DD statement for SYSLIB. Alternatively, define multiple DD statements and include the IN/OF phrase on your COPY statements.

The maximum block size for the copy library depends on the device on which your data set resides.

When compiling in the UNIX System Services shell:

When you compile with the cob2 command, copybooks are included from the HFS. *text-name*, *library-name*, and *literal* are processed as follows:

- User-defined words are folded to uppercase. Literals are not. Because UNIX is case sensitive, if your file-name is lowercase or mixed case, you must specify it as a literal.
- When *text-name* is a literal and *library-name* is omitted, *text-name* is used directly: as a file-name, a relative path name, or an absolute path name (if the first character is /). For example:

  ```
  COPY "MyInc"
  COPY "x/MyInc"
  COPY "/u/user1/MyInc"
  ```
- When *text-name* is a user-defined word and an environment variable of that name is defined, the value of the environment variable is used as the name of the file that contains the copybook.

  If an environment variable of that name is not defined, the copybook is searched for as the following names, in this order:

  1. *text-name*.cpy
  2. *text-name*.CPY
  3. *text-name*.cbl
  4. *text-name*.CBL
  5. *text-name*.cob
  6. *text-name*.COB
  7. *text-name*
- When *library-name* is a literal, it is treated as the actual path, relative or absolute, from which to copy file *text-name*.
- When *library-name* is a user-defined word, it is treated as an environment variable. The value of the environment variable is used as the path. If the environment variable is not set, an error occurs.
- If both *library-name* and *text-name* are specified, the compiler forms the path name for the copybook by concatenating *library-name* and *text-name* with a path separator (/) inserted between the two values. For example, suppose you have the following setting for COPY MYCOPY OF MYLIB:

  ```
  export MYCOPY=mystuff/today.cpy
  export MYLIB=/u/user1
  ```

  These settings result in:

  ```
  /u/user1/mystuff/today.cpy
  ```

When *library-name* is an environment variable that identifies the path from which copybooks are to be copied, use an `export` command such as the following example to define *library-name*:

```
export COPYLIB=/u/mystuff/copybooks
```

The name of the environment variable must be uppercase. To specify more than one copy library, set the environment variable to multiple path names delimited by colon (`:`).

When *library-name* is omitted and *text-name* is not an absolute path name, the copybook is searched for in this order:

1. In the current directory
2. In the paths specified on the `-I cob2` option
3. In the paths specified in the SYSLIB environment variable

`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.

`REPLACE` statement
This statement is used to replace source program text.

`SERVICE LABEL` statement
This statement is generated by the CICS translator to indicate control flow,
+ and should be used at the resume point for a call to CEE3SRP. It is not intended for general use.

`SKIP1/2/3` statement
These statements indicate lines to be skipped in the source listing.

`TITLE` statement
This statement specifies that a title (header) should be printed at the top of each page of the source listing.

`USE` statement
The `USE` statement provides *declaratives* to specify these elements:

- Error-handling procedures: `EXCEPTION/ERROR`
- User label-handling procedures: `LABEL`
- Debugging lines and sections: `DEBUGGING`

RELATED TASKS
"Changing the header of a source listing" on page 7
"Specifying compiler options under z/OS" on page 267

"Specifying compiler options under UNIX" on page 280
"Setting environment variables under UNIX" on page 279
"Eliminating repetitive coding" on page 639

RELATED REFERENCES
"cob2 syntax and options" on page 283
COPY statement (*Enterprise COBOL Language Reference*)

Chapter 19. Debugging

You can choose from two approaches to determine the cause of problems in program behavior of your application: source-language debugging or interactive debugging.

For source-language debugging, COBOL provides several language elements, compiler options, and listing outputs that make debugging easier.

If the problem with your program is not easily detected and you do not have a debugger available, you might need to analyze a storage dump of your program.

For interactive debugging, you can also use Debug Tool, which is available in the Full Function offering of this compiler.

Debug Tool offers these productivity enhancements:

- Interactive debugging (in full-screen or line mode), or debugging in batch mode

  During an interactive full-screen mode session, you can use Debug Tool's full-screen services and session panel windows on a 3270 device to debug your program while it is running.
- COBOL-like commands

  For each high-level language supported, commands for coding actions to be taken at breakpoints are provided in a syntax similar to that programming language.
- Mixed-language debugging

  You can debug an application that contains programs written in a different language. Debug Tool automatically determines the language of the program or subprogram being run.
- COBOL-CICS debugging

  Debug Tool supports the debugging of CICS applications in both interactive and batch mode.
- Support for remote debugging

  Workstation users can use the Debug Perspective of WebSphere Developer for System z for debugging programs that reside on z/OS.

RELATED TASKS
"Debugging with source language" on page 356
"Debugging using compiler options" on page 360
"Using the debugger" on page 365
"Getting listings" on page 365
Debug Tool User's Guide

RELATED REFERENCES
Debug Tool Reference and Messages
Formatting and analyzing system dumps (*Language Environment Debugging Guide*)
Debugging example COBOL programs (*Language Environment Debugging Guide*)

Debugging with source language

You can use several COBOL language features to pinpoint the cause of a failure in a program.

If a failing program is part of a large application that is already in production (precluding source updates), write a small test case to simulate the failing part of the program. Code debugging features in the test case to help detect these problems:

- Errors in program logic
- Input-output errors
- Mismatches of data types
- Uninitialized data
- Problems with procedures

RELATED TASKS
"Tracing program logic"
"Finding and handling input-output errors" on page 357
"Validating data" on page 357
"Finding uninitialized data" on page 358
"Generating information about procedures" on page 358

RELATED REFERENCES
Source language debugging (*Enterprise COBOL Language Reference*)

Tracing program logic

Trace the logic of your program by adding `DISPLAY` statements.

For example, if you determine that the problem is in an `EVALUATE` statement or in a set of nested `IF` statements, use `DISPLAY` statements in each path to see the logic flow. If you determine that the calculation of a numeric value is causing the problem, use `DISPLAY` statements to check the value of some interim results.

If you use explicit scope terminators to end statements in your program, the logic is more apparent and therefore easier to trace.

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, disable the `DISPLAY` statements in one of two ways:

- Put an asterisk in column 7 of each `DISPLAY` statement line to convert it to a comment line.
- Put a `D` in column 7 of each `DISPLAY` statement to convert it to a comment line. When you want to reactivate these statements, include a `WITH DEBUGGING MODE` clause in the `ENVIRONMENT DIVISION`; the `D` in column 7 is ignored and the `DISPLAY` statements are implemented.

Before you put the program into production, delete or disable the debugging aids you used and recompile the program. The program will run more efficiently and use less storage.

RELATED CONCEPTS
"Scope terminators" on page 22

RELATED REFERENCES
DISPLAY statement (*Enterprise COBOL Language Reference*)

Finding and handling input-output errors

File status keys can help you determine whether your program errors are due to input-output errors occurring on the storage media.

To use file status keys in debugging, check for a nonzero value in the status key after each input-output statement. If the value is nonzero (as reported in an error message), look at the coding of the input-output procedures in the program. You can also include procedures to correct the error based on the value of the status key.

If you determine that a problem lies in an input-output procedure, include the `USE EXCEPTION/ERROR` declarative to help debug the problem. Then, when a file fails to open, the appropriate `EXCEPTION/ERROR` declarative is performed. The appropriate declarative might be a specific one for the file or one provided for the open attributes `INPUT`, `OUTPUT`, `I-O`, or `EXTEND`.

+ Code each `USE AFTER STANDARD ERROR` statement in a section that follows the
+ `DECLARATIVES` keyword in the `PROCEDURE DIVISION`.

RELATED TASKS
"Coding ERROR declaratives" on page 238
"Using file status keys" on page 239

RELATED REFERENCES
Status key (*Enterprise COBOL Language Reference*)

Validating data

If you suspect that your program is trying to perform arithmetic on nonnumeric data or is receiving the wrong type of data on an input record, use the class test (the class condition) to validate the type of data.

You can use the class test to check whether the content of a data item is
| `ALPHABETIC`, `ALPHABETIC-LOWER`, `ALPHABETIC-UPPER`, `DBCS`, `KANJI`, or `NUMERIC`. If the
| data item is described implicitly or explicitly as `USAGE NATIONAL`, the class test
| checks the national character representation of the characters associated with the
| specified character class.

RELATED TASKS
"Coding conditional expressions" on page 94
"Testing for valid DBCS characters" on page 142

RELATED REFERENCES
Class condition (*Enterprise COBOL Language Reference*)

Finding uninitialized data

Use an INITIALIZE or SET statement to initialize a table or data item when you suspect that a problem might be caused by residual data in those fields.

If the problem happens intermittently and not always with the same data, it could be that a switch was not initialized but is generally set to the right value (0 or 1) by chance. By using a SET statement to ensure that the switch is initialized, you can determine that the uninitialized switch is the cause of the problem or remove it as a possible cause.

RELATED REFERENCES
INITIALIZE statement (*Enterprise COBOL Language Reference*)
SET statement (*Enterprise COBOL Language Reference*)

Generating information about procedures

Generate information about your program or test case and how it is running by coding the USE FOR DEBUGGING declarative. This declarative lets you include statements in the program and indicate when they should be performed when you run your program.

For example, to determine 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 that control passes to that procedure. You can use the counter technique to check items such as these:

- How many times a PERFORM statement runs, and thus whether a particular routine is being used and whether the control structure is correct
- How many times a loop routine runs, and thus whether the loop is executing and whether the number for the loop is accurate

You can use debugging lines or debugging statements or both in your program.

Debugging lines are statements that are identified by a D in column 7. To make debugging lines in your program active, code the WITH DEBUGGING MODE clause on the SOURCE-COMPUTER line in the ENVIRONMENT DIVISION. Otherwise debugging lines are treated as comments.

Debugging statements are the statements that are coded in the DECLARATIVES section of the PROCEDURE DIVISION. Code each USE FOR DEBUGGING declarative in a separate section. Code the debugging statements as follows:

- Only in a DECLARATIVES section.
- Following the header USE FOR DEBUGGING.
- Only in the outermost program; they are not valid in nested programs. Debugging statements are also never triggered by procedures that are contained in nested programs.

To use debugging statements in your program, you must include the WITH DEBUGGING MODE clause and use the DEBUG runtime option. However, you cannot use the USE FOR DEBUGGING declarative in a program that you compile with the THREAD option.

The WITH DEBUGGING MODE clause and the TEST compiler option (with any suboption value other than NONE) are mutually exclusive. If both are present, the WITH DEBUGGING MODE clause takes precedence.

"Example: USE FOR DEBUGGING"

RELATED REFERENCES
SOURCE-COMPUTER paragraph (*Enterprise COBOL Language Reference*)
Debugging lines (*Enterprise COBOL Language Reference*)
Debugging sections (*Enterprise COBOL Language Reference*)
DEBUGGING declarative (*Enterprise COBOL Language Reference*)

Example: USE FOR DEBUGGING

This example shows the kind of statements that are needed to use a DISPLAY statement and a USE FOR DEBUGGING declarative to test a program.

The DISPLAY statement writes information to the terminal or to an output data set. The USE FOR DEBUGGING declarative is used with a counter to show how many times a routine runs.

```
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.
```

The DISPLAY statement in the DECLARATIVES SECTION issues this message every time the procedure Some-Routine runs:

```
  Trace For Procedure-Name : Some-Routine 22
```

The number at the end of the message, 22, is the value accumulated in the data item Total; it indicates the number of times Some-Routine has run. The statements in the debugging declarative are performed before the named procedure runs.

You can also use the DISPLAY statement to trace program execution and show the flow through the program. You do this by dropping Total from the DISPLAY statement and changing the USE FOR DEBUGGING declarative in the DECLARATIVES SECTION to:

```
USE FOR DEBUGGING ON ALL PROCEDURES.
```

As a result, a message is displayed before each nondebugging procedure in the outermost program runs.

Debugging using compiler options

You can use certain compiler options to help you find errors in your program, find various elements in your program, obtain listings, and prepare your program for debugging.

You can find the following errors by using compiler options (the options are shown in parentheses):

- Syntax errors such as duplicate data-names (NOCOMPILE)
- Missing sections (SEQUENCE)
- Invalid subscript values (SSRANGE)

You can use find these elements in your program by using compiler options:

- Error messages and locations of the associated errors (FLAG)
- Program entity definitions and references (XREF)
- Data items in the DATA DIVISION (MAP)
- Verb references (VBREF)

You can get a copy of your source (SOURCE) or a listing of generated code (LIST).

You prepare your program for debugging by using the TEST compiler option.

RELATED TASKS
"Finding coding errors"
"Finding line sequence problems" on page 361
"Checking for valid ranges" on page 361
"Selecting the level of error to be diagnosed" on page 362
"Finding program entity definitions and references" on page 363
"Listing data items" on page 364
"Getting listings" on page 365

RELATED REFERENCES
Chapter 17, "Compiler options," on page 297

Finding coding errors

Use the NOCOMPILE option to compile conditionally or to only check syntax. When used with the SOURCE option, NOCOMPILE produces a listing that will help you find coding mistakes such as missing definitions, improperly defined data items, and duplicate data-names.

If you are compiling in the TSO foreground, you can send the messages to your screen by using the TERM compiler option and defining your data set as the SYSTERM data set.

Checking syntax only: To only check the syntax of your program, and not produce object code, use NOCOMPILE without parameters. If you also specify the SOURCE option, the compiler produces a listing.

The following compiler options are suppressed when you use NOCOMPILE without parameters: DECK, OFFSET, LIST, OBJECT, OPTIMIZE, SSRANGE, and TEST.

Compiling conditionally: To compile conditionally, use NOCOMPILE(*x*), where *x* is one of the severity levels of errors. Your program is compiled if all the errors are of a lower severity than *x*. The severity levels that you can use, from highest to lowest, are S (severe), E (error), and W (warning).

If an error of level *x* or higher occurs, the compilation stops and your program is only checked for syntax.

RELATED REFERENCES
"COMPILE" on page 305

Finding line sequence problems

Use the SEQUENCE compiler option to find statements that are out of sequence. Breaks in sequence indicate that a section of a source program was moved or deleted.

When you use SEQUENCE, the compiler checks the source statement numbers to determine whether they are in ascending sequence. Two asterisks are placed beside statement numbers that are out of sequence. The total number of these statements is printed as the first line in the diagnostics after the source listing.

RELATED REFERENCES
"SEQUENCE" on page 333

Checking for valid ranges

Use the SSRANGE compiler option to check whether addresses fall within proper ranges.

SSRANGE causes the following addresses to be checked:

- Subscripted or indexed data references: Is the effective address of the desired element 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): Is the actual length positive and within the maximum defined length for the group data item?
- Reference-modified data references: Are the offset and length positive? Is the sum of the offset and length within the maximum length for the data item?

If the SSRANGE option is in effect, checking is performed at run time if both of the following conditions are true:

- The COBOL statement that contains the indexed, subscripted, variable-length, or reference-modified data item is performed.
- The CHECK runtime option is ON.

If an address is generated outside the range of the data item that contains the referenced data, an error message is generated and the program stops. The message identifies the table or identifier that was referenced and the line number where the error occurred. Additional information is provided depending on the type of reference that caused the error.

If all subscripts, indices, and reference modifiers in a given data reference are literals and they result in a reference outside the data item, the error is diagnosed at compile time regardless of the setting of the SSRANGE option.

Performance consideration: SSRANGE can somewhat degrade performance because of the extra overhead to check each subscripted or indexed item.

RELATED REFERENCES
"SSRANGE" on page 337
"Performance-related compiler options" on page 633

Selecting the level of error to be diagnosed

Use the FLAG compiler option to specify the level of error to be diagnosed during compilation and to indicate whether error messages are to be embedded in the listing. Use FLAG(I) or FLAG(I,I) to be notified of all errors.

Specify as the first parameter the lowest severity level of the syntax-error messages to be issued. Optionally specify the second parameter as the lowest level of the syntax-error messages to be embedded in the source listing. This severity level must be the same or higher than the level for the first parameter. If you specify both parameters, you must also specify the SOURCE compiler option.

Table 49. **Severity levels of compiler messages**

Severity level	Resulting messages
U (unrecoverable)	U messages only
S (severe)	All S and U messages
E (error)	All E, S, and U messages
W (warning)	All W, E, S, and U messages
I (informational)	All messages

When you specify the second parameter, each syntax-error message (except a U-level message) is embedded in the source listing at the point where the compiler had enough information to detect that error. All embedded messages (except those issued by the library compiler phase) directly follow the statement to which they refer. The number of the statement that had the error is also included with the message. Embedded messages are repeated with the rest of the diagnostic messages at the end of the source listing.

When you specify the NOSOURCE compiler option, the syntax-error messages are included only at the end of the listing. Messages for unrecoverable errors are not embedded in the source listing, because an error of this severity terminates the compilation.

"Example: embedded messages"

RELATED TASKS
"Generating a list of compiler error messages" on page 275

RELATED REFERENCES
"Severity codes for compiler error messages" on page 276
"Messages and listings for compiler-detected errors" on page 275
"FLAG" on page 314

Example: embedded messages

The following example shows the embedded messages generated by specifying a second parameter to the FLAG option. Some messages in the summary apply to more than one COBOL statement.

```
  LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+  Map and Cross Reference
.
.
.
  090671**            /
  090672**            *****************************************************************
  090673**            ***         I N I T I A L I Z E   P A R A G R A P H           **
  090674**            ***  Open files. Accept date, time and format header lines.    **
  090675**            ***   Load location-table.                                     **
  090676**            *****************************************************************
  090677**               100-initialize-paragraph.
  090678**                 move spaces to ws-transaction-record                                  IMP 331
  090679**                 move spaces to ws-commuter-record                                     IMP 307
  090680**                 move zeroes to commuter-zipcode                                       IMP 318
  090681**                 move zeroes to commuter-home-phone                                    IMP 319
  090682**                 move zeroes to commuter-work-phone                                    IMP 320
  090683**                 move zeroes to commuter-update-date                                   IMP 324
  090684**                 open input update-transaction-file                                    204
==090684==> IGYPS2052-S An error was found in the definition of file "LOCATION-FILE".  The
                        reference to this file was discarded.
  090685**                     location-file                                                     193
  090686**                     i-o commuter-file                                                 181
  090687**                     output print-file                                                 217
  090688**                 if commuter-file-status not = "00" and not = "97"                     241
  090689**    1              display "100-OPEN"
  090690**    1              move 100 to comp-code                                               231
  090691**    1              perform 500-vsam-error                                              91069
  090692**    1              perform 900-abnormal-termination                                    91114
  090693**                 end-if
  090694**                 accept ws-date from date                                              UND
==090694==> IGYPS2121-S "WS-DATE" was not defined as a data-name.  The statement was discarded.
  090695**                 move corr ws-date to header-date                                      UND 455
==090695==> IGYPS2121-S "WS-DATE" was not defined as a data-name.  The statement was discarded.
  090696**                 accept ws-time from time                                              UND
==090696==> IGYPS2121-S "WS-TIME" was not defined as a data-name.  The statement was discarded.
  090697**                 move corr ws-time to header-time                                      UND 449
==090697==> IGYPS2121-S "WS-TIME" was not defined as a data-name.  The statement was discarded.
  090698**                 read location-file                                                    193
==090698==> IGYPS2053-S An error was found in the definition of file "LOCATION-FILE".  This
                        input/output statement was discarded.
  090699**                   at end
  090700**    1                  set location-eof to true                                        256
  090701**                 end-read
.
.
.

LineID  Message code  Message text
         IGYSC0090-W   1700 sequence errors were found in this program.
         IGYSC3002-I   A severe error was found in the program.  The "OPTIMIZE" compiler option was cancelled.
    160  IGYDS1089-S   "ASSIGNN" was invalid.  Scanning was resumed at the next area "A" item, level-number, or
                       the start of the next clause.
    193  IGYGR1207-S   The "ASSIGN" clause was missing or invalid in the "SELECT" entry for file "LOCATION-FILE".
                       The file definition was discarded.
    269  IGYDS1066-S   "REDEFINES" object "WS-DATE" was not the immediately preceding level-1 data item.
                       The "REDEFINES" clause was discarded.
  90602  IGYPS2052-S   An error was found in the definition of file "LOCATION-FILE".  The reference to this file
                       was discarded. Same message on line:  90684
  90694  IGYPS2121-S   "WS-DATE" was not defined as a data-name.  The statement was discarded.
                       Same message on line:  90695
  90696  IGYPS2121-S   "WS-TIME" was not defined as a data-name.  The statement was discarded.
                       Same message on line:  90697
  90698  IGYPS2053-S   An error was found in the definition of file "LOCATION-FILE".  This input/output statement
                       was discarded. Same message on line:  90709
Messages    Total    Informational    Warning    Error    Severe    Terminating
Printed:      13           1             1                  11
* Statistics for COBOL program IGYTCARA:
*    Source records = 1735
*    Data Division statements = 287
*    Procedure Division statements = 471
End of compilation 1,  program IGYTCARA,  highest severity 12.
Return code 12
```

Finding program entity definitions and references

Use the XREF(FULL) compiler option to find out where a data-name, procedure-name, or program-name is defined and referenced. The sorted cross-reference includes the line number where the entity was defined and the line numbers of all references to it.

To include only the explicitly referenced data items, use the XREF(SHORT) option.

Use both the XREF (either FULL or SHORT) and the SOURCE options to print a modified cross-reference to the right of the source listing. This embedded cross-reference shows the line number where the data-name or procedure-name was defined.

If your program contains DBCS user-defined words, these user-defined words are listed before the alphabetic list of EBCDIC user-defined words.

If a DBCS ordering program is specified in the DBCSXREF installation option, the DBCS user-defined words are listed in order according to the specified DBCS collating sequence. Otherwise, the DBCS user-defined words are listed in physical order according to their appearance in the source program.

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.

"Example: XREF output - data-name cross-references" on page 387
"Example: XREF output - program-name cross-references" on page 388
"Example: embedded cross-reference" on page 388

RELATED TASKS
"Getting listings" on page 365

RELATED REFERENCES
"XREF" on page 347

Listing data items

Use the MAP compiler option to produce a listing of the DATA DIVISION items and all implicitly declared items. Use the MAP output to locate the contents of a data item in a system dump.

When you use the MAP option, an embedded MAP summary that contains condensed MAP information is generated to the right of the COBOL source data declaration. When both XREF data and an embedded MAP summary are on the same line, the embedded summary is printed first.

You can select or inhibit parts of the MAP listing and embedded MAP summary by using *CONTROL MAP|NOMAP (or *CBL MAP|NOMAP) statements throughout the source. For example:

```
*CONTROL NOMAP
    01  A
    02  B
*CONTROL MAP
```

"Example: MAP output" on page 370

RELATED TASKS
"Getting listings" on page 365

RELATED REFERENCES
"MAP" on page 320

Using the debugger

The Debug Tool debugger is provided with the Full Function feature of Enterprise COBOL. Use the `TEST` compiler option to prepare your COBOL program so that you can step through the executable program with the debugger.

For remote debugging, the Debug Perspective of WebSphere Developer for System z provides the client graphical user interface to the debug information provided by the Debug Tool engine running under z/OS or UNIX.

You can specify the `TEST` suboption `SEPARATE` to have the symbolic information tables for Debug Tool generated in a data set separate from your object module. Also, you can enable your COBOL program for debugging using overlay hooks (*production debugging*), rather than compiled-in hooks, which have some performance degradation even when the runtime `TEST` option is off. To use this function, compile with `TEST(NONE,SYM)`.

Specify the `NOOPTIMIZE` option to get the most debugging function. Specify the `OPTIMIZE(STD)` or `OPTIMIZE(FULL)` option for a more efficient program, but with these restrictions on debugging:

- The `GOTO` Debug Tool command is not supported.
- Except for the `DESCRIBE ATTRIBUTES` command, Debug Tool commands cannot refer to any data item that was discarded from your program by the `OPTIMIZE(FULL)` option.

RELATED TASKS
Preparing your program for debugging (*Debug Tool User's Guide*)

RELATED REFERENCES
"TEST" on page 338

Getting listings

Get the information that you need for debugging by requesting the appropriate compiler listing with the use of compiler options.

Attention: The listings produced by the compiler are not a programming interface and are subject to change.

Table 50. **Using compiler options to get listings**

Use	Listing	Contents	Compiler option
To check a list of the options in effect for the program, statistics about the content of the program, and diagnostic messages about the compilation	Short listing	• List of options in effect for the program • Statistics about the content of the program • Diagnostic messages about the compilation[1]	`NOSOURCE`, `NOXREF`, `NOVBREF`, `NOMAP`, `NOOFFSET`, `NOLIST`
To aid in testing and debugging your program; to have a record after the program has been debugged	Source listing	Copy of your source	"SOURCE" on page 334

Table 50. **Using compiler options to get listings** *(continued)*

Use	Listing	Contents	Compiler option
To find certain data items in a storage dump; to see the final storage allocation after reentrancy or optimization has been accounted for; to see where programs are defined and check their attributes	Map of DATA DIVISION items	All DATA DIVISION items and all implicitly declared items Embedded map summary (in the right margin of the listing for lines in the DATA DIVISION that contain data declarations) Nested program map (if the program contains nested programs)	"MAP" on page 320[2]
To find where a name is defined, referenced, or modified; to determine the context (such as whether a verb was used in a PERFORM block) in which a procedure is referenced	Sorted cross-reference listing of names	Data-names, procedure-names, and program-names; references to these names Embedded modified cross-reference: provides the line number where the data-name or procedure-name was defined	"XREF" on page 347[2,3]
To find the failing verb in a program or the address in storage of a data item that was moved during the program	PROCEDURE DIVISION code and assembler code produced by the compiler[3]	Generated code	"LIST" on page 319[2,4]
To verify you still have a valid logic path after you move or add PROCEDURE DIVISION sections	Condensed PROCEDURE DIVISION listing	Condensed verb listing, global tables, WORKING-STORAGE information, and literals	"OFFSET" on page 326
To find an instance of a certain verb	Alphabetic listing of verbs	Each verb used, number of times each verb was used, line numbers where each verb was used	"VBREF" on page 346

1. To eliminate messages, turn off the options (such as FLAG) that govern the level of compile diagnostic information.
2. To use your line numbers in the compiled program, use the NUMBER compiler option. The compiler checks the sequence of your source statement line numbers in columns 1 through 6 as the statements are read in. When it finds a line number 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 included in the compilation listing.
3. The context of the procedure reference is indicated by the characters preceding the line number.
4. You can control the listing of generated object code by selectively placing *CONTROL LIST and *CONTROL NOLIST (or equivalently, *CBL LIST and *CBL NOLIST) statements in your source. Note that the *CONTROL statement is different than the PROCESS (or CBL) statement.

   The output is generated if:

   - You specify the COMPILE option (or the NOCOMPILE(*x*) option is in effect and an error level *x* or higher does not occur).
   - You do not specify the OFFSET option. OFFSET and LIST are mutually exclusive options with OFFSET taking precedence.

"Example: short listing"
"Example: SOURCE and NUMBER output" on page 369
"Example: MAP output" on page 370
"Example: embedded map summary" on page 372
"Example: nested program map" on page 375
"Example: XREF output - data-name cross-references" on page 387
"Example: XREF output - program-name cross-references" on page 388
"Example: embedded cross-reference" on page 388
"Example: OFFSET compiler output" on page 389
"Example: VBREF compiler output" on page 390

RELATED TASKS
"Generating a list of compiler error messages" on page 275
"Reading LIST output" on page 376
Debugging COBOL programs (*Language Environment Debugging Guide*)

RELATED REFERENCES
"Messages and listings for compiler-detected errors" on page 275

Example: short listing

The note numbers shown in the following listing correspond to the numbered explanations that follow the listing. For illustrative purposes, some errors that cause diagnostic messages were deliberately introduced.

```
Invocation parameters:      (Note 1)
C,NODU,FLAG(I),X,MAP,NOLIST,RENT,OPT,SSR,TEST(NONE,SYM),
PROCESS(CBL) statements:
CBL NODECK                  (Note 2)
CBL NOADV,NODYN,NONAME,NONUMBER,QUOTE,SEQ,DUMP
CBL NOSOURCE, NOXREF, NOVBREF, NOMAP, NOOFFSET, NOLIST          IGY00010
Options in effect:          (Note 3)
    NOADATA
    NOADV
      QUOTE
      ARITH(COMPAT)
    NOAWO
      BUFSIZE(4096)
    NOCICS
      CODEPAGE(1140)
      COMPILE
    NOCURRENCY
      DATA(31)
    NODATEPROC
      DBCS
    NODECK
    NODIAGTRUNC
    NODLL
    NODUMP
    NODYNAM
    NOEXIT
    NOEXPORTALL
    NOFASTSRT
      FLAG(I)
    NOFLAGSTD
      INTDATE(ANSI)
      LANGUAGE(EN)
    NOLIB
      LINECOUNT(60)
    NOLIST
    NOMAP
    NOMDECK
    NONAME
      NSYMBOL(NATIONAL)
    NONUMBER
      NUMPROC(NOPFD)
      OBJECT
    NOOFFSET
      OPTIMIZE(STD)
      OUTDD(SYSOUT)
      PGMNAME(COMPAT)
      RENT
      RMODE(AUTO)
      SEQUENCE
      SIZE(MAX)
    NOSOURCE
      SPACE(1)
    NOSQL
      SQLCCSID
      SSRANGE
    NOTERM
      TEST(NONE,SYM,NOSEPARATE)
    NOTHREAD
      TRUNC(STD)
    NOVBREF
    NOWORD
    NOXREF
      YEARWINDOW(1900)
      ZWB
```

```
  LineID  Message code  Message text   (Note 4)
          IGYDS0139-W   Diagnostic messages were issued during processing of compiler options.
                        These messages are located at the beginning of the listing.
          IGYSC0090-W   3 sequence errors were found in this program.
     160  IGYDS1089-S   "ASSIGNN" was invalid.  Scanning was resumed at the next area "A" item,
                        level-number,or the start of the next clause.
     193  IGYGR1207-S   The "ASSIGN" clause was missing or invalid in the "SELECT" entry for file
                        "LOCATION-FILE".  The file definition was discarded.
     269  IGYDS1066-S   "REDEFINES" object "WS-DATE" was not the immediately preceding level-1 data item.
                        The "REDEFINES" clause was discarded.
     901  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:   1008
  Messages    Total    Informational    Warning    Error    Severe    Terminating   (Note 5)
Printed:      14                          3                  11
  * Statistics for COBOL program IGYTCARA:    (Note 6)
*    Source records = 1735
*    Data Division statements = 287
*    Procedure Division statements = 471
End of compilation 1,  program IGYTCARA,  highest severity 12.  (Note 7)
Return code 12
```

(1) Message about options passed to the compiler at compiler invocation. This message does not appear if no options were passed.

LIST Produces an assembler-language expansion of the source code.

(2) Options coded in the PROCESS (or CBL) statement.

TEST(NONE,SYM,SEPARATE)
The program was compiled for use with Debug Tool or formatted dumps.

OFFSET Produces a condensed listing of the PROCEDURE DIVISION.

MAP Produces a map report of the items in the DATA DIVISION.

(3) Status of options at the start of this compilation.

(4) Program diagnostics. The first message refers you to any library phase diagnostics. Diagnostics for the library phase are presented at the beginning of the listing.

(5) Count of diagnostic messages in this program, grouped by severity level.

(6) Program statistics for the program IGYTCARA.

(7) Program statistics for the compilation unit. When you perform a batch compilation, the return code is the highest message severity level for the entire compilation.

Example: SOURCE and NUMBER output

In the portion of the listing shown below, the programmer numbered two of the statements out of sequence. The note numbers in the listing correspond to numbered explanations that follow the listing.

```
LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Cross-Reference (1)
  (2)    (3)    (4)
               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.
               087700   000-do-main-logic.
               087800     display "PROGRAM IGYTCARA - Beginning"
               087900     perform 050-create-vsam-master-file.                                   90633
               088150     display "perform 050-create-vsam-master finished".
088151**       088125     perform 100-initialize-paragraph                                       90677
               088200     display "perform 100-initialize-paragraph finished"
               088300     read update-transaction-file into ws-transaction-record                204 331
               088400         at end
            1  088500       set transaction-eof to true                                          254
               088600     end-read
               088700     display "READ completed"
               088800     perform until transaction-eof                                          254
            1  088900       display "inside perform until loop"
            1  089000       perform 200-edit-update-transaction                                  90733
            1  089100       display "After perform 200-edit   "
            1  089200       if no-errors                                                         365
            2  089300         perform 300-update-commuter-record                                 90842
            2  089400         display "After perform 300-update "
            1  089650       else
089651**    2  089600         perform 400-print-transaction-errors                               90995
            2  089700         display "After perform 400-errors "
            1  089800       end-if
            1  089900       perform 410-re-initialize-fields                                     91056
            1  090000       display "After perform 410-reinitialize"
            1  090100       read update-transaction-file into ws-transaction-record              204 331
            1  090200           at end
            2  090300         set transaction-eof to true                                        254
            1  090400       end-read
            1  090500       display "After '2nd READ'   "
               090600     end-perform
```

(1) Scale line, which labels Area A, Area B, and source-code column numbers

(2) Source-code line number assigned by the compiler

(3) Program (PL) and statement (SL) nesting level

(4) Columns 1 through 6 of program (the sequence number area)

Example: MAP output

The following example shows output from the MAP option. The numbers used in the explanation below correspond to the numbers that annotate the output.

```
Data Division Map

(1)
Data Definition Attribute codes (rightmost column) have the following meanings:
    D = Object of OCCURS DEPENDING    G = GLOBAL                             S = Spanned file
    E = EXTERNAL                      O = Has OCCURS clause                  U = Undefined format file
    F = Fixed-length file             OG= Group has own length definition    V = Variable-length file
    FB= Fixed-length blocked file     R = REDEFINES                          VB= Variable-length blocked file

(2)      (3) (4)                           (5)        (6)  (7)
                                                                        (8)           (9)            (10)
Source   Hierarchy and                     Base       Hex-Displacement  Asmblr Data                  Data Def
LineID   Data Name                         Locator    Blk   Structure   Definition    Data Type      Attributes
     4  PROGRAM-ID IGYTCARA------------------------------------------------------------------------------------*
   181   FD COMMUTER-FILE . . . . . . . .                                             VSAM            F
   183   1  COMMUTER-RECORD . . . . . . .  BLF=00000  000               DS OCL80      Group
   184     2  COMMUTER-KEY. . . . . . . .  BLF=00000  000   0 000 000   DS 16C        Display
   185     2  FILLER. . . . . . . . . . .  BLF=00000  010   0 000 010   DS 64C        Display
   187   FD COMMUTER-FILE-MST . . . . . .                                             VSAM            F
   189   1  COMMUTER-RECORD-MST . . . . .  BLF=00001  000               DS OCL80      Group
   190     2  COMMUTER-KEY-MST. . . . . .  BLF=00001  000   0 000 000   DS 16C        Display
   191     2  FILLER. . . . . . . . . . .  BLF=00001  010   0 000 010   DS 64C        Display
   193   FD LOCATION-FILE . . . . . . . .                                             QSAM            FB
   198   1  LOCATION-RECORD . . . . . . .  BLF=00002  000               DS OCL80      Group
   199     2  LOC-CODE. . . . . . . . . .  BLF=00002  000   0 000 000   DS 2C         Display
   200     2  LOC-DESCRIPTION . . . . . .  BLF=00002  002   0 000 002   DS 20C        Display
   201     2  FILLER. . . . . . . . . . .  BLF=00002  016   0 000 016   DS 58C        Display
   204   FD UPDATE-TRANSACTION-FILE . . .                                             QSAM            FB
   209   1  UPDATE-TRANSACTION-RECORD . .  BLF=00003  000               DS 80C        Display
   217   FD PRINT-FILE. . . . . . . . . .                                             QSAM            FB
   222   1  PRINT-RECORD. . . . . . . . .  BLF=00004  000               DS 121C       Display
   229   1  WORKING-STORAGE-FOR-IGYCARA .  BLW=00000  000               DS 1C         Display
   231   77 COMP-CODE . . . . . . . . . .  BLW=00000  008               DS 2C         Binary
   232   77 WS-TYPE . . . . . . . . . . .  BLW=00000  010               DS 3C         Display
   235   1  I-F-STATUS-AREA . . . . . . .  BLW=00000  018               DS OCL2       Group
   236     2  I-F-FILE-STATUS . . . . . .  BLW=00000  018   0 000 000   DS 2C         Display
   237     88 I-O-SUCCESSFUL. . . . . . .
   240   1  STATUS-AREA . . . . . . . . .  BLW=00000  020               DS OCL8       Group
   241     2  COMMUTER-FILE-STATUS. . . .  BLW=00000  020   0 000 000   DS 2C         Display
   242     88 I-O-OKAY. . . . . . . . . .
   243     2  COMMUTER-VSAM-STATUS. . . .  BLW=00000  022   0 000 002   DS OCL6       Group
   244       3  VSAM-R15-RETURN-CODE. . .  BLW=00000  022   0 000 002   DS 2C         Binary
   245   77 UNUSED-DATA-ITEM. . . . . . .  BLW=XXXXX  022               DS 10C        Display  (11)
```

(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 level 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 in source order.

(5) Base locator used for this data item.

(6) Hexadecimal displacement from the beginning of the base locator value.

(7) Hexadecimal displacement from the beginning of the containing structure.

(8) Pseudoassembler code showing how the data is defined. When a structure contains variable-length fields, the maximum length of the structure is shown.

(9) Data type and usage.

(10) Data definition attribute codes. The definitions are explained at the top of the `DATA DIVISION` map.

(11) `UNUSED-DATA-ITEM` was not referenced in the `PROCEDURE DIVISION`. Because `OPTIMIZE(FULL)` was specified, `UNUSED-DATA-ITEM` was deleted, resulting in the base locator being set to `XXXXX`.

"Example: embedded map summary"
"Example: nested program map" on page 375

RELATED REFERENCES
"Terms used in MAP output" on page 373
"Symbols used in LIST and MAP output" on page 374

Example: embedded map summary

The following example shows an embedded map summary from specifying the MAP option. The summary appears in the right margin of the listing for lines in the DATA DIVISION that contain data declarations.

```
000002    Identification Division.
000003
000004    Program-id.    IGYTCARA.
  . . .
000177    Data division.
000178    File section.
000179
000180
000181    FD  COMMUTER-FILE
000182        record 80 characters.                                     (1)       (2) (3)       (4)
  . . .
000222      01 print-record                pic x(121).                 BLF=00004+000           121C
  . . .
000228    Working-storage section.
000229      01 Working-storage-for-IGYCARA    pic x.                   BLW=00000+000           1C
000230
000231      77 comp-code                   pic S9999 comp.             BLW=00000+008           2C
000232      77 ws-type                     pic x(3)   value spaces.    BLW=00000+010           3C
000233
000234
000235      01 i-f-status-area.                                        BLW=00000+018           0CL2
000236        05 i-f-file-status           pic x(2).                   BLW=00000+018,0000000   2C
000237          88 i-o-successful          value zeroes.
000238
000239
000240      01 status-area.                                            BLW=00000+020           0CL8
000241        05 commuter-file-status      pic x(2).                   BLW=00000+020,0000000   2C
000242          88 i-o-okay                value zeroes.
000243        05 commuter-vsam-status.                                 BLW=00000+022,0000002   0CL6
000244          10 vsam-r15-return-code    pic 9(2) comp.              BLW=00000+022,0000002   2C
000245          10 vsam-function-code      pic 9(1) comp.              BLW=00000+024,0000004   2C
000246          10 vsam-feedback-code      pic 9(3) comp.              BLW=00000+026,0000006   2C
000247
000248      77 update-file-status          pic xx.                     BLW=00000+028           2C
000249      77 loccode-file-status         pic xx.                     BLW=00000+030           2C
000250      77 updprint-file-status        pic xx.                     BLW=00000+038           2C
000251
000252      01 flags.                                                  BLW=00000+040           0CL3
000253        05 transaction-eof-flag      pic x value space.          BLW=00000+040,0000000   1C
000254            88 transaction-eof       value "Y".
000255        05 location-eof-flag         pic x value space.          BLW=00000+041,0000001   1C
000256            88 location-eof          value "Y".
000257        05 transaction-match-flag    pic x.                      BLW=00000+042,0000002   1C
  . . .
000876    procedure division.
000877      000-do-main-logic.
000878        display "PROGRAM IGYTCARA - Beginning"
000879        perform 050-create-vsam-master-file.
```

(1) Base locator used for this data item

(2) Hexadecimal displacement from the beginning of the base locator value

(3) Hexadecimal displacement from the beginning of the containing structure

(4) Pseudoassembler code showing how the data is defined

RELATED REFERENCES
"Symbols used in LIST and MAP output" on page 374

Terms used in MAP output

The following table describes the terms used in the listings produced by the MAP compiler option.

Table 51. **Terms used in MAP output**

Term	Definition	Description
ALPHABETIC	DS *n*C	Alphabetic data item (PICTURE A)
ALPHA-EDIT	DS *n*C	Alphabetic-edited data item
AN-EDIT	DS *n*C	Alphanumeric-edited data item
BINARY	DS 1H[2], 1F[2], 2F[2], 2C, 4C, or 8C	Binary data item (USAGE BINARY, COMPUTATIONAL, or COMPUTATIONAL-5)
COMP-1	DS 4C	Single-precision internal floating-point data item (USAGE COMPUTATIONAL-1)
COMP-2	DS 8C	Double-precision internal floating-point data item (USAGE COMPUTATIONAL-2)
DBCS	DS *n*C	DBCS data item (USAGE DISPLAY-1)
DBCS-EDIT	DS *n*C	DBCS-edited data item (USAGE DISPLAY-1)
| DISP-FLOAT	DS *n*C	Display floating-point data item (USAGE DISPLAY)
DISPLAY	DS *n*C	Alphanumeric data item (PICTURE X)
| DISP-NUM	DS *n*C	Zoned decimal data item (USAGE DISPLAY)
| DISP-NUM-EDIT	DS *n*C	Numeric-edited data item (USAGE DISPLAY)
FD		File definition
| FUNCTION-PTR	DS *n*C	Function pointer (USAGE FUNCTION-POINTER)
| GROUP	DS 0CL*n*[1]	Fixed-length alphanumeric group data item
| GRP-VARLEN	DS 0CL*n*[1]	Variable-length alphanumeric group data item
| INDEX	DS *n*C	Index data item (USAGE INDEX)
| INDEX-NAME	DS *n*C	Index name
| NATIONAL	DS *n*C	Category national data item (USAGE NATIONAL)
| NAT-EDIT	DS *n*C	National-edited data item (USAGE NATIONAL)
| NAT-FLOAT	DS *n*C	National floating-point data item (USAGE NATIONAL)
| NAT-GROUP	DS 0CL*n*[1]	National group (GROUP-USAGE NATIONAL)
| NAT-GRP-VARLEN	DS 0CL*n*[1]	National variable-length group (GROUP-USAGE NATIONAL)
| NAT-NUM	DS *n*C	National decimal data item (USAGE NATIONAL)
| NAT-NUM-EDIT	DS *n*C	National numeric-edited data item (USAGE NATIONAL)
| OBJECT-REF	DS *n*C	Object-reference data item (USAGE OBJECT REFERENCE)
PACKED-DEC	DS *n*P	Internal decimal data item (USAGE PACKED-DECIMAL or COMPUTATIONAL-3)
| POINTER	DS *n*C	Pointer data item (USAGE POINTER)
| PROCEDURE-PTR	DS *n*C	Procedure pointer (USAGE PROCEDURE-POINTER)
SD		Sort file definition
VSAM, QSAM, LINESEQ		File processing method

Table 51. **Terms used in MAP output** *(continued)*

Term	Definition	Description
1-49, 77		Level-numbers for data descriptions
66		Level-number for RENAMES
88		Level-number for condition-names
1. *n* is the size in bytes for fixed-length groups and the maximum size in bytes for variable-length groups. 2. If the SYNCHRONIZED clause appears, these fields are used.		

Symbols used in LIST and MAP output

The following table describes the symbols used in the listings produced by the LIST or MAP option.

Table 52. **Symbols used in LIST and MAP output**

Symbol	Definition
APBdisp=n[1]	ALL subscript parameter block displacement
AVN=n[1]	Variable name cell for ALTER statement
BL=n[1]	Base locator for special registers
BLA=n[1]	Base locator for alphanumeric temporaries[4]
BLF=n[1]	Base locator for files
BLK=n[1]	Base locator for LOCAL-STORAGE
BLL=n[1]	Base locator for LINKAGE SECTION
BLM=n[1]	Base locator for factory data
BLO=n[1]	Base locator for object instance data
BLS=n[1]	Base locator for sort items
BLT=n[1]	Base locator for XML-TEXT and XML-NTEXT
BLV=n[1]	Base locator for variably located data
BLW=n[1]	Base locator for WORKING-STORAGE
BLX=n[1]	Base locator for external data
CBL=n[1]	Base locator for constant global table (CGT)
CLLE=@=n[1]	Load list entry address in TGT
CLO=n[1]	Class object cell
DOV=n[1]	DSA overflow cell
EVALUATE=n[1]	Evaluate Boolean cell
FCB=n[1]	File control block (FCB) address
GN=n(hhhhh)[2.]	Generated procedure-name and its offset in hexadecimal
IDX=n[1]	Base locator for index-names
IDX=n[1]	Index cell number
ILS=n[1]	Index cell for LOCAL-STORAGE table or instance variable
ODOSAVE=n[1]	ODO save cell number
OPT=nnnn[3]	Optimizer temporary storage cell
PBL=n[1]	Base locator for procedure code
PFM=n[1]	PERFORM n times cells
PGMLIT AT + nnnn[3]	Displacement for program literal from beginning of literal pool

Table 52. **Symbols used in LIST and MAP output** *(continued)*

Symbol	Definition
PSV=n[1]	Perform save cell number
PVN=n[1]	Variable name cell for PERFORM statement
RBKST=n[1]	Register backstore cell
SFCB=n[1]	Secondary file control block for external file
SYSLIT AT + nnnn[3]	Displacement for system literal from beginning of system literal pool
TGT FDMP TEST INFO. AREA + nnnn[3]	FDUMP/TEST information area
TGTFIXD + nnnn[3]	Offset from beginning of fixed portion of task global table (TGT)
TOV=n[1]	TGT overflow cell number
TS1=aaaa	Temporary storage cell number in subpool 1
TS2=aaaa	Temporary storage cell number in subpool 2
TS3=aaaa	Temporary storage cell number in subpool 3
TS4=aaaa	Temporary storage cell number in subpool 4
V(routine name)	Assembler VCON for external routine
VLC=n[1]	Variable-length name cell number (ODO)
VNI=n[1]	Variable name initialization
WHEN=n[1]	Evaluate WHEN cell number

1. n is the number of the entry. For base locators, it can also be XXXXX, indicating a data item that was deleted by OPTIMIZE(FULL) processing.
2. (hhhhh) is the program offset in hexadecimal.
3. nnnn is the offset in decimal from the beginning of the entry.
4. Alphanumeric temporaries are temporary data values used in processing alphanumeric intrinsic function and alphanumeric EVALUATE statement subjects.

Example: nested program map

This example shows a map of nested procedures produced by specifying the MAP compiler option. Numbers in parentheses refer to notes that follow the example.

```
Nested Program Map
Program Attribute codes (rightmost column) have the following meanings:
    C = COMMON
    I = INITIAL (1)
    U = PROCEDURE DIVISION USING...                        (5)
Source Nesting                                             Program
LineID Level   Program Name from PROGRAM-ID paragraph       Attributes
     2     0   NESTMAIN. . . . . . . . . . . . . . . . . . . U
   120     1 (4) SUBPRO1 . . . . . . . . . . . . . . . . . . I,C,U
(2)199     2       NESTED1 . . . . . . . . . . . . . . . . . I,C,U
   253     1     SUBPRO2 . . . . . . . . . . . . . . . . . . U
   335     2       NESTED2 . . . . . . . . . . . . . . . . . C,U
          (3)
```

(1) Explanations of the program attribute codes

(2) Source line number where the program was defined

(3) Depth of program nesting

(4) Program-name

(5) Program attribute codes

Reading LIST output

Parts of the `LIST` compiler output might be useful to you for debugging a program.

The `LIST` compiler option produces seven pieces of output:

- An assembler listing of the initialization code for the program (program signature information bytes) from which you can verify program characteristics such as these:
  - Compiler options in effect
  - Types of data items present
  - Verbs used in the `PROCEDURE DIVISION`
- An assembler listing of the source code for the program

  From the address in storage of the instruction that was executing when an abend occurred, you can find the COBOL verb that corresponds to that instruction. After you find the address of the failing instruction, go to the assembler listing and find the verb for which that instruction was generated.
- Location of compiler-generated tables in the object module
- A map of the task global table (TGT), including information about the program global table (PGT) and constant global table (CGT)

  Use the TGT to find information about the environment in which your program is running.
- Information about the location and size of `WORKING-STORAGE` and control blocks

  You can use the `WORKING-STORAGE` portion of `LIST` output to find the location of data items defined in `WORKING-STORAGE`. (The beginning location of `WORKING-STORAGE` is not shown for programs compiled with the `RENT` option.)
- Map of the dynamic save area (DSA)

  The map of the DSA (also known as the *stack frame*) contains information about the contents of the storage acquired each time a separately compiled procedure is entered.
- Information about the location of literals and code for dynamic storage usage

You do not need to be able to program in assembler language to understand the `LIST` output. The comments that accompany most of the assembler code provide you with a conceptual understanding of the functions performed by the code.

"Example: program initialization code"
"Example: assembler code generated from source code" on page 384
"Example: TGT memory map" on page 385
"Example: DSA memory map" on page 386
"Example: location and size of WORKING-STORAGE" on page 387

RELATED REFERENCES
"Signature information bytes: compiler options" on page 378
"Signature information bytes: DATA DIVISION" on page 380
"Signature information bytes: ENVIRONMENT DIVISION" on page 380
"Signature information bytes: PROCEDURE DIVISION verbs" on page 381
"Signature information bytes: more PROCEDURE DIVISION items" on page 383
Stack storage overview (*Language Environment Programming Guide*)

Example: program initialization code

A listing of the program initialization code gives you information about the characteristics of the COBOL source program. Interpret the program signature information bytes to verify characteristics of your program.

```
  (1)     (2)                     (3)                            (4)
000000           IMIN           DS    0H                      PROGRAM:IMIN
                                USING *,15
000000  47F0 F028               B     40(,15)                 BYPASS CONSTANTS. BRANCH TO @STM
000004  00                      DC    AL1(0)                  ZERO NAME LENGTH FOR DUMPS
000005  C3C5C5                  DC    CL3'CEE'                CEE EYE CATCHER                       (5)
000008  00000110                DC    X'00000110'             STACK FRAME SIZE
00000C  00000014                DC    A(@PPA1-IMIN)           OFFSET TO PPA1 FROM PRIMARY ENTRY
000010  47F0 F001               B     1(,15)                  RESERVED
000014                 @PPA1    DS    0H                      PPA1 STARTS HERE
000014  98                      DC    X'98'                   OFFSET TO LENGTH OF NAME FROM PPA1
000015  CE                      DC    X'CE'                   CEL SIGNATURE
000016  AC                      DC    X'AC'                   CEL FLAGS: '10101100'B
000017  00                      DC    X'00'                   MEMBER FLAGS FOR COBOL
000018  000000B6                DC    A(@PPA2)                ADDRESS OF PPA2
00001C  00000000                DC    F'0'                    OFFSET TO THE BDI (NONE)
000020  00000000                DC    F'0'                    ADDRESS OF ENTRY POINT DESCRIPTORS
000024  0000                    DC    X'0000'                 RESERVED
000026  00                      DC    X'00'                   DSA FPR 8-15 SAVE AREA OFFSET/16
000027  00                      DC    X'00'                   DSA FPR 8-15 SAVE AREA BIT MASK
000028                 @STM     DS    0H                      STM STARTS HERE
000028  90EC D00C               STM   14,12,12(13)      @STM: SAVE CALLER'S REGISTERS
00002C  4110 F038               LA    1,56(,15)               GET ADDRESS OF PARMLIST INTO R1
000030  98EF F04C               LM    14,15,76(15)            LOAD ADDRESSES FROM @BRVAL
000034  07FF                    BR    15                      DO ANY NECESSARY INITIALIZATION
000036  0000                    DC    AL2'0'                  AVAILABLE HALF-WORD
000038                 @MAINENT DS    0H                      PRIMARY ENTRY POINT ADDRESS
000038  00000000                DC    A(IMIN)         @PARMS: 1) PRIMARY ENTRY POINT ADDRESS
00003C  00000000                DC    AL4'0'                  2) Available
000040  000003C0                DC    A(DAB)                  3) DAB ADDRESS                        (6)
000044  000000AE                DC    A(@EPNAM)               4) ENTRY POINT NAME ADDRESS
000048  00000000                DC    A(IMIN)                 5) CURRENT ENTRY POINT ADDRESS
00004C  00000272                DC    A(START)        @BRVAL: 6) PROCEDURE CODE ADDRESS
000050  00000000                DC    V(IGZCBSO)              7) INITIALIZATION ROUTINE
000054  000000CA                DC    A(@CEEPARM)             8) ADDRESS OF PARM LIST FOR CEEINT
000058  00104001                DC    X'00104001'             DSA WORD 0 CONSTANT
00005C  00000000                DC    AL4'0'                  AVAILABLE WORD
000060  00000000                DC    AL4'0'                  AVAILABLE WORD
000064  00000000                DC    AL4'0'                  AVAILABLE WORD
000068  F2F0F0F4                DC    CL4'2005'     @TIMEVRS: YEAR OF COMPILATION                   (7)
00006C  F1F2F1F6                DC    CL4'0624'               MONTH/DAY OF COMPILATION              (8)
000070  F1F0F4F8                DC    CL4'1048'               HOURS/MINUTES OF COMPILATION          (9)
000074  F1F2                    DC    CL2'16'                 SECONDS FOR COMPILATION DATE
000076  F0F3F0F4F0F0            DC    CL6'030400'             VERSION/RELEASE/MOD LEVEL OF PROD     (10)
00007C  0474                    DC    X'0474'                 UNSIGNED BINARY CODE PAGE CCSID VALUE (11)
00007E  0000                    DC    AL2'0'                  AVAILABLE HALF-WORD
000080  0000                    DC    X'0000'                 INFO. BYTES 28-29                     (12)
000082  076C                    DC    X'076C'                 SIGNED BINARY YEARWINDOW OPTION VALUE
000084  A0487C4C2000            DC    X'A0487C4C2000'         INFO. BYTES 1-6
00008A  000000080000            DC    X'000000080000'         INFO. BYTES 7-12
000090  000000000800            DC    X'000000000800'         INFO. BYTES 13-18                     (12)
000096  0000000000              DC    X'0000000000'           INFO. BYTES 19-23
00009B  00                      DC    X'00'                   COBOL SIGNATURE LEVEL
00009C  00000001                DC    X'00000001'             # DATA DIVISION STATEMENTS            (13)
0000A0  00000003                DC    X'00000003'             # PROCEDURE DIVISION STATEMENTS       (14)
0000A4  000080                  DC    X'000080'               INFO. BYTES 24-26                     (12)
0000A7  00                      DC    X'00'                   INFO. BYTE 27
0000A8  40404040                DC    C'    '                 USER LEVEL INFO (LVLINFO)             (15)
0000AC  0004                    DC    X'0004'                 LENGTH OF PROGRAM NAME
0000AE                 @EPNAM   DS    0H                      ENTRY POINT NAME
0000AE  C9D4C9D540404040        DC    C'IMIN    '             PROGRAM NAME                          (16)
0000B6                 @PPA2    DS    0H                      PPA2 STARTS HERE
0000B6  05                      DC    X'05'                   CEL MEMBER IDENTIFIER
0000B7  00                      DC    X'00'                   CEL MEMBER SUB-IDENTIFIER
0000B8  00                      DC    X'00'                   CEL MEMBER DEFINED BYTE
0000B9  01                      DC    X'01'                   CONTROL LEVEL OF PROLOG
0000BA  00000000                DC    V(CEESTART)             VCON FOR LOAD MODULE
0000BE  00000000                DC    F'0'                    OFFSET TO THE CDI (NONE)
0000C2  FFFFFFB2                DC    A(@TIMEVRS-@PPA2)       OFFSET TO TIMESTAMP/VERSION INFO
0000C6  00000000                DC    A(IMIN)                 ADDRESS OF CU PRIMARY ENTRY POINT
0000CA                 @CEEPARM DS    0H                      PARM LIST FOR CEEINT
0000CA  00000038                DC    A(@MAINENT)             POINTER TO PRIMARY ENTRY PT ADDR
0000CE  00000008                DC    A(@PARMCEE-@CEEPARM)    OFFSET TO PARAMETERS FOR CEEINT
0000D2                 @PARMCEE DS    0H                      PARAMETERS FOR CEEINT
0000D2  00000006                DC    F'6'                    1) NUMBER OF ENTRIES IN PARM LIST
0000D6  00000038                DC    A(@MAINENT)             2) POINTER TO PRIMARY ENTRY PT ADDR
0000DA  00000000                DC    V(CEESTART)             3) ADDRESS OF CEESTART
0000DE  00000000                DC    V(CEEBETBL)             4) ADDRESS OF CEEBETBL
0000E2  00000005                DC    F'5'                    5) CEL MEMBER IDENTIFIER
0000E6  00000000                DC    F'0'                    6) FOR CEL MEMBER USE
. . .
```

(1) Offset from the start of the COBOL program.

(2) Hexadecimal representation of assembler instructions.

(3) Pseudoassembler code generated for the COBOL program.

(4) Comments explaining the assembler code.

(5) Eye-catcher indicating that the COBOL compiler is Language Environment-enabled.

(6) Address of the task global table (TGT), or the address of the dynamic access block (DAB), if the program is reentrant.

(7) Four-digit year when the program was compiled.

(8) Month and the day when the program was compiled.

(9) Time when the program was compiled.

(10) Version, release, and modification level of the COBOL compiler used to compile this program (each represented in two digits).

(11) Code page CCSID value (from `CODEPAGE` compiler option).

(12) Program signature information bytes. These provide information about these elements of the program:

- Compiler options
- `DATA DIVISION`
- `ENVIRONMENT DIVISION`
- `PROCEDURE DIVISION`

(13) Number of statements in the `DATA DIVISION`.

(14) Number of statements in the `PROCEDURE DIVISION`.

(15) 4-byte user-controlled level information field. The value of this field is controlled by the `LVLINFO`.

(16) Program-name as used in the `IDENTIFICATION DIVISION`.

RELATED REFERENCES
"Signature information bytes: compiler options"
"Signature information bytes: DATA DIVISION" on page 380
"Signature information bytes: ENVIRONMENT DIVISION" on page 380
"Signature information bytes: PROCEDURE DIVISION verbs" on page 381
"Signature information bytes: more PROCEDURE DIVISION items" on page 383

Signature information bytes: compiler options

This table shows program signature information that is part of the listing of program initialization code provided when you use the `LIST` compiler option.

Table 53. **Signature information bytes for compiler options**

Byte	Bit	On	Off
1	0	ADV	NOADV
	1	APOST	QUOTE
	2	DATA(31)	DATA(24)
	3	DECK	NODECK
	4	DUMP	NODUMP
	5	DYNAM	NODYNAM
	6	FASTSRT	NOFASTSRT
	7	Reserved	

Table 53. **Signature information bytes for compiler options** *(continued)*

Byte	Bit	On	Off
2	0	LIB	NOLIB
	1	LIST	NOLIST
	2	MAP	NOMAP
	3	NUM	NONUM
	4	OBJ	NOOBJ
	5	OFFSET	NOOFFSET
	6	OPTIMIZE	NOOPTIMIZE
	7	ddname supplied in `OUTDD` option will be used	`OUTDD(SYSOUT)` is in effect.
3	0	NUMPROC(PFD)	NUMPROC(NOPFD)
	1	RENT	NORENT
	2	Reserved	
	3	SEQUENCE	NOSEQUENCE
	4	SIZE(MAX)	SIZE(*value*)
	5	SOURCE	NOSOURCE
	6	SSRANGE	NOSSRANGE
	7	TERM	NOTERM
4	0	TEST	NOTEST
	1	TRUNC(STD)	TRUNC(OPT)
	2	`WORD` option was specified.	NOWORD
	3	VBREF	NOVBREF
	4	XREF	NOXREF
	5	ZWB	NOZWB
	6	NAME	NONAME
5	0	NUMPROC(MIG)	
	1	NUMCLS(ALT)	NUMCLS(PRIM)
	2	DBCS	NODBCS
	3	AWO	NOAWO
	4	TRUNC(BIN)	Not TRUNC(BIN)
	6	CURRENCY	NOCURRENCY
	7	Compilation unit is a class.	Compilation unit is a program.
26	0	RMODE(ANY)	RMODE(24)
	1	TEST(STMT)	Not TEST(STMT)
	2	TEST(PATH)	Not TEST(PATH)
	3	TEST(BLOCK)	Not TEST(BLOCK)
	4	OPT(FULL)	OPT(STD) or NOOPT
	5	INTDATE(LILIAN)	INTDATE(ANSI)
	6	TEST(SEPARATE)	Not TEST(SEPARATE)

Table 53. **Signature information bytes for compiler options** *(continued)*

Byte	Bit	On	Off
27	0	PGMNAME(LONGUPPER)	Not PGMNAME(LONGUPPER)
	1	PGMNAME(LONGMIXED)	Not PGMNAME(LONGMIXED)
	2	DLL	NODLL
	3	EXPORTALL	NOEXPORTALL
	4	DATEPROC	NODATEPROC
	5	ARITH(EXTEND)	ARITH(COMPAT)
	6	THREAD	NOTHREAD
+ 28	0	SQL	NOSQL
+	1	CICS	NOCICS
+	2	MDECK(COMPILE)	NOMDECK
+	3	SQLCCSID	NOSQLCCSID

Signature information bytes: DATA DIVISION

This table shows program signature information that is part of the listing of program initialization code provided when you use the LIST compiler option.

Table 54. **Signature information bytes for the DATA DIVISION**

Byte	Bit	Item
6	0	QSAM file descriptor
	1	VSAM sequential file descriptor
	2	VSAM indexed file descriptor
	3	VSAM relative file descriptor
	4	CODE-SET clause (ASCII files) in file descriptor
	5	Spanned records
	6	PIC G or PIC N (DBCS data item)
	7	OCCURS DEPENDING ON clause in data description entry
7	0	SYNCHRONIZED clause in data description entry
	1	JUSTIFIED clause in data description entry
	2	USAGE IS POINTER item
	3	Complex OCCURS DEPENDING ON clause
	4	External floating-point items in the DATA DIVISION
	5	Internal floating-point items in the DATA DIVISION
	6	Line-sequential file
	7	USAGE IS PROCEDURE-POINTER or FUNCTION-POINTER item

RELATED REFERENCES
"LIST" on page 319

Signature information bytes: ENVIRONMENT DIVISION

This table shows program signature information that is part of the listing of program initialization code provided when you use the LIST compiler option.

Table 55. **Signature information bytes for the ENVIRONMENT DIVISION**

Byte	Bit	Item
8	0	FILE STATUS clause in FILE-CONTROL paragraph
	1	RERUN clause in I-O-CONTROL paragraph of INPUT-OUTPUT SECTION
	2	UPSI switch defined in SPECIAL-NAMES paragraph

Signature information bytes: PROCEDURE DIVISION verbs

The following table shows program signature information that is part of the listing of program initialization code provided when you use the LIST compiler option.

Table 56. **Signature information bytes for PROCEDURE DIVISION verbs**

Byte	Bit	Item
9	0	ACCEPT
	1	ADD
	2	ALTER
	3	CALL
	4	CANCEL
	6	CLOSE
10	0	COMPUTE
	2	DELETE
	4	DISPLAY
	5	DIVIDE
11	1	END-PERFORM
	2	ENTER
	3	ENTRY
	4	EXIT
	5	EXEC
	6	GO TO
	7	IF
12	0	INITIALIZE
	1	INVOKE
	2	INSPECT
	3	MERGE
	4	MOVE
	5	MULTIPLY
	6	OPEN
	7	PERFORM

Table 56. **Signature information bytes for PROCEDURE DIVISION verbs** *(continued)*

Byte	Bit	Item
13	0	READ
	2	RELEASE
	3	RETURN
	4	REWRITE
	5	SEARCH
	7	SET
14	0	SORT
	1	START
	2	STOP
	3	STRING
	4	SUBTRACT
	7	UNSTRING
15	0	USE
	1	WRITE
	2	CONTINUE
	3	END-ADD
	4	END-CALL
	5	END-COMPUTE
	6	END-DELETE
	7	END-DIVIDE
16	0	END-EVALUATE
	1	END-IF
	2	END-MULTIPLY
	3	END-READ
	4	END-RETURN
	5	END-REWRITE
	6	END-SEARCH
	7	END-START
17	0	END-STRING
	1	END-SUBTRACT
	2	END-UNSTRING
	3	END-WRITE
	4	GOBACK
	5	EVALUATE
	7	SERVICE
18	0	END-INVOKE
	1	END-EXEC
	2	XML
	3	END-XML

Check return code: A return code greater than 4 from the compiler could mean that some of the verbs shown in the information bytes might have been discarded from the program.

Signature information bytes: more PROCEDURE DIVISION items

This table shows program signature information that is part of the listing of program initialization code provided when you use the LIST compiler option.

Table 57. **Signature information bytes for more PROCEDURE DIVISION items**

Byte	Bit	Item
21	0	Hexadecimal literal
	1	Altered GO TO
	2	I-O ERROR declarative
	3	LABEL declarative
	4	DEBUGGING declarative
	5	Program segmentation
	6	OPEN . . . EXTEND
	7	EXIT PROGRAM
22	0	CALL literal
	1	CALL identifier
	2	CALL . . . ON OVERFLOW
	3	CALL . . . LENGTH OF
	4	CALL . . . ADDRESS OF
	5	CLOSE . . . REEL/UNIT
	6	Exponentiation used
	7	Floating-point items used
23	0	COPY
	1	BASIS
	2	DBCS name in program
	3	Shift-out and Shift-in in program
	4-7	Highest error severity at entry to ASM2 module IGYBINIT
24	0	DBCS literal
	1	REPLACE
	2	Reference modification was used.
	3	Nested program
	4	INITIAL
	5	COMMON
	6	SELECT . . . OPTIONAL
	7	EXTERNAL

Table 57. **Signature information bytes for more PROCEDURE DIVISION items** *(continued)*

Byte	Bit	Item
25	0	GLOBAL
	1	RECORD IS VARYING
	2	ACCEPT FROM SYSIPT used in LABEL declarative
	3	DISPLAY UPON SYSLST used in LABEL declarative
	4	DISPLAY UPON SYSPCH used in LABEL declarative
	5	Intrinsic function was used
29	0	Java-based OO syntax in program
	1	FUNCTION RANDOM used in program
	2	NATIONAL data used in program

RELATED REFERENCES
"LIST" on page 319

Example: assembler code generated from source code

The following example shows a listing of the assembler code that is generated from source code when you use the LIST compiler option. You can use this listing to find the COBOL verb that corresponds to the instruction that failed.

```
DATA VALIDATION AND UPDATE PROGRAM    IGYTCARA Date 06/30/2005 Time 10:48:16
000433  MOVE
000435  READ
000436  SET   (1)

    (2)         (3)              (5)                        (6)
    000F26  92E8 A00A             MVI  10(10),X'E8'          LOCATION-EOF-FLAG
    000F2A             GN=13     EQU  *
    000F2A  47F0 B426             BC   15,1062(0,11)         GN=75(000EFA)
    000F2E             GN=74     EQU  *
000439  IF
    000F2E  95E8 A00A             CLI  10(10),X'E8'          LOCATION-EOF-FLAG
    000F32  4780 B490             BC   8,1168(0,11)          GN=14(000F64)
000440  DISPLAY
    000F36  5820 D05C             L    2,92(0,13)            TGTFIXD+92
    000F3A  58F0 202C             L    15,44(0,2)            V(IGZCDSP )
    000F3E  4110 97FF             LA   1,2047(0,9)           PGMLIT AT +1999
    000F42  05EF                  BALR 14,15
000443  CALL
    000F44  4130 A012             LA   3,18(0,10)            COMP-CODE
    000F48  5030 D21C             ST   3,540(0,13)           TS2=4
    000F4C  9680 D21C             OI   540(13),X'80'         TS2=4
    000F50  4110 D21C             LA   1,540(0,13)           TS2=4
    000F54  58F0 9000             L    15,0(0,9)             V(ILBOABN0)
    000F58  05EF                  BALR 14,15
    000F5A  50F0 D078             ST   15,120(0,13)          TGTFIXD+120
    000F5E  BF38 D089             ICM  3,8,137(13)           TGTFIXD+137
    000F62  0430                  SPM  3,0
    000F64        (4)    GN=14    EQU  *
    000F64  5820 D154             L    2,340(0,13)           VN=3
    000F68  07F2                  BCR  15,2
```

(1) Source line number and COBOL verb, paragraph name, or section name

In line 000436, SET is the COBOL verb. An asterisk (*) before a name indicates that the name is a paragraph name or a section name.

(2) Relative location of the object code instruction in the module, in hexadecimal notation

(3) Object code instruction, in hexadecimal notation

The first two or four hexadecimal digits are the instruction, and the remaining digits are the instruction operands. Some instructions have two operands.

(4) Compiler-generated names (GN) for code sequences

(5) Object code instruction in a form that closely resembles assembler language

(6) Comments about the object code instruction:

- One or two operands that participate in the machine instructions are displayed on the right. An asterisk immediately follows the data-names that are defined in more than one structure (in that way made unique by qualification in the source program).
- The relative location of any generated label that appears as an operand is displayed in parentheses.

RELATED REFERENCES
"Symbols used in LIST and MAP output" on page 374

Example: TGT memory map

The following example shows LIST output for the task global table (TGT) with information about the environment in which your program runs.

```
DATA VALIDATION AND UPDATE PROGRAM    IGYTCARA Date 06/30/2005 Time 10:48:16
                   *** TGT MEMORY MAP ***
                   (1)      (2)
                   TGTLOC

                   000000  RESERVED - 72 BYTES
                   000048  TGT IDENTIFIER
                   00004C  RESERVED - 4 BYTES
                   000050  TGT LEVEL INDICATOR
                   000051  RESERVED - 3 BYTES
                   000054  32 BIT SWITCH
                   000058  POINTER TO RUNCOM
                   00005C  POINTER TO COBVEC
                   000060  POINTER TO PROGRAM DYNAMIC BLOCK TABLE
                   000064  NUMBER OF FCB'S
                   000068  WORKING-STORAGE LENGTH
                   00006C  RESERVED - 4 BYTES
                   000070  ADDRESS OF IGZESMG WORK AREA
                   000074  ADDRESS OF 1ST GETMAIN BLOCK (SPACE MGR)
                   000078  RESERVED - 2 BYTES
                   00007A  RESERVED - 2 BYTES
                   00007C  RESERVED - 2 BYTES
                   00007E  MERGE FILE NUMBER
                   000080  ADDRESS OF CEL COMMON ANCHOR AREA
                   000084  LENGTH OF TGT
                   000088  RESERVED - 1 SINGLE BYTE FIELD
                   000089  PROGRAM MASK USED BY THIS PROGRAM
                   00008A  RESERVED - 2 SINGLE BYTE FIELDS
                   00008C  NUMBER OF SECONDARY FCB CELLS
                   000090  LENGTH OF THE ALTER VN(VNI) VECTOR
                   000094  COUNT OF NESTED PROGRAMS IN COMPILE UNIT
                   000098  DDNAME FOR DISPLAY OUTPUT
                   0000A0  RESERVED - 8 BYTES
                   0000A8  POINTER TO COM-REG SPECIAL REGISTER
                   0000AC  RESERVED - 52 BYTES
                   0000E0  ALTERNATE COLLATING SEQUENCE TABLE PTR.
```

```
                    0000E4  ADDRESS OF SORT G.N. ADDRESS BLOCK
                    0000E8  ADDRESS OF PGT
                    0000EC  RESERVED - 4 BYTES
                    0000F0  POINTER TO 1ST IPCB
                    0000F4  ADDRESS OF THE CLLE FOR THIS PROGRAM
                    0000F8  POINTER TO ABEND INFORMATION TABLE
                    0000FC  POINTER TO TEST INFO FIELDS IN THE TGT
                    000100  ADDRESS OF START OF COBOL PROGRAM
                    000104  POINTER TO ALTER VNI'S IN CGT
                    000108  POINTER TO ALTER VN'S IN TGT
                    00010C  POINTER TO FIRST PBL IN THE PGT
                    000110  POINTER TO FIRST FCB CELL
                    000114  WORKING-STORAGE ADDRESS
                    000118  POINTER TO FIRST SECONDARY FCB CELL
                    00011C  POINTER TO STATIC CLASS INFO BLOCK 1
                    000120  POINTER TO STATIC CLASS INFO BLOCK 2

                    *** VARIABLE PORTION OF TGT ***

                    000124  BASE LOCATORS FOR SPECIAL REGISTERS
                    00012C  BASE LOCATORS FOR WORKING-STORAGE   (3)
                    000134  BASE LOCATORS FOR LINKAGE-SECTION
                    000138  BASE LOCATORS FOR FILES
                    00014C  CLLE ADDR. CELLS FOR CALL LIT. SUB-PGMS.
                    000170  INDEX CELLS
                    000194  FCB CELLS
                    0001A8  INTERNAL PROGRAM CONTROL BLOCKS
```

(1) Hexadecimal offset of the TGT field from the start of the TGT

(2) Explanation of the contents of the TGT field

(3) TGT fields for the base locators of COBOL data areas

Example: DSA memory map

The following example shows `LIST` output for the dynamic save area (DSA). The DSA contains information about the contents of the storage acquired when a separately compiled procedure is entered.

```
DATA VALIDATION AND UPDATE PROGRAM   IGYTCARA Date 06/30/2005 Time 10:48:16
           *** DSA MEMORY MAP ***
           (1)     (2)
           DSALOC

           000000  REGISTER SAVE AREA
           00004C  STACK NAB (NEXT AVAILABLE BYTE)
           000058  ADDRESS OF INLINE-CODE PRIMARY DSA
           00005C  ADDRESS OF TGT
           000060  ADDRESS OF CAA
           000084  SWITCHES
           000088  CURRENT INT. PROGRAM OR METHOD NUMBER
           00008C  ADDRESS OF CALL STATEMENT PROGRAM NAME
           000090  CALC ROUTINE REGISTER SAVE AREA
           0000C4  ADDRESS OF FILE MUTEX USE COUNT CELLS
           0000C8  PROCEDURE DIVISION RETURNING VALUE

           *** VARIABLE PORTION OF DSA ***

           0000D0  BACKSTORE CELLS FOR SYMBOLIC REGISTERS
           000158  BASE LOCATORS FOR ALPHANUMERIC TEMPS
           00015C  VARIABLE-LENGTH CELLS
           000170  ODO SAVE CELLS
           00017C  VARIABLE NAME (VN) CELLS FOR PERFORM
           0001EC  PERFORM SAVE CELLS
           000320  TEMPORARY STORAGE-1
```

```
000330  TEMPORARY STORAGE-2
000500  ALL PARAMETER BLOCK
000564  ALPHANUMERIC TEMPORARY STORAGE
```

(1) Hexadecimal offset of the DSA field from the start of the DSA

(2) Explanation of the contents of the DSA field

Example: location and size of WORKING-STORAGE

The following example shows `LIST` output about the `WORKING-STORAGE` for a program compiled with the `RENT` option.

```
   (1)                         (2)
WRK-STOR WILL BE ALLOCATED FOR 000015B0 BYTES
```

(1) `WORKING-STORAGE` identification

(2) Length of `WORKING-STORAGE` in hexadecimal notation

RELATED CONCEPTS
"Storage and its addressability" on page 41

Example: XREF output - data-name cross-references

The following example shows a sorted cross-reference of data-names that is produced by the `XREF` compiler option. Numbers in parentheses refer to notes that follow the example.

```
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-STL-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
    1342   400-PRINT-TRANSACTION-ERRORS .  P896
```

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 `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

"Example: XREF output - program-name cross-references"
"Example: embedded cross-reference"

Example: XREF output - program-name cross-references

The following example shows a sorted cross-reference of program-names produced by the `XREF` compiler option. Numbers in parentheses refer to notes that follow the example.

```
   (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
```

(1) Line number where the program-name was defined. If the program is external, the word `EXTERNAL` is displayed instead of a definition line number.

(2) Program-name.

(3) Line numbers where the program is referenced.

Example: embedded cross-reference

The following example shows a modified cross-reference that is embedded in the source listing. The cross-reference is produced by the `XREF` compiler option.

```
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
  . . .
```

(1) Line number of the definition of the data-name or procedure-name in the program

(2) Special definition symbols:

UND The user name is undefined.

DUP The user name is defined more than once.

IMP Implicitly defined name, such as special registers and figurative constants.

IFN Intrinsic function reference.

EXT External reference.

* The program-name is unresolved because the `NOCOMPILE` option is in effect.

Example: OFFSET compiler output

The following example shows a compiler listing that has a condensed verb listing, global tables, `WORKING-STORAGE` information, and literals. The listing is output from the `OFFSET` compiler option.

```
DATA VALIDATION AND UPDATE PROGRAM   IGYTCARA Date 06/30/2005  Time 10:48:16
  . . .
 (1)     (2)     (3)
LINE #  HEXLOC  VERB         LINE #  HEXLOC  VERB         LINE #  HEXLOC  VERB
000880 0026F0 DISPLAY        000881 002702 PERFORM        000933 002702 OPEN
000934 002722 IF             000935 00272C DISPLAY        000936 002736 PERFORM
001389 002736 DISPLAY        001390 002740 DISPLAY        001391 00274A DISPLAY
001392 002754 DISPLAY        001393 00275E DISPLAY        001394 002768 DISPLAY
001395 002772 DISPLAY        000937 00277C PERFORM        001434 00277C DISPLAY
001435 002786 STOP           000939 0027A2 MOVE           000940 0027AC WRITE
000941 0027D6 IF             000942 0027E0 DISPLAY        000943 0027EA PERFORM
001389 0027EA DISPLAY        001390 0027F4 DISPLAY        001391 0027FE DISPLAY
001392 002808 DISPLAY        001393 002812 DISPLAY        001394 00281C DISPLAY
```

```
001395 002826 DISPLAY      000944 002830 DISPLAY      000945 00283A PERFORM
001403 00283A DISPLAY      001404 002844 DISPLAY      001405 00284E DISPLAY
001406 002858 DISPLAY      001407 002862 CALL         000947 002888 CLOSE
```

(1) Line number. Your line numbers or compiler-generated line numbers are listed.

(2) Offset, from the start of the program, of the code generated for this verb (in hexadecimal notation).

The verbs are listed in the order in which they occur and are listed once for each time they are used.

(3) Verb used.

RELATED REFERENCES
"OFFSET" on page 326

Example: VBREF compiler output

The following example shows an alphabetic listing of all the verbs in a program, and shows where each is referenced. The listing is produced by the VBREF compiler option.

```
(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
```

(1) Number of times the verb is used in the program

(2) Verb

(3) Line numbers where the verb is used

Part 3. Targeting COBOL programs for certain environments

Chapter 20. Developing COBOL programs for CICS

COBOL programs that are written for CICS can run under CICS Transaction Server. CICS COBOL application programs that use CICS services must use the CICS command-level interface.

When you use the `CICS` compiler option, the Enterprise COBOL compiler handles both native COBOL and embedded CICS statements in the source program. Compilers before COBOL for OS/390 & VM Version 2 Release 2 require a separate translation step to convert `EXEC CICS` commands to COBOL code. You can still translate embedded CICS statements separately, but use of the integrated CICS translator is recommended.

| To use the integrated CICS translator, CICS Transaction Server Version 2 or later is required.

After you compile and link-edit your program, you need to do some other steps such as updating CICS tables before you can run the COBOL program under CICS. However, these CICS topics are beyond the scope of this COBOL information. See the related references for further information about CICS.

You can determine how runtime errors are handled by setting the `CBLPSHPOP` runtime option. See the related tasks for information about CICS `HANDLE` and `CBLPSHPOP`.

RELATED CONCEPTS
"Integrated CICS translator" on page 399

RELATED TASKS
"Coding COBOL programs to run under CICS"
"Compiling with the CICS option" on page 397
"Using the separate CICS translator" on page 400
"Handling errors by using CICS HANDLE" on page 402
Condition handling under CICS: using the CBLPSHPOP run-time option (*Language Environment Programming Guide*)
CICS Application Programming Guide

RELATED REFERENCES
"CICS" on page 303

Coding COBOL programs to run under CICS

To code a program to run under CICS, code CICS commands in the `PROCEDURE DIVISION` by using the `EXEC CICS` command format.

```
EXEC CICS command-name command-options
END-EXEC
```

CICS commands have the basic format shown above. Within `EXEC` commands, use the space as a word separator; do not use a comma or a semicolon.

Restrictions: COBOL class definitions and methods (object-oriented COBOL) cannot be run in a CICS environment. In addition, when you code your programs to run under CICS, do not use the following code:

- FILE-CONTROL entry in the ENVIRONMENT DIVISION, unless the FILE-CONTROL entry is used for a SORT statement
- FILE SECTION of the DATA DIVISION, unless the FILE SECTION is used for a SORT statement
- User-specified parameters to the main program
- USE declaratives (except USE FOR DEBUGGING)
- These COBOL language statements:
  - ACCEPT format 1: data transfer (you can use format-2 ACCEPT to retrieve the system date and time)
  - CLOSE
  - DELETE
  - DISPLAY UPON CONSOLE
  - DISPLAY UPON SYSPUNCH
  - MERGE
  - OPEN
  - READ
  - RERUN
  - REWRITE
  - START
  - STOP *literal*
  - WRITE

If you plan to use the separate CICS translator, you must put any REPLACE statements that contain EXEC commands after the PROCEDURE DIVISION header for the program, otherwise the commands will not be translated.

Coding file input and output: You must use CICS commands for most input and output processing. Therefore, do not describe files or code any OPEN, CLOSE, READ, START, REWRITE, WRITE, or DELETE statements. Instead, use CICS commands to retrieve, update, insert, and delete data.

Coding a COBOL program to run above the 16-MB line: Under Enterprise COBOL, the following restrictions apply when you code a COBOL program to run above the 16-MB line:

- If you use IMS/ESA(R) Version 6 (or later) without DBCTL, DL/I CALL statements are supported only if all the data passed in the call resides below the 16-MB line. Therefore, you must specify the DATA(24) compiler option. However, if you use IMS/ESA Version 6 (or later) with DBCTL, you can use the DATA(31) compiler option instead and pass data that resides above the 16-MB line.

  If you use EXEC DLI instead of DL/I CALL statements, you can specify DATA(31) regardless of the level of the IMS product.
- If the receiving program is link-edited with AMODE 31, addresses that are passed must be 31 bits long, or 24 bits long with the leftmost byte set to zeros.
- If the receiving program is link-edited with AMODE 24, addresses that are passed must be 24 bits long.

Displaying the contents of data items: DISPLAY to the system logical output device (SYSOUT, SYSLIST, SYSLST) is supported under CICS. The DISPLAY output is written to the Language Environment message file (transient data queue CESE). DISPLAY . . . UPON CONSOLE and DISPLAY . . . UPON SYSPUNCH, however, are not allowed.

RELATED CONCEPTS
"Integrated CICS translator" on page 399

RELATED TASKS
"Sorting under CICS" on page 231
"Getting the system date under CICS"
"Calling to or from COBOL programs"
"Determining the success of ECI calls" on page 397
"Using the separate CICS translator" on page 400

RELATED REFERENCES
"CICS SORT application restrictions" on page 231

Getting the system date under CICS

To retrieve the system date in a CICS program, use a format-2 `ACCEPT` statement or the `CURRENT-DATE` intrinsic function.

You can use any of these format-2 `ACCEPT` statements in the CICS environment to get the system date:

- `ACCEPT` *identifier-2* `FROM DATE` (two-digit year)
- `ACCEPT` *identifier-2* `FROM DATE YYYYMMDD`
- `ACCEPT` *identifier-2* `FROM DAY` (two-digit year)
- `ACCEPT` *identifier-2* `FROM DAY YYYYDDD`
- `ACCEPT` *identifier-2* `FROM DAY-OF-WEEK` (one-digit integer, where 1 represents Monday)

You can use this format-2 `ACCEPT` statement in the CICS environment to get the system time:

- `ACCEPT` *identifier-2* `FROM TIME`

Alternatively, you can use the `CURRENT-DATE` intrinsic function, which can also provide the time.

These methods work in both CICS and non-CICS environments.

Do not use a format-1 `ACCEPT` statement in a CICS program.

RELATED TASKS
"Assigning input from a screen or file (ACCEPT)" on page 36

RELATED REFERENCES
CURRENT-DATE (*Enterprise COBOL Language Reference*)

Calling to or from COBOL programs

You can make calls to or from VS COBOL II, COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL programs by using the `CALL` statement. However, these programs cannot call or be called by OS/VS COBOL programs with the `CALL` statement. You must use `EXEC CICS LINK` instead.

If you are calling a separately compiled COBOL program that was processed with either the separate CICS translator or the integrated CICS translator, you must pass `DFHEIBLK` and `DFHCOMMAREA` as the first two parameters in the `CALL` statement.

Called programs that are processed by the separate CICS translator or the integrated CICS translator can contain any function that is supported by CICS for the language.

You can use COBOL dynamic calls when running under CICS. When a COBOL program has been processed with the separate CICS translator or the integrated CICS translator, or contains `EXEC SQL` statements, the `NODYNAM` compiler option is required. In this case you can use `CALL` *identifier* with the `NODYNAM` compiler option to dynamically call a program. When a COBOL program has no `EXEC SQL` statements and has not been processed by the separate CICS translator or the integrated CICS translator, there is no requirement to compile with the `NODYNAM` compiler option. In this case you can use either `CALL` *literal* with the `DYNAM` compiler option or `CALL` *identifier* to dynamically call a program.

You must define dynamically called programs in the CICS program processing table (PPT) if you are not using CICS autoinstall. Under CICS, COBOL programs do not support dynamic calls to subprograms that have the `RELOAD=YES` option coded in their CICS `PROGRAM` definition. Dynamic calls to programs that are defined with `RELOAD=YES` can cause a storage shortage. Use the `RELOAD=NO` option for programs that are to be dynamically called by COBOL.

Support for interlanguage communication (ILC) with other high-level languages is available. Where ILC is not supported, you can use `CICS LINK`, `XCTL`, and `RETURN` instead.

The following table shows the calling relationship between COBOL and assembler programs. In the table, assembler programs that conform to the interface that is described in the *Language Environment Programming Guide* are called *Language Environment-conforming* assembler programs. Those that do not conform to the interface are *non-Language Environment-conforming* assembler programs.

Table 58. **Calls between COBOL and assembler under CICS**

Calls between COBOL and assembler programs	Language Environment-conforming assembler program	Non-Language Environment-conforming assembler program
From an Enterprise COBOL program to the assembler program?	Yes	Yes
From the assembler program to an Enterprise COBOL program?	Yes, if the assembler program is not a main program	No

Coding nested programs: When you compile with the integrated CICS translator, the translator generates the `DFHEIBLK` and `DFHCOMMAREA` control blocks with the `GLOBAL` clause in the outermost program. Therefore when you code nested programs, you do not have to pass these control blocks as arguments on calls to the nested programs.

When you code nested programs and you plan to use the separate CICS translator, pass `DFHEIBLK` and `DFHCOMMAREA` as parameters to the nested programs that contain `EXEC` commands or references to the `EXEC` interface block (`EIB`). You must pass the same parameters also to any program that forms part of the control hierarchy between such a program and its top-level program.

RELATED CONCEPTS
"Integrated CICS translator" on page 399

RELATED TASKS
"Using the separate CICS translator" on page 400
"Choosing the DYNAM or NODYNAM compiler option" on page 415
"Handling errors when calling programs" on page 244
ILC under CICS (*Language Environment Writing ILC Applications*)
CICS External Interfaces Guide
Language Environment Programming Guide

RELATED REFERENCES
"DYNAM" on page 313

Determining the success of ECI calls

After calls to the external CICS interface (ECI), the content of the `RETURN-CODE` special register is set to an unpredictable value. Therefore, even if your COBOL program terminates normally after successfully using the external CICS interface, the job step could end with an undefined return code.

To ensure that a meaningful return code occurs at termination, set the `RETURN-CODE` special register before you terminate your program. To make the job return code reflect the status of the last call to CICS, set the `RETURN-CODE` special register based on the response codes from the last call to the external CICS interface.

RELATED TASKS
CICS External Interfaces Guide

Compiling with the CICS option

Use the `CICS` compiler option to enable the integrated CICS translator and to specify CICS suboptions.

If you specify the `NOCICS` option, the compiler diagnoses and discards any CICS statements that it finds in your source program. If you have already used the separate CICS translator, you must use `NOCICS`.

You can specify the `CICS` option in any of the compiler option sources: compiler invocation, `PROCESS` or `CBL` statements, or installation default. When the `CICS` option is the COBOL installation default, you cannot specify CICS suboptions. However, making the `CICS` option the installation default is not recommended, because the changes that are made by the integrated CICS translator are not appropriate for non-CICS applications.

All `CBL` or `PROCESS` statements must precede any comment lines, in accordance with the rules for Enterprise COBOL.

The COBOL compiler makes available to the integrated CICS translator the CICS suboption string that you provide in the `CICS` compiler option. Only that translator views the contents of the string.

When you use the integrated CICS translator, you must compile with the following options:

Table 59. **Compiler options required for the integrated CICS translator**

Compiler option	**Comment**
`CICS`	If you specify `NOLIB`, `DYNAM`, or `NORENT`, the compiler forces `LIB`, `NODYNAM`, and `RENT` on.
`LIB`	Must be in effect with `CICS`
`NODYNAM`	Must be in effect with `CICS`
`RENT`	Must be in effect with `CICS`
`SIZE(xxx)`	`xxx` must be a size value (not `MAX`) that leaves enough storage in your user region for the integrated CICS translation process.

In addition, IBM recommends that you use the compiler option `WORD(CICS)` to cause the compiler to flag language elements that are not supported under CICS.

To compile your program with the integrated CICS translator, you can use the standard JCL procedural statements that are supplied with COBOL. In addition to specifying the above compiler options, you must change your JCL in two ways:

- Specify the `STEPLIB` override for the COBOL step.
- Add the data set that contains the integrated CICS translator services, unless these services are in the linklist.

The default name of the data set for CICS Transaction Server V2R2 is CICSTS22.CICS.SDFHLOAD, but your installation might have changed the name. For example, you might have the following line in your JCL:

```
//STEPLIB  DD  DSN=CICSTS22.CICS.SDFHLOAD,DISP=SHR
```

The COBOL compiler listing includes the error diagnostics (such as syntax errors in the CICS statements) that the integrated CICS translator generates. The listing reflects the input source; it does not include the COBOL statements that the integrated CICS translator generates.

Compiling a sequence of programs: When you use the `CICS` option to compile a source file that contains a sequence of COBOL programs, the order of precedence of the options from highest to lowest is:

- Options that are specified in the `CBL` or `PROCESS` card that initiates the unit of compilation
- Options that are specified when the compiler is started
- `CICS` default options

RELATED CONCEPTS
"Integrated CICS translator" on page 399

RELATED TASKS
"Coding COBOL programs to run under CICS" on page 393
"Separating CICS suboptions" on page 399
CICS Application Programming Guide

RELATED REFERENCES
"CICS" on page 303
"Conflicting compiler options" on page 300

Separating CICS suboptions

You can partition the specification of CICS suboptions into multiple CBL statements. CICS suboptions are cumulative. The compiler concatenates them from multiple sources in the order that they are specified.

For example, suppose that a JCL file has the following code:

```
//STEP1 EXEC IGYWC, . . .
//PARM.COBOL="CICS("FLAG(I)")"
//COBOL.SYSIN DD *
  CBL CICS("DEBUG")
  CBL CICS("LINKAGE")
  IDENTIFICATION DIVISION.
  PROGRAM-ID. COBOL1.
```

During compilation, the compiler passes the following CICS suboption string to the integrated CICS translator:

```
"FLAG(I) DEBUG LINKAGE"
```

The concatenated strings are delimited with single spaces and with a quotation mark or single quotation mark around the group. When the compiler finds multiple instances of the same CICS suboption, the last specification of the suboption in the concatenated string takes effect. The compiler limits the length of the concatenated CICS suboption string to 4 KB.

RELATED REFERENCES
"CICS" on page 303

Integrated CICS translator

When you compile a COBOL program using the CICS compiler option, the COBOL compiler works with the integrated CICS translator to handle both native COBOL and embedded CICS statements in the source program.

When the compiler encounters CICS statements, and at other significant points in the source program, the compiler interfaces with the integrated CICS translator. The translator takes appropriate actions and then returns to the compiler, typically indicating which native language statements to generate.

Restriction: If you are using CICS Transaction Server 1.3, you must continue to use the separate translator.

Although you can still translate embedded CICS statements separately, using the integrated CICS translator is recommended. Certain restrictions that apply when you use the separate translator do not apply when you use the integrated translator, and using the integrated translator provides several advantages:

- You can use Debug Tool to debug the original source instead of the expanded source that the separate CICS translator provides.
- You do not need to separately translate the EXEC CICS or EXEC DLI statements that are in copybooks.
- There is no intermediate data set for the translated but not compiled version of the source program.
- Only one output listing instead of two is produced.
- Using nested programs that contain EXEC CICS statements is simpler. DFHCOMAREA and DFHEIBLK are generated with the GLOBAL attribute in the outermost program.

You do not need to pass them as arguments on calls to nested programs or specify them in the USING phrase of the PROCEDURE DIVISION header of nested programs.

- You can keep nested programs that contain EXEC CICS statements in separate files, and include those nested programs by using COPY statements.
- REPLACE statements can affect EXEC CICS statements.
- + You can compile programs that contain CICS statements in batch.
- Because the compiler generates binary fields in CICS control blocks with format COMP-5 instead of BINARY, there is no dependency on the setting of the TRUNC compiler option. You can use any setting of the TRUNC option in CICS programs, subject only to the requirements of the application logic and use of user-defined binary fields.

RELATED TASKS
"Coding COBOL programs to run under CICS" on page 393
"Compiling with the CICS option" on page 397

RELATED REFERENCES
"CICS" on page 303
"TRUNC" on page 343

Using the separate CICS translator

To run a COBOL program under CICS, you can use the separate CICS translator to convert the CICS commands to COBOL statements, and then compile and link the program to create the executable module. However, using the CICS translator that is integrated with Enterprise COBOL is recommended.

To translate CICS statements separately, use the COBOL3 translator option. This option causes the following line to be inserted:

```
CBL RENT,NODYNAM,LIB
```

You can suppress the insertion of a CBL statement by using the CICS translator option NOCBLCARD.

CICS provides the translator option ANSI85, which supports the following language features (introduced by Standard COBOL 85):

- Blank lines intervening in literals
- Sequence numbers containing any character
- Lowercase characters supported in all COBOL words
- REPLACE statement
- Batch compilation
- Nested programs
- Reference modification
- GLOBAL variables
- Interchangeability of comma, semicolon, and space
- Symbolic character definition

After you use the separate CICS translator, use the following compiler options when you compile the program:

Table 60. **Compiler options required for the separate CICS translator**

Required compiler option	Condition
RENT	
NODYNAM	The program is translated by the CICS translator.
LIB	The program contains a COPY or BASIS statement.

In addition, IBM recommends that you use the compiler option WORD(CICS) to cause the compiler to flag language elements that are not supported under CICS.

The following TRUNC compiler option recommendations are based on expected values for binary data items:

Table 61. **TRUNC compiler options recommended for the separate CICS translator**

Recommended compiler option	Condition
TRUNC(OPT)	All binary data items conform to the PICTURE and USAGE clause for those data items.
TRUNC(BIN)	Not all binary data items conform to the PICTURE and USAGE clause for those data items.

For example, if you use the separate CICS translator and have a data item defined as PIC S9(8) BINARY that might receive a value greater than eight digits, use the TRUNC(BIN) compiler option, change the item to USAGE COMP-5, or change the PICTURE clause.

You might also want to avoid using these options, which have no effect:
- ADV
- FASTSRT
- OUTDD

The input data set for the compiler is the data set that you received as a result of translation, which is SYSPUNCH by default.

RELATED CONCEPTS
"Integrated CICS translator" on page 399

RELATED TASKS
"Compiling with the CICS option" on page 397

CICS reserved-word table

COBOL provides an alternate reserved-word table (IGYCCICS) for CICS application programs. If you use the compiler option WORD(CICS), COBOL words that are not supported under CICS are flagged with an error message.

In addition to the COBOL words restricted by the IBM-supplied default reserved-word table, the IBM-supplied CICS reserved-word table restricts the following COBOL words:
- CLOSE
- DELETE
- FD

- **FILE**
- **FILE-CONTROL**
- **INPUT-OUTPUT**
- I-O-CONTROL
- MERGE
- OPEN
- READ
- RERUN
- REWRITE
- **SD**
- **SORT**
- START
- WRITE

If you intend to use the SORT statement under CICS (COBOL supports an interface for the SORT statement under CICS), you must change the CICS reserved-word table to remove the words in **bold** above from the list of words marked as restricted.

RELATED TASKS
"Compiling with the CICS option" on page 397
"Sorting under CICS" on page 231

RELATED REFERENCES
"WORD" on page 346

Handling errors by using CICS HANDLE

The setting of the CBLPSHPOP runtime option affects the state of the HANDLE specifications when a program calls COBOL subprograms using a CALL statement.

When CBLPSHPOP is ON and a COBOL subprogram (not a nested program) is called with a CALL statement, the following actions occur:

1. As part of program initialization, the run time suspends the HANDLE specifications of the calling program (using EXEC CICS PUSH HANDLE).
2. The default actions for HANDLE apply until the called program issues its own HANDLE commands.
3. As part of program termination, the run time reinstates the HANDLE specifications of the calling program (using EXEC CICS POP HANDLE).

If you use the CICS HANDLE CONDITION or CICS HANDLE AID commands, the LABEL specified for the CICS HANDLE command must be in the same PROCEDURE DIVISION as the CICS command that causes branching to the CICS HANDLE label. You cannot use the CICS HANDLE commands with the LABEL option to handle conditions, aids, or abends that were caused by another program invoked with the COBOL CALL statement. Attempts to perform cross-program branching by using the CICS HANDLE command with the LABEL option result in a transaction abend.

If a condition, aid, or abend occurs in a nested program, the LABEL for the condition, aid, or abend must be in the same nested program; otherwise unpredictable results occur.

Performance considerations: When CBLPSHPOP is OFF, the run time does not perform CICS PUSH or POP on a CALL to any COBOL subprogram. If the subprograms do not use any of the EXEC CICS condition-handling commands, you can run with CBLPSHPOP(OFF), thus eliminating the overhead of the PUSH HANDLE and POP HANDLE commands. As a result, performance can be improved compared to running with CBLPSHPOP(ON).

If you are migrating an application from the VS COBOL II run time to the Language Environment run time, see the related reference for information about the CBLPSHPOP option for additional considerations.

"Example: handling errors by using CICS HANDLE"

RELATED TASKS
"Running efficiently with CICS, IMS, or VSAM" on page 637

RELATED REFERENCES
CICS HANDLE commands and CBLPSHPOP (*Enterprise COBOL Compiler and Runtime Migration Guide*)
Enterprise COBOL Version 3 Performance Tuning

Example: handling errors by using CICS HANDLE

The following example shows the use of CICS HANDLE in COBOL programs.

Program A has a CICS HANDLE CONDITION command and program B has no CICS HANDLE commands. Program A calls program B; program A also calls nested program A1. A condition is handled in one of three scenarios.

(1) CBLPSHPOP(ON): If the CICS READ command in program B causes a condition, the condition is not handled by program A (the HANDLE specifications are suspended because the run time performs a CICS PUSH HANDLE). The condition turns into a transaction abend.

(2) CBLPSHPOP(OFF): If the CICS READ command in program B causes a condition, the condition is not handled by program A (the run time diagnoses the attempt to perform cross-program branching by using a CICS HANDLE command with the LABEL option). The condition turns into a transaction abend.

(3) If the CICS READ command in nested program A1 causes a condition, the flow of control goes to label ERR-1, and unpredictable results occur.

```
***********************************************************
*  Program A                                              *
***********************************************************
 ID DIVISION.
 PROGRAM-ID. A.
 . . .
 PROCEDURE DIVISION.
     EXEC CICS HANDLE CONDITION
               ERROR(ERR-1)
               END-EXEC.
     CALL 'B' USING DFHEIBLK DFHCOMMAREA.
     CALL 'A1'.
     . . .
 THE-END.
     EXEC CICS RETURN END-EXEC.
 ERR-1.
 . . .
* Nested program A1.
 ID DIVISION.
 PROGRAM-ID. A1.
 PROCEDURE DIVISION.
     EXEC CICS READ              (3)
               FILE('LEDGER')
               INTO(RECORD)
               RIDFLD(ACCTNO)
               END-EXEC.
 END PROGRAM A1.
 END PROGRAM A.
*
************************************************************
*  Program B                                               *
************************************************************
 ID DIVISION.
 PROGRAM-ID. B.
 . . .
 PROCEDURE DIVISION.
     EXEC CICS READ              (1)  (2)
               FILE('MASTER')
               INTO(RECORD)
               RIDFLD(ACCTNO)
               END-EXEC.
     . . .
 END PROGRAM B.
```

Chapter 21. Programming for a DB2 environment

In general, the coding for your COBOL program will be the same if you want the program 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, do these steps:

- Code any SQL statements that you need, delimiting them with `EXEC SQL` and `END-EXEC` statements.
- Either use the DB2 stand-alone precompiler, or compile with the `SQL` compiler option (if you are using DB2 UDB for z/OS and OS/390 Version 7 or later) and use the DB2 coprocessor.

RELATED CONCEPTS
"DB2 coprocessor"
"COBOL and DB2 CCSID determination" on page 411

RELATED TASKS
"Coding SQL statements" on page 406
"Compiling with the SQL option" on page 409
"Choosing the DYNAM or NODYNAM compiler option" on page 415

RELATED REFERENCES
"SQL" on page 335
"Differences in how the DB2 precompiler and DB2 coprocessor behave" on page 413

DB2 coprocessor

When you use the DB2 coprocessor (called *SQL statement coprocessor* by DB2), the compiler handles your source program that contains embedded SQL statements without your having to use a separate precompile step.

When the compiler encounters SQL statements in the source program, it interfaces with the DB2 coprocessor. This coprocessor takes appropriate actions for the SQL statements and indicates to the compiler which native COBOL statements to generate for them.

Although the use of a separate precompile step continues to be supported, use of the coprocessor is recommended. Interactive debugging with Debug Tool is enhanced when you use the coprocessor because you see the SQL statements in the listing (and not the generated COBOL source). However, you must have DB2 UDB for z/OS and OS/390 Version 7 or later.

Compiling with the DB2 coprocessor generates a DB2 database request module (DBRM) along with the usual COBOL compiler outputs such as object module and listing. The DBRM writes to the data set that you specified in the `DBRMLIB DD` statement in the JCL for the COBOL compile step. As input to the DB2 bind process, the DBRM data set contains information about the SQL statements and host variables in the program.

The COBOL compiler listing includes the error diagnostics (such as syntax errors in the SQL statements) that the DB2 coprocessor generates.

Certain restrictions on the use of COBOL language that apply when you use the precompile step do not apply when you use the DB2 coprocessor:

- You can use SQL statements in any nested program. (With the precompiler, SQL statements are restricted to the outermost program.)
- You can use SQL statements in copybooks.
- `REPLACE` statements work in SQL statements.

+ You must specify the `SQL` compiler option to compile programs that use the DB2
+ coprocessor.

RELATED CONCEPTS
"COBOL and DB2 CCSID determination" on page 411

RELATED TASKS
"Compiling with the SQL option" on page 409

RELATED REFERENCES
"Differences in how the DB2 precompiler and DB2 coprocessor behave" on page 413

Coding SQL statements

Delimit SQL statements with `EXEC SQL` and `END-EXEC`. The `EXEC SQL` and `END-EXEC` delimiters must each be complete on one line. You cannot continue them across multiple lines.

You also need to do these special steps:

- Code an `EXEC SQL INCLUDE` statement to include an SQL communication area (SQLCA) in the `WORKING-STORAGE SECTION` or `LOCAL-STORAGE SECTION` of the outermost program. `LOCAL-STORAGE` is recommended for recursive programs and programs that use the `THREAD` compiler option.
- Declare all host variables that you use in SQL statements in the `WORKING-STORAGE SECTION`, `LOCAL-STORAGE SECTION`, or `LINKAGE SECTION`. However, you do not need to identify them with `EXEC SQL BEGIN DECLARE SECTION` and `EXEC SQL END DECLARE SECTION`.
- Provide all programs that use the `SQL` compiler option with a `WORKING-STORAGE SECTION` or a `LOCAL-STORAGE SECTION`. If such programs are recursive or use the `THREAD` compiler option, they must have a `LOCAL-STORAGE SECTION`.

Restriction: You cannot use SQL statements in object-oriented classes or methods.

RELATED TASKS
"Using SQL INCLUDE with the DB2 coprocessor" on page 407
"Using character data in SQL statements" on page 407
"Using national decimal data in SQL statements" on page 408
"Using national group items in SQL statements" on page 409
"Using binary items in SQL statements" on page 409
"Determining the success of SQL statements" on page 409
Coding SQL statements in a COBOL application (*DB2 UDB Application Programming and SQL Guide*)

RELATED REFERENCES
"Code-page determination for string host variables in SQL statements" on page 412
DB2 UDB SQL Reference

Using SQL INCLUDE with the DB2 coprocessor

An SQL `INCLUDE` statement is treated identically to a native COBOL `COPY` statement when you use the `SQL` compiler option.

The following two lines are therefore treated the same way. (The period that ends the `EXEC SQL INCLUDE` statement is required.)

```
EXEC SQL INCLUDE name END-EXEC.
COPY "name".
```

The processing of the *name* in an SQL `INCLUDE` statement follows the same rules as those of the literal in a `COPY` *literal-1* statement that does not have a `REPLACING` phrase.

The library search order for SQL `INCLUDE` statements is the same SYSLIB concatenation as the compiler uses to resolve COBOL `COPY` statements that do not specify a library-name.

RELATED REFERENCES
Chapter 18, "Compiler-directing statements," on page 351
"Differences in how the DB2 precompiler and DB2 coprocessor behave" on page 413
COPY statement (*Enterprise COBOL Language Reference*)

Using character data in SQL statements

You can code any of the following `USAGE` clauses to describe host variables for character data that you use in `EXEC SQL` statements: `USAGE DISPLAY` for single-byte or UTF-8 data, `USAGE DISPLAY-1` for DBCS data, or `USAGE NATIONAL` for UTF-16 data.

When you use the stand-alone DB2 precompiler, you must specify the code page (CCSID) in `EXEC SQL DECLARE` statements for host variables that are declared with `USAGE NATIONAL`. You must specify the code page for host variables that are declared with `USAGE DISPLAY` or `DISPLAY-1` only if the CCSID that is in effect for the COBOL `CODEPAGE` compiler option does not match the CCSIDs that are used by DB2 for character and graphic data.

Consider the following code. The two highlighted statements are unnecessary when you use the integrated DB2 coprocessor (with the `SQLCCSID` compiler option, as detailed in the related concept below), because the code-page information is handled implicitly.

```
CBL CODEPAGE(1140) NSYMBOL(NATIONAL)
. . .
WORKING-STORAGE SECTION.
    EXEC SQL INCLUDE SQLCA END-EXEC.
01  INT1 PIC S9(4) USAGE COMP.
01  C1140.
    49 C1140-LEN  PIC S9(4) USAGE COMP.
    49 C1140-TEXT PIC X(50).
    EXEC SQL DECLARE :C1140 VARIABLE CCSID 1140 END-EXEC.
01  G1200.
    49 G1200-LEN  PIC S9(4) USAGE COMP.
    49 G1200-TEXT PIC N(50) USAGE NATIONAL.
    EXEC SQL DECLARE :G1200 VARIABLE CCSID 1200 END-EXEC.
    . . .
    EXEC SQL FETCH C1 INTO :INT1, :C1140, :G1200 END-EXEC.
```

If you specify `EXEC SQL DECLARE` *variable-name* `VARIABLE CCSID` *nnnn* `END-EXEC`, that specification overrides the implied CCSID. For example, the following code would cause DB2 to treat `C1208-TEXT` as encoded in UTF-8 (CCSID 1208) rather than as encoded in the CCSID in effect for the COBOL `CODEPAGE` compiler option:

```
01 C1208.
    49 C1208-LEN  PIC S9(4) USAGE COMP.
    49 C1208-TEXT PIC X(50).
    EXEC SQL DECLARE :C1208 VARIABLE CCSID 1208 END-EXEC.
```

The `NSYMBOL` compiler option has no effect on a character literal inside an `EXEC SQL` statement. Character literals in an `EXEC SQL` statement follow the SQL rules for character constants.

RELATED CONCEPTS
"COBOL and DB2 CCSID determination" on page 411

RELATED TASKS
Coding SQL statements in a COBOL application (*DB2 UDB Application Programming and SQL Guide*)

RELATED REFERENCES
"Differences in how the DB2 precompiler and DB2 coprocessor behave" on page 413
"CODEPAGE" on page 305
DB2 UDB SQL Reference

Using national decimal data in SQL statements

You can use national decimal host variables in `EXEC SQL` statements when you use either the integrated DB2 coprocessor or the DB2 precompiler. You do not need to specify the CCSID in `EXEC SQL DECLARE` statements in either case. CCSID 1200 is used automatically.

Any national decimal host variable that you specify in an `EXEC SQL` statement must have the following characteristics:

- It must be signed.
- It must be specified with the `SIGN LEADING SEPARATE` clause.
- `USAGE NATIONAL` must be in effect implicitly or explicitly.

RELATED CONCEPTS
"Formats for numeric data" on page 49

RELATED TASKS
"Defining national numeric data items" on page 129

RELATED REFERENCES
"Differences in how the DB2 precompiler and DB2 coprocessor behave" on page 413

Using national group items in SQL statements

| You can use a national group item as a host variable in an EXEC SQL statement. The national group item is treated with group semantics (that is, as shorthand for the set of host variables that are subordinate to the group item) rather than as an elementary item.

| Because all subordinate items in a national group must have USAGE NATIONAL, a national group item cannot describe a variable-length string.

RELATED TASKS
"Using national groups" on page 130

Using binary items in SQL statements

For binary data items that you specify in an EXEC SQL statement, you can declare the data items as either USAGE COMP-5 or as USAGE BINARY, COMP, or COMP-4.

If you declare the binary data items as USAGE BINARY, COMP, or COMP-4, use the TRUNC(BIN) option. (This technique might have a larger effect on performance than using USAGE COMP-5 on individual data items.) If instead TRUNC(OPT) or TRUNC(STD) are in effect, the compiler accepts the items but the data might not be valid because of the decimal truncation rules. You need to ensure that truncation does not affect the validity of the data.

RELATED CONCEPTS
"Formats for numeric data" on page 49

RELATED REFERENCES
"TRUNC" on page 343

Determining the success of SQL statements

When DB2 finishes executing an SQL statement, DB2 sends a return code in the SQLCA structure (with one exception) to indicate whether the operation succeeded or failed. Your program should test the return code and take any necessary action.

The exception occurs when a program runs under DSN from one of the alternate entry points of the TSO batch mode module IKJEFT01 (IKJEFT1A or IKJEFT1B). In this case, the return code is passed in register 15.

After execution of SQL statements, the content of the RETURN-CODE special register might not be valid. Therefore, even if your COBOL program terminates normally after successfully using SQL statements, the job step could end with an undefined return code. To ensure that a meaningful return code is given at termination, set the RETURN-CODE special register before terminating your program.

RELATED TASKS
Coding SQL statements in a COBOL application (*DB2 UDB Application Programming and SQL Guide*)

Compiling with the SQL option

You use the SQL compiler option to enable the DB2 coprocessor and to specify DB2 suboptions. The SQL compiler option works with embedded SQL statements only if the compiler has access to DB2 UDB for z/OS and OS/390 Version 7 or later.

You can specify the SQL option in any of the compiler option sources: compiler invocation, PROCESS or CBL statements, or installation default. You cannot specify DB2 suboptions when the SQL option is the COBOL installation default, but you can specify default DB2 suboptions by customizing the DB2 product installation defaults.

The DB2 suboption string that you provide in the SQL compiler option is made available to the DB2 coprocessor. Only the DB2 coprocessor views the contents of the string.

When you use the DB2 coprocessor, you must compile with the options that are shown below.

Table 62. **Compiler options required with the DB2 coprocessor**

Compiler option	Comment
SQL	If you use also NOLIB, LIB is forced on.
LIB	Must be specified with SQL
SIZE(*xxx*)	*xxx* is a size value (not MAX) that leaves enough storage in the user region for the DB2 coprocessor services.

You can use standard JCL procedural statements to compile your program with the DB2 coprocessor. In addition to specifying the above compiler options, specify the following items in your JCL:

- DBRMLIB DD statement with the location for the generated database request module (DBRM).
- STEPLIB override for the COBOL step, adding the data set that contains the DB2 coprocessor services, unless these services are in the LNKLST. Typically, this data set is DSN710.SDSNLOAD, but your installation might have changed the name.

For example, you might have the following lines in your JCL:

```
//DBRMLIB  DD  DSN=PAYROLL.MONTHLY.DBRMLIB.DATA(MASTER),DISP=SHR
//STEPLIB  DD  DSN=DSN710.SDSNLOAD,DISP=SHR
```

Compiling in batch: When you use the SQL option to compile a source file that contains a sequence of COBOL programs (a batch compile sequence), the option must be in effect for the first program of the batch sequence. If the SQL option is specified in CBL or PROCESS cards, the CBL or PROCESS cards must precede the first program in the batch compile sequence.

RELATED CONCEPTS
"DB2 coprocessor" on page 405
"COBOL and DB2 CCSID determination" on page 411

RELATED TASKS
"Separating DB2 suboptions" on page 411
"Choosing the DYNAM or NODYNAM compiler option" on page 415

RELATED REFERENCES
"DYNAM" on page 313
"SQL" on page 335
DB2 UDB Command Reference

Separating DB2 suboptions

Because of the concatenation of multiple SQL option specifications, you can separate DB2 suboptions (which might not fit in one CBL statement) into multiple CBL statements.

The options that you include in the suboption string are cumulative. The compiler concatenates these suboptions from multiple sources in the order that they are specified. For example, suppose that your source file has the following code:

```
//STEP1 EXEC IGYWC, . . .
//  PARM.COBOL='SQL("string1")'
//COBOL.SYSIN DD *
    CBL SQL("string2")
    CBL SQL("string3")
    IDENTIFICATION DIVISION.
    PROGRAM-ID. DRIVER1.
```

During compilation, the compiler passes the following suboption string to the DB2 coprocessor:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces. If the compiler finds multiple instances of the same SQL suboption, the last specification of that suboption in the concatenated string takes effect. The compiler limits the length of the concatenated DB2 suboption string to 4 KB.

COBOL and DB2 CCSID determination

All DB2 string data other than BLOB, BINARY, and VARBINARY data has an associated encoding scheme and a coded character set ID (CCSID). This is true for fixed-length and variable-length character strings, fixed-length and variable-length graphic character strings, CLOB host variables, and DBCLOB host variables.

When you use the integrated DB2 coprocessor, the determination of the code page CCSID that will be associated with the string host variables used in SQL statement processing depends on the setting of the COBOL SQLCCSID option, on the programming techniques used, and on various DB2 configuration options.

When you use the SQL and SQLCCSID COBOL compiler options, the CCSID *nnnnn* that is specified in the CODEPAGE compiler option, or that is determined from the COBOL data type of a host variable, is communicated automatically from COBOL to DB2. DB2 associates the COBOL CCSID with host variables, overriding the CCSID that would otherwise be implied by DB2 external mechanisms and defaults. This associated CCSID is used for the processing of the SQL statements that reference host variables.

When you use the SQL and NOSQLCCSID compiler options, the CCSID *nnnnn* that is specified in the CODEPAGE compiler option is used only for processing COBOL statements within the COBOL program; that CCSID is not used for the processing of SQL statements. Instead, DB2 assumes in processing SQL statements that host variable data values are encoded according to the CCSID or CCSIDs that are specified through DB2 external mechanisms and defaults.

RELATED CONCEPTS
"DB2 coprocessor" on page 405

RELATED TASKS
"Programming with the SQLCCSID or NOSQLCCSID option"

RELATED REFERENCES
"Code-page determination for string host variables in SQL statements"
"CODEPAGE" on page 305
"SQL" on page 335
"SQLCCSID" on page 337

Code-page determination for string host variables in SQL statements

When you use the integrated DB2 coprocessor (`SQL` compiler option), the code page for processing string host variables in SQL statements is determined as shown below, in descending order of precedence.

- A host variable that has `USAGE NATIONAL` is always processed by DB2 using CCSID 1200 (Unicode UTF-16). For example:

  ```
  01  hostvariable pic n(10) usage national.
  ```
- An alphanumeric host variable that has an explicit `FOR BIT DATA` declaration is set by DB2 to CCSID 66535, which indicates that the variable does not represent encoded characters. For example:

  ```
  EXEC SQL DECLARE hostvariable VARIABLE FOR BIT DATA END-EXEC
  ```
- A BLOB, BINARY, or VARBINARY host variable has no CCSID association. These string types do not represent encoded characters.
- A host variable for which you specify an explicit CCSID override in the SQLDA is processed with that CCSID.
- A host variable that you specify in a declaration with an explicit CCSID is processed with that CCSID. For example:

  ```
  EXEC SQL DECLARE hostvariable VARIABLE CCSID nnnnn END-EXEC
  ```
- An alphanumeric host variable, if the `SQLCCSID` compiler option is in effect, is processed with the CCSID *nnnnn* from the `CODEPAGE` compiler option.
- A DBCS host variable, if the `SQLCCSID` option is in effect, is processed with the mapped value *mmmmm*, which is the pure DBCS CCSID component of the mixed (MBCS) CCSID *nnnnn* from the `CODEPAGE(nnnnn)` compiler option.
- An alphanumeric or DBCS host variable, if the `NOSQLCCSID` option is in effect, is processed with the CCSID from the DB2 ENCODING bind option, if specified, or from the APPLICATION ENCODING set in DSNHDECP through the DB2 installation panel DSNTIPF.

RELATED REFERENCES
"CODEPAGE" on page 305
"SQLCCSID" on page 337

Programming with the SQLCCSID or NOSQLCCSID option

In general, the `SQLCCSID` option is recommended for new applications that use the integrated DB2 coprocessor, and as a long-term direction for existing applications. The `NOSQLCCSID` option is recommended as a mechanism for migrating existing precompiler-based applications to use the integrated DB2 coprocessor.

The `SQLCCSID` option is recommended for COBOL-DB2 applications that have any of these characteristics:

- Use COBOL Unicode support

- Use other COBOL syntax that is indirectly sensitive to CCSID encoding, such as XML support or object-oriented syntax for Java interoperability
- Process character data that is encoded in a CCSID that is different from the default CCSID assumed by DB2

The `NOSQLCCSID` option is recommended for applications that require the highest compatibility with the behavior of the DB2 precompiler.

For applications that use COBOL alphanumeric data items as host variables interacting with DB2 string data that is defined with the `FOR BIT DATA` subtype, you must either:

- Use the `NOSQLCCSID` compiler option
- Specify explicit `FOR BIT DATA` declarations for those host variables, for example:

  ```
  EXEC SQL DECLARE hostvariable VARIABLE FOR BIT DATA END-EXEC
  ```

Usage notes

- If you use the DB2 `DCLGEN` command to generate COBOL declarations for a table, you can optionally create `FOR BIT DATA` declarations automatically. To do so, specify the `DCLBIT(YES)` option of the `DCLGEN` command.
- **Performance consideration:** Using the `SQLCCSID` compiler option could result in some performance overhead in SQL processing, because with `SQLCCSID` in effect the default DB2 CCSID association mechanism is overridden with a mechanism that works on a per-host-variable basis.

RELATED CONCEPTS
"DB2 coprocessor" on page 405

RELATED REFERENCES
"SQLCCSID" on page 337

Differences in how the DB2 precompiler and DB2 coprocessor behave

The sections that follow enumerate the differences in behavior between the stand-alone COBOL DB2 precompiler and the integrated COBOL DB2 coprocessor.

Period at the end of EXEC SQL INCLUDE statements

Precompiler: The DB2 precompiler does not require that a period end each `EXEC SQL INCLUDE` statement. If a period is specified, the precompiler processes it as part of the statement. If a period is not specified, the precompiler accepts the statement as if a period had been specified.

Coprocessor: The DB2 coprocessor treats each `EXEC SQL INCLUDE` statement like a `COPY` statement, and requires that a period end the statement. For example:

```
IF A = B THEN
    EXEC SQL INCLUDE some_code_here END-EXEC.
ELSE
    . . .
END-IF
```

Note that the period does not terminate the `IF` statement.

Source code after an END-EXEC statement

Precompiler: The DB2 precompiler ignores any code that follows `END-EXEC` statements on the same line.

Coprocessor: The DB2 coprocessor processes code that follows `END-EXEC` statements on the same line.

Multiple definitions of host variables

Precompiler: The DB2 precompiler does not require that host variable references be unique. The first definition that maps to a valid DB2 data type is used.

Coprocessor: The DB2 coprocessor requires that each host variable reference be unique. The coprocessor diagnoses nonunique references to host variables. You must fully qualify host variable references to make them unique.

EXEC SQL statement continuation lines

Precompiler: The DB2 precompiler requires that `EXEC SQL` statements start in columns 12 through 72. Continuation lines of the statements can start anywhere in columns 8 through 72.

Coprocessor: The DB2 coprocessor requires that all lines of an `EXEC SQL` statement, including continuation lines, be coded in columns 12 through 72.

Bit-data host variables

Precompiler: With the DB2 precompiler, a COBOL alphanumeric data item can be used as a host variable to hold DB2 character data that has subtype `FOR BIT DATA`. An explicit `EXEC SQL DECLARE VARIABLE` statement that declares that host variable as `FOR BIT DATA` is not required.

Coprocessor: With the DB2 coprocessor, a COBOL alphanumeric data item can be used as a host variable to hold DB2 character data that has subtype `FOR BIT DATA` if an explicit `EXEC SQL DECLARE VARIABLE` statement for that host variable is specified in the COBOL program. For example:

```
EXEC SQL DECLARE :HV1 VARIABLE FOR BIT DATA END-EXEC.
```

As an alternative to adding `EXEC SQL DECLARE . . . FOR BIT DATA` statements, you can use the `NOSQLCCSID` compiler option. For details, see the related reference about code-page determination below.

SQL-INIT-FLAG

Precompiler: With the DB2 precompiler, if you pass host variables that might be located at different addresses when the program is called more than once, the called program must reset `SQL-INIT-FLAG`. Resetting this flag indicates to DB2 that storage must be initialized when the next SQL statement runs. To reset the flag, insert the statement `MOVE ZERO TO SQL-INIT-FLAG` in the `PROCEDURE DIVISION` of the called program ahead of any executable SQL statements that use those host variables.

Coprocessor: With the DB2 coprocessor, the called program does not need to reset `SQL-INIT-FLAG`. An `SQL-INIT-FLAG` is automatically defined in the program to aid program portability. However, statements that modify `SQL-INIT-FLAG`, such as `MOVE ZERO TO SQL-INIT-FLAG`, have no effect on the SQL processing in the program.

RELATED CONCEPTS
"DB2 coprocessor" on page 405

RELATED REFERENCES
"Code-page determination for string host variables in SQL statements" on page 412
"SQLCCSID" on page 337

Choosing the DYNAM or NODYNAM compiler option

For COBOL programs that have `EXEC SQL` statements, your choice of the compiler option `DYNAM` or `NODYNAM` depends on the operating environment.

When you run under:

- TSO or IMS: You can use either the `DYNAM` or `NODYNAM` compiler option.

  Note that IMS and DB2 share a common alias name, DSNHLI, for the language interface module. You must concatenate your libraries as follows:
  - If you use IMS with the `DYNAM` option, concatenate the IMS library first.
  - If you run your application only under DB2, concatenate the DB2 library first.
- CICS or the DB2 call attach facility (CAF): You must use the `NODYNAM` compiler option.

  Because stored procedures use CAF, you must also compile COBOL stored procedures with the `NODYNAM` option.

RELATED TASKS
"Compiling with the SQL option" on page 409
Programming for the call attachment facility (*DB2 UDB Application Programming and SQL Guide*)

RELATED REFERENCES
"DYNAM" on page 313

Chapter 22. Developing COBOL programs for IMS

Although much of the coding of a COBOL program will be the same when running under IMS, be aware of the following recommendations and restrictions.

In COBOL, IMS message processing programs (MPPs) do not use non-IMS input or output statements such as `READ`, `WRITE`, `REWRITE`, `OPEN`, and `CLOSE`.

With Enterprise COBOL, you can invoke IMS facilities using the following interfaces:

- CBLTDLI call
- Language Environment callable service CEETDLI

You code calls to CEETDLI the same way as you code calls to CBLTDLI. CEETDLI behaves essentially the same way as CBLTDLI.

You can also run object-oriented COBOL programs in an IMS Java dependent region. You can mix the object-oriented COBOL and Java languages in a single application.

RELATED TASKS
"Compiling and linking COBOL programs for running under IMS"
"Using object-oriented COBOL and Java under IMS" on page 418
"Calling a COBOL method from an IMS Java application" on page 418
"Building a mixed COBOL and Java application that starts with COBOL" on page 419
"Writing mixed-language IMS applications" on page 420

Compiling and linking COBOL programs for running under IMS

For best performance in the IMS environment, use the `RENT` compiler option. It causes COBOL to generate reentrant code. You can then run your application programs in either *preloaded* mode (the programs are always resident in storage) or *nonpreload* mode without having to recompile with different options.

IMS allows COBOL programs to be preloaded. Preloading can boost performance because subsequent requests for the program can be handled faster when the program is already in storage (rather than being fetched from a library each time it is needed).

For IMS programs, IBM recommends the `RENT` compiler option. You must use the `RENT` compiler option for a program that is to be run preloaded or as both preloaded and nonpreloaded. When you preload a load module that contains COBOL programs, all of the COBOL programs in that load module must be compiled with the `RENT` option.

You can place programs compiled with the `RENT` option in the z/OS link pack area. There they can be shared among the IMS dependent regions.

To run above the 16-MB line, an application program must be compiled with either RENT or NORENT RMODE(ANY). The data for IMS application programs can reside above the 16-MB line, and you can use DATA(31) RENT or RMODE(ANY) NORENT for programs that use IMS services.

The recommended link-edit attributes for proper execution of COBOL programs under IMS are as follows:

- To link load modules that contain only COBOL programs compiled with the RENT compiler option, link as RENT.
- To link load modules that contain a mixture of COBOL RENT programs and other programs, use the link-edit attributes recommended for the other programs.

RELATED CONCEPTS
"Storage and its addressability" on page 41

RELATED TASKS
"Choosing the DYNAM or NODYNAM compiler option" on page 415
Condition handling under IMS (*Language Environment Programming Guide*)

RELATED REFERENCES
"DATA" on page 307
"RENT" on page 331
IMS considerations (*Enterprise COBOL Compiler and Runtime Migration Guide*)

Using object-oriented COBOL and Java under IMS

You can mix object-oriented COBOL and Java in an application that runs in an IMS Java dependent region.

For example, you can:

- Call a COBOL method from an IMS Java application. You can build the messaging portion of your application in Java and call COBOL methods to access IMS databases.
- Build a mixed COBOL and Java application that starts with the main method of a COBOL class and that invokes Java routines.

You must run these applications in either a Java message processing (JMP) dependent region or a Java batch processing (JBP) dependent region. A program that reads from the message queue (regardless of the language) must run in a JMP dependent region.

RELATED TASKS
"Defining a factory section" on page 557
Chapter 30, "Writing object-oriented programs," on page 525
Chapter 31, "Communicating with Java methods," on page 569
Chapter 16, "Compiling, linking, and running OO applications," on page 287
IMS Java Guide and Reference

Calling a COBOL method from an IMS Java application

You can use the object-oriented language support in Enterprise COBOL to write COBOL methods that an IMS Java program can call.

When you define a COBOL class and compile it with the Enterprise COBOL compiler, the compiler generates a Java class definition with native methods, and the object code that implements those native methods. You can then create an instance and invoke the methods of this class from an IMS Java program that runs in an IMS Java dependent region, just as you would use any other class.

For example, you can define a COBOL class that uses the appropriate DL/I calls to access an IMS database. To make the implementation of this class available to an IMS Java program, do the following steps:

1. Compile the COBOL class with the Enterprise COBOL compiler to generate a Java source file (.java) that contains the class definition and an object module (.o) that contains the implementation of the native methods.
2. Compile the generated Java source file with the Java compiler to create a class file (.class).
3. Link the object code into a dynamic link library (DLL) in the HFS (.so). The HFS directory that contains the COBOL DLLs must be listed in the LIBPATH, as specified in the IMS.PROCLIB member that is indicated by the ENVIRON= parameter of the IMS region procedure.
4. Update the sharable application class path in the master JVM options member (ibm.jvm.sharable.application.class.path in the IMS.PROCLIB member that is specified by the JVMOPMAS= parameter of the IMS region procedure) to enable the JVM to access the Java class file.

When you write the initial routine of a mixed-language application in Java, you must implement a class that is derived from the IMS Java IMSApplication class.

A Java program cannot call procedural COBOL programs directly. To reuse existing COBOL IMS code, use one of the following techniques:

- Restructure the COBOL code as a method in a COBOL class.
- Write a COBOL class definition and method that serves as a wrapper for the existing procedural code. The wrapper code can use COBOL `CALL` statements to access procedural COBOL programs.

RELATED TASKS
Chapter 16, "Compiling, linking, and running OO applications," on page 287
"Structuring OO applications" on page 566
"Wrapping procedure-oriented COBOL programs" on page 566
IMS Java Guide and Reference

Building a mixed COBOL and Java application that starts with COBOL

An application that runs in an IMS Java dependent region must start with the `main` method of a class. A COBOL class definition with a `main` factory method meets this requirement; therefore, you can use it as the first routine of a mixed COBOL and Java IMS application.

Enterprise COBOL generates a Java class with a `main` method that the IMS Java dependent region can find, instantiate, and invoke in the same way that the region does for the `main` method of an IMS Java IMSApplication subclass. Although you can code the entire application in COBOL, you would probably build this type of application to call a Java routine. When COBOL runtime support runs within the JVM of an IMS Java dependent region, it automatically finds and uses this JVM to invoke methods on Java classes.

When the `main` factory method of a COBOL class is the initial routine of a mixed-language application in an IMS Java dependent region, the program is subject to the same requirements as a Java program that runs in the dependent region. That is, the program must explicitly commit resources before it reads subsequent messages from the message queue or exits the application. (A COBOL DL/I `GU` call does not commit resources when it is issued in an IMS Java dependent region, as it does when issued in a message processing region (MPR).)

However, the COBOL application is not derived from the IMSApplication class, and it should not use the IMS Java classes for processing messages or synchronizing transactions. Instead, it should use DL/I calls in COBOL for processing messages (`GU` and `GN`) and synchronizing transactions (`CHKP`). A `CHKP` call in an IMS Java dependent region does not result in the automatic retrieval of a message from the message queue, unlike a `CHKP` call in a non-IMS Java region.

RELATED TASKS
"Structuring OO applications" on page 566
IMS Java Guide and Reference
Persistent Reusable Java Virtual Machine User's Guide

Writing mixed-language IMS applications

When you write mixed-language IMS applications, you need to be aware of the effects of the `STOP RUN` statement, and to understand how to process messages and synchronize transactions, access databases, and use the application interface block (AIB).

RELATED TASKS
"Using the STOP RUN statement"
"Processing messages and synchronizing transactions"
"Accessing databases"
"Using the application interface block" on page 421

Using the STOP RUN statement

If you use the `STOP RUN` statement in the COBOL portion of your application, the statement terminates all COBOL and Java routines (including the JVM).

Control is returned immediately to IMS. The program and the transaction are left in a stopped state.

Processing messages and synchronizing transactions

IMS message-processing applications must do all message processing and transaction synchronization either in COBOL or Java, rather than distributing this logic between application components written in both languages.

COBOL components use `CALL` statements to DL/I services to process messages (`GU` and `GN`) and synchronize transactions (`CHKP`). Java components use IMS Java classes to do these functions. You can use object instances of classes derived from IMSFieldMessage to communicate entire IMS messages between the COBOL and Java components of the application.

RELATED TASKS
IMS Java Guide and Reference
IMS Application Programming: Transaction Manager

Accessing databases

You can use either Java, COBOL, or a mixture of the two languages to access IMS databases.

Limitation: EXEC SQL statements for DB2 database access are not currently supported in COBOL routines that run in IMS Java dependent regions.

Recommendation: Do not access the same database program communication block (PCB) from both Java and COBOL. The Java and COBOL parts of the application share the same database position. Changes in database position from calls in one part of the application affect the database position in another part of the application. (This problem occurs whether the affected parts of an application are written in the same language or in different languages.)

Suppose that a Java component of a mixed application builds an SQL SELECT clause and uses Java Database Connectivity (JDBC) to query and retrieve results from an IMS database. The IMS Java class library constructs the appropriate request to IMS to establish the correct position in the database. If you then invoke a COBOL method that builds a segment search argument (SSA) and issues a GU (Get Unique) request to IMS against the same database PCB, the request probably altered the position in the database for that PCB. If so, subsequent JDBC requests to retrieve more records by using the initial SQL SELECT clause are incorrect because the database position changed. If you must access the same PCB from multiple languages, reestablish the database position after an interlanguage call before you access more records in the database.

RELATED TASKS
IMS Java Guide and Reference

Using the application interface block

COBOL applications that run in an IMS Java dependent region are required to use the AIB interface because the IMS Java dependent region does not provide PCB addresses to its application.

To use the AIB interface, specify the PCB requested for the call by placing the PCB name (which must be defined as part of the PSBGEN) in the resource name field of the AIB. (The AIB requires that all PCBs in a program specification block (PSB) definition have a name.) You do not specify the PCB address directly, and your application does not need to know the relative PCB position in the PCB list. Upon the completion of the call, the AIB returns the PCB address that corresponds to the PCB name that the application passed.

"Example: using the application interface block"

RELATED TASKS
IMS Java Guide and Reference

Example: using the application interface block: The following example shows how you can use the AIB interface in a COBOL application.

```
Local-storage section.
   copy AIB.
   . . .
Linkage section.
01 IOPCB.
   05 logtterm     pic x(08).
   05              pic x(02).
   05 tpstat       pic x(02).
   05 iodate       pic x(04).
   05 iotime       pic x(04).
   05              pic x(02).
   05 seqnum       pic x(02).
   05 mod          pic x(08).
```

```
 Procedure division.
    Move spaces to input-area
    Move spaces to AIB
    Move "DFSAIB" to AIBRID
    Move length of AIB to AIBRLEN
    Move "IOPCB" to AIBRSNM1
    Move length of input-area to AIBOALEN
    Call "CEETDLI" using GU, AIB, input-area
    Set address of IOPCB to AIBRESA1
    If tpstat = spaces
* . . process input message
```

Chapter 23. Running COBOL programs under UNIX

To run COBOL programs in the UNIX environment, compile them with Enterprise COBOL or COBOL for OS/390 & VM. The programs must be reentrant, so use the compiler and linker option `RENT`.

If you are going to run them from the HFS, use the linker option `AMODE 31`. Any `AMODE 24` program that you call from within a UNIX application must reside in an MVS PDS or PDSE.

The following restrictions apply to running under UNIX:

- `SORT` and `MERGE` statements are not supported.
- You cannot use the old COBOL interfaces for preinitialization (runtime option `RTEREUS` and functions IGZERRE and ILBOSTP0) to establish a reusable environment.
- You cannot run a COBOL program compiled with the `NOTHREAD` option in more than one thread. If you start a COBOL application in a second thread, you get a software condition from the COBOL run time. You can run `NOTHREAD` COBOL programs in the initial process thread (IPT) or in one non-IPT that you create from a C or PL/I routine.

  You can run a COBOL program in more than one thread when you compile all the COBOL programs in the application with the `THREAD` option.

You can use Debug Tool to debug Unix System Services programs in remote debug mode, for example, by using the Debug Perspective of WebSphere Developer for System z, or in full-screen mode (MFI) using a VTAM$^{(R)}$ terminal.

RELATED TASKS
Chapter 15, "Compiling under UNIX," on page 279
"Running OO applications under UNIX" on page 289
"Running in UNIX environments"
"Setting and accessing environment variables" on page 424
"Calling UNIX/POSIX APIs" on page 426
"Accessing main program parameters" on page 428
Language Environment Programming Guide

RELATED REFERENCES
"RENT" on page 331

Running in UNIX environments

You can run UNIX COBOL programs in any of the UNIX execution environments, either from a UNIX shell or from outside a shell.

- From a UNIX shell, you can run programs in either the OMVS shell (OMVS) or the ISPF shell (ISHELL).

  Enter the program-name at the shell prompt. The program must be in the current directory or in your search path.

  You can specify runtime options only by setting the environment variable _CEE_RUNOPTS before starting the program.

  You can run programs that reside in a cataloged MVS data set from a shell by using the `tso` utility. For example:

```
tso "call 'my.loadlib(myprog)'"
```

The ISPF shell can direct stdout and stderr only to an HFS file, not to your terminal.

- From outside a shell, you can run programs either under TSO/E or in batch.

  To call a UNIX COBOL program that resides in an HFS file from the TSO/E prompt, use the BPXBATCH utility or a spawn() syscall in a REXX exec.

  To call a UNIX COBOL program that resides in an HFS file with the JCL `EXEC` statement, use the BPXBATCH utility.

RELATED TASKS
"Running OO applications under UNIX" on page 289
"Setting and accessing environment variables"
"Calling UNIX/POSIX APIs" on page 426
"Accessing main program parameters" on page 428
"Defining and allocating QSAM files" on page 166
"Defining and allocating line-sequential files" on page 209
"Allocating VSAM files" on page 201
"Displaying values on a screen or in a file (DISPLAY)" on page 37
Running z/OS UNIX C/C++ application programs: running POSIX-enabled programs (*Language Environment Programming Guide*)

RELATED REFERENCES
"TEST" on page 338
The BPXBATCH utility (*UNIX System Services User's Guide*)
Language Environment Programming Reference

Setting and accessing environment variables

You can set environment variables for UNIX COBOL programs either from the shell with commands `export` and `set`, or from the program.

Although setting and resetting environment variables from the shell before you begin to run a program is a typical procedure, you can set, reset, and access environment variables from the program while it is running.

If you are running a program with BPXBATCH, you can set environment variables by using an `STDENV DD` statement.

To reset an environment variable as if it had not been set, use the UNIX shell command `unset`. To reset an environment variable from a COBOL program, call the setenv() function.

To see the values of all environment variables, use the `export` command with no parameters. To access the value of an environment variable from a COBOL program, call the getenv() function.

"Example: setting and accessing environment variables" on page 426

RELATED TASKS
"Running in UNIX environments" on page 423
"Setting environment variables that affect execution" on page 425
"Accessing main program parameters" on page 428
"Running OO applications under UNIX" on page 289
"Setting environment variables under UNIX" on page 279

RELATED REFERENCES
"Runtime environment variables"
Language Environment Programming Reference
MVS JCL Reference

Setting environment variables that affect execution

To set environment variables for UNIX COBOL programs from a shell, use the `export` or `set` command. To set environment variables from within the program, call POSIX functions setenv() or putenv().

For example, to set the environment variable MYFILE:

```
export MYFILE=/usr/mystuff/notes.txt
```

"Example: setting and accessing environment variables" on page 426

RELATED TASKS
"Calling UNIX/POSIX APIs" on page 426
"Setting environment variables under UNIX" on page 279

RELATED REFERENCES
"Runtime environment variables"

Runtime environment variables

Several runtime variables are of interest for COBOL programs.

These are the runtime environment variables:

_CEE_ENVFILE
Specifies a file from which to read environment variables.

_CEE_RUNOPTS
Specifies runtime options.

CLASSPATH
Specifies directory paths of Java .class files required for an OO application.

COBJVMINITOPTIONS
Specifies Java virtual machine (JVM) options used when COBOL initializes a JVM.

_IGZ_SYSOUT
Specifies where to direct `DISPLAY` output. `stdout` and `stderr` are the only allowable values.

LIBPATH
Specifies directory paths of dynamic link libraries.

PATH
Specifies directory paths of executable programs.

STEPLIB
Specifies location of programs that are not in the LNKLST.

RELATED TASKS
"Displaying data on the system logical output device" on page 38

RELATED REFERENCES
_CEE_ENVFILE (*C/C++ Programming Guide*)
Language Environment Programming Reference

Example: setting and accessing environment variables

The following example shows how you can access and set environment variables from a COBOL program by calling the standard POSIX functions getenv() and putenv().

Because getenv() and putenv() are C functions, you must pass arguments BY VALUE. Pass character strings as BY VALUE pointers that point to null-terminated strings. Compile programs that call these functions with the NODYNAM and PGMNAME(LONGMIXED) options.

```
 CBL pgmname(longmixed),nodynam
 Identification division.
 Program-id. "envdemo".
 Data division.
 Working-storage section.
 01 P pointer.
 01 PATH pic x(5) value Z"PATH".
 01 var-ptr pointer.
 01 var-len pic 9(4) binary.
 01 putenv-arg pic x(14) value Z"MYVAR=ABCDEFG".
 01 rc pic 9(9) binary.
 Linkage section.
 01 var pic x(5000).
 Procedure division.
* Retrieve and display the PATH environment variable
     Set P to address of PATH
     Call "getenv" using by value P returning var-ptr
     If var-ptr = null then
         Display "PATH not set"
     Else
         Set address of var to var-ptr
         Move 0 to var-len
         Inspect var tallying var-len
           for characters before initial X"00"
         Display "PATH = " var(1:var-len)
     End-if
* Set environment variable MYVAR to ABCDEFG
     Set P to address of putenv-arg
     Call "putenv" using by value P returning rc
     If rc not = 0 then
         Display "putenv failed"
         Stop run
     End-if
     Goback.
```

Calling UNIX/POSIX APIs

You can call standard UNIX/POSIX functions from UNIX programs and from traditional z/OS COBOL programs by using the CALL *literal* statement. These functions are part of Language Environment.

Because these are C functions, you must pass arguments BY VALUE. Pass character strings as BY VALUE pointers that point to null-terminated strings. You must use the compiler options NODYNAM and PGMNAME(LONGMIXED) when you compile programs that call these functions.

You can call the fork(), exec(), and spawn() functions from a COBOL program or from a non-COBOL program in the same process as COBOL programs. However, be aware of these restrictions:

- From a forked process you cannot access any COBOL sequential, indexed, or relative files that were open when you issued the fork. File status code 92 is returned if you attempt such access (CLOSE, READ, WRITE, REWRITE, DELETE, or START). You can access line-sequential files that were open at the time of a fork.

- You cannot use the fork() function in a process in which any of the following conditions are true:
  - A COBOL SORT or MERGE is running.
  - A declarative is running.
  - The process has more than one Language Environment enclave (COBOL run unit).
  - The process has used any of the COBOL reusable environment interfaces.
  - The process has ever run an OS/VS COBOL or VS COBOL II program.
- With one exception, DD allocations are not inherited from a parent process to a child process. The exception is the local spawn, which creates a child process in the same address space as the parent process. You request a local spawn by setting the environment variable _BPX_ SHAREAS=YES before you invoke the spawn() function.

The exec() and spawn() functions start a new Language Environment enclave in the new UNIX process. Therefore the target program of the exec() or spawn() function is a main program, and all COBOL programs in the process start in initial state with all files closed.

Sample code for calling some of the POSIX routines is provided in the SIGYSAMP data set.

Table 63. **Samples with POSIX function calls**

Purpose	**Sample**	**Functions used**
Shows how to use some of the file and directory routines	IGYTFL1	• getcwd() • mkdir() • rmdir() • access()
Shows how to use the iconv routines to convert data	IGYTCNV	• iconv_open() • iconv() • iconv_close()
Shows the use of the exec() routine to run a new program along with other process-related routines	IGYTEXC, IGYTEXC1	• fork() • getpid() • getppid() • execl() • perror() • wait()
Shows how to get the errno value	IGYTERNO, IGYTGETE	• perror() • fopen()

Table 63. **Samples with POSIX function calls** *(continued)*

Purpose	Sample	Functions used
Shows the use of the interprocess communication message routines	IGYTMSQ, IGYTMSQ2	• ftok() • msgget() • msgsnd() • perror() • fopen() • fclose() • msgrcv() • msgctl() • perror()

RELATED TASKS
"Running in UNIX environments" on page 423
"Setting and accessing environment variables" on page 424
"Accessing main program parameters"
Language Environment Programming Guide

RELATED REFERENCES
C/C++ Run-Time Library Reference
UNIX System Services Programming: Assembler Callable Services Reference

Accessing main program parameters

When you run a COBOL program from the UNIX shell command line or with an exec() or spawn() function, the parameter list consists of three parameters passed by reference. You can access these parameters with standard COBOL coding.

argument count
A binary fullword integer that contains the number of elements in each of the arrays that are passed in the second and third parameters.

argument length list
An array of pointers. The *n*th entry in the array is the address of a fullword binary integer that contains the length of the *n*th entry in the argument list.

argument list
An array of pointers. The *n*th entry in the array is the address of the *n*th character string passed as an argument in the spawn() or exec() function or in the command invocation. Each character string is null-terminated.

This array is never empty. The first argument is the character string that represents the name of the file associated with the process being started.

"Example: accessing main program parameters"

RELATED TASKS
"Running in UNIX environments" on page 423
"Setting and accessing environment variables" on page 424
"Calling UNIX/POSIX APIs" on page 426

Example: accessing main program parameters

The following example shows the three parameters that are passed by reference.

```
 Identification division.
 Program-id. "EXECED".
****************************************************************
* This sample program displays arguments received via exec()   *
* function of UNIX System Services                             *
****************************************************************
 Data division.
 Working-storage section.
 01 curr-arg-count pic 9(9) binary value zero.
 Linkage section.
 01 arg-count pic 9(9) binary.                       (1)
 01 arg-length-list.                                 (2)
     05 arg-length-addr pointer occurs 1 to 99999
           depending on curr-arg-count.
 01 arg-list.                                        (3)
     05 arg-addr pointer occurs 1 to 99999
           depending on curr-arg-count.
 01 arg-length pic 9(9) binary.
 01 arg pic X(65536).
 Procedure division using arg-count arg-length-list arg-list.
*****************************************************************
* Display number of arguments received                          *
*****************************************************************
     Display "Number of arguments received: " arg-count
*****************************************************************
* Display each argument passed to this program                  *
*****************************************************************
     Perform arg-count times
       Add 1 to curr-arg-count
* *******************************************************
* * Set address of arg-length to address of current     *
* * argument length and display                         *
* *******************************************************
       Set Address of arg-length
         to arg-length-addr(curr-arg-count)
       Display
         "Length of Arg " curr-arg-count " = " arg-length
* *******************************************************
* * Set address of arg to address of current argument   *
* * and display                                         *
* *******************************************************
       Set Address of arg to arg-addr(curr-arg-count)
       Display "Arg " curr-arg-count " = " arg (1:arg-length)
     End-Perform
     Display "Display of arguments complete."
     Goback.
```

(1) This count contains the number of elements in the arrays that are passed in the second and third parameters.

(2) This array includes a pointer to the length of the *n*th entry in the argument list.

(3) This array includes a pointer to the *n*th character string passed as an argument on the spawn() or exec() function or the command invocation.

Part 4. Structuring complex applications

Chapter 24. Using subprograms

Many applications consist of several separately compiled programs linked together. A *run unit* (the COBOL term that is synonymous with the Language Environment term *enclave*) includes one or more object programs and can include object programs written in other Language Environment member languages.

Language Environment provides interlanguage support that allows your Enterprise COBOL programs to call and be called by programs that meet the requirements of Language Environment.

Name prefix alert: Do not use program-names that start with prefixes used by IBM products. If you use programs whose names have any of the following prefixes, `CALL` statements might resolve to IBM library or compiler routines rather than to the intended program:

- AFB
- AFH
- CBC
- CEE
- EDC
- IBM
- IFY
- IGY
- IGZ
- ILB

RELATED CONCEPTS
"Main programs, subprograms, and calls"

RELATED TASKS
"Ending and reentering main programs or subprograms" on page 434
"Transferring control to another program" on page 435
"Making recursive calls" on page 447
"Calling to and from object-oriented programs" on page 447
"Using procedure and function pointers" on page 447
"Making programs reentrant" on page 450
"Handling COBOL limitations with multithreading" on page 482
Language Environment Writing ILC Applications

RELATED REFERENCES
Register conventions (*Language Environment Programming Guide*)

Main programs, subprograms, and calls

If a COBOL program is the first program in a run unit, that COBOL program is the *main program*. Otherwise, it and all other COBOL programs in the run unit are *subprograms*. No specific source-code statements or options identify a COBOL program as a main program or subprogram.

Whether a COBOL program is a main program or subprogram can be significant for either of two reasons:

- Effect of program termination statements
- State of the program when it is reentered after returning

In the `PROCEDURE DIVISION`, a program can call another program (generally called a *subprogram*), 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 processing of the called program is completed, the called 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`.

RELATED TASKS
"Ending and reentering main programs or subprograms"
"Transferring control to another program" on page 435
"Making recursive calls" on page 447

RELATED REFERENCES
Language Environment Programming Guide

Ending and reentering main programs or subprograms

Whether a program is left in its last-used state or its initial state, and to what caller it returns, can depend on the termination statements that you use.

You can use any of three termination statements in a program, but they have different effects, as shown in the table below.

Table 64. **Effects of termination statements**

Termination statement	**Main program**	**Subprogram**
`EXIT PROGRAM`	No action taken	Return to calling program without ending the run unit. An implicit `EXIT PROGRAM` statement is generated if the called program has no next executable statement. In a threaded environment, the thread is not terminated unless the program is the first (oldest) one in the thread.
`STOP RUN`	Return to calling program.[1] (Might be the operating system, and application will end.) `STOP RUN` terminates the run unit, and deletes all dynamically called programs in the run unit and all programs link-edited with them. (It does not delete the main program.) In a threaded environment, the entire Language Environment enclave is terminated, including all threads running within the enclave.	Return directly to the program that called the main program.[1] (Might be the operating system, and application will end.) `STOP RUN` terminates the run unit, and deletes all dynamically called programs in the run unit and all programs link-edited with them. (It does not delete the main program.) In a threaded environment, the entire Language Environment enclave is terminated, including all threads running within the enclave.

Table 64. **Effects of termination statements** (continued)

Termination statement	Main program	Subprogram
GOBACK	Return to calling program.[1] (Might be the operating system, and application will end.) GOBACK terminates the run unit, and deletes all dynamically called programs in the run unit and all programs link-edited with them. (It does not delete the main program.) In a threaded environment, the thread is terminated.[2]	Return to calling program. In a threaded environment, if the program is the first program in a thread, the thread is terminated.[2]

1. If the main program is called by a program written in another language that does not follow Language Environment linkage conventions, return is to this calling program.
2. If the thread is the initial thread of execution in an enclave, the enclave is terminated.

A subprogram is usually left in its *last-used state* when it terminates with EXIT PROGRAM or GOBACK. The next time the subprogram is called in the run unit, its internal values are as they were left, except that return values for PERFORM statements are reset to their initial values. (In contrast, a main program is initialized each time it is called.)

There are some cases where programs will be in their initial state:
- A subprogram that is dynamically called and then canceled will be in the initial state the next time it is called.
- A program that has the INITIAL attribute will be in the initial state each time it is called.
- Data items defined in the LOCAL-STORAGE SECTION will be reset to the initial state specified by their VALUE clauses each time the program is called.

RELATED CONCEPTS
"Comparison of WORKING-STORAGE and LOCAL-STORAGE" on page 15
Language Environment termination: thread termination (*Language Environment Programming Guide*)

RELATED TASKS
"Calling nested COBOL programs" on page 444
"Making recursive calls" on page 447

Transferring control to another program

You can use several different methods to transfer control to another program: static calls, dynamic calls, calls to nested programs, and calls to dynamic link libraries (DLLs).

In addition to making calls between Enterprise COBOL programs, you can also make static and dynamic calls between Enterprise COBOL and programs compiled with older compilers in all environments including CICS.

When you use OS/VS COBOL with Enterprise COBOL, there are differences in support between non-CICS and CICS:

In a non-CICS environment

You can make static and dynamic calls between Enterprise COBOL and other COBOL programs.

Exception: You cannot call VS COBOL II or OS/VS COBOL programs in the UNIX environment.

In a CICS environment

You cannot call OS/VS COBOL programs in the CICS environment. You must use `EXEC CICS LINK` to transfer control between OS/VS COBOL programs and other COBOL programs.

Calls to nested programs allow you to create applications using structured programming techniques. You can use nested programs in place of `PERFORM` procedures to prevent unintentional modification of data items. Call nested programs using either the `CALL` *literal* or `CALL` *identifier* statement.

Calls to dynamic link libraries (DLLs) are an alternative to COBOL dynamic `CALL`, and are well suited to object-oriented COBOL applications, UNIX programs, and applications that interoperate with C/C++.

Under z/OS, linking two load modules together results logically in a single program with a primary entry point and an alternate entry point, each with its own name. Each name by which a subprogram is to be dynamically called must be known to the system. You must specify each such name in linkage-editor or binder control statements as either a `NAME` or an `ALIAS` of the load module that contains the subprogram.

RELATED CONCEPTS
"AMODE switching" on page 439
"Performance considerations of static and dynamic calls" on page 441
"Nested programs" on page 444

RELATED TASKS
"Making static calls"
"Making dynamic calls" on page 437
"Making both static and dynamic calls" on page 442
"Calling nested COBOL programs" on page 444

Making static calls

When you use the `CALL` *literal* statement in a program that is compiled using the `NODYNAM` and `NODLL` compiler options, a static call occurs. With these options, all `CALL` *literal* calls are handled as static calls.

With static calls statement, the COBOL program and all called programs are part of the same load module. When control is transferred, the called program already resides in storage, and a branch to it takes place. Subsequent executions of the `CALL` statement make the called program available in its last-used state unless the called program has the `INITIAL` attribute. In that case, the called program and each program directly or indirectly contained within it are placed into their initial state each time the called program is called within a run unit.

If you specify alternate entry points, a static `CALL` statement can use any alternate entry point to enter the called subprogram.

"Examples: static and dynamic CALL statements" on page 442

RELATED CONCEPTS
"Performance considerations of static and dynamic calls" on page 441

RELATED TASKS
"Making dynamic calls"
"Making both static and dynamic calls" on page 442
"Calling to and from object-oriented programs" on page 447

RELATED REFERENCES
"DLL" on page 311
"DYNAM" on page 313
CALL statement (*Enterprise COBOL Language Reference*)

Making dynamic calls

When you use a CALL *literal* statement in a program that is compiled using the DYNAM and the NODLL compiler options, or when you use the CALL *identifier* statement in a program that is compiled using the NODLL compiler option, a dynamic call occurs.

In these forms of the CALL statement, the called COBOL subprogram is not link-edited with the main program. Instead, it is link-edited into a separate load module, and is loaded at run time only when it is required (that is, when called). The program-name in the PROGRAM-ID paragraph or ENTRY statement must be identical to the corresponding load module name or load module alias of the load module that contains the program.

Each subprogram that you call with a dynamic CALL statement can be part of a different load module that is a member of either the system link library or a private library that you supply. In either case it must be in an MVS load library; it cannot reside in the hierarchical file system. When a dynamic CALL statement calls a subprogram that is not resident in storage, the subprogram is loaded from secondary storage into the region or partition that contains the main program, and a branch to the subprogram is performed.

The first dynamic call to a subprogram within a run unit obtains a fresh copy of the subprogram. Subsequent calls to the same subprogram (by either the original caller or any other subprogram within the same run unit) result in a branch to the same copy of the subprogram in its last-used state, provided the subprogram does not possess the INITIAL attribute. Therefore, the reinitialization of either of the following items is your responsibility:

- GO TO statements that have been altered
- Data items

If you call the same COBOL program in different run units, a separate copy of WORKING-STORAGE is allocated for each run unit.

Restrictions: You cannot make dynamic calls to:

- COBOL DLL programs
- COBOL programs compiled with the PGMNAME(LONGMIXED) option, unless the program-name is less than or equal to eight characters in length and is all uppercase
- COBOL programs compiled with the PGMNAME(LONGUPPER) option, unless the program-name is less than or equal to eight characters in length

- More than one entry point in the same COBOL program (unless an intervening CANCEL statement was executed)

"Examples: static and dynamic CALL statements" on page 442

RELATED CONCEPTS
"When to use a dynamic call with subprograms"
"Performance considerations of static and dynamic calls" on page 441

RELATED TASKS
"Canceling a subprogram"
"Making static calls" on page 436
"Making both static and dynamic calls" on page 442

RELATED REFERENCES
"DLL" on page 311
"DYNAM" on page 313
ENTRY statement (*Enterprise COBOL Language Reference*)
CALL statement (*Enterprise COBOL Language Reference*)
Language Environment Programming Reference

Canceling a subprogram

When you issue a CANCEL statement for a subprogram, the storage that is occupied by the subprogram is freed. A subsequent call to the subprogram functions as though it were the first call. You can cancel a subprogram from a program other than the original caller.

If the called subprogram has more than one entry point, ensure that an intervening CANCEL statement is issued before you specify different entry points in a dynamic CALL statement to that subprogram.

After a CANCEL statement is processed for a dynamically called contained program, the program will be in its first-used state. However, the program is not loaded with the initial call, and storage is not freed after the program is canceled.

"Examples: static and dynamic CALL statements" on page 442

RELATED CONCEPTS
"Performance considerations of static and dynamic calls" on page 441

When to use a dynamic call with subprograms

Your decision to use dynamic calls with subprograms depends on factors such as location of the load module, frequency of calls to the subprograms, size of the subprograms, ease of maintenance, the need to call subprograms in their unused state, the need for AMODE switching, and when the program-names are known.

The load module that you want to dynamically call must be in an MVS load library rather than in the hierarchical file system.

If subprograms are called in only a few conditions, you can use dynamic calls to bring in the subprograms only when needed.

If the subprograms are very large or there are many of them, using static calls might require too much main storage. Less total storage might be required to call and cancel one, then call and cancel another, than to statically call both.

If you are concerned about ease of maintenance, dynamic calls can help. Applications do not have to be link-edited again when dynamically called subprograms are changed.

When you cannot use the `INITIAL` attribute to ensure that a subprogram is placed in its unused state each time that it is called, you can set the unused state by using a combination of dynamic `CALL` and `CANCEL` statements. When you cancel a subprogram that was first called by a COBOL program, the next call causes the subprogram to be reinitialized to its unused state.

Using the `CANCEL` statement to explicitly cancel a subprogram that was dynamically loaded and branched to by a non-COBOL program does not result in any action being taken to release the subprogram's storage or to delete the subprogram.

Suppose you have an OS/VS COBOL or other `AMODE 24` program in the same run unit with Enterprise COBOL programs that you want to run in 31-bit addressing mode. COBOL dynamic call processing includes `AMODE` switching for `AMODE 24` programs that call `AMODE 31` programs, and vice versa. To have this implicit `AMODE` switching done, you must use the Language Environment runtime option `ALL31(OFF)`. `AMODE` switching is not performed when `ALL31(ON)` is set.

When `AMODE` switching is performed, control is passed from the caller to a Language Environment library routine. After the switching is performed, control passes to the called program; the save area for the library routine will be positioned between the save area for the caller program and the save area for the called program.

If you do not know the program-name to be called until run time, use the format `CALL` *identifier*, where *identifier* is a data item that will contain the name of the called program at run time. For example, you could use `CALL` *identifier* when the program to be called varies depending on conditional processing in your program. `CALL` *identifier* is always dynamic, even if you use the `NODYNAM` compiler option.

"Examples: static and dynamic CALL statements" on page 442

RELATED CONCEPTS
"AMODE switching"
"Performance considerations of static and dynamic calls" on page 441

RELATED TASKS
"Making dynamic calls" on page 437

RELATED REFERENCES
"DYNAM" on page 313
CALL statement (*Enterprise COBOL Language Reference*)
Language Environment Programming Reference

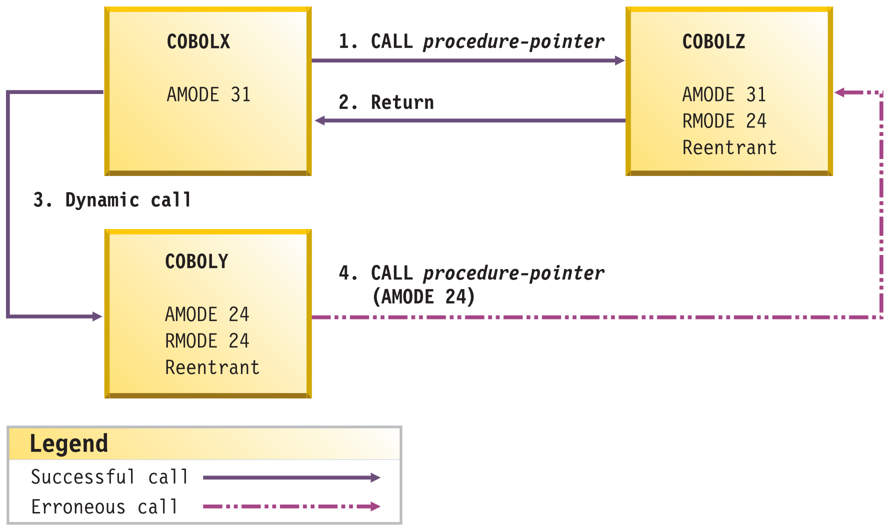
AMODE switching

When you have an application that has COBOL subprograms, some of the COBOL subprograms can be `AMODE 31` and some can be `AMODE 24`.

If your application consists of only COBOL programs, and you are using only static and dynamic calls, each COBOL subprogram will always be entered in the proper AMODE. For example, if you are using a dynamic call from an AMODE 31 COBOL program to an AMODE 24 COBOL program, the AMODE is automatically switched.

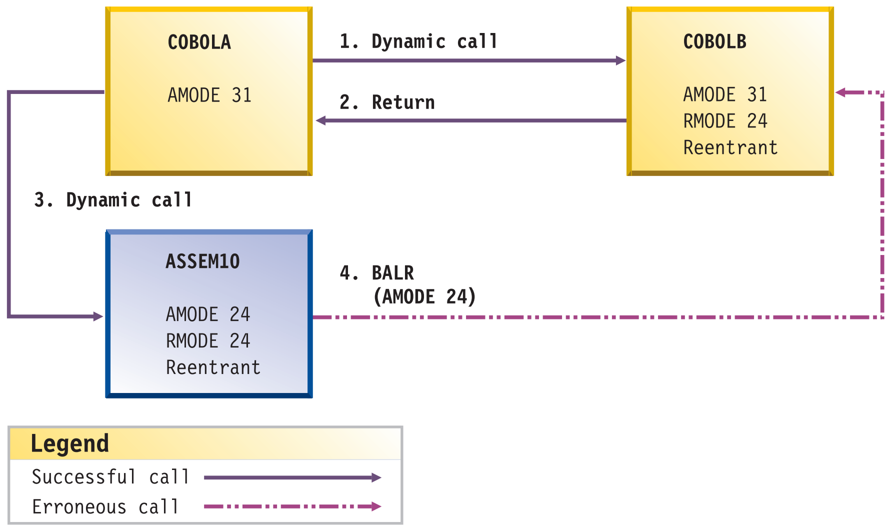
However, if you are using procedure pointers, function pointers, or other languages that call COBOL subprograms, you must ensure that when a COBOL program is called more than once in an enclave, it is entered in the same AMODE each time that it is called. The AMODE is not automatically switched in this case.

The following scenario shows that AMODE problems can arise when procedure pointers are used to call COBOL subprograms. This scenario is not supported because the COBOL program COBOLY is not entered in the same AMODE each time that it is called.



1. COBOLX is AMODE 31. It uses the SET statement to set a procedure pointer to COBOLZ. COBOLZ is a reentrant load module and is AMODE 31 and RMODE 24. COBOLX calls COBOLZ using the procedure pointer. COBOLZ is entered in AMODE 31.
2. COBOLZ returns to COBOLX.
3. COBOLX dynamically calls COBOLY, passing the procedure pointer for COBOLZ. COBOLY is a reentrant load module, and is AMODE 24 and RMODE 24. COBOLY is entered in AMODE 24.
4. COBOLY calls COBOLZ using the procedure pointer. This call causes COBOLZ to be entered in AMODE 24, which is not the same AMODE in which COBOLZ was entered when it was called the first time.

The following scenario uses a mix of COBOL and assembler language. This scenario is not supported because the COBOL program COBOLB is not entered in the same AMODE each time that it is called.



1. COBOLA is `AMODE 31`. COBOLA dynamically calls COBOLB. COBOLB is a reentrant load module and is `AMODE 31` and `RMODE 24`. COBOLB is entered in `AMODE 31`.
2. COBOLB returns to COBOLA.
3. COBOLA dynamically calls ASSEM10, which is in assembler language. ASSEM10 is a reentrant load module, and is `AMODE 24` and `RMODE 24`. ASSEM10 is entered in `AMODE 24`.
4. ASSEM10 loads COBOLB. ASSEM10 does a `BALR` instruction to COBOLB. COBOLB is entered in `AMODE 24`, which is not the same `AMODE` in which COBOLB was entered when it was called the first time.

RELATED CONCEPTS
"Storage and its addressability" on page 41
"When to use a dynamic call with subprograms" on page 438

RELATED TASKS
"Making dynamic calls" on page 437

RELATED REFERENCES
ALL31 (*Language Environment Programming Reference*)

Performance considerations of static and dynamic calls

Because a statically called program is link-edited into the same load module as the calling program, a static call is faster than a dynamic call. A static call is the preferred method if your application does not require the services of the dynamic call.

Statically called programs cannot be deleted using `CANCEL`, so static calls might take more main storage. If storage is a concern, think about using dynamic calls. Storage usage of calls depends on whether:

- The subprogram is called only a few times. Regardless of whether it is called, a statically called program is loaded into storage; a dynamically called program is loaded only when it is called.
- You subsequently delete the dynamically called subprogram with a `CANCEL` statement.

You cannot delete a statically called program, but you can delete a dynamically called program. Using a dynamic call and then a CANCEL statement to delete the dynamically called program after it is no longer needed in the application (and not after each call to it) might require less storage than using a static call.

RELATED CONCEPTS
"When to use a dynamic call with subprograms" on page 438

RELATED TASKS
"Making static calls" on page 436
"Making dynamic calls" on page 437

Making both static and dynamic calls

You can use both static and dynamic CALL statements in the same program if you compile the program with the NODYNAM compiler option.

In this case, with the CALL *literal* statement, the called subprogram will be link-edited with the main program into one load module. The CALL *identifier* statement results in the dynamic invocation of a separate load module.

When a dynamic CALL statement and a static CALL statement to the same subprogram are issued within one program, a second copy of the subprogram is loaded into storage. Because this arrangement does not guarantee that the subprogram will be left in its last-used state, results can be unpredictable.

RELATED REFERENCES
"DYNAM" on page 313

Examples: static and dynamic CALL statements

This example shows how you can code static and dynamic calls.

The example has three parts:
- Code that uses a static call to call a subprogram
- Code that uses a dynamic call to call the same subprogram
- The subprogram that is called by the two types of calls

The following example shows how you would code static calls:

```
PROCESS NODYNAM NODLL
IDENTIFICATION DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RECORD-2                PIC X.            (6)
01  RECORD-1.                                 (2)
    05  PAY                 PICTURE S9(5)V99.
    05  HOURLY-RATE         PICTURE S9V99.
    05  HOURS               PICTURE S99V9.
. . .
PROCEDURE DIVISION.
    CALL "SUBPROG" USING RECORD-1.            (1)
    CALL "PAYMASTR" USING RECORD-1 RECORD-2.  (5)
    STOP RUN.
```

The following example shows how you would code dynamic calls:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  PGM-NAME                PICTURE X(8).
01  RECORD-2                PIC x.            (6)
```

```
01  RECORD-1.                                       (2)
    05  PAY                   PICTURE S9(5)V99.
    05  HOURLY-RATE           PICTURE S9V99.
    05  HOURS                 PICTURE S99V9.
. . .
PROCEDURE DIVISION.
. . .
    MOVE "SUBPROG" TO PGM-NAME.
    CALL PGM-NAME USING RECORD-1.                   (1)
    CANCEL PGM-NAME.
    MOVE "PAYMASTR" TO PGM-NAME.                    (4)
    CALL PGM-NAME USING RECORD-1 RECORD-2.          (5)
    STOP RUN.
```

The following example shows a called subprogram that is called by each of the two preceding calling programs:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUBPROG.
DATA DIVISION.
LINKAGE SECTION.
01  PAYREC.                                         (2)
    10 PAY           PICTURE S9(5)V99.
    10 HOURLY-RATE   PICTURE S9V99.
    10 HOURS         PICTURE S99V9.
77  PAY-CODE         PICTURE 9.                     (6)
PROCEDURE DIVISION USING PAYREC.                    (1)
. . .
    EXIT PROGRAM.                                   (3)
    ENTRY "PAYMASTR" USING PAYREC PAY-CODE.         (5)
    . . .
    GOBACK.                                         (7)
```

(1) Processing begins in the calling program. When the first CALL statement is executed, control is transferred to the first statement of the PROCEDURE DIVISION in SUBPROG, which is the called program.

In each of the CALL statements, the operand of the first USING option is identified as RECORD-1.

(2) When SUBPROG receives control, the values within RECORD-1 are made available to SUBPROG; however, in SUBPROG they are referred to as PAYREC.

The PICTURE character-strings within PAYREC and PAY-CODE contain the same number of characters as RECORD-1 and RECORD-2, although the descriptions are not identical.

(3) When processing within SUBPROG reaches the EXIT PROGRAM statement, control is returned to the calling program. Processing continues in that program until the second CALL statement is issued.

(4) In the example of a dynamically called program, because the second CALL statement refers to another entry point within SUBPROG, a CANCEL statement is issued before the second CALL statement.

(5) With the second CALL statement in the calling program, control is again transferred to SUBPROG, but this time processing begins at the statement following the ENTRY statement in SUBPROG.

(6) The values within RECORD-1 are again made available to PAYREC. In addition, the value in RECORD-2 is now made available to SUBPROG through the corresponding USING operand, PAY-CODE.

When control is transferred the second time from the statically linked program, SUBPROG is made available in its last-used state (that is, if any values in SUBPROG storage were changed during the first execution, those changed values are still in effect). When control is transferred from the dynamically linked program, however, SUBPROG is made available in its initial state, because of the CANCEL statement that has been executed.

(7) When processing reaches the GOBACK statement, control is returned to the calling program at the statement immediately after the second CALL statement.

In any given execution of the called program and either of the two calling programs, if the values within RECORD-1 are changed between the time of the first CALL and the second, the values passed at the time of the second CALL statement will be the changed, not the original, values. If you want to use the original values, you must save them.

Calling nested COBOL programs

By calling nested programs, you can create applications that use structured programming techniques. You can also call nested programs instead of PERFORM procedures to prevent unintentional modification of data items. Use either CALL *literal* or CALL *identifier* statements to make calls to nested programs.

You can call a contained program only from its directly containing program unless you identify the contained program as COMMON in its PROGRAM-ID paragraph. In that case, you can call the *common program* from any program that is contained (directly or indirectly) in the same program as the common program. Only contained programs can be identified as COMMON. Recursive calls are not allowed.

Follow these guidelines when using nested program structures:
- Code an IDENTIFICATION DIVISION in each program. All other divisions are optional.
- Optionally make the name of each contained program unique. Although the names of contained programs are not required to be unique (as described in the related reference about scope of names), making the names unique could help make your application more maintainable. You can use any valid user-defined word or an alphanumeric literal as the name of a contained program.
- In the outermost program, code any CONFIGURATION SECTION entries that might be required. Contained programs cannot have a CONFIGURATION SECTION.
- Include each contained program in the containing program immediately before the END PROGRAM marker of the containing program.
- Use an END PROGRAM marker to terminate contained and containing programs.

You cannot use the THREAD option when compiling programs that contain nested programs.

RELATED CONCEPTS
"Nested programs"

RELATED REFERENCES
"Scope of names" on page 446

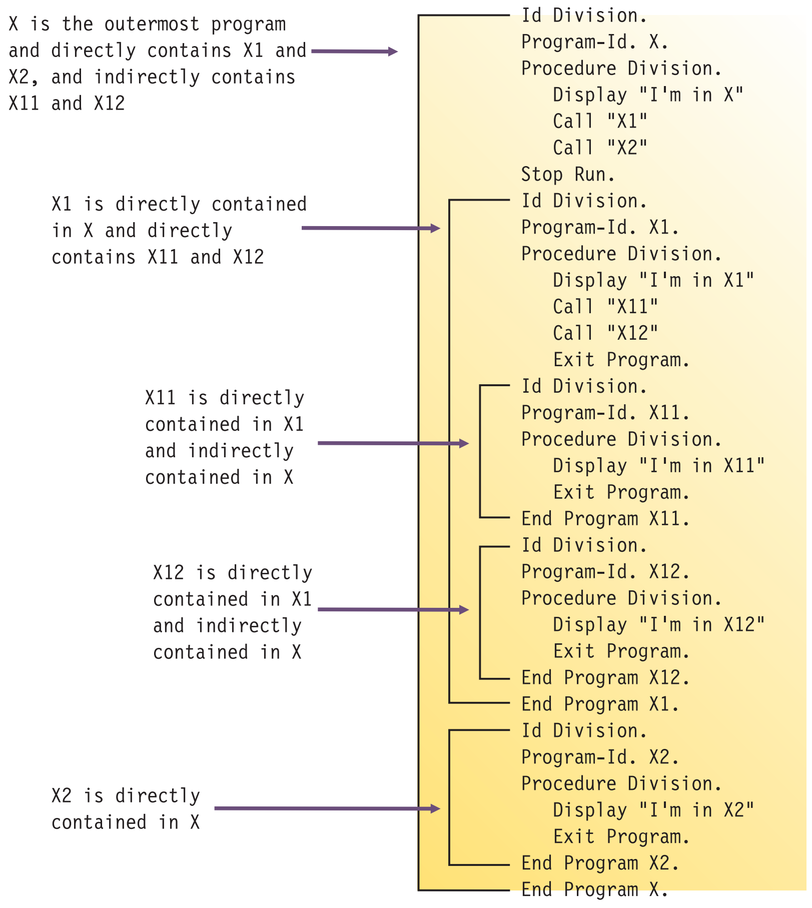
Nested programs

A COBOL program can *nest*, or contain, other COBOL programs. The nested programs can themselves contain other programs. A nested program can be directly or indirectly contained in a program.

There are four main advantages to nesting called programs:

- Nested programs provide a method for creating modular functions and maintaining structured programming techniques. They can be used analogously to PERFORM procedures, but with more structured control flow and with the ability to protect local data items.
- Nested programs let you debug a program before including it in the application.
- Nested programs let you compile an application with a single invocation of the compiler.
- Calls to nested programs have the best performance of all the forms of COBOL CALL statements.

The following example describes a nested structure that has directly and indirectly contained 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.
```

"Example: structure of nested programs"

RELATED TASKS
"Calling nested COBOL programs" on page 444

RELATED REFERENCES
"Scope of names" on page 446

Example: structure of nested programs

The following example shows a nested structure with some contained programs that are 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 A13.
End Program A.
```

The following table describes the calling hierarchy for the structure that is shown in the example above. Programs A12, A2, and A3 are identified as `COMMON`, and the calls associated with them differ.

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

In this example, 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.

Scope of names

Names in nested structures 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 (except the program-name) are local unless declared to be otherwise. Local names are visible or accessible only within the program in which they are declared. They are not visible or accessible to contained and containing programs.

Global names: A name that is global (indicated by 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. Therefore, the contained programs can share common data and files from the containing program simply by referencing the names of the items.

Any item that is subordinate to a global item (including condition-names and indexes) is automatically global.

You can declare the same name with the GLOBAL clause more than one time, provided that each declaration occurs in a different program. Be aware that you can mask, or hide, a name in a nested structure by having the same name occur in different programs in the same containing structure. However, such masking could cause problems during a search for a name declaration.

Searches 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 the containing programs until a match is found. The search follows this process:

1. Declarations in the program are searched.
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. If no match is found, an error exists.

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.

Making recursive calls

A called program can directly or indirectly execute its caller. For example, program X calls program Y, program Y calls program Z, and program Z then calls program X. This type of call is *recursive*.

To make a recursive call, you must code the RECURSIVE clause in the PROGRAM-ID paragraph of the recursively called program. If you try to recursively call a COBOL program that does not have the RECURSIVE clause in the PROGRAM-ID paragraph, a condition is signaled. If the condition remains unhandled, the run unit will end.

RELATED TASKS
"Identifying a program as recursive" on page 6

RELATED REFERENCES
PROGRAM-ID paragraph (*Enterprise COBOL Language Reference*)

Calling to and from object-oriented programs

When you create applications that contain object-oriented (OO) programs, the OO COBOL programs are DLL programs and can be in one or more dynamic link libraries (DLLs). Each class definition must be in a separate DLL, however.

Calls to or from COBOL DLL programs must either use DLL linkage or be static calls. COBOL dynamic calls to or from COBOL DLL programs are not supported.

If you must call a COBOL DLL program from a COBOL non-DLL program, other means to ensure that the DLL linkage mechanism is followed are available.

Using procedure and function pointers

You can set procedure-pointer and function-pointer data items only by using format 6 of the SET statement.

Procedure pointers are data items defined with the USAGE IS PROCEDURE-POINTER clause. *Function pointers* are data items defined with the USAGE IS FUNCTION-POINTER clause. In this information, "pointer" refers to either a procedure-pointer data item or a function-pointer data item. You can set either of these data items to contain entry addresses of, or pointers to, these entry points:

- Another COBOL program that is not nested. For example, to have a user-written error-handling routine take control when an exception condition occurs, you must first pass the entry address of the routine to CEEHDLR, a condition-management Language Environment callable service, so that the routine is registered.
- A program written in another language. For example, to receive the entry address of a C function, call the function with the CALL RETURNING statement. It will return a pointer that you can either use as a function pointer or convert to a procedure pointer by using a form of the SET statement.
- An alternate entry point in another COBOL program (as defined in an ENTRY statement).

The SET statement sets the pointer to refer either to an entry point in the same load module as your program, to a separate load module, or to an entry point that is exported from a DLL, depending on the DYNAM|NODYNAM and DLL|NODLL compiler options. Therefore, consider these factors when using these pointer data items:

- If you compile a program with the NODYNAM and NODLL options and set a pointer item to a literal value (to an actual name of an entry point), the value must refer to an entry point in the same load module. Otherwise the reference cannot be resolved.
- If you compile a program with the NODLL option and either set a pointer item to an identifier that will contain the name of the entry point at run time or set the pointer item to a literal and compile with the DYNAM option, then the pointer item, whether a literal or variable, must point to an entry point in a separate load module. The entry point can be either the primary entry point or an alternate entry point named in an ALIAS linkage-editor or binder statement.
- If you compile with the NODYNAM and DLL options and set a pointer item to a literal value (the actual name of an entry point), the value must refer to an entry point in the same load module or to an entry-point name that is exported from a DLL module. In the latter case you must include the DLL side file for the target DLL module in the link edit of your program load module.
- If you compile with the NODYNAM and DLL options and set a pointer item to an identifier (a data item that contains the entry point name at run time), the identifier value must refer to the entry-point name that is exported from a DLL module. In this case the DLL module name must match the name of the exported entry point.

If you set a pointer item to an entry address in a dynamically called load module, and your program subsequently cancels that dynamically called module, then that pointer item becomes undefined. Reference to it thereafter is not reliable.

RELATED TASKS
"Deciding which type of pointer to use" on page 449
"Calling alternate entry points" on page 449
"Using procedure or function pointers with DLLs" on page 472

RELATED REFERENCES
"DLL" on page 311
"DYNAM" on page 313

CANCEL statement (*Enterprise COBOL Language Reference*)
Format 6: SET for procedure-pointer and function-pointer data items (*Enterprise COBOL Language Reference*)
ENTRY statement (*Enterprise COBOL Language Reference*)

Deciding which type of pointer to use

Use procedure pointers to call other COBOL programs and to call Language Environment callable services. Use function pointers to communicate with C/C++ programs or with services provided by the Java Native Interface.

Procedure pointers are more efficient than function pointers for COBOL-to-COBOL calls, and are required for calls to Language Environment condition-handling services.

Many callable services written in C return function pointers. You can call such a C function pointer from your COBOL program by using COBOL function pointers as shown below.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. DEMO.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
*
 WORKING-STORAGE SECTION.
 01 FP USAGE FUNCTION-POINTER.
*
 PROCEDURE DIVISION.
     CALL "c-function" RETURNING FP.
     CALL FP.
```

RELATED TASKS
"Using procedure or function pointers with DLLs" on page 472
"Accessing JNI services" on page 569

Calling alternate entry points

Static calls to alternate entry points work without restriction.

Dynamic calls to alternate entry points require the following elements:

+ • Either explicitly specified `NAME` or `ALIAS` linkage-editor or binder control
+ statements, or use of the `NAME` compiler option which generates them
+ automatically.
- An intervening `CANCEL` for any dynamic call to the same module at a different entry point. `CANCEL` causes the program to be invoked in initial state when it is called at a new entry point.

You can specify another entry point at which a program will begin running by using the `ENTRY` label in the called program. However, this method is not recommended in a structured program.

"Examples: static and dynamic CALL statements" on page 442

RELATED REFERENCES
"NAME" on page 323
CANCEL statement (*Enterprise COBOL Language Reference*)
ENTRY statement (*Enterprise COBOL Language Reference*)
MVS Program Management: User's Guide and Reference

Making programs reentrant

If more than one user will run an application program at the same time (for example, users in different address spaces accessing a program that resides in the link pack area), you must make the program *reentrant* by compiling with the `RENT` option.

You do not need to worry about multiple copies of variables. The compiler creates the necessary reentrancy controls in the object module.

The following Enterprise COBOL programs must be reentrant:

- Programs to be used with CICS
- Programs to be preloaded with IMS
- Programs to be used as DB2 stored procedures
- Programs to be run in the UNIX environment
- Programs that are enabled for DLL support
- Programs that use object-oriented syntax

For reentrant programs, use the `DATA` compiler option and the `HEAP` and `ALL31` runtime options to control whether dynamic data areas, such as `WORKING-STORAGE`, are obtained from storage below or above the 16-MB line.

RELATED CONCEPTS
"Storage and its addressability" on page 41

RELATED TASKS
"Compiling programs to create DLLs" on page 466
Chapter 16, "Compiling, linking, and running OO applications," on page 287

RELATED REFERENCES
"RENT" on page 331
"DATA" on page 307
ALL31 run-time option (*Language Environment Programming Reference*)
HEAP run-time option (*Language Environment Programming Reference*)

Chapter 25. Sharing data

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 access to common data.

This information describes how you can write programs that share data with other programs. In this information, a *subprogram* is any program that is called by another program.

RELATED TASKS
"Passing data"
"Coding the LINKAGE SECTION" on page 455
"Coding the PROCEDURE DIVISION for passing arguments" on page 455
"Passing return-code information" on page 459
"Specifying CALL . . . RETURNING" on page 460
"Sharing data by using the EXTERNAL clause" on page 461
"Sharing files between programs (external files)" on page 461
"Sharing data with Java" on page 573

Passing data

You can choose among three ways of passing data between programs: `BY REFERENCE`, `BY CONTENT`, or `BY VALUE`.

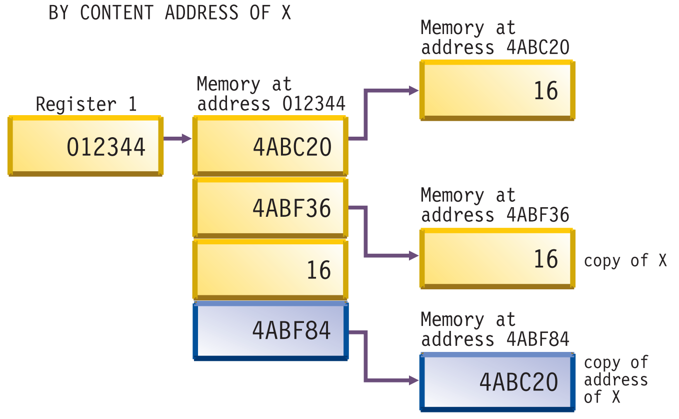
`BY REFERENCE`
The subprogram refers to and processes the data items in the storage of the calling program rather than working on a copy of the data. `BY REFERENCE` is the assumed passing mechanism for a parameter if none of the three ways is specified or implied for the parameter.

`BY CONTENT`
The calling program passes only the contents of the *literal* or *identifier*. The called program cannot change the value of the *literal* or *identifier* in the calling program, even if it modifies the data item in which it received the *literal* or *identifier*.

`BY VALUE`
The calling program or method passes the value of the *literal* or *identifier*, not a reference to the sending data item. The called program or invoked method can change the parameter. However, because the subprogram or method has access only to a temporary copy of the sending data item, any change does not affect the argument in the calling program.

The following figure shows the differences in values passed `BY REFERENCE`, `BY CONTENT`, and `BY VALUE`:

```
MOVE 16 TO X.
CALL ABC USING
    BY REFERENCE X
    BY CONTENT X
    BY VALUE X
    BY CONTENT ADDRESS OF X
```



Determine which of these data-passing methods to use based on what you want your program to do with the data.

Table 65. **Methods for passing data in the CALL statement**

Code	Purpose	Comments
`CALL . . . BY REFERENCE` *identifier*	To have the definition of the argument of the `CALL` statement in the calling program and the definition of the parameter in the called program share the same memory	Any changes made by the subprogram to the parameter affect the argument in the calling program.
`CALL . . . BY REFERENCE ADDRESS OF` *identifier*	To pass the address of *identifier* to a called program, where *identifier* is an item in the `LINKAGE SECTION`	Any changes made by the subprogram to the address affect the address in the calling program.
`CALL . . . BY REFERENCE` *file-name*	To pass a Data Control Block (DCB) to assembler programs	The file-name must reference a QSAM sequential file.[1]
`CALL . . . BY CONTENT ADDRESS OF` *identifier*	To pass a copy of the address of *identifier* to a called program	Any changes to the copy of the address will not affect the address of *identifier*, but changes to *identifier* using the copy of the address will cause changes to *identifier*.
`CALL . . . BY CONTENT` *identifier*	To pass a copy of the identifier to the subprogram	Changes to the parameter by the subprogram will not affect the caller's identifier.
`CALL . . . BY CONTENT` *literal*	To pass a copy of a literal value to a called program	
`CALL . . . BY CONTENT LENGTH OF` *identifier*	To pass a copy of the length of a data item	The calling program passes the length of the *identifier* from its `LENGTH` special register.
A combination of `BY REFERENCE` and `BY CONTENT` such as: `CALL 'ERRPROC' USING BY REFERENCE A BY CONTENT LENGTH OF A.`	To pass both a data item and a copy of its length to a subprogram	
`CALL . . . BY VALUE` *identifier*	To pass data to a program, such as a C/C++ program, that uses `BY VALUE` parameter linkage conventions	A copy of the identifier is passed directly in the parameter list.

Table 65. **Methods for passing data in the CALL statement** *(continued)*

Code	Purpose	Comments
`CALL . . . BY VALUE` *literal*	To pass data to a program, such as a C/C++ program, that uses `BY VALUE` parameter linkage conventions	A copy of the literal is passed directly in the parameter list.
`CALL . . . BY VALUE ADDRESS OF` *identifier*	To pass the address of *identifier* to a called program. This is the recommended way to pass data to a C/C++ program that expects a pointer to the data.	Any changes to the copy of the address will not affect the address of *identifier*, but changes to *identifier* using the copy of the address will cause changes to *identifier*.
`CALL . . . RETURNING`	To call a C/C++ function with a function return value	
1. File-names as `CALL` operands are allowed as an IBM extension to COBOL. Any use of the extension generally depends on the specific internal implementation of the compiler. Control block field settings might change in future releases. Any changes made to the control block are the user's responsibility and are not supported by IBM.		

RELATED CONCEPTS
"Storage and its addressability" on page 41

RELATED TASKS
"Describing arguments in the calling program"
"Describing parameters in the called program" on page 454
"Testing for OMITTED arguments" on page 454
"Specifying CALL . . . RETURNING" on page 460
"Sharing data by using the EXTERNAL clause" on page 461
"Sharing files between programs (external files)" on page 461
"Sharing data with Java" on page 573

RELATED REFERENCES
CALL statement (*Enterprise COBOL Language Reference*)
The USING phrase (*Enterprise COBOL Language Reference*)
INVOKE statement (*Enterprise COBOL Language Reference*)

Describing arguments in the calling program

In the calling program, describe arguments in the `DATA DIVISION` in the same manner as other data items in the `DATA DIVISION`.

Storage for arguments is allocated only in the highest outermost program. For example, program A calls program B, which calls program C. Data items are allocated in program A. They are described in the `LINKAGE SECTION` of programs B and C, making the one set of data 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` phrase of the `CALL` statement to pass the arguments. If you pass a
| data item `BY VALUE`, it must be an elementary item.

Do not pass parameters allocated in storage above the 16-MB line to `AMODE 24` subprograms. Use the `DATA(24)` option if the `RENT` option is in effect, or the `RMODE(24)` option if the `NORENT` option is in effect.

RELATED CONCEPTS
"Storage and its addressability" on page 41

RELATED TASKS
"Coding the LINKAGE SECTION" on page 455
"Coding the PROCEDURE DIVISION for passing arguments" on page 455

RELATED REFERENCES
The USING phrase (*Enterprise COBOL Language Reference*)

Describing parameters in the called program

You must know what data is being passed from the calling program and describe it in the LINKAGE SECTION of each program that is called directly or indirectly by the calling program.

Code the USING phrase after the PROCEDURE DIVISION header to name the parameters that receive the data that is passed from the calling program.

When arguments are passed to the subprogram BY REFERENCE, it is invalid for the subprogram to specify any relationship between its parameters and any fields other than those that are passed and defined in the main program. The subprogram must not:

- Define a parameter to be larger in total number of bytes than the corresponding argument.
- Use subscript references to refer to elements beyond the limits of tables that are passed as arguments by the calling program.
- Use reference modification to access data beyond the length of defined parameters.
- Manipulate the address of a parameter in order to access other data items that are defined in the calling program.

If any of the rules above are violated, unexpected results might occur if the calling program was compiled with the OPTIMIZE compiler option.

RELATED TASKS
"Coding the LINKAGE SECTION" on page 455

RELATED REFERENCES
The USING phrase (*Enterprise COBOL Language Reference*)

Testing for OMITTED arguments

You can specify that one or more BY REFERENCE arguments are not to be passed to a called program by coding the OMITTED keyword in place of those arguments in the CALL statement.

For example, to omit the second argument when calling program sub1, code this statement:

```
Call 'sub1' Using PARM1, OMITTED, PARM3
```

The arguments in the USING phrase of the CALL statement must match the parameters of the called program in number and position.

In a called program, you can test whether an argument was passed as OMITTED by comparing the address of the corresponding parameter to NULL. For example:

```
Program-ID. sub1.
. . .
Procedure Division Using RPARM1, RPARM2, RPARM3.
    If Address Of RPARM2 = Null Then
        Display 'No 2nd argument was passed this time'
    Else
        Perform Process-Parm-2
    End-If
```

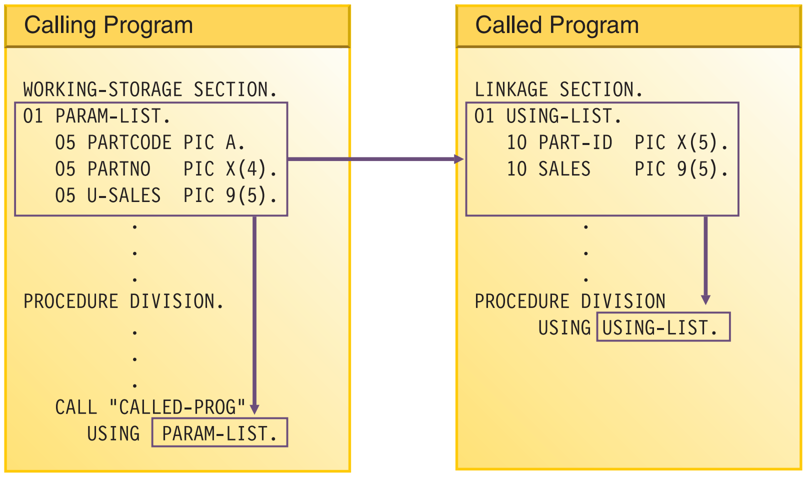
RELATED REFERENCES
CALL statement (*Enterprise COBOL Language Reference*)
The USING phrase (*Enterprise COBOL Language Reference*)

Coding the LINKAGE SECTION

Code the same number of data-names in the identifier list of the called program as the number of arguments in the calling program. Synchronize by position, because the compiler passes the first argument from the calling program to the first identifier of the called program, and so on.

You will introduce errors if the number of data-names in the identifier list of a called program is greater than the number of arguments passed from the calling program. The compiler does not try to match arguments and parameters.

The following figure shows a data item being passed from one program to another (implicitly BY REFERENCE):


Calling Program

```
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.
```

Called Program

```
LINKAGE SECTION.
01 USING-LIST.
   10 PART-ID  PIC X(5).
   10 SALES    PIC 9(5).
          .
          .
          .
PROCEDURE DIVISION
      USING USING-LIST.
```

In the calling program, the code for parts (PARTCODE) and the part number (PARTNO) are distinct data items. In the called program, by contrast, 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 these items.

Coding the PROCEDURE DIVISION for passing arguments

If you pass an argument BY VALUE, code the USING BY VALUE clause in the PROCEDURE DIVISION header of the subprogram. If you pass an argument BY REFERENCE or BY CONTENT, you do not need to indicate in the header how the argument was passed.

```
PROCEDURE DIVISION USING BY VALUE. . .

PROCEDURE DIVISION USING. . .
PROCEDURE DIVISION USING BY REFERENCE. . .
```

The first header above indicates that the data items are passed `BY VALUE`; the second or third headers indicate that the items are passed `BY REFERENCE` or `BY CONTENT`.

RELATED REFERENCES
The procedure division header (*Enterprise COBOL Language Reference*)
The USING phrase (*Enterprise COBOL Language Reference*)
CALL statement (*Enterprise COBOL Language Reference*)

Grouping data to be passed

Consider grouping all the data items that you need to pass between programs and putting them under one level-01 item. If you do so, you can pass a single level-01 record.

| Note that if you pass a data item `BY VALUE`, it must be an elementary item.

To lessen the possibility of mismatched records, put the level-01 record into a copy library and copy it into both programs. That is, copy it in the `WORKING-STORAGE SECTION` of the calling program and in the `LINKAGE SECTION` of the called program.

RELATED TASKS
"Coding the LINKAGE SECTION" on page 455

RELATED REFERENCES
CALL statement (*Enterprise COBOL Language Reference*)

Handling null-terminated strings

COBOL supports null-terminated strings when you use string-handling verbs together with null-terminated literals and the hexadecimal literal `X'00'`.

You can manipulate null-terminated strings (passed from a C program, for example) by using string-handling mechanisms such as those in the following code:

```
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!'.
```

To determine the length of a null-terminated string, and display the value of the string and its length, code:

```
Inspect N tallying N-length for characters before initial X'00'
Display 'N: ' N(1:N-length) ' Length: ' N-length
```

To move a null-terminated string to an alphanumeric string, but delete the null, code:

```
Unstring N  delimited by X'00' into X
```

To create a null-terminated string, code:

```
String Y      delimited by size
       X'00'  delimited by size
       into N.
```

To concatenate two null-terminated strings, code:

```
String L      delimited by x'00'
       M      delimited by x'00'
       X'00'  delimited by size
       into N.
```

RELATED TASKS
"Manipulating null-terminated strings" on page 106

RELATED REFERENCES
Null-terminated alphanumeric literals (*Enterprise COBOL Language Reference*)

Using pointers to process a chained list

When you need to pass and receive addresses of record areas, you can use pointer data items, which are either data items that are 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 in which each record points to the next.

When you pass addresses between programs in a chained list, you can use NULL to assign the value of an address that is not valid (nonnumeric 0) to a pointer item in either of two ways:

- Use a VALUE IS NULL clause in its data definition.
- Use NULL as 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, you can use this code to check for the end of the list:

```
IF PTR-NEXT-REC = NULL
. . .
 (logic for end of chain)
```

If the program has not reached the end of the list, the program can process the record and move on to the next record.

The data passed from a calling program might contain header information that you want to ignore. Because pointer data items are not numeric, you cannot directly perform arithmetic on them. However, to bypass header information, you can use the SET statement to increment the passed address.

"Example: using pointers to process a chained list"

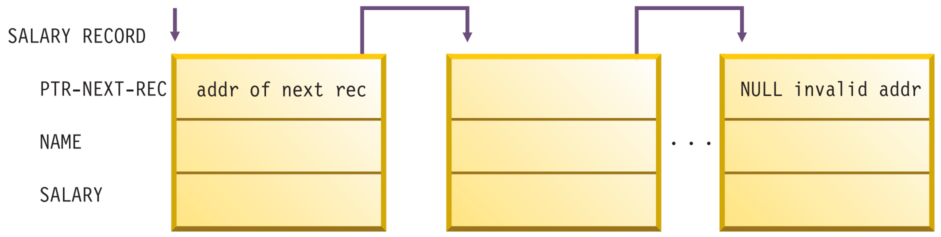
RELATED TASKS
"Coding the LINKAGE SECTION" on page 455
"Coding the PROCEDURE DIVISION for passing arguments" on page 455

RELATED REFERENCES
SET statement (*Enterprise COBOL Language Reference*)

Example: using pointers to process a chained list

The following example shows how you might process a linked list, that is, a chained list of data items.

For this example, picture a chained list of data that consists of individual salary records. The following figure shows one way to visualize how the records are linked in storage. The first item in each record except the last points to the next record. The first item in the last record contains a null value (instead of a valid address) to indicate that it is the last record.



The high-level logic of an application that processes these records might be:

```
Obtain address of first record in chained list from routine
Check for end of the list
DO UNTIL end of the list
   Process record
   Traverse to the next record
END
```

The following code contains an outline of the calling program, `LISTS`, used in this example of processing a chained list.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. LISTS.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
******
 WORKING-STORAGE SECTION.
 77  PTR-FIRST         POINTER  VALUE IS NULL.                 (1)
 77  DEPT-TOTAL        PIC 9(4) VALUE IS 0.
******
 LINKAGE SECTION.
 01  SALARY-REC.
     02  PTR-NEXT-REC    POINTER.                              (2)
     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                         (3)
     SET ADDRESS OF SALARY-REC TO PTR-FIRST                    (4)
******
     PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL (5)

      IF DEPT = DEPT-X
        THEN ADD SALARY TO DEPT-TOTAL
        ELSE CONTINUE
      END-IF
      SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC                (6)
     END-PERFORM
******
     DISPLAY DEPT-TOTAL
     GOBACK.
```

(1) PTR-FIRST is defined as a pointer data item with an initial value of NULL. On a successful return from the call to CHAIN-ANCH, PTR-FIRST contains the address of the first record in the chained list. If something goes wrong 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.

(2) 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.

(3) To obtain the address of the first SALARY-REC record area, the LISTS program calls the program CHAIN-ANCH:

(4) The SET statement bases the record description SALARY-REC on the address contained in PTR-FIRST.

(5) The chained list in this example is set up so that the last record contains an address that is not valid. This check for the end of the chained list is accomplished with a do-while structure where the value NULL is assigned to the pointer data item in the last record.

(6) The address of the record in the LINKAGE-SECTION is set equal to the address of the next record by means of the pointer data item sent as the first field in SALARY-REC. The record-processing routine repeats, processing the next record in the chained list.

To increment addresses received from another program, you could set up the LINKAGE SECTION and PROCEDURE DIVISION 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).
. . .
PROCEDURE DIVISION USING DEPT-X.
. . .
    SET ADDRESS OF SALARY-REC TO ADDRESS OF REAL-SALARY-REC
```

The address of SALARY-REC is now based on the address of REAL-SALARY-REC, or RECORD-A + 12.

RELATED TASKS
"Using pointers to process a chained list" on page 457

Passing return-code information

Use the RETURN-CODE special register to pass 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 in the PROCEDURE DIVISION header of a 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.

Understanding the RETURN-CODE special register

When a COBOL program returns to its caller, the contents of the RETURN-CODE special register are stored into register 15.

When control is returned to a COBOL program or method from a call, the contents of register 15 are stored into the RETURN-CODE special register of the calling program or method. When control is returned from a COBOL program to the operating system, the special register contents are returned as a user return code.

You might need to think about this handling of the RETURN-CODE special register when control is returned to a COBOL program from a non-COBOL program. If the non-COBOL program does not use register 15 to pass back the return code, the RETURN-CODE special register of the COBOL program might be updated with an invalid value. Unless you set this special register to a meaningful value before your Enterprise COBOL program returns to the operating system, a return code that is invalid will be passed to the system.

For equivalent function between COBOL and C programs, have your COBOL program call the C program with the RETURNING phrase. If the C program (function) correctly declares a function value, the RETURNING value of the calling COBOL program will be set.

You cannot set the RETURN-CODE special register by using the INVOKE statement.

Using PROCEDURE DIVISION RETURNING . . .

Use the RETURNING phrase in the PROCEDURE DIVISION header of a program to return information to the calling program.

```
PROCEDURE DIVISION RETURNING dataname2
```

When the called program in the example above successfully returns to its caller, the value in *dataname2* is stored into the identifier that you specified in the RETURNING phrase of the CALL statement:

```
CALL . . . RETURNING dataname2
```

CEEPIPI: The results of specifying PROCEDURE DIVISION RETURNING in programs that are called with the Language Environment preinitialization service (CEEPIPI) are undefined.

Specifying CALL . . . RETURNING

You can specify the RETURNING phrase of the CALL statement for calls to C/C++ functions or to COBOL subroutines.

The RETURNING phrase has the following format.

```
CALL . . . RETURNING dataname2
```

The return value of the called program is stored into *dataname2*. You must define *dataname2* in the DATA DIVISION of the calling program. The data type of the return value that is declared in the target function must be identical to the data type of *dataname2*.

Sharing data by using the EXTERNAL clause

Use the EXTERNAL clause to allow separately compiled programs and methods (including programs in a batch sequence) to share data items. Code EXTERNAL in the level-01 data description in the WORKING-STORAGE SECTION.

The following rules apply:

- Items that are subordinate to an EXTERNAL group item are themselves EXTERNAL.
- You cannot use the name of an EXTERNAL data item as the name for another EXTERNAL item in the same program.
- You cannot code the VALUE clause for any group item or subordinate item that is EXTERNAL.

In the run unit, any COBOL program or method that has the same data description for the item as the program that contains the item can access and process that item. For example, suppose program A has the following data description:

```
01 EXT-ITEM1     EXTERNAL     PIC 99.
```

Program B can access that data item if it has the identical data description in its WORKING-STORAGE SECTION.

Any program that has access to an EXTERNAL data item can change the value of that item. Therefore do not use this clause for data items that you need to protect.

Sharing files between programs (external files)

To enable separately compiled programs or methods in a run unit to access a file as a common file, use the EXTERNAL clause for the file.

It is recommended that you follow these guidelines:

- Use the same data-name in the FILE STATUS clause of all the programs that check the file status code.
- For each program that checks the same file status field, code the EXTERNAL clause on the level-01 data definition for the file status field.

Using an external file has these benefits:

- Even though the main program does not contain any input or output statements, it can reference the record area of the file.
- Each subprogram can control a single input or output function, such as OPEN or READ.
- Each program has access to the file.

“Example: using external files” on page 462

RELATED TASKS
“Using data in input and output operations” on page 13

RELATED REFERENCES
EXTERNAL clause (*Enterprise COBOL Language Reference*)

Example: using external files

The following example shows the use of an external file in several programs. COPY statements ensure that each subprogram contains an identical description of the file.

The table below describes the main program and subprograms.

Name	Function
ef1	The main program, which calls all the subprograms and then verifies the contents of a record area
ef1openo	Opens the external file for output and checks the file status code
ef1write	Writes a record to the external file and checks the file status code
ef1openi	Opens the external file for input and checks the file status code
ef1read	Reads a record from the external file and checks the file status code
ef1close	Closes the external file and checks the file status code

Each program uses three copybooks:

- efselect is placed in the FILE-CONTROL paragraph.

```
Select ef1
Assign To ef1
File Status Is efs1
Organization Is Sequential.
```

- effile 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).
```

- efwrkstg is placed in the WORKING-STORAGE SECTION.

```
01  efs1         Pic 99 External.
```

Input-output using external files

```
 Identification Division.
 Program-Id.
     ef1.
*
* This main program controls 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.
     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.
     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.
     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.
```

Chapter 26. Creating a DLL or a DLL application

Creating a dynamic link library (DLL) or a DLL application is similar to creating a regular COBOL application. It involves writing, compiling, and linking your source code.

Special considerations when writing a DLL or a DLL application include:

- Determining how the parts of the load module or the application relate to each other or to other DLLs
- Deciding what linking or calling mechanisms to use

Depending on whether you want to create a DLL load module or a load module that references a separate DLL, you need to use slightly different compiler and linkage-editor or binder options.

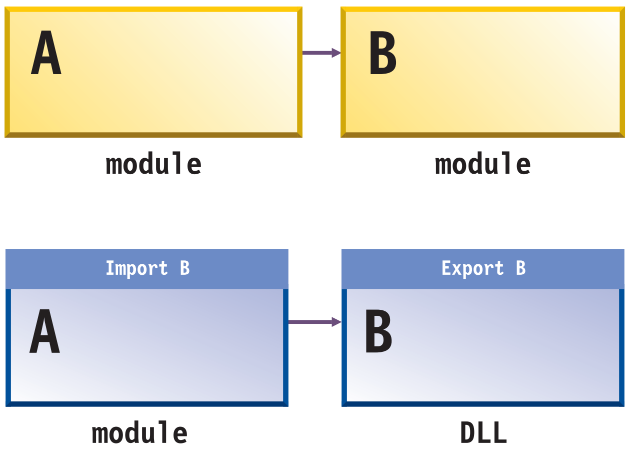
RELATED CONCEPTS
"Dynamic link libraries (DLLs)"

RELATED TASKS
"Creating a DLL under UNIX" on page 282
"Compiling programs to create DLLs" on page 466
"Linking DLLs" on page 467
"Using CALL identifier with DLLs" on page 469
"Using DLL linkage and dynamic calls together" on page 470
"Using COBOL DLLs with C/C++ programs" on page 474
"Using DLLs in OO COBOL applications" on page 475
"Using procedure or function pointers with DLLs" on page 472

Dynamic link libraries (DLLs)

A DLL is a load module or a program object that can be accessed from other separate load modules.

A DLL differs from a traditional load module in that it *exports* definitions of programs, functions, or variables to DLLs, DLL applications, or non-DLLs. Therefore, you do not need to link the target routines into the same load module as the referencing routine. When an application references a separate DLL for the first time, the system automatically loads the DLL into memory. In other words, calling a program in a DLL is similar to calling a load module with a dynamic `CALL`.



A DLL application is an application that references imported definitions of programs, functions, or variables.

Although some functions of z/OS DLLs overlap the functions provided by COBOL dynamic `CALL` statements, DLLs have several advantages over regular z/OS load modules and dynamic calls:

- DLLs are common across COBOL and C/C++, thus providing better interoperation for applications that use multiple programming languages. Reentrant COBOL and C/C++ DLLs can also interoperate smoothly.
- You can make calls to programs in separate DLL modules that have long program-names. (Dynamic call resolution truncates program-names to eight characters.) Using the COBOL option `PGMNAME(LONGUPPER)` or `PGMNAME(LONGMIXED)` and the COBOL DLL support, you can make calls between load modules with names of up to 160 characters.

DLLs are supported by IBM z/OS Language Environment, based on function provided by the z/OS program management binder. DLL support is available for applications running under z/OS in batch or in TSO, CICS, UNIX, or IMS environments.

RELATED REFERENCES
"PGMNAME" on page 329
Binder support for DLLs (*MVS Program Management: User's Guide and Reference*)

Compiling programs to create DLLs

When you compile a COBOL program with the `DLL` option, it becomes enabled for DLL support. Applications that use DLL support must be reentrant. Therefore, you must compile them with the `RENT` compiler option and link them with the `RENT` binder option.

In an application with DLL support, use the following compiler options depending on where the programs or classes are:

Table 66. **Compiler options for DLL applications**

Programs or classes in:	**Compile with:**
Root load module	`DLL`, `RENT`, `NOEXPORTALL`
DLL load modules used by other load modules	`DLL`, `RENT`, `EXPORTALL`

If a DLL load module includes some programs that are used only from within the DLL module, you can hide these routines by compiling them with `NOEXPORTALL`.

"Example: sample JCL for a procedural DLL application" on page 468

RELATED TASKS
"Creating a DLL under UNIX" on page 282
"Linking DLLs"
"Prelinking certain DLLs" on page 469
Chapter 26, "Creating a DLL or a DLL application," on page 465

RELATED REFERENCES
"DLL" on page 311
"EXPORTALL" on page 313
"RENT" on page 331

Linking DLLs

You can link DLL-enabled object modules into separate DLL load modules, or you can link them together statically. You can decide whether to package the application as one module or as several DLL modules at link time.

The DLL support in the z/OS binder is recommended for linking DLL applications. The binder can directly receive the output of COBOL compilers, thus eliminating the prelink step. However, you must use the Language Environment prelinker before standard linkage editing if your DLL must reside in a PDS load library.

A binder-based DLL must reside in a PDSE or in an HFS file rather than in a PDS.

When using the binder to link a DLL application, use the following options:

Table 67. **Binder options for DLL applications**

Type of code	Link using binder parameters:
DLL applications	`DYNAM(DLL)`, `RENT`
Applications that use mixed-case exported program-names Class definitions or `INVOKE` statements	`CASE(MIXED)`

You must specify a `SYSDEFSD DD` statement to indicate the data set in which the binder should create a DLL definition side file. This side file contains `IMPORT` control statements for each symbol exported by a DLL. The binder SYSLIN input (the binding code that references the DLL code) must include the DLL definition side files for DLLs that are to be referenced from the module being linked.

If there are programs in the module that you do not want to make available with DLL linkage, you can edit the definition side file to remove these programs.

"Example: sample JCL for a procedural DLL application" on page 468

RELATED TASKS
"Creating a DLL under UNIX" on page 282

Chapter 26, “Creating a DLL or a DLL application,” on page 465
“Compiling programs to create DLLs” on page 466
“Prelinking certain DLLs” on page 469

RELATED REFERENCES
Binder support for DLLs (*MVS Program Management: User's Guide and Reference*)

Example: sample JCL for a procedural DLL application

The following example shows how to create an application that consists of a main program that calls a DLL subprogram.

The first step creates the DLL load module that contains the subprogram `DemoDLLSubprogram`. The second step creates the main load module that contains the program `MainProgram`. The third step runs the application.

```
//DLLSAMP  JOB ,
//  TIME=(1),MSGLEVEL=(1,1),MSGCLASS=H,CLASS=A,
//  NOTIFY=&SYSUID,USER=&SYSUID
// SET LEPFX='SYS1'
//*---------------------------------------------------------------------
//* Compile COBOL subprogram, bind to form a DLL.
//*---------------------------------------------------------------------
//STEP1 EXEC IGYWCL,REGION=80M,GOPGM=DEMODLL,
//       PARM.COBOL='RENT,PGMN(LM),DLL,EXPORTALL',
//       PARM.LKED='RENT,LIST,XREF,LET,MAP,DYNAM(DLL),CASE(MIXED)'
//COBOL.SYSIN    DD *
       Identification division.
       Program-id. "DemoDLLSubprogram".
       Procedure division.
           Display "Hello from DemoDLLSubprogram!".
       End program "DemoDLLSubprogram".
/*
//LKED.SYSDEFSD DD DSN=&&SIDEDECK,UNIT=SYSDA,DISP=(NEW,PASS),
//            SPACE=(TRK,(1,1))
//LKED.SYSLMOD   DD DSN=&&GOSET(&GOPGM),DSNTYPE=LIBRARY,DISP=(MOD,PASS)
//LKED.SYSIN     DD DUMMY
//*---------------------------------------------------------------------
//* Compile and bind COBOL main program
//*---------------------------------------------------------------------
//STEP2 EXEC IGYWCL,REGION=80M,GOPGM=MAINPGM,
//       PARM.COBOL='RENT,PGMNAME(LM),DLL',
//       PARM.LKED='RENT,LIST,XREF,LET,MAP,DYNAM(DLL),CASE(MIXED)'
//COBOL.SYSIN    DD *
       Identification division.
       Program-id. "MainProgram".
       Procedure division.
           Call "DemoDLLSubprogram"
           Stop Run.
       End program "MainProgram".
/*
//LKED.SYSIN    DD DSN=&&SIDEDECK,DISP=(OLD,DELETE)
//*---------------------------------------------------------------------
//* Execute the main program, calling the subprogram DLL.
//*---------------------------------------------------------------------
//STEP3 EXEC PGM=MAINPGM,REGION=80M
//STEPLIB DD DSN=&&GOSET,DISP=(OLD,DELETE)
//        DD DSN=&LEPFX..SCEERUN,DISP=SHR
//SYSOUT  DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
```

Prelinking certain DLLs

You must use the Language Environment prelinker before standard linkage editing if a DLL must reside in a PDS load library rather than in a PDSE or an HFS file.

After compiling the DLL source, prelink the object modules to form a single object module:

1. Specify a SYSDEFSD DD statement for the prelink step to indicate the data set in which the prelinker should create a DLL definition side file. The side file contains IMPORT prelinker control statements for each symbol exported by the DLL. The prelinker uses this side file to prelink other modules that reference the new DLL.
2. Specify the DLLNAME(*xxx*) prelinker option to indicate the DLL load module name for the prelinker to use in constructing the IMPORT control statements in the side file. Alternatively, the prelinker can obtain the DLL load module name from the NAME prelinker control statement or from the PDS member name in the SYSMOD DD statement for the prelink step.
3. If the new DLL references any other DLLs, include the definition side files for these DLLs together with the object decks that are input to this prelink step. These side files instruct the prelinker to resolve the symbolic references in the current module to the symbols exported from the other DLLs.

Use the linkage editor or binder as usual to create the DLL load module from the object module produced by the prelinker. Specify the RENT option of the linkage editor or binder.

RELATED TASKS
"Compiling programs to create DLLs" on page 466
"Linking DLLs" on page 467

Using CALL identifier with DLLs

In a COBOL program that has been compiled with the DLL option, you can use CALL *identifier* and CALL *literal* statements to make calls to DLLs. However, there are a few additional considerations for the CALL *identifier* case.

For the content of the *identifier* or for the *literal*, use the name of either of the following programs:

- A nested program in the same compilation unit that is eligible to be called from the program that contains the CALL *identifier* statement.
- A program in a separately bound DLL module. The target program-name must be exported from the DLL, and the DLL module name must match the exported name of the target program.

In the nonnested case, the runtime environment interprets the program-name in the *identifier* according to the setting of the PGMNAME compiler option of the program that contains the CALL statement, and interprets the program-name that is exported from the target DLL according to the setting of the PGMNAME option used when the target program was compiled.

The search for the target DLL in the hierarchical file system (HFS) is case sensitive. If the target DLL is a PDS or PDSE member, the DLL member name must be eight characters or less. For the purpose of the search for the DLL as a PDS or PDSE member, the run time automatically converts the name to uppercase.

If the runtime environment cannot resolve the CALL statement in either of these cases, control is transferred to the ON EXCEPTION or ON OVERFLOW phrase of the CALL statement. If the CALL statement does not specify one of these phrases in this situation, Language Environment raises a severity-3 condition.

RELATED TASKS
"Using DLL linkage and dynamic calls together"
"Compiling programs to create DLLs" on page 466
"Linking DLLs" on page 467

RELATED REFERENCES
"DLL" on page 311
"PGMNAME" on page 329
CALL statement (*Enterprise COBOL Language Reference*)
"Search order for DLLs in the HFS"

Search order for DLLs in the HFS

When you use the hierarchical file system (HFS), the search order for resolving a DLL reference in a CALL statement depends on the setting of the Language Environment POSIX runtime option.

If the POSIX runtime option is ON, the search order is as follows:

1. The runtime environment looks for the DLL in the HFS. If the LIBPATH environment variable is set, the run time searches each directory listed. Otherwise, it searches just the current directory. The search for the DLL in the HFS is case sensitive.
2. If the runtime environment does not find the DLL in the HFS, it tries to load the DLL from the MVS load library search order of the caller. In this case, the DLL name must be eight characters or less. The run time automatically converts the DLL name to uppercase for this search.

If the POSIX runtime option is set to OFF, the search order is reversed:

1. The runtime environment tries to load the DLL from the search order for the load library of the caller.
2. If the runtime environment cannot load the DLL from this load library, it tries to load the DLL from the HFS.

RELATED TASKS
"Using CALL identifier with DLLs" on page 469

RELATED REFERENCES
POSIX (*Language Environment Programming Reference*)

Using DLL linkage and dynamic calls together

For applications (that is, Language Environment enclaves) that are structured as multiple, separately bound modules, you should use exclusively one form of linkage between modules: either dynamic call linkage or DLL linkage.

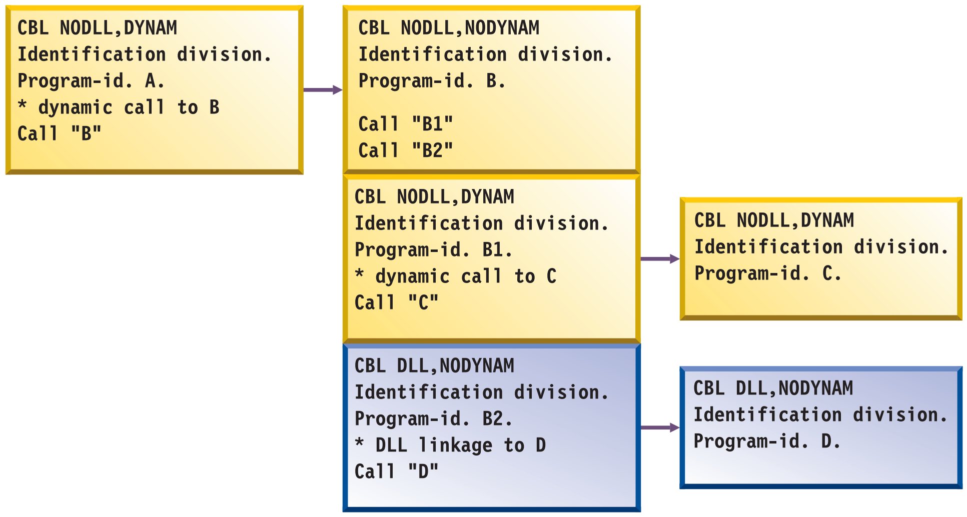
DLL linkage refers to a call in a program that is compiled with the DLL and NODYNAM options in which the call resolves to an exported name in a separate module. DLL linkage can also refer to an invocation of a method that is defined in a separate module.

However, some applications require more flexibility. If so, you can use both DLL linkage and COBOL dynamic call linkage within a Language Environment enclave if the programs are compiled as follows:

Program A	Program B	Compile both with:
Contains dynamic call	Target of dynamic call	`NODLL`
Uses DLL linkage	Contains target program or method	`DLL`

If a program contains a `CALL` statement for a separately compiled program and you compile one program with the `DLL` compiler option and the other program with `NODLL`, then the call is supported only if you bind the two programs together in the same module.

The following diagram shows several separately bound modules that mix dynamic calls and DLL linkage.



```
CBL NODLL,DYNAM
Identification division.
Program-id. A.
* dynamic call to B
Call "B"
```

```
CBL NODLL,NODYNAM
Identification division.
Program-id. B.

Call "B1"
Call "B2"
```

```
CBL NODLL,DYNAM
Identification division.
Program-id. B1.
* dynamic call to C
Call "C"
```

```
CBL DLL,NODYNAM
Identification division.
Program-id. B2.
* DLL linkage to D
Call "D"
```

```
CBL NODLL,DYNAM
Identification division.
Program-id. C.
```

```
CBL DLL,NODYNAM
Identification division.
Program-id. D.
```

All components of a DLL application must have the same AMODE. The automatic AMODE switching normally provided by COBOL dynamic calls is not available for DLL linkages.

You cannot cancel programs that are called using DLL linkage.

RELATED CONCEPTS
"Dynamic link libraries (DLLs)" on page 465

RELATED TASKS
"Compiling programs to create DLLs" on page 466
"Linking DLLs" on page 467
"Using procedure or function pointers with DLLs" on page 472
"Calling DLLs from non-DLLs" on page 472

RELATED REFERENCES
"DLL" on page 311
"EXPORTALL" on page 313

Using procedure or function pointers with DLLs

In run units that contain both DLLs and non-DLLs, use procedure- and function-pointer data items with care.

When you use the `SET` *procedure-pointer-1* `TO ENTRY` *entry-name* or `SET` *function-pointer-1* `TO ENTRY` *entry-name* statement in a program that is compiled with the `NODLL` option, you must not pass the pointer to a program that is compiled with the `DLL` option. However, when you use this statement in a program that is compiled with the `DLL` option, you can pass the pointer to a program that is in a separately bound DLL module.

If you compile with the `NODYNAM` and `DLL` options, and *entry-name* is an identifier, the identifier value must refer to the entry-point name that is exported from a DLL module. The DLL module name must match the name of the exported entry point. In this case, note also that:

- The program-name that is contained in the identifier is interpreted according to the setting of the `PGMNAME(COMPAT|LONGUPPER|LONGMIXED)` compiler option of the program that contains the `CALL` statement.
- The program-name that is exported from the target DLL is interpreted according to the setting of the `PGMNAME` option used when compiling the target program.
- The search for the target DLL in the HFS is case sensitive.
- If the target DLL is a PDS or PDSE member, the DLL member name must have eight characters or less. For the purpose of the search for the DLL as a PDS or PDSE member, the name is automatically converted to uppercase.

RELATED TASKS
"Using CALL identifier with DLLs" on page 469
"Using procedure and function pointers" on page 447
"Compiling programs to create DLLs" on page 466
"Linking DLLs" on page 467

RELATED REFERENCES
"DLL" on page 311
"EXPORTALL" on page 313

Calling DLLs from non-DLLs

It is possible to call a DLL from a COBOL program that is compiled with the `NODLL` option, but there are restrictions.

You can use the following methods to ensure that the DLL linkage is followed:

- Put the COBOL DLL programs that you want to call from the COBOL non-DLL programs in the load module that contains the main program. Use static calls from the COBOL non-DLL programs to call the COBOL DLL programs.

  The COBOL DLL programs in the load module that contains the main program can call COBOL DLL programs in other DLLs.
- Put the COBOL DLL programs in DLLs and call them from COBOL non-DLL programs with `CALL` *function-pointer*, where *function-pointer* is set to a function descriptor of the target program. You can obtain the address of the function descriptor for the program in the DLL by calling a C routine that uses `dllload` and `dllqueryfn`.

"Example: calling DLLs from non-DLLs" on page 473

RELATED TASKS
"Using procedure and function pointers" on page 447

Example: calling DLLs from non-DLLs

The following example shows how a COBOL program that is not in a DLL (COBOL1) can call a COBOL program that is in a DLL (program ooc05R in DLL OOC05R).

```
CBL NODYNAM
       IDENTIFICATION DIVISION.
       PROGRAM-ID. 'COBOL1'.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       01 DLL-INFO.
           03 DLL-LOADMOD-NAME PIC X(12).
           03 DLL-PROGRAM-NAME PIC X(160).
           03 DLL-PROGRAM-HANDLE FUNCTION-POINTER.
       77 DLL-RC               PIC S9(9) BINARY.
       77 DLL-STATUS           PIC X(1) VALUE 'N'.
           88 DLL-LOADED       VALUE 'Y'.
           88 DLL-NOT-LOADED   VALUE 'N'.

       PROCEDURE DIVISION.

           IF DLL-NOT-LOADED
           THEN
      *      Move the names in. They must be null terminated.
             MOVE Z'OOC05R' TO DLL-LOADMOD-NAME
             MOVE Z'ooc05r' TO DLL-PROGRAM-NAME

      *      Call the C routine to load the DLL and to get the
      *      function descriptor address.
             CALL 'A1CCDLGT' USING BY REFERENCE DLL-INFO
                                   BY REFERENCE DLL-RC
             IF DLL-RC = 0
             THEN
               SET DLL-LOADED TO TRUE
             ELSE
               DISPLAY 'A1CCLDGT failed with rc = '
                 DLL-RC
               MOVE 16 TO RETURN-CODE
               STOP RUN
             END-IF
           END-IF

      *    Use the function pointer on the call statement to call the
      *    program in the DLL.
      *    Call the program in the DLL.
           CALL DLL-PROGRAM-HANDLE

           GOBACK.


#include <stdio.h>
#include <dll.h>
#pragma linkage (A1CCDLGT,COBOL)

typedef struct dll_lm {
  char         dll_loadmod_name[(12)];
  char         dll_func_name[(160)];
  void         (*fptr) (void); /* function pointer */
  } dll_lm;

void A1CCDLGT (dll_lm *dll, int *rc)
{
  dllhandle *handle;
  void (*fptr1)(void);
  *rc = 0;
   /* Load the DLL                               */
  handle = dllload(dll->dll_loadmod_name);
  if (handle == NULL) {
     perror("A1CCDLGT failed on call to load DLL./n");
     *rc = 1;
     return;
  }

  /* Get the address of the function            */
  fptr1 = (void (*)(void))
          dllqueryfn(handle,dll->dll_func_name);
  if (fptr1 == NULL) {
     perror("A1CCDLGT failed on retrieving function./n");
     *rc = 2;
     return;
  }
  /* Return the function pointer                */
  dll->fptr = fptr1;
  return;
}
```

Using COBOL DLLs with C/C++ programs

COBOL support for DLLs interoperates with the DLL support in the z/OS C/C++ products, except for COBOL EXTERNAL data. In particular, COBOL applications can call functions that are exported from C/C++ DLLs, and C/C++ applications can call COBOL programs that are exported from COBOL DLLs.

COBOL data items that are declared with the EXTERNAL attribute are independent of DLL support. These data items are accessible by name from any COBOL program in the run unit that declares them, regardless of whether the programs are in DLLs.

The COBOL options DLL, RENT, and EXPORTALL work much the same way as the C/C++ DLL, RENT, and EXPORTALL options. (The DLL option applies only to C.) However, the C/C++ compiler produces DLL-enabled code by default.

You can pass a C/C++ DLL function pointer to COBOL and use it within COBOL, receiving the C/C++ function pointer as a function-pointer data item. The following example shows a COBOL call to a C function that returns a function pointer to a service, followed by a COBOL call to the service.

```
Identification Division.
Program-id. Demo.
Data Division.
Working-Storage section.
01  fp usage function-pointer.
Procedure Division.
    Call "c-function" returning fp.
    Call fp.
```

RELATED TASKS
"Compiling programs to create DLLs" on page 466
"Linking DLLs" on page 467

RELATED REFERENCES
"DLL" on page 311
"EXPORTALL" on page 313
"RENT" on page 331
EXTERNAL clause (*Enterprise COBOL Language Reference*)

Using DLLs in OO COBOL applications

You must compile each COBOL class definition using the DLL, THREAD, RENT, and DBCS compiler options, and link-edit it into a separate DLL module using the RENT binder option.

RELATED TASKS
Chapter 16, "Compiling, linking, and running OO applications," on page 287
"Compiling programs to create DLLs" on page 466
"Linking DLLs" on page 467

RELATED REFERENCES
"DLL" on page 311
"THREAD" on page 342
"RENT" on page 331
"DBCS" on page 309

Chapter 27. Preparing COBOL programs for multithreading

You can run COBOL programs in multiple threads within a process under batch, TSO, IMS, or UNIX.

There is no explicit COBOL language to use for multithreaded execution; rather, you compile with the `THREAD` compiler option.

COBOL does not directly support managing program threads. However, you can run COBOL programs that you compile with the `THREAD` compiler option in multithreaded application servers, in applications that use a C/C++ driver program to create the threads, in programs that interoperate with Java and use Java threads, and in applications that use PL/I tasking. In other words, other programs can call COBOL programs in such a way that the COBOL programs run in multiple threads within a process or as multiple program invocation instances within a thread. Your threaded application must run within a single Language Environment enclave.

Choosing `LOCAL-STORAGE` or `WORKING-STORAGE`: Because you must code your multithreaded programs as recursive, the persistence of data is that of any recursive program:

- Data items in the `LOCAL-STORAGE SECTION` are automatically allocated for each instance of a program invocation. When a program runs in multiple threads simultaneously, each invocation has a separate copy of `LOCAL-STORAGE` data.
- Data items in the `WORKING-STORAGE SECTION` are allocated once for each program and are thus available in their last-used state to all invocations of the program.

For the data that you want to isolate to an individual program invocation instance, define the data in the `LOCAL-STORAGE SECTION`. In general, this choice is appropriate for working data in threaded programs. If you declare data in `WORKING-STORAGE` and your program changes the contents of the data, you must take one of the following actions:

- Structure your application so that you do not access data in `WORKING-STORAGE` simultaneously from multiple threads.
- If you do access data simultaneously from separate threads, write appropriate serialization code.

RELATED CONCEPTS
"Multithreading" on page 478

RELATED TASKS
"Choosing THREAD to support multithreading" on page 479
"Transferring control to multithreaded programs" on page 479
"Ending multithreaded programs" on page 480
"Processing files with multithreading" on page 480
"Handling COBOL limitations with multithreading" on page 482

RELATED REFERENCES
"THREAD" on page 342
PROGRAM-ID paragraph (*Enterprise COBOL Language Reference*)

Multithreading

To use COBOL support for multithreading, you need to understand how processes, threads, run units, and program invocation instances relate to each other.

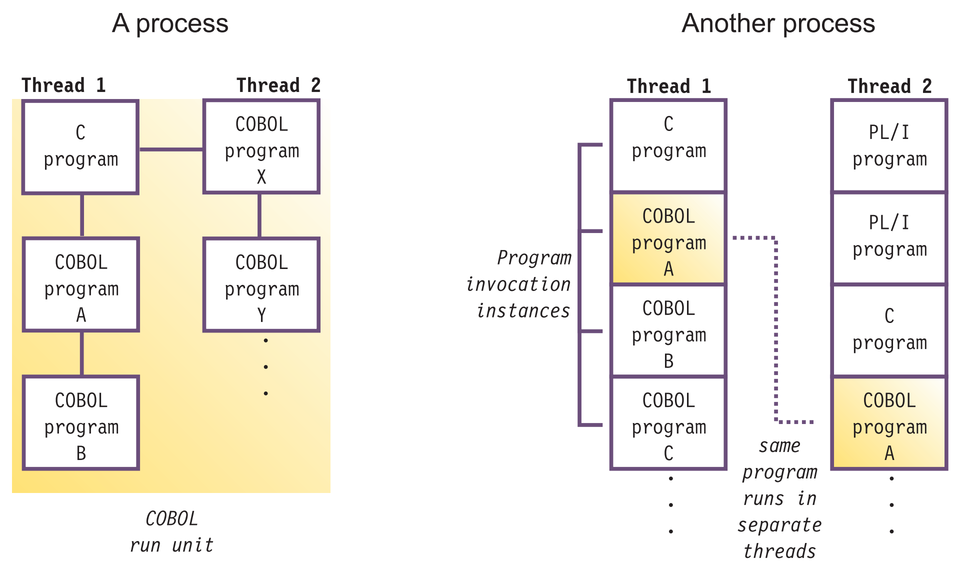
The operating system and multithreaded applications can handle execution flow within a *process*, which is the course of events when all or part of a program runs. Programs that run within a process can share resources. Processes can be manipulated. For example, they can have a high or low priority in terms of the amount of time that the system devotes to running the process.

Within a process, an application can initiate one or more *threads*, each of which is a stream of computer instructions that controls that thread. A multithreaded process begins with one stream of instructions (one thread) and can later create other instruction streams to perform tasks. These multiple threads can run concurrently. Within a thread, control is transferred between executing programs.

In a multithreaded environment, a COBOL *run unit* is the portion of the process that includes threads that have 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.

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 called program is a *program invocation instance*. Program invocation instances of a particular program can exist in multiple threads within a given process.

The following illustration shows these relationships between processes, threads, run units, and program invocation instances.



RELATED CONCEPTS
Program management model (*Language Environment Programming Guide*)
Understanding the basics: threads (*Language Environment Programming Guide*)

RELATED TASKS
"Choosing THREAD to support multithreading"
"Transferring control to multithreaded programs"
"Ending multithreaded programs" on page 480
"Processing files with multithreading" on page 480
"Handling COBOL limitations with multithreading" on page 482

RELATED REFERENCES
"THREAD" on page 342

Choosing THREAD to support multithreading

Use the THREAD compiler option for multithreading support. Use THREAD if your program will be called in more than one thread in a single process by an application. However, THREAD might adversely affect performance because of the serialization logic that is automatically generated.

In order to run COBOL programs in more than one thread, you must compile all of the COBOL programs in the run unit with the THREAD compiler option. You must also compile them with the RENT compiler option and link them with the RENT option of the binder or linkage editor.

Use the THREAD option when you compile object-oriented (OO) clients and classes.

Language restrictions: When you use the THREAD option, you cannot use certain language elements. For details, see the related reference below.

Recursion: Before you compile a program with the THREAD compiler option, you must specify the RECURSIVE phrase in the PROGRAM-ID paragraph. If you do not do so, an error will occur.

RELATED TASKS
"Sharing data in recursive or multithreaded programs" on page 18
"Compiling OO applications under UNIX" on page 287

RELATED REFERENCES
"THREAD" on page 342

Transferring control to multithreaded programs

When you write COBOL programs for a multithreaded environment, choose appropriate program linkage statements.

As in single-threaded environments, a called program is in its initial state when it is first called within a run unit and when it is first called after a CANCEL to the called program. Ensure that the program that you name on a CANCEL statement is not active on any thread. If you try to cancel an active program, a severity-3 Language Environment condition occurs.

If your threaded application requires preinitialization, use the Language Environment services (CEEPIPI interface). You cannot use the COBOL-specific interfaces for preinitialization (runtime option RTEREUS and functions IGZERRE and ILBOSTP0) to establish a reusable environment from any program that has been compiled with the THREAD option.

RELATED CONCEPTS
Language Environment termination: enclave termination (*Language Environment Programming Guide*)

RELATED TASKS
"Ending multithreaded programs"
"Ending and reentering main programs or subprograms" on page 434

Ending multithreaded programs

You can end a multithreaded program by using `GOBACK`, `EXIT PROGRAM`, or `STOP RUN`.

Use `GOBACK` to return to the caller of the program. When you use `GOBACK` from the first program in a thread, the thread is terminated. If that thread is the initial thread in an enclave, the entire enclave is terminated.

Use `EXIT PROGRAM` as you would `GOBACK`, except from a main program where it has no effect.

Use `STOP RUN` to terminate the entire Language Environment enclave and to return control to the caller of the main program (which might be the operating system). All threads that are executing within the enclave are terminated.

RELATED CONCEPTS
Language Environment termination: enclave termination (*Language Environment Programming Guide*)

RELATED TASKS
"Ending and reentering main programs or subprograms" on page 434

Processing files with multithreading

In threaded applications, you can code COBOL statements for input and output in QSAM, VSAM, and line-sequential files.

Each file definition (`FD`) has an implicit serialization lock. This lock is used with automatic serialization logic during the input or output operation that is associated with the execution of the following statements:

- `OPEN`
- `CLOSE`
- `READ`
- `WRITE`
- `REWRITE`
- `START`
- `DELETE`

Automatic serialization also occurs for the implicit `MOVE` that is associated with the following statements:

```
WRITE record-name FROM identifier
READ  file-name   INTO identifier
```

Automatic serialization is not applied to any statements specified within the following conditional phrases:

- `AT END`

- `NOT AT END`
- `INVALID KEY`
- `NOT INVALID KEY`
- `AT END-OF-PAGE`
- `NOT AT END-OF-PAGE`

RELATED CONCEPTS
"File-definition (FD) storage"

RELATED TASKS
"Closing QSAM files" on page 164
"Closing VSAM files" on page 195
"Coding ERROR declaratives" on page 238
"Serializing file access with multithreading"

File-definition (FD) storage

On all program invocations, the storage that is associated with a file definition (such as `FD` records and the record area that is associated with the `SAME RECORD AREA` clause) is allocated and available in its last-used state.

All threads of execution share this storage. You can depend on automatic serialization for this storage during the execution of the `OPEN`, `CLOSE`, `READ`, `WRITE`, `REWRITE`, `START`, and `DELETE` statements, but not between uses of these statements.

RELATED TASKS
"Serializing file access with multithreading"

Serializing file access with multithreading

To take full advantage of automatic serialization and to avoid explicitly writing your own serialization logic, use one of the recommended file organizations and usage patterns when you access files in threaded programs.

Use one of the following file organizations:
- Sequential organization
- Line-sequential organization
- Relative organization with sequential access
- Indexed organization with sequential access

Use the following pattern for input:

```
  OPEN INPUT fn
  . . .
  READ fn INTO local-storage-item
  . . .
* Process the record from the local-storage item
  . . .
  CLOSE fn
```

Use the following pattern for output:

```
  OPEN OUTPUT fn
  . . .
* Construct output record in local-storage item
  . . .
  WRITE rec FROM local-storage-item
  . . .
  CLOSE fn
```

With other usage patterns, you must take one of the following actions:

- Verify the safety of your application logic. Ensure that two instances of the program are never simultaneously active on different threads.
- Code explicit serialization logic by using calls to POSIX services.

To avoid serialization problems when you access a file from multiple threads, define the data items that are associated with the file (such as file-status data items and key arguments) in the `LOCAL-STORAGE SECTION`.

"Example: usage patterns of file input and output with multithreading"

RELATED TASKS
"Calling UNIX/POSIX APIs" on page 426

Example: usage patterns of file input and output with multithreading

The following examples show the need for explicit serialization logic when you deviate from the recommended usage pattern for file input and output in your multithreaded applications. These examples also explain the unexpected behavior that might result if you fail to handle serialization properly.

In each example, two instances of a program that contains the sample operations are running within one run unit on two different threads.

```
READ F1
. . .
REWRITE R1
```

In the example above, the second thread might execute the `READ` statement after the `READ` statement is executed on the first thread but before the `REWRITE` statement is executed on the first thread. The `REWRITE` statement might not update the record that you intended. To ensure the results that you want, write explicit serialization logic.

```
 READ F1
 . . .
* Process the data in the FD record description entry for F1
 . . .
```

In the example above, the second thread might execute the `READ` statement while the first thread is still processing a record in the `FD` record description entry. The second `READ` statement would overlay the record that the first thread is processing. To avoid this problem, use the recommended technique `READ F1 INTO` *`LOCAL-STORAGE-item`*.

Other cases: You must give similar consideration to other usage patterns that involve a sequence of related input and output operations, such as `START` followed by `READ NEXT`, or `READ` followed by `DELETE`. Take appropriate steps to ensure the correct processing of file input and output.

Handling COBOL limitations with multithreading

Some COBOL applications depend on subsystems or other applications. In a multithreaded environment, these dependencies and others result in some limitations on COBOL programs.

In general, you must synchronize access to resources that are visible to the application within a run unit. Exceptions to this requirement are `DISPLAY` and `ACCEPT`, which you can use from multiple threads, and supported COBOL file I/O statements that have the recommended usage pattern; all synchronization is provided for these by the runtime environment.

CICS: You cannot run multithreaded applications in the CICS environment. In the CICS environment you can run a COBOL program that has been compiled with the `THREAD` option and that is part of an application that has no multiple threads or PL/I tasks.

Recursive: Because you must code the programs in a multithreaded application as recursive, you must adhere to all the restrictions and programming considerations that apply to recursive programs, such as not coding nested programs.

Reentrancy: You must compile your multithreading programs with the `RENT` compiler option and link them with the `RENT` option of the binder or linkage editor.

POSIX and PL/I: If you use POSIX threads in your multithreaded application, you must specify the Language Environment runtime option `POSIX(ON)`. If the application uses PL/I tasking, you must specify `POSIX(OFF)`. You cannot mix POSIX threads and PL/I tasks in the same application.

PL/I tasking: To include COBOL programs in applications that contain multiple PL/I tasks, follow these guidelines:
- Compile all COBOL programs that you run in multiple PL/I tasks with the `THREAD` option. If you compile any COBOL program with the `NOTHREAD` option, all of the COBOL programs must run in one PL/I task.
- You can call COBOL programs compiled with the `THREAD` option from one or more PL/I tasks. However, calls from PL/I programs to COBOL programs cannot include the `TASK` or `EVENT` option. The PL/I tasking call must first call a PL/I program or function that in turn calls the COBOL program. This indirection is required because you cannot specify the COBOL program directly as the target of a PL/I `CALL` statement that includes the `TASK` or `EVENT` option.
- Be aware that issuing a `STOP RUN` statement from a COBOL program or a `STOP` statement from a PL/I program terminates the entire Language Environment enclave, including all the tasks of execution.
- Do not code explicit POSIX threading (calls to pthread_create()) in any run unit that includes PL/I tasking.

C and Language Environment-enabled assembler: You can combine your multithreaded COBOL programs with C programs and Language Environment-enabled assembler programs in the same run unit when those programs are also appropriately coded for multithreaded execution.

AMODE: You must run your multithreaded applications with `AMODE 31`. You can run a COBOL program that has been compiled with the `THREAD` option with `AMODE 24` as part of an application that does not have multiple threads or PL/I tasks.

Asynchronous signals: In a threaded application your COBOL program might be interrupted by an asynchronous signal or interrupt. If your program contains logic that cannot tolerate such an interrupt, you must disable the interrupts for the duration of that logic. Call a C/C++ function to set the signal mask appropriately.

Older COBOL programs: To run your COBOL programs on multiple threads of a multithreaded application, you must compile them with Enterprise COBOL and use the THREAD option. If you run programs that have been compiled with older compilers, you must follow these restrictions:

- Run applications that contain OS/VS COBOL programs only on the initial thread (IPT).
- Run applications that contain programs compiled by other older compilers only on one thread, although it can be a thread other than the initial thread.

IGZBRDGE, IGZETUN, and IGZEOPT: Do not use IGZBRDGE, the macro for converting static calls to dynamic calls, with programs that have been compiled with the THREAD option; this macro is not supported. Do not use the modules IGZETUN (for storage tuning) or IGZEOPT (for runtime options) for applications in which the main program has been compiled with the THREAD option; these CSECTs are ignored.

UPSI switches: All programs and all threads in an application share a single copy of UPSI switches. If you modify switches in a threaded application, you must code appropriate serialization logic.

RELATED TASKS
"Making recursive calls" on page 447
"Serializing file access with multithreading" on page 481
Using threads in z/OS UNIX System Services applications (*C/C++ Programming Guide*)
Language Environment Writing ILC Applications

Part 5. Using XML and COBOL together

Chapter 28. Processing XML input

You can process XML input in your COBOL program by using the XML PARSE statement. The XML PARSE statement is the COBOL language interface to the high-speed XML parser, which is part of the COBOL run time.

Processing XML input involves control being passed to and received from the XML parser. You start this exchange of control with the XML PARSE statement, which specifies a processing procedure that receives control from the XML parser to handle the parser events. You use special registers in your processing procedure to exchange information with the parser.

Use these COBOL facilities to process XML input:

- XML PARSE statement to begin XML parsing and to identify the document and your processing procedure
- Processing procedure to control the parsing: receive and process the XML events and associated document fragments, and optionally handle exceptions
- Special registers to receive and pass information:
  - XML-CODE to determine the status of XML parsing
  - XML-EVENT to receive the name of each XML event
  - XML-TEXT to receive XML document fragments from an alphanumeric document
  - XML-NTEXT to receive XML document fragments from a national document

+ **Link-edit considerations:** COBOL programs that contain the XML PARSE statement
+ must be link-edited with AMODE 31.

RELATED CONCEPTS
"XML parser in COBOL"

RELATED TASKS
"Accessing XML documents" on page 489
"Parsing XML documents" on page 489
"Understanding the encoding of XML documents" on page 502
"Handling exceptions that the XML parser finds" on page 505
"Terminating XML parsing" on page 509

RELATED REFERENCES
Appendix D, "XML reference material," on page 669
Extensible Markup Language (XML)

XML parser in COBOL

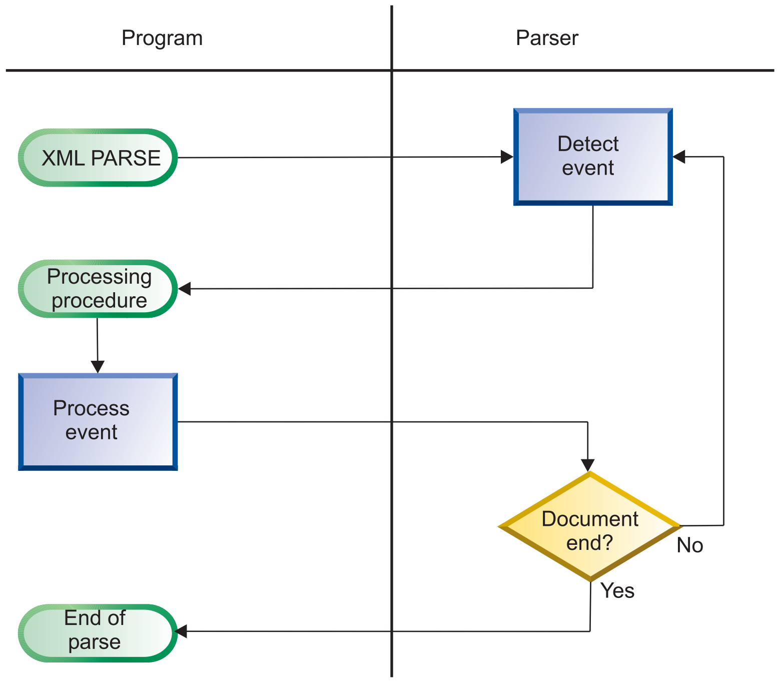
Enterprise COBOL provides an event-based interface that enables you to parse XML documents and transform them to COBOL data structures.

The XML parser finds fragments (associated with XML events) within the document, and your processing procedure acts on these fragments. You code your procedure to handle each XML event. Throughout this operation, control passes back and forth between the parser and your procedure.

You start this exchange with the parser by using the XML PARSE statement, in which you designate your processing procedure. Execution of this XML PARSE statement begins the parsing and establishes your processing procedure with the parser. Each execution of your procedure causes the XML parser to continue analyzing the XML document and report the next event, passing back to your procedure the fragment that it finds, such as the start of a new element. You can also specify in the XML PARSE statement two imperative statements to which you want control to be passed at the end of the parsing: one when a normal end occurs and one when an exception condition exists.

The following figure shows a high-level overview of the basic exchange of control between the parser and your program.

XML parsing flow overview



Normally, parsing continues until the entire XML document has been parsed.

When the XML parser parses XML documents, it checks them for most aspects of well formedness. A document is *well formed* if it adheres to the XML syntax in the *XML specification* and follows some additional rules such as proper use of end tags and uniqueness of attribute names.

RELATED TASKS
"Accessing XML documents" on page 489
"Parsing XML documents" on page 489
"Writing procedures to process XML" on page 495
"Understanding the encoding of XML documents" on page 502
"Handling exceptions that the XML parser finds" on page 505
"Terminating XML parsing" on page 509

RELATED REFERENCES
"XML conformance" on page 675
XML specification

Accessing XML documents

Before you can parse an XML document with an XML PARSE statement, you must make the document available to your program. Common methods of acquiring the document are by retrieval from a WebSphere MQ message, a CICS transient queue or communication area, or an IMS message processing queue.

If the XML document that you want to parse is held in a file, use ordinary COBOL facilities to place the document into a data item in your program:

- A FILE-CONTROL entry to define the file to your program
- An OPEN statement to open the file
- READ statements to read all the records from the file into a data item (either an elementary item of category alphanumeric or national, or an alphanumeric or national group) that is defined in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION
- Optionally the STRING statement to string all of the separate records together into one continuous stream, to remove extraneous blanks, and to handle variable-length records

RELATED TASKS
"Coding COBOL programs to run under CICS" on page 393
Chapter 22, "Developing COBOL programs for IMS," on page 417

Parsing XML documents

To parse XML documents, use the XML PARSE statement, specifying the XML document that is to be parsed and the procedure for handling XML events that occur during parsing. You can also optionally specify what action should be taken after parsing finishes.

```
XML PARSE XMLDOCUMENT
    PROCESSING PROCEDURE XMLEVENT-HANDLER
  ON EXCEPTION
     DISPLAY 'XML document error ' XML-CODE
     STOP RUN
  NOT ON EXCEPTION
     DISPLAY 'XML document was successfully parsed.'
END-XML
```

In the XML PARSE statement you first identify the data item (XMLDOCUMENT in the example above) that contains the XML document character stream. In the DATA DIVISION, you can declare the identifier as national (either a national group item or an elementary item of category national) or as alphanumeric (either an alphanumeric group item or an elementary item of category alphanumeric). If it is national, its content must be encoded in Unicode UTF-16BE, CCSID 1200. If it is alphanumeric, its content must be encoded with one of the supported single-byte character sets. See the related reference below about coded characters sets for more information.

If the identifier is alphanumeric and the XML document does not contain an encoding declaration, the document is parsed with the code page that is specified by the CODEPAGE compiler option in effect.

XML declaration: If the document that you are parsing contains an XML declaration, the declaration must begin in the first byte of the document. If the string <?xml starts after the first byte of the document, the parser generates an exception code. The attribute names that are coded in the XML declaration must all be in lowercase characters.

Next you specify the name of the procedure (XMLEVENT-HANDLER in the example) that is to handle the XML events from the document.

In addition, you can specify either or both of the following phrases to receive control after parsing finishes:

- ON EXCEPTION, to receive control when an unhandled exception occurs during parsing
- NOT ON EXCEPTION, to receive control otherwise

You can end the XML PARSE statement with the explicit scope terminator END-XML. Use END-XML to nest an XML PARSE statement that uses the ON EXCEPTION or NOT ON EXCEPTION phrase in a conditional statement (for example, in another XML PARSE statement or in an XML GENERATE statement).

The parser passes control to the processing procedure for each XML event. Control returns to the parser when the end of the processing procedure is reached. This exchange of control between the XML parser and the processing procedure continues until one of the following events occurs:

- The entire XML document has been parsed, as indicated by the END-OF-DOCUMENT event.
- The parser detects an error in the document and signals an EXCEPTION event, and the processing procedure does not reset the special register XML-CODE to zero before returning to the parser.
- You terminate the parsing process deliberately by setting the special register XML-CODE to -1 before returning to the parser.

Special registers: Use the XML-EVENT special register to determine which event the parser passes to your processing procedure. XML-EVENT contains an event name such as 'START-OF-ELEMENT'. The parser passes the content for the event in special register XML-TEXT or XML-NTEXT, depending on the type of the XML identifier specified in the XML PARSE statement.

RELATED TASKS
"Understanding the encoding of XML documents" on page 502
"Writing procedures to process XML" on page 495

RELATED REFERENCES
"Coded character sets for XML documents" on page 503
"The content of XML-EVENT"
XML PARSE statement (*Enterprise COBOL Language Reference*)

The content of XML-EVENT

When an event occurs during XML parsing, the XML parser writes the appropriate event name shown below to the XML-EVENT special register. (The event is passed to your processing procedure, and the text that corresponds to the event is written to either the XML-TEXT or XML-NTEXT special register.)

ATTRIBUTE-CHARACTER
Occurs in attribute values for the predefined entity references '&amp;', '&apos;', '&gt;', '&lt;', and '&quot;'. See *XML specification* for details about predefined entities.

ATTRIBUTE-CHARACTERS
: Occurs for each fragment of an attribute value. XML text contains the fragment. An attribute value normally consists only of a single string, even if it is split across lines. The attribute value might consist of multiple events, however.

ATTRIBUTE-NAME
: Occurs for each attribute in an element start tag or empty element tag, after a valid name is recognized. XML text contains the attribute name.

ATTRIBUTE-NATIONAL-CHARACTER
: Occurs in attribute values for numeric character references (Unicode code points or “scalar values”) of the form ’&#*dd*..;’ or ’&#*hh*..;’, where *d* and *h* represent decimal and hexadecimal digits, respectively. If the scalar value of the national character is greater than 65,535 (`NX’FFFF’`), `XML-NTEXT` contains two encoding units (a *surrogate pair*) and has a length of 4 bytes. This pair of encoding units represents a single character. Do not create characters that are not valid by splitting this pair. (See the related reference below about code-page-sensitive characters for information about coding the number sign (#).)

COMMENT
: Occurs for any comments in the XML document. XML text contains the data between the opening and closing comment delimiters, ’`<!--`’ and ’`-->`’, respectively. (See the related reference below about code-page-sensitive characters for information about coding the exclamation point (!).)

CONTENT-CHARACTER
: Occurs in element content for the predefined entity references ’&amp;’, ’&apos;’, ’&gt;’, ’&lt;’, and ’&quot;’. See *XML specification* for details about predefined entities.

CONTENT-CHARACTERS
: This event represents the principal part of an XML document: the character data between element start and end tags. XML text contains this data, which usually consists only of a single string even if it is split across lines. If the content of an element includes any references or other elements, the complete content might consist of several events. The parser also uses the `CONTENT-CHARACTERS` event to pass the text of CDATA sections to your program.

CONTENT-NATIONAL-CHARACTER
: Occurs in element content for numeric character references (Unicode code points or “scalar values”) of the form ’&#*dd*..;’ or ’&#*hh*..;’, where *d* and *h* represent decimal and hexadecimal digits, respectively. If the scalar value of the national character is greater than 65,535 (`NX’FFFF’`), `XML-NTEXT` contains two encoding units (a surrogate pair) and has a length of 4 bytes. This pair of encoding units represents a single character. Do not create characters that are not valid by splitting this pair. (See the related reference below about code-page-sensitive characters for information about coding the number sign (#).)

DOCUMENT-TYPE-DECLARATION
: Occurs when the parser finds a document type declaration. Document type declarations begin with the character sequence ’<!DOCTYPE’ and end with a right angle bracket (’>’) character; some fairly complicated grammar rules describe the content in between. See *XML specification* for details. (Also see the related reference below about code-page-sensitive characters for

information about coding the exclamation point (!).) For this event, XML text contains the entire declaration, including the opening and closing character sequences. This is the only event for which XML text includes the delimiters.

ENCODING-DECLARATION
: Occurs within the XML declaration for the optional encoding declaration. XML text contains the encoding value.

END-OF-CDATA-SECTION
: Occurs when the parser recognizes the end of a CDATA section. (See the related reference below about code-page-sensitive characters for information about coding the right square bracket (]).)

END-OF-DOCUMENT
: Occurs when document parsing has completed.

END-OF-ELEMENT
: Occurs once for each element end tag or empty element tag when the parser recognizes the closing angle bracket of the tag. XML text contains the element name.

EXCEPTION
: Occurs when an error in processing the XML document is detected. For encoding conflict exceptions, which are signaled before parsing begins, `XML-TEXT` (for XML documents in an alphanumeric data item) or `XML-NTEXT` (for XML documents in a national data item) either is zero length or contains only the encoding declaration value from the document.

PROCESSING-INSTRUCTION-DATA
: Occurs for the data that follows the PI target, up to but not including the PI closing character sequence, '?>'. XML text contains the PI data, which includes trailing, but not leading, white-space characters.

PROCESSING-INSTRUCTION-TARGET
: Occurs when the parser recognizes the name that follows the opening character sequence, '<?', of a processing instruction (PI). PIs allow XML documents to contain special instructions for applications.

STANDALONE-DECLARATION
: Occurs within the XML declaration for the optional `standalone` declaration. XML text contains the `standalone` value.

START-OF-CDATA-SECTION
: Occurs at the start of a CDATA section. CDATA sections begin with the string '<![CDATA[' and end with the string ']]>'. Such sections are used to "escape" blocks of text that contain characters that would otherwise be recognized as XML markup. XML text always contains the opening character sequence '<![CDATA['. The parser passes the content of a CDATA section between these delimiters as a single `CONTENT-CHARACTERS` event. (See the related reference below about code-page-sensitive characters for information about coding the exclamation point (!) and left square bracket ([).)

START-OF-DOCUMENT
: Occurs once, at the beginning of the parsing of the document. XML text is the entire document, including any line-control characters such as LF (Line Feed) or NL (New Line).

START-OF-ELEMENT
Occurs once for each element start tag or empty element tag. XML text is set to the element name.

UNKNOWN-REFERENCE-IN-ATTRIBUTE
Occurs within attribute values for entity references other than the five predefined entity references, as shown for ATTRIBUTE-CHARACTER above.

UNKNOWN-REFERENCE-IN-CONTENT
Occurs within element content for entity references other than the predefined entity references, as shown for CONTENT-CHARACTER above.

VERSION-INFORMATION
Occurs within the optional XML declaration for the version information. XML text contains the version value. An *XML declaration* is XML text that specifies the version of XML that is used and the encoding of the document.

"Example: processing XML events"

RELATED CONCEPTS
"The content of XML-TEXT and XML-NTEXT" on page 501

RELATED TASKS
"Parsing XML documents" on page 489
"Writing procedures to process XML" on page 495

RELATED REFERENCES
"Code-page-sensitive characters in XML markup" on page 503
XML-EVENT (*Enterprise COBOL Language Reference*)
4.6 Predefined entities (*XML specification*)

Example: processing XML events

Although short, the following sample XML document contains many XML events. These events are shown below the document in the order in which they would occur during parsing, and the exact text for each XML event is delimited by angle brackets (<<>>).

In general, this text can be the content of either XML-TEXT or XML-NTEXT. The sample XML document below does not contain any text that requires XML-NTEXT, however, and thus uses only XML-TEXT.

This example begins with an XML declaration. If an XML declaration occurs in a document that you are parsing, the declaration must begin in the first byte of the document; otherwise, the parser generates an exception code. The attribute names that are coded in the XML declaration must all be in lowercase characters.

```
<?xml version="1.0" encoding="ibm-1140" standalone="yes" ?>
<!--This document is just an example-->
<sandwich>
  <bread type="baker&apos;s best" />
  <?spread please use real mayonnaise ?>
  <meat>Ham &amp; turkey</meat>
  <filling>Cheese, lettuce, tomato, etc.</filling>
  <![CDATA[We should add a <relish> element in future!]]>
</sandwich>junk
```

START-OF-DOCUMENT
The text for this sample is 336 characters in length.

VERSION-INFORMATION
: `<<1.0>>`

ENCODING-DECLARATION
: `<<ibm-1140>>`

STANDALONE-DECLARATION
: `<<yes>>`

DOCUMENT-TYPE-DECLARATION
: The sample does not have a document type declaration.

COMMENT
: `<<This document is just an example>>`

START-OF-ELEMENT
: In the order that they occur as `START-OF-ELEMENT` events:

1. `<<sandwich>>`
2. `<<bread>>`
3. `<<meat>>`
4. `<<filling>>`

ATTRIBUTE-NAME
: `<<type>>`

ATTRIBUTE-CHARACTERS
: In the order in which they occur as `ATTRIBUTE-CHARACTERS` events:

1. `<<baker>>`
2. `<<s best>>`

Notice that the value of the `type` attribute in the sample consists of three fragments: the string 'baker', the single character ''', and the string 's best'. The single-character fragment ''' is passed separately as an `ATTRIBUTE-CHARACTER` event.

ATTRIBUTE-CHARACTER
: `<<'>>`

ATTRIBUTE-NATIONAL-CHARACTER
: The sample does not contain a numeric character reference.

END-OF-ELEMENT
: In the order that they occur as `END-OF-ELEMENT` events:

1. `<<bread>>`
2. `<<meat>>`
3. `<<filling>>`
4. `<<sandwich>>`

PROCESSING-INSTRUCTION-TARGET
: `<<spread>>`

PROCESSING-INSTRUCTION-DATA
: `<<please use real mayonnaise >>`

CONTENT-CHARACTERS
: In the order that they occur as `CONTENT-CHARACTERS` events:

1. `<<Ham >>`
2. `<< turkey>>`
3. `<<Cheese, lettuce, tomato, etc.>>`

4. `<<We should add a <relish> element in future!>>`

Notice that the content of the `meat` element in the sample consists of the string 'Ham ', the character '&', and the string ' turkey'. The single-character fragment '&' is passed separately as a `CONTENT-CHARACTER` event. Also notice the trailing and leading spaces, respectively, in these two string fragments.

CONTENT-CHARACTER
: <<&>>

CONTENT-NATIONAL-CHARACTER
: The sample does not contain a numeric character reference.

START-OF-CDATA-SECTION
: <<`<![CDATA[`>>

END-OF-CDATA-SECTION
: <<`]]>`>>

UNKNOWN-REFERENCE-IN-ATTRIBUTE
: The sample does not have any unknown entity references.

UNKNOWN-REFERENCE-IN-CONTENT
: The sample does not have any unknown entity references.

END-OF-DOCUMENT
: XML text is empty for the `END-OF-DOCUMENT` event.

EXCEPTION
: The part of the document that was parsed up to and including the point where the exception (the superfluous 'junk' after the `</sandwich>` tag) was detected.

RELATED CONCEPTS
"The content of XML-TEXT and XML-NTEXT" on page 501

RELATED TASKS
"Parsing XML documents" on page 489
"Writing procedures to process XML"

RELATED REFERENCES
"The content of XML-EVENT" on page 490
"Code-page-sensitive characters in XML markup" on page 503
XML-EVENT (*Enterprise COBOL Language Reference*)
4.6 Predefined entities (*XML specification*)
2.8 Prolog and document type declaration (*XML specification*)

Writing procedures to process XML

In your processing procedure, code statements to handle XML events.

For each event that the parser encounters, it passes information to your processing procedure in several special registers, as shown in the following table. Use these registers to populate the data structures and to control the processing.

When used in nested programs, these special registers are implicitly defined as `GLOBAL` in the outermost program.

Table 68. **Special registers used by the XML parser**

Special register	**Content**	**Implicit definition and usage**
`XML-EVENT`[1]	The name of the XML event	`PICTURE X(30) USAGE DISPLAY VALUE SPACE`
`XML-CODE`[2]	An exception code or zero for each XML event	`PICTURE S9(9) USAGE BINARY VALUE ZERO`
`XML-TEXT`[1]	Text (corresponding to the event that the parser encountered) from the XML document if you specify an alphanumeric item for the `XML PARSE` identifier	Variable-length elementary category alphanumeric item; size limit of 134,217,727 bytes
`XML-NTEXT`[1]	Text (corresponding to the event that the parser encountered) from the XML document if you specify a national item for the `XML PARSE` identifier	Variable-length elementary category national item; size limit of 134,217,727 bytes

1. You cannot use this special register as a receiving data item.
2. The `XML GENERATE` statement also uses `XML-CODE`. Therefore, if you code an `XML GENERATE` statement in the processing procedure, save the value of `XML-CODE` before the `XML GENERATE` statement and restore the saved value after the `XML GENERATE` statement.

Restriction: A processing procedure must not directly execute an `XML PARSE` statement. However, if a processing procedure passes control to a method or outermost program by using an `INVOKE` or `CALL` statement, the target method or program can execute the same or a different `XML PARSE` statement. You can also execute the same XML statement or different XML statements simultaneously from a program that is running on multiple threads.

The compiler inserts a return mechanism after the last statement in each processing procedure. You can code a `STOP RUN` statement in a processing procedure to end the run unit. However, an `EXIT PROGRAM` statement (when a `CALL` statement is active) or a `GOBACK` statement does not return control to the parser. Using either of these statements in a processing procedure results in a severe error.

"Example: parsing XML" on page 497

RELATED CONCEPTS
"The content of XML-CODE" on page 499
"The content of XML-TEXT and XML-NTEXT" on page 501

RELATED TASKS
"Transforming XML text to COBOL data items" on page 501
"Converting to or from national (Unicode) representation" on page 133

RELATED REFERENCES
"XML PARSE exceptions that allow continuation" on page 669
"XML PARSE exceptions that do not allow continuation" on page 672
XML-CODE (*Enterprise COBOL Language Reference*)
XML-EVENT (*Enterprise COBOL Language Reference*)
XML-NTEXT (*Enterprise COBOL Language Reference*)
XML-TEXT (*Enterprise COBOL Language Reference*)

Example: parsing XML

This example shows the basic organization of an XML PARSE statement and of a processing procedure.

The XML document is shown in the source so that you can follow the flow of the parsing. The output of the program is also shown below. Compare the document to the output of the program to follow the interaction of the parser and the processing procedure, and to match events to document fragments.

```
 Process flag(i,i)
 Identification division.
   Program-id. xmlsampl.

 Data division.
  Working-storage section.
******************************************************************
* XML document, encoded as initial values of data items.        *
******************************************************************
   1 xml-document.
    2 pic x(39) value '<?xml version="1.0" encoding="ibm-1140"'.
    2 pic x(19) value ' standalone="yes"?>'.
    2 pic x(39) value '<!--This document is just an example-->'.
    2 pic x(10) value '<sandwich>'.
    2 pic x(35) value '  <bread type="baker&apos;s best"/>'.
    2 pic x(41) value '  <?spread please use real mayonnaise  ?>'.
    2 pic x(31) value '  <meat>Ham &amp; turkey</meat>'.
    2 pic x(40) value '  <filling>Cheese, lettuce, tomato, etc.'.
    2 pic x(10) value '</filling>'.
    2 pic x(35) value '  <![CDATA[We should add a <relish>'.
    2 pic x(22) value ' element in future!]]>'.
    2 pic x(31) value '  <listprice>$4.99 </listprice>'.
    2 pic x(27) value '  <discount>0.10</discount>'.
    2 pic x(11) value '</sandwich>'.
   1 xml-document-length computational pic 999.

******************************************************************
* Sample data definitions for processing numeric XML content.   *
******************************************************************
   1 current-element pic x(30).
   1 xfr-ed pic x(9) justified.
   1 xfr-ed-1 redefines xfr-ed pic 999999.99.
   1 list-price computational pic 9v99 value 0.
   1 discount computational pic 9v99 value 0.
   1 display-price pic $$9.99.

 Procedure division.
  mainline section.

     XML PARSE xml-document PROCESSING PROCEDURE xml-handler
       ON EXCEPTION
         display 'XML document error ' XML-CODE
       NOT ON EXCEPTION
         display 'XML document successfully parsed'
     END-XML

******************************************************************
*    Process the transformed content and calculate promo price.  *
******************************************************************
     display ' '
     display '-----+++++++***** Using information from XML '
         '*****+++++++-----'
     display ' '
     move list-price to display-price
     display '  Sandwich list price: ' display-price
     compute display-price = list-price * (1 - discount)
     display '  Promotional price:   ' display-price
     display '  Get one today!'

     goback.

  xml-handler section.
     evaluate XML-EVENT
* ==> Order XML events most frequent first
       when 'START-OF-ELEMENT'
         display 'Start element tag: <' XML-TEXT '>'
         move XML-TEXT to current-element
       when 'CONTENT-CHARACTERS'
         display 'Content characters: <' XML-TEXT '>'
* ==> Transform XML content to operational COBOL data item...
         evaluate current-element
           when 'listprice'
* ==> Using function NUMVAL-C...
             compute list-price = function numval-c(XML-TEXT)
           when 'discount'
* ==> Using de-editing of a numeric edited item...
             move XML-TEXT to xfr-ed
             move xfr-ed-1 to discount
         end-evaluate
       when 'END-OF-ELEMENT'
         display 'End element tag: <' XML-TEXT '>'
         move spaces to current-element
       when 'START-OF-DOCUMENT'
         compute xml-document-length = function length(XML-TEXT)
         display 'Start of document: length=' xml-document-length
             ' characters.'
       when 'END-OF-DOCUMENT'
         display 'End of document.'
       when 'VERSION-INFORMATION'
         display 'Version: <' XML-TEXT '>'
       when 'ENCODING-DECLARATION'
         display 'Encoding: <' XML-TEXT '>'
       when 'STANDALONE-DECLARATION'
         display 'Standalone: <' XML-TEXT '>'
       when 'ATTRIBUTE-NAME'
         display 'Attribute name: <' XML-TEXT '>'
       when 'ATTRIBUTE-CHARACTERS'
         display 'Attribute value characters: <' XML-TEXT '>'
       when 'ATTRIBUTE-CHARACTER'
         display 'Attribute value character: <' XML-TEXT '>'
       when 'START-OF-CDATA-SECTION'
         display 'Start of CData: <' XML-TEXT '>'
       when 'END-OF-CDATA-SECTION'
         display 'End of CData: <' XML-TEXT '>'
       when 'CONTENT-CHARACTER'
         display 'Content character: <' XML-TEXT '>'
       when 'PROCESSING-INSTRUCTION-TARGET'
         display 'PI target: <' XML-TEXT '>'
       when 'PROCESSING-INSTRUCTION-DATA'
         display 'PI data: <' XML-TEXT '>'
       when 'COMMENT'
         display 'Comment: <' XML-TEXT '>'
       when 'EXCEPTION'
         compute xml-document-length = function length (XML-TEXT)
         display 'Exception ' XML-CODE ' at offset '
             xml-document-length '.'
       when other
         display 'Unexpected XML event: ' XML-EVENT '.'
     end-evaluate
     .
 End program xmlsampl.
```

Output from parse example: From the following output you can see which parsing events came from which fragments of the document:

```
Start of document: length=390 characters.
Version: <1.0>
Encoding: <ibm-1140>
Standalone: <yes>
Comment: <This document is just an example>
Start element tag: <sandwich>
Content characters: <  >
Start element tag: <bread>
Attribute name: <type>
Attribute value characters: <baker>
Attribute value character: <'>
Attribute value characters: <s best>
End element tag: <bread>
Content characters: <  >
PI target: <spread>
PI data: <please use real mayonnaise  >
Content characters: <  >
Start element tag: <meat>
Content characters: <Ham >
Content character: <&>
Content characters: < turkey>
End element tag: <meat>
Content characters: <  >
Start element tag: <filling>
Content characters: <Cheese, lettuce, tomato, etc.>
End element tag: <filling>
Content characters: <  >
Start of CData: <<![CDATA[>
Content characters: <We should add a <relish> element in future!>
End of CData: <]]>>
Content characters: <  >
Start element tag: <listprice>
Content characters: <$4.99 >
End element tag: <listprice>
Content characters: <  >
Start element tag: <discount>
Content characters: <0.10>
End element tag: <discount>
End element tag: <sandwich>
End of document.
XML document successfully parsed

-----+++++++++** Using information from XML **********+-----

  Sandwich list price:  $4.99
  Promotional price:    $4.49
  Get one today!
```

The content of XML-CODE

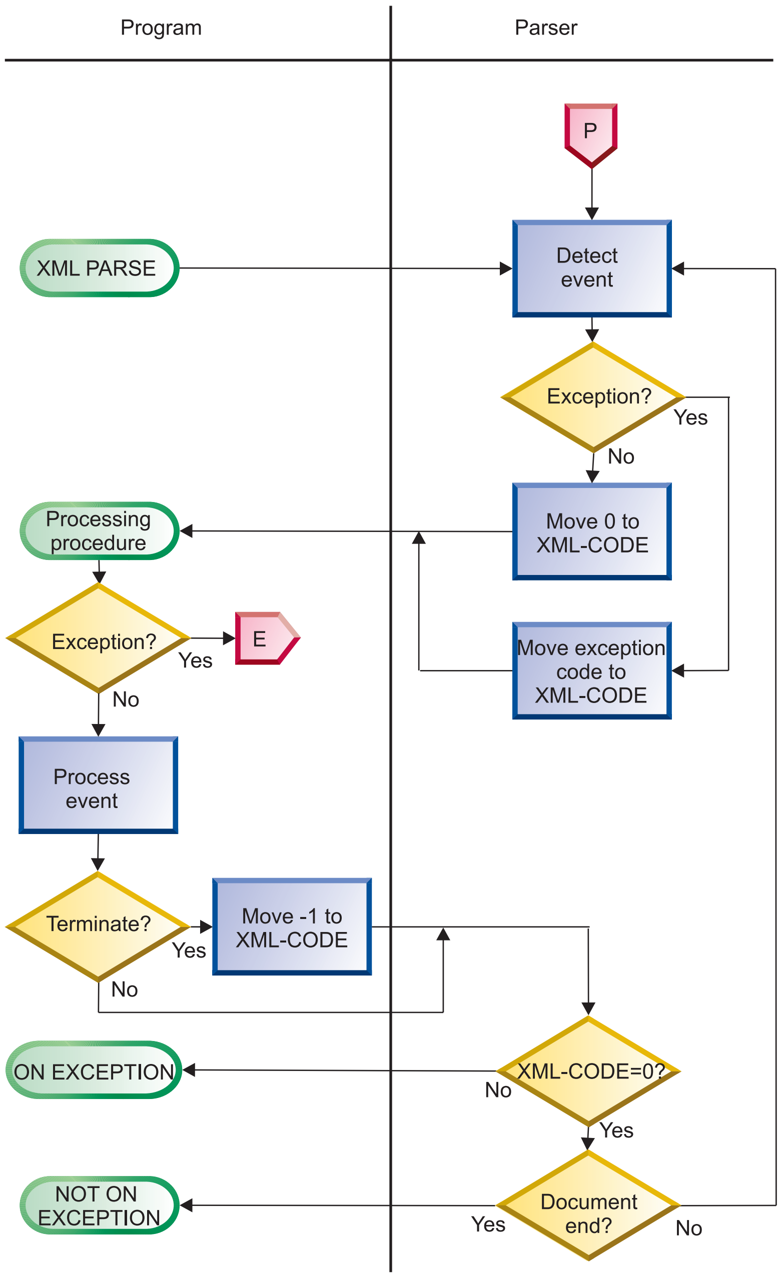
When the parser returns control to your XML PARSE statement, special register XML-CODE contains the most recent value that was set by the parser or by your processing procedure.

For each event except the EXCEPTION event, the value of XML-CODE is zero. If you set XML-CODE to -1 before you return control to the XML parser for an event other than EXCEPTION, processing stops with a user-initiated exception (as indicated by the returned XML-CODE value of -1). The result of changing XML-CODE to any other nonzero value before returning from any event is undefined.

For the EXCEPTION event, special register XML-CODE contains the exception code.

The following figure shows the flow of control between the parser and your processing procedure and shows how XML-CODE is used to pass information between them. The off-page connectors (for example, P) connect the multiple charts in this information. In particular, E in the following figure connects to the chart "Control flow for XML exceptions" on page 506, and P connects from "XML CCSID exception flow control" on page 508.

Control flow between XML parser and program, showing XML-CODE usage



RELATED CONCEPTS
"The content of XML-TEXT and XML-NTEXT" on page 501

RELATED TASKS
"Writing procedures to process XML" on page 495
"Handling exceptions that the XML parser finds" on page 505

RELATED REFERENCES
Appendix D, "XML reference material," on page 669
XML-CODE (*Enterprise COBOL Language Reference*)

The content of XML-TEXT and XML-NTEXT

The special registers `XML-TEXT` and `XML-NTEXT` are mutually exclusive. When the parser sets `XML-TEXT`, `XML-NTEXT` is undefined (length 0). When the parser sets `XML-NTEXT`, `XML-TEXT` is undefined (length 0).

The type of the `XML PARSE` identifier determines which of these two special registers is set, except for the `ATTRIBUTE-NATIONAL-CHARACTER` and `CONTENT-NATIONAL-CHARACTER` events. For these two events, `XML-NTEXT` is set regardless of the data item that you specify as the `XML PARSE` identifier.

To determine how many national characters `XML-NTEXT` contains, use the `LENGTH` function. `LENGTH OF XML-NTEXT` contains the number of bytes, rather than characters, used by `XML-NTEXT`.

RELATED CONCEPTS
"The content of XML-CODE" on page 499

RELATED TASKS
"Writing procedures to process XML" on page 495

RELATED REFERENCES
XML-NTEXT (*Enterprise COBOL Language Reference*)
XML-TEXT (*Enterprise COBOL Language Reference*)

Transforming XML text to COBOL data items

Because XML data is neither fixed length nor fixed format, you need to use special techniques when you move XML data to COBOL data items.

For alphanumeric items, decide whether the XML data should go at the left (default) end of a COBOL item or at the right end. If it should go at the right end, specify the `JUSTIFIED RIGHT` clause in the declaration of the COBOL item.

Give special consideration to numeric XML values, particularly "decorated" monetary values such as '$1,234.00' or '$1234'. These two strings mean the same thing in XML, but would need completely different declarations as COBOL sending fields. Use one of these techniques when you move XML data to COBOL data items:

- If the format is reasonably regular, code a `MOVE` to an alphanumeric item that is redefined appropriately as a numeric-edited item. Then do the final move to a numeric (operational) item by moving from, and thus de-editing, the numeric-edited item. (A regular format would have the same number of digits after the decimal point, a comma separator for values greater than 999, and so on.)
- For simplicity and vastly increased flexibility, use the following functions for alphanumeric XML data:
  - `NUMVAL` to extract and decode simple numeric values from XML data that represents plain numbers
  - `NUMVAL-C` to extract and decode numeric values from XML data that represents monetary quantities

  However, use of these functions is at the expense of performance.

RELATED TASKS
"Using national data (Unicode) in COBOL" on page 126
"Writing procedures to process XML" on page 495

Understanding the encoding of XML documents

The parser decides how to process your document by using certain sources of information about the document encoding. These sources of encoding information must be consistent with one another. The parser signals an XML exception event if it finds a conflict.

The sources of encoding information are:
- The *basic document encoding*
- The *type of the data item* that contains the document
- Any *document encoding declaration*
- The *external code page*, if applicable
- The set of *supported code pages*

The *basic document encoding* is one of the following encoding categories that the parser determines by examining the first few bytes of the XML document:
- ASCII
- EBCDIC
- Unicode UTF-16, either big-endian or little-endian
- Other unsupported encoding
- No recognizable encoding

The *type of the data item* that contains the document is also relevant. The parser supports the following combinations:
- Category national data items whose contents are encoded using Unicode UTF-16
- Category alphanumeric data items whose contents are encoded using one of the supported single-byte character sets

The discovery that the encoding is UTF-16 also provides the code-page information, CCSID 1200 UTF-16BE (big-endian), since Unicode is effectively a single large code page. Thus, if the parser finds a UTF-16 document in a national data item, it ignores external code-page information.

But if the basic document encoding is ASCII or EBCDIC, the parser needs specific code-page information to be able to parse correctly. This additional code-page information is acquired from the document encoding declaration or from the external code page.

The *document encoding declaration* is an optional part of the XML declaration at the beginning of the document. (See the related task about specifying the code page for details.)

The *external code page* is the value in effect for the CODEPAGE compiler option.

Finally, the encoding must be one of the *supported code pages*. (See the related reference below about coded character sets for XML documents for details.)

RELATED TASKS
"Specifying the code page" on page 504

RELATED REFERENCES
"Coded character sets for XML documents"
"XML PARSE exceptions that allow continuation" on page 669
"XML PARSE exceptions that do not allow continuation" on page 672

Coded character sets for XML documents

Documents in national data items must be encoded with Unicode UTF-16, CCSID 1200.

Documents in alphanumeric data items must be encoded with one of the single-byte EBCDIC code pages shown in the table below.

Table 69. **Supported EBCDIC code pages**

CCSID	Description
1047	Latin 1 / Open Systems
1140, 37	USA, Canada, . . . Euro Country Extended Code Page (ECECP), Country Extended Code Page (CECP)
1141, 273	Austria, Germany ECECP, CECP
1142, 277	Denmark, Norway ECECP, CECP
1143, 278	Finland, Sweden ECECP, CECP
1144, 280	Italy ECECP, CECP
1145, 284	Spain, Latin America (Spanish) ECECP, CECP
1146, 285	UK ECECP, CECP
1147, 297	France ECECP, CECP
1148, 500	International ECECP, CECP
1149, 871	Iceland ECECP, CECP

To parse XML documents that are encoded in other code pages, first convert the documents to national characters by using the `NATIONAL-OF` intrinsic function. You can convert the individual pieces of document text that are passed to the processing procedure in special register `XML-NTEXT` back to the original code page by using the `DISPLAY-OF` intrinsic function.

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 133
"Specifying the code page" on page 504

Code-page-sensitive characters in XML markup

Several special characters that are used in XML markup have different hexadecimal representations in various EBCDIC code pages.

The following table shows the special characters and their hexadecimal values for the supported code page CCSIDs.

Table 70. **Hexadecimal values of special characters for code page CCSIDs**

Character	1047	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149
[	X'AD'	X'BA'	X'63'	X'9E'	X'B5'	X'90'	X'4A'	X'B1'	X'90'	X'4A'	X'AE'
]	X'BD'	X'BB'	X'FC'	X'9F'	X'9F'	X'51'	X'5A'	X'BB'	X'B5'	X'5A'	X'9E'
!	X'5A'	X'5A'	X'4F'	X'4F'	X'4F'	X'4F'	X'BB'	X'5A'	X'4F'	X'4F'	X'4F'

Table 70. **Hexadecimal values of special characters for code page CCSIDs** *(continued)*

Character	1047	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149
|	X′4F′	X′4F′	X′BB′	X′BB′	X′BB′	X′BB′	X′4F′	X′4F′	X′BB′	X′BB′	X′BB′
#	X′7B′	X′7B′	X′7B′	X′4A′	X′63′	X′B1′	X′69′	X′7B′	X′B1′	X′7B′	X′7B′

Specifying the code page

The preferred way to specify the code page for parsing an XML document is to omit the encoding declaration from the document and to rely on external code-page information.

Omitting the XML declaration makes it possible to transmit an XML document between heterogeneous systems without requiring that you update the encoding declaration to reflect any translation imposed by the transmission process.

Unicode XML documents do not require additional code-page information because Unicode effectively consists of a single large code page. The code page used for parsing EBCDIC XML documents that do not have an encoding declaration is the value in effect for the `CODEPAGE` compiler option.

You can also specify the encoding information for an XML document in the XML declaration with which most XML documents begin. (Note that the XML parser generates an exception if it encounters an XML declaration that does not begin in the first byte of an XML document.) This is an example of an XML declaration that includes an encoding declaration:

```
<?xml version="1.0" encoding="ibm-1140" ?>
```

Specify the encoding declaration in either of the following ways:
- Specify the CCSID number (with or without any number of leading zeros) optionally prefixed by any of the following strings (in any mixture of uppercase and lowercase):
  - IBM-
  - IBM_
  - CCSID-
  - CCSID_
- Use one of the following supported aliases (in any mixture of uppercase and lowercase):

Table 71. **Aliases for XML encoding declarations**

Code page	Supported aliases
037	EBCDIC-CP-US, EBCDIC-CP-CA, EBCDIC-CP-WT, EBCDIC-CP-NL
500	EBCDIC-CP-BE, EBCDIC-CP-CH
1200	UTF-16

RELATED TASKS
“Understanding the encoding of XML documents” on page 502

Handling exceptions that the XML parser finds

If the exception code that the XML parser passes in `XML-CODE` is within a certain range, you can handle the exception event with a processing procedure.

To handle an exception event in your processing procedure, do these steps:

1. Check the contents of `XML-CODE`.
2. Handle the exception as appropriate.
3. Set `XML-CODE` to zero to indicate that you handled the exception.
4. Return control to the parser. The exception condition then no longer exists.

You can handle exceptions this way only if the exception code that is passed in `XML-CODE` is within one of the following ranges, which indicate that an encoding conflict was detected:

- 50-99
- 100,001-165,535

You can do limited handling of exceptions for which the exception code passed in `XML-CODE` is within the range 1-49. After an exception in this range occurs, the parser does not signal any further normal events, except the `END-OF-DOCUMENT` event, even if you set `XML-CODE` to zero before returning. If you set `XML-CODE` to zero, the parser continues parsing the document and signals any exceptions that it finds. (Doing so can be useful as a way of discovering multiple errors in the document.) At the end of parsing after an exception in this range, control is passed to the statement that you specify in the `ON EXCEPTION` phrase, if any, otherwise to the end of the `XML PARSE` statement. The special register `XML-CODE` contains the code for the most recent exception that the parser detected.

For all other exceptions, the parser signals no further events, and passes control to the statement that you specify in the `ON EXCEPTION` phrase. In this case, `XML-CODE` contains the original exception number even if you set `XML-CODE` to zero in the processing procedure before returning control to the parser.

If you do not want to handle an exception, return control to the parser without changing the value of `XML-CODE`. The parser transfers control to the statement that you specify in the `ON EXCEPTION` phrase. If you do not code an `ON EXCEPTION` phrase, control is transferred to the end of the `XML PARSE` statement.

If you return control to the parser with `XML-CODE` set to a nonzero value that is different from the original exception code, the results are undefined.

If no unhandled exceptions occur before the end of parsing, control is passed to the statement that you specify in the `NOT ON EXCEPTION` phrase (normal end of parsing). If you do not code a `NOT ON EXCEPTION` phrase, control is passed to the end of the `XML PARSE` statement. The special register `XML-CODE` contains zero.

RELATED CONCEPTS
"How the XML parser handles errors" on page 506
"The content of XML-CODE" on page 499

RELATED TASKS
"Writing procedures to process XML" on page 495
"Understanding the encoding of XML documents" on page 502
"Handling conflicts in code pages" on page 508

RELATED REFERENCES
"XML PARSE exceptions that allow continuation" on page 669
"XML PARSE exceptions that do not allow continuation" on page 672
XML-CODE (*Enterprise COBOL Language Reference*)

How the XML parser handles errors

When the XML parser detects an error in an XML document, it generates an XML exception event and passes control to your processing procedure.

The parser provides the following information in special registers:

- `XML-EVENT` contains `'EXCEPTION'`.
- `XML-CODE` contains a numeric exception code.
- `XML-TEXT` (for XML documents in an alphanumeric data item) or `XML-NTEXT` (for XML documents in a national data item) contains the document text up to and including the point where the exception was detected.

You might be able to handle the exception in your processing procedure and continue parsing if the numeric exception code is within one of the following ranges:

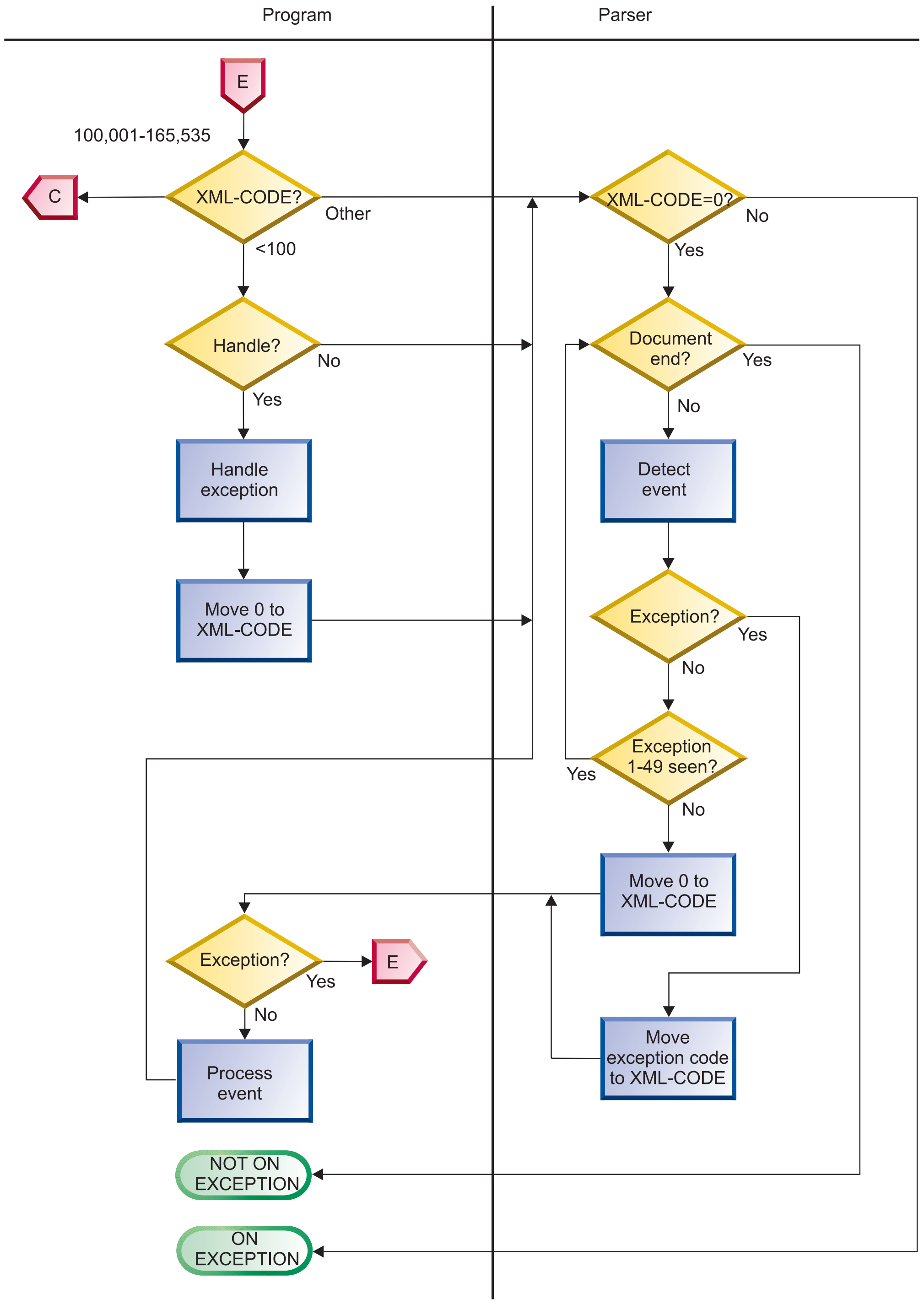
- 1-99
- 100,001-165,535

If the exception code has any other nonzero value, parsing cannot continue. The exceptions for encoding conflicts (50-99 and 300-399) are signaled before the parsing of the document begins. For these exceptions, `XML-TEXT` (or `XML-NTEXT`) either is zero length or contains only the encoding declaration value from the document.

Exceptions in the range 1-49 are fatal errors according to the XML specification. Therefore the parser does not continue normal parsing even if you handle the exception. However, the parser does continue scanning for further errors until it reaches the end of the document or encounters an error that does not allow continuation. For these exceptions, the parser does not signal any further normal events except the `END-OF-DOCUMENT` event.

Use the following figure to understand the flow of control between the parser and your processing procedure. The figure illustrates how you can handle certain exceptions and can use `XML-CODE` to identify the exceptions. The off-page connectors (for example, P) connect the multiple charts in this information. In particular, C connects to the chart "XML CCSID exception flow control" on page 508. Within this figure, E and F serve both as off-page and on-page connectors.

Control flow for XML exceptions



RELATED TASKS
"Understanding the encoding of XML documents" on page 502
"Handling exceptions that the XML parser finds" on page 505
"Handling conflicts in code pages" on page 508
"Terminating XML parsing" on page 509

RELATED REFERENCES
"XML PARSE exceptions that allow continuation" on page 669
"XML PARSE exceptions that do not allow continuation" on page 672
XML-CODE (*Enterprise COBOL Language Reference*)

Handling conflicts in code pages

Exception events in which the document item is alphanumeric and the exception code in `XML-CODE` is between 100,001 and 165,535 indicate that the code page of the document (as specified by its encoding declaration) conflicts with the external code-page information.

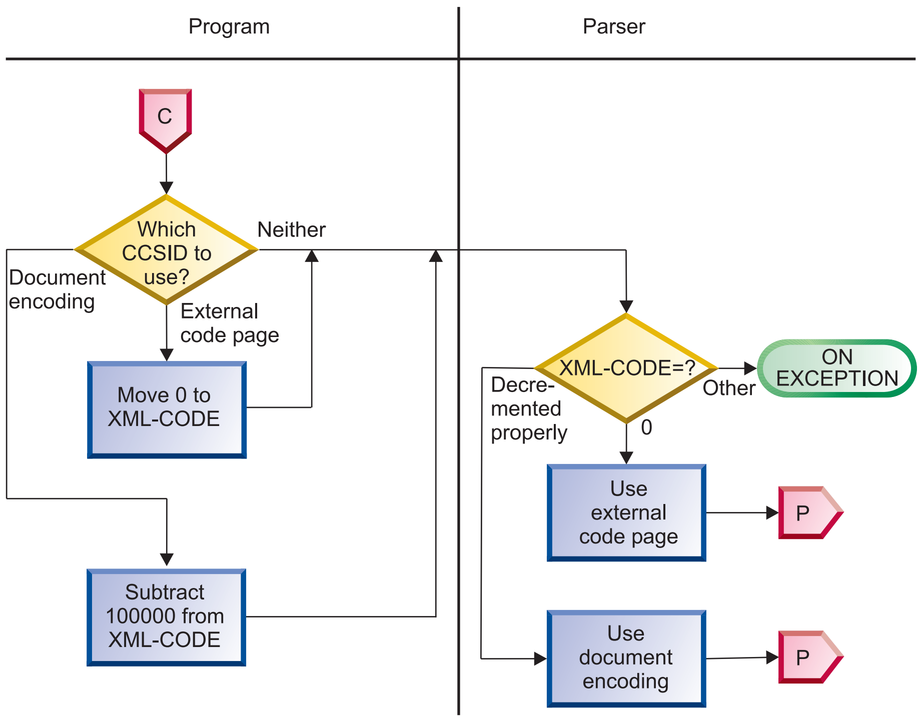
In this special case, you can choose to parse with the code page of the document by subtracting 100,000 from the value in `XML-CODE`. For instance, if `XML-CODE` contains 101,140, the code page of the document is 1140. Alternatively, you can choose to parse with the external code page by setting `XML-CODE` to zero before returning to the parser.

The parser takes one of three actions after returning from a processing procedure for a code-page conflict exception event:

- If you set `XML-CODE` to zero, the parser uses the external code page: the `CODEPAGE` compiler option value.
- If you set `XML-CODE` to the code page of the document (that is, the original `XML-CODE` value minus 100,000), the parser uses the code page of the document. This is the only case where the parser continues when `XML-CODE` has a nonzero value upon returning from a processing procedure.
- Otherwise, the parser stops processing the document and returns control to the `XML PARSE` statement with an exception condition. `XML-CODE` contains the exception code that was originally passed to the exception event.

The following figure illustrates these actions. The off-page connectors (for example, P) connect the multiple charts in this information. In particular, P in the following figure connects to "Control flow between XML parser and program, showing XML-CODE usage" on page 500, and C connects from "Control flow for XML exceptions" on page 506.

XML CCSID exception flow control



In this figure, *CCSID* (coded character set identifier) is a value from 1 to 65,536 that denotes the code page.

RELATED CONCEPTS
"How the XML parser handles errors" on page 506

RELATED TASKS
"Understanding the encoding of XML documents" on page 502
"Handling exceptions that the XML parser finds" on page 505

RELATED REFERENCES
"XML PARSE exceptions that allow continuation" on page 669
"XML PARSE exceptions that do not allow continuation" on page 672
XML-CODE (*Enterprise COBOL Language Reference*)

Terminating XML parsing

You can terminate parsing deliberately by setting `XML-CODE` to -1 in your processing procedure before returning to the parser from any normal XML event (that is, not an `EXCEPTION` event). You can use this technique when you have seen enough of the document or have detected some irregularity in the document that precludes further meaningful processing.

In this case, the parser does not signal any further events although an exception condition exists. Therefore, control returns to the `ON EXCEPTION` phrase if specified. In the imperative statement of the `ON EXCEPTION` phrase, you can test whether `XML-CODE` is -1, which indicates that you terminated parsing deliberately. If you do not specify an `ON EXCEPTION` phrase, control returns to the end of the `XML PARSE` statement.

You can also terminate parsing after any XML exception event by returning to the parser without changing `XML-CODE`. The result is similar to the result of deliberate termination except that the parser returns to the `XML PARSE` statement with `XML-CODE` containing the exception number.

RELATED CONCEPTS
"How the XML parser handles errors" on page 506

RELATED TASKS
"Handling exceptions that the XML parser finds" on page 505

RELATED REFERENCES
XML-CODE (*Enterprise COBOL Language Reference*)

Chapter 29. Producing XML output

You can produce XML output from a COBOL program by using the XML GENERATE statement. In the XML GENERATE statement, you can also identify a field to receive a count of the number of characters of XML output generated, and a statement to receive control if an exception occurs.

To produce XML output, use:

- The XML GENERATE statement to identify the source and target data items, count field, and ON EXCEPTION statement
- The special register XML-CODE to determine the status of XML generation

After you transform COBOL data items to XML, you can use the resulting XML output in various ways, such as deploying it in a Web service, passing it as a message to WebSphere MQ, or transmitting it for subsequent conversion to a CICS communication area.

+ **Link-edit considerations:** COBOL programs that contain the XML GENERATE
+ statement must be link-edited with AMODE 31.

RELATED TASKS
"Generating XML output"
"Enhancing XML output" on page 517
"Controlling the encoding of generated XML output" on page 521
"Handling errors in generating XML output" on page 522

Generating XML output

To transform COBOL data to XML, use the XML GENERATE statement.

```
XML GENERATE XML-OUTPUT FROM SOURCE-REC
      COUNT IN XML-CHAR-COUNT
  ON EXCEPTION
    DISPLAY 'XML generation error ' XML-CODE
    STOP RUN
  NOT ON EXCEPTION
    DISPLAY 'XML document was successfully generated.'
END-XML
```

In the XML GENERATE statement, you first identify the data item (XML-OUTPUT in the example above) that is to receive the XML output. Define the data item to be large enough to contain the generated XML output, typically five to eight times the size of the COBOL source data depending on the length of its data-name or data-names.

| In the DATA DIVISION, you can declare the receiving identifier as alphanumeric
| (either an alphanumeric group item or an elementary item of category
| alphanumeric) or as national (either a national group item or an elementary item
| of category national).

| The receiving identifier must be national if the CODEPAGE compiler option specifies a code page that includes DBCS characters or the XML output will contain any data from the COBOL source record that has any of the following characteristics:

- | Is of class national or class DBCS

- Has a DBCS name (that is, is a data item whose name contains DBCS characters)
- Is an alphanumeric item that contains DBCS characters

Next you identify the source data item that is to be transformed to XML format (SOURCE-REC in the example). The source data item can be an alphanumeric group item, national group item, or elementary data item of class alphanumeric or national. Do not specify the RENAMES clause in the data description of that data item.

If the source data item is an alphanumeric group item or a national group item, the source data item is processed with group semantics, not as an elementary item. Any groups that are subordinate to the source data item are also processed with group semantics.

Some COBOL data items are not transformed to XML, but are ignored. Subordinate data items of an alphanumeric group item or national group item that you transform to XML are ignored if they:

- Specify the REDEFINES clause, or are subordinate to such a redefining item
- Specify the RENAMES clause

These items in the source data item are also ignored when you generate XML:

- Elementary FILLER (or unnamed) data items
- Slack bytes inserted for SYNCHRONIZED data items

There must be at least one elementary data item that is not ignored when you generate XML. For the data items that are not ignored, ensure that the identifier that you transform to XML satisfies these conditions when you declare it in the DATA DIVISION:

- Each elementary data item is either an index data item or belongs to one of these classes:
  - Alphabetic
  - Alphanumeric
  - DBCS
  - Numeric
  - National

  That is, no elementary data item is described with the USAGE POINTER, USAGE FUNCTION-POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE phrase.
- Each data-name other than FILLER is unique within the immediately containing group, if any.
- Any DBCS data-names, when converted to Unicode, are legal as names in the XML specification, version 1.0.
- The data item or items do not specify the DATE FORMAT clause, or the DATEPROC compiler option is not in effect.

An XML declaration is not generated. No white space (for example, new lines or indentation) is inserted to make the generated XML more readable.

Optionally, you can code the COUNT IN phrase to obtain the number of XML character positions that are filled during generation of the XML output. Declare the count field as an integer data item that does not have the symbol P in its PICTURE string. You can use the count field and reference modification to obtain only that portion of the receiving data item that contains the generated XML output. For example, XML-OUTPUT(1:XML-CHAR-COUNT) references the first XML-CHAR-COUNT character positions of XML-OUTPUT.

In addition, you can specify either or both of the following phrases to receive control after generation of the XML document:

- ON EXCEPTION, to receive control if an error occurs during XML generation
- NOT ON EXCEPTION, to receive control if no error occurs

You can end the XML GENERATE statement with the explicit scope terminator END-XML. Code END-XML to nest an XML GENERATE statement that has the ON EXCEPTION or NOT ON EXCEPTION phrase in a conditional statement.

XML generation continues until either the COBOL source record has been transformed to XML or an error occurs. If an error occurs, the results are as follows:

- Special register XML-CODE contains a nonzero exception code.
- Control is passed to the ON EXCEPTION phrase, if specified, otherwise to the end of the XML GENERATE statement.

If no error occurs during XML generation, special register XML-CODE contains zero, and control is passed to the NOT ON EXCEPTION phrase if specified or to the end of the XML GENERATE statement otherwise.

"Example: generating XML"

RELATED TASKS
"Controlling the encoding of generated XML output" on page 521
"Handling errors in generating XML output" on page 522

RELATED REFERENCES
Classes and categories of data (*Enterprise COBOL Language Reference*)
XML GENERATE statement (*Enterprise COBOL Language Reference*)
Operation of XML GENERATE (*Enterprise COBOL Language Reference*)
Extensible Markup Language (XML)

Example: generating XML

The following example simulates the building of a purchase order in a group data item, and generates an XML version of that purchase order.

Program XGFX uses XML GENERATE to produce XML output in elementary data item xmlPO from the source record, group data item purchaseOrder. Elementary data items in the source record are converted to character format as necessary, and the characters are inserted in XML elements whose names are derived from the data-names in the source record.

XGFX calls program Pretty, which uses the XML PARSE statement with processing procedure p to format the XML output with new lines and indentation so that the XML content can more easily be verified.

Program XGFX

```
Identification division.
  Program-id. XGFX.
Data division.
 Working-storage section.
```

```
   01 numItems pic 99 global.
   01 purchaseOrder global.
     05 orderDate pic x(10).
     05 shipTo.
       10 country pic xx value 'US'.
       10 name pic x(30).
       10 street pic x(30).
       10 city pic x(30).
       10 state pic xx.
       10 zip pic x(10).
     05 billTo.
       10 country pic xx value 'US'.
       10 name pic x(30).
       10 street pic x(30).
       10 city pic x(30).
       10 state pic xx.
       10 zip pic x(10).
     05 orderComment pic x(80).
     05 items occurs 0 to 20 times depending on numItems.
       10 item.
         15 partNum pic x(6).
         15 productName pic x(50).
         15 quantity pic 99.
         15 USPrice pic 999v99.
         15 shipDate pic x(10).
         15 itemComment pic x(40).
   01 numChars comp pic 999.
   01 xmlPO pic x(999).
Procedure division.
  m.
    Move 20 to numItems
    Move spaces to purchaseOrder

    Move '1999-10-20' to orderDate

    Move 'US' to country of shipTo
    Move 'Alice Smith' to name of shipTo
    Move '123 Maple Street' to street of shipTo
    Move 'Mill Valley' to city of shipTo
    Move 'CA' to state of shipTo
    Move '90952' to zip of shipTo

    Move 'US' to country of billTo
    Move 'Robert Smith' to name of billTo
    Move '8 Oak Avenue' to street of billTo
    Move 'Old Town' to city of billTo
    Move 'PA' to state of billTo
    Move '95819' to zip of billTo
    Move 'Hurry, my lawn is going wild!' to orderComment

    Move 0 to numItems
    Call 'addFirstItem'
    Call 'addSecondItem'
    Move space to xmlPO
    Xml generate xmlPO from purchaseOrder count in numChars
    Call 'pretty' using xmlPO value numChars
    Goback
    .

Identification division.
  Program-id. 'addFirstItem'.
Procedure division.
    Add 1 to numItems
    Move '872-AA' to partNum(numItems)
    Move 'Lawnmower' to productName(numItems)
    Move 1 to quantity(numItems)
    Move 148.95 to USPrice(numItems)
    Move 'Confirm this is electric' to itemComment(numItems)
    Goback.
End program 'addFirstItem'.

Identification division.
  Program-id. 'addSecondItem'.
Procedure division.
    Add 1 to numItems
    Move '926-AA' to partNum(numItems)
    Move 'Baby Monitor' to productName(numItems)
    Move 1 to quantity(numItems)
    Move 39.98 to USPrice(numItems)
    Move '1999-05-21' to shipDate(numItems)
    Goback.
End program 'addSecondItem'.

End program XGFX.
```

Program Pretty

```
Identification division.
  Program-id. Pretty.
Data division.
 Working-storage section.
   01 prettyPrint.
     05 pose pic 999.
     05 posd pic 999.
     05 depth pic 99.
     05 element pic x(30).
     05 indent pic x(20).
     05 buffer pic x(100).
 Linkage section.
  1 doc.
   2 pic x occurs 16384 times depending on len.
  1 len comp-5 pic 9(9).
Procedure division using doc value len.
  m.
    Move space to prettyPrint
    Move 0 to depth posd
    Move 1 to pose
    Xml parse doc processing procedure p
    Goback.
  p.
    Evaluate xml-event
      When 'START-OF-ELEMENT'
        If element not = space
          If depth > 1
            Display indent(1:2 * depth - 2) buffer(1:pose - 1)
          Else
            Display buffer(1:pose - 1)
          End-if
        End-if
        Move xml-text to element
        Add 1 to depth
        Move 1 to pose
        String '<' xml-text '>' delimited by size into buffer
            with pointer pose
        Move pose to posd
      When 'CONTENT-CHARACTERS'
        String xml-text delimited by size into buffer
            with pointer posd
      When 'CONTENT-CHARACTER'
        String xml-text delimited by size into buffer
            with pointer posd
      When 'END-OF-ELEMENT'
        Move space to element
        String '</' xml-text '>' delimited by size into buffer
            with pointer posd
        If depth > 1
          Display indent(1:2 * depth - 2) buffer(1:posd - 1)
        Else
          Display buffer(1:posd - 1)
        End-if
        Subtract 1 from depth
        Move 1 to posd
      When other
        Continue
    End-evaluate
    .
End program Pretty.
```

Output from program XGFX

```
<purchaseOrder>
  <orderDate>1999-10-20</orderDate>
  <shipTo>
    <country>US</country>
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo>
    <country>US</country>
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <orderComment>Hurry, my lawn is going wild!</orderComment>
  <items>
    <item>
      <partNum>872-AA</partNum>
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <shipDate> </shipDate>
      <itemComment>Confirm this is electric</itemComment>
    </item>
  </items>
  <items>
    <item>
      <partNum>926-AA</partNum>
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
      <itemComment> </itemComment>
    </item>
  </items>
</purchaseOrder>
```

RELATED TASKS

Chapter 28, “Processing XML input,” on page 487

RELATED REFERENCES

Operation of XML GENERATE (*Enterprise COBOL Language Reference*)

Enhancing XML output

It might happen that the information that you want to express in XML format already exists in a group item in the DATA DIVISION, but you are unable to use that item directly to generate an XML document because of one or more factors.

For example:

- In addition to the required data, the item has subordinate data items that contain values that are irrelevant to the XML output document.
- The names of the required data items are unsuitable for external presentation, and are possibly meaningful only to programmers.
- The definition of the data is not of the required data type. Perhaps only the redefinitions (which are ignored by the XML GENERATE statement) have the appropriate format.
- The required data items are nested too deeply within irrelevant subordinate groups. The XML output should be "flattened" rather than hierarchical as it would be by default.
- The required data items are broken up into too many components, and should be output as the content of the containing group.
- The group item contains the required information but in the wrong order.

There are various ways that you can deal with such situations. One possible technique is to define a new data item that has the appropriate characteristics, and move the required data to the appropriate fields of this new data item. However, this approach is somewhat laborious and requires careful maintenance to keep the original and new data items synchronized.

An alternative approach that has some advantages is to provide a redefinition of the original group data item, and to generate the XML output from that redefinition. To do so, start from the original set of data descriptions, and make these changes:

- Exclude elementary data items from the generated XML either by renaming them to FILLER or by deleting their names.
- Provide more meaningful and appropriate names for the selected elementary items and for the group items that contain them.
- Remove unneeded intermediate group items to flatten the hierarchy.
- Specify different data types to obtain the desired trimming behavior.
- Choose a different order for the output by using a sequence of XML GENERATE statements.

The safest way to accomplish these changes is to use another copy of the original declarations accompanied by one or more REPLACE compiler-directing statements.

"Example: enhancing XML output" on page 518

You might also find when you generate an XML document that some of the element names and element values contain hyphens. You might want to convert the hyphens in the element names to underscores without changing the hyphens that are in the element values. The example that is referenced below shows a way to do so.

"Example: converting hyphens in element names to underscores" on page 520

RELATED REFERENCES
Operation of XML GENERATE (*Enterprise COBOL Language Reference*)

Example: enhancing XML output

The following example shows how you can improve XML output.

Consider the following data structure. The XML that is generated from the structure suffers from several problems that can be corrected.

```
01  CDR-LIFE-BASE-VALUES-BOX.
    15  CDR-LIFE-BASE-VAL-DATE    PIC X(08).
    15  CDR-LIFE-BASE-VALUE-LINE  OCCURS  2 TIMES.
        20  CDR-LIFE-BASE-DESC.
            25 CDR-LIFE-BASE-DESC1 PIC X(15).
            25  FILLER             PIC X(01).
            25  CDR-LIFE-BASE-LIT  PIC X(08).
            25  CDR-LIFE-BASE-DTE  PIC X(08).
        20  CDR-LIFE-BASE-PRICE.
            25  CDR-LIFE-BP-SPACE  PIC X(02).
            25  CDR-LIFE-BP-DASH   PIC X(02).
            25  CDR-LIFE-BP-SPACE1 PIC X(02).
        20  CDR-LIFE-BASE-PRICE-ED  REDEFINES
             CDR-LIFE-BASE-PRICE  PIC $$$.$$.
        20  CDR-LIFE-BASE-QTY.
            25  CDR-LIFE-QTY-SPACE   PIC X(08).
            25  CDR-LIFE-QTY-DASH    PIC X(02).
            25  CDR-LIFE-QTY-SPACE1  PIC X(02).
            25  FILLER               PIC X(02).
        20  CDR-LIFE-BASE-QTY-ED    REDEFINES
                CDR-LIFE-BASE-QTY PIC ZZ,ZZZ,ZZZ.ZZZ.
        20  CDR-LIFE-BASE-VALUE   PIC X(15).
        20  CDR-LIFE-BASE-VALUE-ED  REDEFINES
                CDR-LIFE-BASE-VALUE
                                PIC $(4),$$$,$$9.99.
    15  CDR-LIFE-BASE-TOT-VALUE-LINE.
        20  CDR-LIFE-BASE-TOT-VALUE   PIC X(15).
```

When this data structure is populated with some sample values, and XML is generated directly from it and then formatted using program `Pretty` (shown in “Example: generating XML” on page 513), the result is as follows:

```
<CDR-LIFE-BASE-VALUES-BOX>
  <CDR-LIFE-BASE-VAL-DATE>01/02/03</CDR-LIFE-BASE-VAL-DATE>
  <CDR-LIFE-BASE-VALUE-LINE>
    <CDR-LIFE-BASE-DESC>
      <CDR-LIFE-BASE-DESC1>First</CDR-LIFE-BASE-DESC1>
      <CDR-LIFE-BASE-LIT> </CDR-LIFE-BASE-LIT>
      <CDR-LIFE-BASE-DTE>01/01/01</CDR-LIFE-BASE-DTE>
    </CDR-LIFE-BASE-DESC>
    <CDR-LIFE-BASE-PRICE>
      <CDR-LIFE-BP-SPACE>$2</CDR-LIFE-BP-SPACE>
      <CDR-LIFE-BP-DASH>3.</CDR-LIFE-BP-DASH>
      <CDR-LIFE-BP-SPACE1>00</CDR-LIFE-BP-SPACE1>
    </CDR-LIFE-BASE-PRICE>
    <CDR-LIFE-BASE-QTY>
      <CDR-LIFE-QTY-SPACE>       1</CDR-LIFE-QTY-SPACE>
      <CDR-LIFE-QTY-DASH>23</CDR-LIFE-QTY-DASH>
      <CDR-LIFE-QTY-SPACE1>.0</CDR-LIFE-QTY-SPACE1>
    </CDR-LIFE-BASE-QTY>
    <CDR-LIFE-BASE-VALUE>        $765.00</CDR-LIFE-BASE-VALUE>
  </CDR-LIFE-BASE-VALUE-LINE>
  <CDR-LIFE-BASE-VALUE-LINE>
    <CDR-LIFE-BASE-DESC>
      <CDR-LIFE-BASE-DESC1>Second</CDR-LIFE-BASE-DESC1>
      <CDR-LIFE-BASE-LIT> </CDR-LIFE-BASE-LIT>
      <CDR-LIFE-BASE-DTE>02/02/02</CDR-LIFE-BASE-DTE>
    </CDR-LIFE-BASE-DESC>
    <CDR-LIFE-BASE-PRICE>
      <CDR-LIFE-BP-SPACE>$3</CDR-LIFE-BP-SPACE>
      <CDR-LIFE-BP-DASH>4.</CDR-LIFE-BP-DASH>
      <CDR-LIFE-BP-SPACE1>00</CDR-LIFE-BP-SPACE1>
    </CDR-LIFE-BASE-PRICE>
    <CDR-LIFE-BASE-QTY>
      <CDR-LIFE-QTY-SPACE>       2</CDR-LIFE-QTY-SPACE>
      <CDR-LIFE-QTY-DASH>34</CDR-LIFE-QTY-DASH>
      <CDR-LIFE-QTY-SPACE1>.0</CDR-LIFE-QTY-SPACE1>
    </CDR-LIFE-BASE-QTY>
    <CDR-LIFE-BASE-VALUE>        $654.00</CDR-LIFE-BASE-VALUE>
  </CDR-LIFE-BASE-VALUE-LINE>
  <CDR-LIFE-BASE-TOT-VALUE-LINE>
    <CDR-LIFE-BASE-TOT-VALUE>Very high!</CDR-LIFE-BASE-TOT-VALUE>
  </CDR-LIFE-BASE-TOT-VALUE-LINE>
</CDR-LIFE-BASE-VALUES-BOX>
```

This generated XML suffers from several problems:

- The element names are long and not very meaningful.
- There is unwanted data, for example, `CDR-LIFE-BASE-LIT` and `CDR-LIFE-BASE-DTE`.
- Required data has an unnecessary parent. For example, `CDR-LIFE-BASE-DESC1` has parent `CDR-LIFE-BASE-DESC`.
- Other required fields are split into too many subcomponents. For example, `CDR-LIFE-BASE-PRICE` has three subcomponents for one amount.

These and other characteristics of the XML output can be remedied by redefining the storage as follows:

```
1 BaseValues redefines CDR-LIFE-BASE-VALUES-BOX.
 2 BaseValueDate pic x(8).
 2 BaseValueLine occurs 2 times.
  3 Description pic x(15).
  3 pic x(9).
  3 BaseDate pic x(8).
  3 BasePrice pic x(6) justified.
  3 BaseQuantity pic x(14) justified.
  3 BaseValue pic x(15) justified.
 2 TotalValue pic x(15).
```

The result of generating and formatting XML from the set of definitions of the data values shown above is more usable:

```
<BaseValues>
  <BaseValueDate>01/02/03</BaseValueDate>
  <BaseValueLine>
    <Description>First</Description>
    <BaseDate>01/01/01</BaseDate>
    <BasePrice>$23.00</BasePrice>
    <BaseQuantity>123.000</BaseQuantity>
    <BaseValue>$765.00</BaseValue>
  </BaseValueLine>
  <BaseValueLine>
    <Description>Second</Description>
    <BaseDate>02/02/02</BaseDate>
    <BasePrice>$34.00</BasePrice>
    <BaseQuantity>234.000</BaseQuantity>
    <BaseValue>$654.00</BaseValue>
  </BaseValueLine>
  <TotalValue>Very high!</TotalValue>
</BaseValues>
```

You can redefine the original data definition directly, as shown above. However, it is generally safer to use the original definition but to modify it suitably using the text-manipulation capabilities of the compiler. An example is shown in the REPLACE compiler-directing statement below. This REPLACE statement might appear complicated, but it has the advantage of being self-maintaining if the original data definitions are modified.

```
replace ==CDR-LIFE-BASE-VALUES-BOX== by
            ==BaseValues redefines CDR-LIFE-BASE-VALUES-BOX==
        ==CDR-LIFE-BASE-VAL-DATE== by ==BaseValueDate==
        ==CDR-LIFE-BASE-VALUE-LINE== by ==BaseValueLine==
        ==20  CDR-LIFE-BASE-DESC.== by ====
        ==CDR-LIFE-BASE-DESC1== by ==Description==
        ==CDR-LIFE-BASE-LIT== by ====
        ==CDR-LIFE-BASE-DTE== by ==BaseDate==
        ==20  CDR-LIFE-BASE-PRICE.== by ====
        ==25  CDR-LIFE-BP-SPACE PIC X(02).== by ====
        ==25  CDR-LIFE-BP-DASH PIC X(02).== by ====
        ==25  CDR-LIFE-BP-SPACE1 PIC X(02).== by ====
        ==CDR-LIFE-BASE-PRICE-ED== by ==BasePrice==
        ==REDEFINES CDR-LIFE-BASE-PRICE PIC $$$.$$.== by
            ==pic x(6) justified.==
        ==20  CDR-LIFE-BASE-QTY.
            25  CDR-LIFE-QTY-SPACE PIC X(08).
            25  CDR-LIFE-QTY-DASH PIC X(02).
            25  CDR-LIFE-QTY-SPACE1 PIC X(02).
            25  FILLER PIC X(02).== by ====
        ==CDR-LIFE-BASE-QTY-ED== by ==BaseQuantity==
        ==REDEFINES CDR-LIFE-BASE-QTY PIC ZZ,ZZZ,ZZZ.ZZZ.== by
            ==pic x(14) justified.==
        ==CDR-LIFE-BASE-VALUE-ED== by ==BaseValue==
        ==20  CDR-LIFE-BASE-VALUE PIC X(15).== by ====
        ==REDEFINES CDR-LIFE-BASE-VALUE PIC $(4),$$$,$$9.99.==
            by ==pic x(15) justified.==
        ==CDR-LIFE-BASE-TOT-VALUE-LINE. 20== by ====
        ==CDR-LIFE-BASE-TOT-VALUE== by ==TotalValue==.
```

The result of this REPLACE statement followed by a second instance of the original set of definitions is similar to the suggested redefinition of group item BaseValues shown above. This REPLACE statement illustrates a variety of techniques for eliminating unwanted definitions and for modifying the definitions that should be retained. Use whichever technique is appropriate for your situation.

RELATED REFERENCES
Operation of XML GENERATE (*Enterprise COBOL Language Reference*)
REPLACE statement (*Enterprise COBOL Language Reference*)

Example: converting hyphens in element names to underscores

| When you generate an XML document from a data structure whose items have
| data-names that contain hyphens, the generated XML has element names that
| contain hyphens. This example shows a way to convert the hyphens in the element
| names without changing hyphens that occur in the element values.

```
1 Customer-Record.
 2 Customer-Number pic 9(9).
 2 First-Name      pic x(10).
 2 Last-Name       pic x(20).
```

| When the data structure above is populated with some sample values, and XML is
| generated directly from it and then formatted using program Pretty (shown in
| “Example: generating XML” on page 513), the result is as follows:

```
<Customer-Record>
  <Customer-Number>12345</Customer-Number>
  <First-Name>John</First-Name>
  <Last-Name>Smith-Jones</Last-Name>
</Customer-Record>
```

The element names contain hyphens, and the content of the element `Last-Name` also contains a hyphen.

Assuming that this XML document is the content of data item `xmldoc`, and that `charcnt` is set to the length of this XML document, you can change all the hyphens in the element names to underscores but leave the element values unchanged by using the following code:

```
1 xmldoc         pic x(16384).
1 charcnt  comp-5 pic 9(5).
1 pos      comp-5 pic 9(5).
1 tagstate comp-5 pic 9  value zero.
. . .
dash-to-underscore.
  perform varying pos from 1 by 1
          until pos > charcnt
    if xmldoc(pos:1) = '<'
      move 1 to tagstate
    end-if
    if tagstate = 1 and xmldoc(pos:1) = '-'
      move '_' to xmldoc(pos:1)
    else
      if xmldoc(pos:1) = '>'
        move 0 to tagstate
      end-if
    end-if
  end-perform.
```

The revised XML document in data item `xmldoc` has underscores instead of hyphens in the element names, as shown below:

```
<Customer_Record>
  <Customer_Number>12345</Customer_Number>
  <First_Name>John</First_Name>
  <Last_Name>Smith-Jones</Last_Name>
</Customer_Record>
```

Controlling the encoding of generated XML output

When you generate XML output by using the `XML GENERATE` statement, you can control the encoding of the output by the category of the data item that receives the XML output.

Table 72. **Encoding of generated XML output**

If you define the receiving XML identifier as:	**The generated XML output is encoded in:**
Alphanumeric	The code page specified by the `CODEPAGE` compiler option in effect when the source was compiled
National	UTF-16 big-endian (UTF-16BE, CCSID 1200)[1]

1. A *byte order mark* is not generated.

For details about how data items are converted to XML and how the XML element names are formed from the COBOL data-names, see the related reference below about the operation of the `XML GENERATE` statement.

RELATED REFERENCES
"CODEPAGE" on page 305
Operation of XML GENERATE (*Enterprise COBOL Language Reference*)

Handling errors in generating XML output

When an error is detected during generation of XML output, an exception condition exists. You can write code to check the special register `XML-CODE`, which contains a numeric exception code that indicates the error type.

To handle errors, use either or both of the following phrases of the `XML GENERATE` statement:

- `ON EXCEPTION`
- `COUNT IN`

If you code the `ON EXCEPTION` phrase in the `XML GENERATE` statement, control is transferred to the imperative statement that you specify. You might code an imperative statement, for example, to display the `XML-CODE` value. If you do not code an `ON EXCEPTION` phrase, control is transferred to the end of the `XML GENERATE` statement.

When an error occurs, one problem might be that the data item that receives the XML output is not large enough. In that case, the XML output is not complete, and special register `XML-CODE` contains error code 400.

You can examine the generated XML output by doing these steps:

1. Code the `COUNT IN` phrase in the `XML GENERATE` statement.

   The count field that you specify holds a count of the XML character positions that are filled during XML generation. If you define the XML output as national, the count is in national character positions (UTF-16 character encoding units); otherwise the count is in bytes.
2. Use the count field with reference modification to refer to the substring of the receiving data item that contains the generated XML output.

   For example, if `XML-OUTPUT` is the data item that receives the XML output, and `XML-CHAR-COUNT` is the count field, then `XML-OUTPUT(1:XML-CHAR-COUNT)` references the XML output.

Use the contents of `XML-CODE` to determine what corrective action to take. For a list of the exceptions that can occur during XML generation, see the related reference below.

RELATED TASKS
"Referring to substrings of data items" on page 107

RELATED REFERENCES
"XML GENERATE exceptions" on page 677

Part 6. Developing object-oriented programs

Chapter 30. Writing object-oriented programs

When you write an object-oriented (OO) program, you have to determine what classes you need and the methods and data that the classes need to do their work.

OO programs are based on *objects* (entities that encapsulate state and behavior) and their classes, methods, and data. A *class* is a template that defines the state and the capabilities of an object. Usually a program creates and works with multiple *object instances* (or simply, *instances*) of a class, that is, multiple objects that are members of that class. The state of each instance is stored in data known as *instance data*, and the capabilities of each instance are called *instance methods*. A class can define data that is shared by all instances of the class, known as *factory* or *static* data, and methods that are supported independently of any object instance, known as *factory* or *static* methods.

Using Enterprise COBOL, you can:
- Define classes, with methods and data implemented in COBOL.
- Create instances of Java and COBOL classes.
- Invoke methods on Java and COBOL objects.
- Write classes that inherit from Java classes or other COBOL classes.
- Define and invoke overloaded methods.

In Enterprise COBOL programs, you can call the services provided by the Java Native Interface (JNI) to obtain Java-oriented capabilities in addition to the basic OO capabilities available directly in the COBOL language.

In Enterprise COBOL classes, you can code `CALL` statements to interface with procedural COBOL programs. Thus COBOL class definition syntax can be especially useful for writing *wrapper* classes for procedural COBOL logic, enabling existing COBOL code to be accessed from Java.

Java code can create instances of COBOL classes, invoke methods of these classes, and can extend COBOL classes.

It is recommended that you develop and run OO COBOL programs and Java programs in the z/OS UNIX System Services environment.

Restrictions:
- COBOL class definitions and methods cannot contain `EXEC SQL` statements and cannot be compiled using the `SQL` compiler option.
- COBOL class definitions and methods cannot contain `EXEC CICS` statements, and cannot be run in a CICS environment. They cannot be compiled using the `CICS` compiler option.

"Example: accounts" on page 526

RELATED TASKS
"Defining a class" on page 528
"Defining a class instance method" on page 533
"Defining a client" on page 541
"Defining a subclass" on page 553

"Defining a factory section" on page 557
Chapter 16, "Compiling, linking, and running OO applications," on page 287
Upgrading IBM COBOL source programs (*Enterprise COBOL Compiler and Runtime Migration Guide*)

RELATED REFERENCES
The Java Language Specification

Example: accounts

Consider the example of a bank in which customers can open accounts and make deposits to and withdrawals from their accounts. You could represent an account by a general-purpose class, called Account. Because there are many customers, multiple instances of the Account class could exist simultaneously.

After you determine the classes that you need, the next step is to determine the methods that the classes need to do their work. An Account class must provide the following services:
- Open the account.
- Get the current balance.
- Deposit to the account.
- Withdraw from the account.
- Report account status.

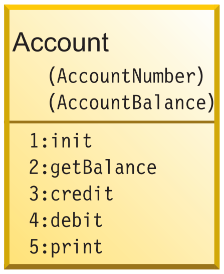
The following methods for an Account class meet those needs:

init	Open an account and assign it an account number.
getBalance	Return the current balance of the account.
credit	Deposit a given sum to the account.
debit	Withdraw a given sum from the account.
print	Display account number and account balance.

As you design an Account class and its methods, you discover the need for the class to keep some instance data. Typically, an Account object needs the following instance data:
- Account number
- Account balance
- Customer information: name, address, home phone, work phone, social security number, and so forth

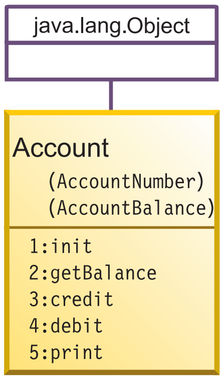
To keep the example simple, however, it is assumed that the account number and account balance are the only instance data that the Account class needs.

Diagrams are helpful when you design classes and methods. The following diagram depicts a first attempt at a design of the Account class:



The words in parentheses in the diagrams are the names of the instance data, and the words that follow a number and colon are the names of the instance methods.

The structure below shows how the classes relate to each other, and is known as the *inheritance hierarchy*. The Account class inherits directly from the class java.lang.Object.



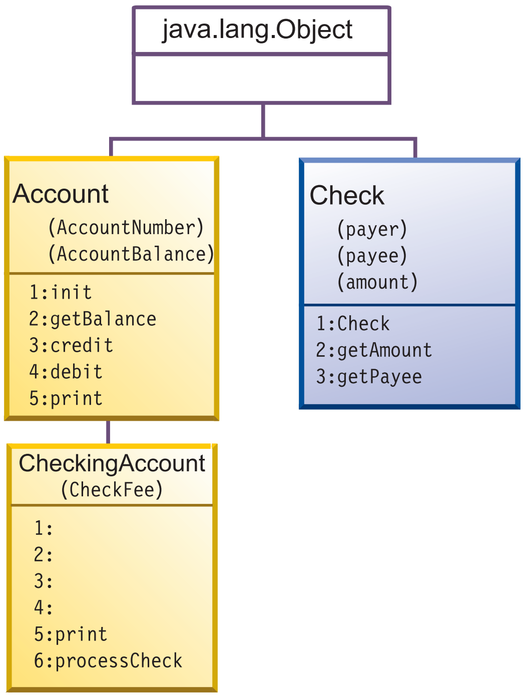
Subclasses

In the account example, Account is a general-purpose class. However, a bank could have many different types of accounts: checking accounts, savings accounts, mortgage loans, and so forth, all of which have all the general characteristics of accounts but could have additional characteristics not shared by all types of accounts.

For example, a CheckingAccount class could have, in addition to the account number and account balance that all accounts have, a check fee that applies to each check written on the account. A CheckingAccount class also needs a method to process checks (that is, to read the amount, debit the payer, credit the payee, and so forth). So it makes sense to define CheckingAccount as a subclass of Account, and to define in the subclass the additional instance data and instance methods that the subclass needs.

As you design the CheckingAccount class, you discover the need for a class that models checks. An instance of class Check needs, at a minimum, instance data for payer, payee, and the check amount.

Many additional classes (and database and transaction-processing logic) would need to be designed in a real-world OO account system, but have been omitted to keep the example simple. The updated inheritance diagram is shown below.



A number and colon with no method-name following them indicate that the method with that number is inherited from the superclass.

Multiple inheritance: You cannot use *multiple inheritance* in OO COBOL applications. All classes that you define must have exactly one parent, and java.lang.Object must be at the root of every inheritance hierarchy. The class structure of any object-oriented system defined in an OO COBOL application is thus a tree.

"Example: defining a method" on page 540

RELATED TASKS
"Defining a class"
"Defining a class instance method" on page 533
"Defining a subclass" on page 553

Defining a class

A COBOL class definition consists of an IDENTIFICATION DIVISION and ENVIRONMENT DIVISION, followed by an optional factory definition and optional object definition, followed by an END CLASS marker.

Table 73. **Structure of class definitions**

Section	Purpose	Syntax
IDENTIFICATION DIVISION (required)	Name the class. Provide inheritance information for it.	"CLASS-ID paragraph for defining a class" on page 530 (required) AUTHOR paragraph (optional) INSTALLATION paragraph (optional) DATE-WRITTEN paragraph (optional) DATE-COMPILED paragraph (optional)

Table 73. **Structure of class definitions** *(continued)*

Section	Purpose	Syntax
ENVIRONMENT DIVISION (required)	Describe the computing environment. Relate class-names used within the class definition to the corresponding external class-names known outside the compilation unit.	CONFIGURATION SECTION (required) "REPOSITORY paragraph for defining a class" on page 530 (required) SOURCE-COMPUTER paragraph (optional) OBJECT-COMPUTER paragraph (optional) SPECIAL-NAMES paragraph (optional)

Factory definition (optional)	Define data to be shared by all instances of the class, and methods supported independently of any object instance.	`IDENTIFICATION DIVISION.`
		`FACTORY.`
		`DATA DIVISION.`
		`WORKING-STORAGE SECTION.`
		`*   (Factory data here)`
		`PROCEDURE DIVISION.`
		`*   (Factory methods here)`
		`END FACTORY.`

Object definition (optional)	Define instance data and instance methods.	`IDENTIFICATION DIVISION.`
		`OBJECT.`
		`DATA DIVISION.`
		`WORKING-STORAGE SECTION.`
		`*   (Instance data here)`
		`PROCEDURE DIVISION.`
		`*   (Instance methods here)`
		`END OBJECT.`

If you specify the SOURCE-COMPUTER, OBJECT-COMPUTER, or SPECIAL-NAMES paragraphs in a class CONFIGURATION SECTION, they apply to the entire class definition including all methods that the class introduces.

A class CONFIGURATION SECTION can consist of the same entries as a program CONFIGURATION SECTION, except that a class CONFIGURATION SECTION cannot contain an INPUT-OUTPUT SECTION. You define an INPUT-OUTPUT SECTION only in the individual methods that require it rather than defining it at the class level.

As shown above, you define instance data and methods in the DATA DIVISION and PROCEDURE DIVISION, respectively, within the OBJECT paragraph of the class definition. In classes that require data and methods that are to be associated with the class itself rather than with individual object instances, define a separate DATA DIVISION and PROCEDURE DIVISION within the FACTORY paragraph of the class definition.

Each COBOL class definition must be in a separate source file.

"Example: defining a class" on page 533

RELATED TASKS
"WORKING-STORAGE SECTION for defining class instance data" on page 532
"Defining a class instance method" on page 533
"Defining a subclass" on page 553
"Defining a factory section" on page 557
"Describing the computing environment" on page 7
Chapter 16, "Compiling, linking, and running OO applications," on page 287

RELATED REFERENCES
COBOL class definition structure (*Enterprise COBOL Language Reference*)

CLASS-ID paragraph for defining a class

Use the CLASS-ID paragraph in the IDENTIFICATION DIVISION to name a class and provide inheritance information for it.

```
Identification Division.         Required
Class-id. Account inherits Base.  Required
```

Use the CLASS-ID paragraph to identify these classes:

- The class that you are defining (Account in the example above).
- The immediate superclass from which the class that you are defining inherits its characteristics. The superclass can be implemented in Java or COBOL.

  In the example above, inherits Base indicates that the Account class inherits methods and data from the class known within the class definition as Base. It is recommended that you use the name Base in your OO COBOL programs to refer to java.lang.Object.

A class-name must use single-byte characters and must conform to the normal rules of formation for a COBOL user-defined word.

Use the REPOSITORY paragraph in the CONFIGURATION SECTION of the ENVIRONMENT DIVISION to associate the superclass name (Base in the example) with the name of the superclass as it is known externally (java.lang.Object for Base). You can optionally also specify the name of the class that you are defining (Account in the example) in the REPOSITORY paragraph and associate it with its corresponding external class-name.

You must derive all classes directly or indirectly from the java.lang.Object class.

RELATED TASKS
"REPOSITORY paragraph for defining a class"

RELATED REFERENCES
CLASS-ID paragraph (*Enterprise COBOL Language Reference*)
User-defined words (*Enterprise COBOL Language Reference*)

REPOSITORY paragraph for defining a class

Use the REPOSITORY paragraph to declare to the compiler that the specified words are class-names when you use them within a class definition, and to optionally relate the class-names to the corresponding external class-names (the class-names as they are known outside the compilation unit).

External class-names are case sensitive and must conform to Java rules of formation. For example, in the Account class definition you might code this:

```
Environment Division.              Required
Configuration Section.             Required
Repository.                        Required
    Class Base is "java.lang.Object"  Required
    Class Account is "Account".       Optional
```

The REPOSITORY paragraph entries indicate that the external class-names of the classes referred to as Base and Account within the class definition are java.lang.Object and Account, respectively.

In the REPOSITORY paragraph, you must code an entry for each class-name that you explicitly reference in the class definition. For example:

- Base
- A superclass from which the class that you are defining inherits
- The classes that you reference in methods within the class definition

In a `REPOSITORY` paragraph entry, you must specify the external class-name if the name contains non-COBOL characters. You must also specify the external class-name for any referenced class that is part of a Java *package*. For such a class, specify the external class-name as the fully qualified name of the package, followed by period (`.`), followed by the simple name of the Java class. For example, the Object class is part of the java.lang package, so specify its external name as java.lang.Object as shown above.

An external class-name that you specify in the `REPOSITORY` paragraph must be an alphanumeric literal that conforms to the rules of formation for a fully qualified Java class-name.

If you do not include the external class-name in a `REPOSITORY` paragraph entry, the external class-name is formed from the class-name in the following manner:
- The class-name is converted to uppercase.
- Each hyphen is changed to zero.
- The first character, if a digit, is changed:
  - 1-9 are changed to A-I.
  - 0 is changed to J.

In the example above, class Account is known externally as Account (in mixed case) because the external name is spelled using mixed case.

You can optionally include in the `REPOSITORY` paragraph an entry for the class that you are defining (Account in this example). You must include an entry for the class that you are defining if the external class-name contains non-COBOL characters, or to specify a fully package-qualified class-name if the class is to be part of a Java package.

"Example: external class-names and Java packages"

RELATED TASKS
"Declaring arrays and strings for Java" on page 575

RELATED REFERENCES
REPOSITORY paragraph (*Enterprise COBOL Language Reference*)
Identifiers (*The Java Language Specification*)
Packages (*The Java Language Specification*)

Example: external class-names and Java packages

The following example shows how external class-names are determined from entries in a `REPOSITORY` paragraph.

```
Environment division.
Configuration section.
Repository.
    Class Employee is "com.acme.Employee"
    Class JavaException is "java.lang.Exception"
    Class Orders.
```

The local class-names (the class-names as used within the class definition), the Java packages that contain the classes, and the associated external class-names are as shown in the table below.

Local class-name	Java package	External class-name
Employee	com.acme	com.acme.Employee
JavaException	java.lang	java.lang.Exception
Orders	(unnamed)	ORDERS

The external class-name (the name after the class-name and optional IS in the REPOSITORY paragraph entry) is composed of the fully qualified name of the package (if any) followed by a period, followed by the simple name of the class.

RELATED TASKS
"REPOSITORY paragraph for defining a class" on page 530

RELATED REFERENCES
REPOSITORY paragraph (*Enterprise COBOL Language Reference*)

WORKING-STORAGE SECTION for defining class instance data

Use the WORKING-STORAGE SECTION in the DATA DIVISION of the OBJECT paragraph to describe the *instance data* that a COBOL class needs, that is, the data to be allocated for each instance of the class.

The OBJECT keyword, which you must immediately precede with an IDENTIFICATION DIVISION declaration, indicates the beginning of the definitions of the instance data and instance methods for the class. For example, the definition of the instance data for the Account class might look like this:

```
Identification division.
Object.
  Data division.
  Working-storage section.
  01 AccountNumber  pic 9(6).
  01 AccountBalance pic S9(9) value zero.
  . . .
End Object.
```

The instance data is allocated when an object instance is created, and exists until garbage collection of the instance by the Java run time.

You can initialize simple instance data by using VALUE clauses as shown above. You can initialize more complex instance data by coding customized methods to create and initialize instances of classes.

COBOL instance data is equivalent to Java private nonstatic member data. No other class or subclass (nor factory method in the same class, if any) can reference COBOL instance data directly. Instance data is global to all instance methods that the OBJECT paragraph defines. If you want to make instance data accessible from outside the OBJECT paragraph, define attribute (get or set) instance methods for doing so.

The syntax of the WORKING-STORAGE SECTION for instance data declaration is generally the same as in a program, with these exceptions:

- You cannot use the EXTERNAL attribute.
- You can use the GLOBAL attribute, but it has no effect.

RELATED TASKS
"Creating and initializing instances of classes" on page 550
"Freeing instances of classes" on page 552
"Defining a factory method" on page 558
"Coding attribute (get and set) methods" on page 539

Example: defining a class

The following example shows a first attempt at the definition of the Account class, excluding method definitions.

```
 cbl dll,thread,pgmname(longmixed)
 Identification Division.
 Class-id. Account inherits Base.
 Environment Division.
 Configuration section.
 Repository.
     Class Base    is "java.lang.Object"
     Class Account is "Account".
*
 Identification division.
 Object.
  Data division.
  Working-storage section.
  01 AccountNumber  pic 9(6).
  01 AccountBalance pic S9(9) value zero.
*
  Procedure Division.
*
*   (Instance method definitions here)
*
 End Object.
*
 End class Account.
```

RELATED TASKS
Chapter 16, "Compiling, linking, and running OO applications," on page 287
"Defining a client" on page 541

Defining a class instance method

Define COBOL *instance methods* in the PROCEDURE DIVISION of the OBJECT paragraph of a class definition. An instance method defines an operation that is supported for each object instance of a class.

A COBOL instance method definition consists of four divisions (like a COBOL program), followed by an END METHOD marker.

Table 74. **Structure of instance method definitions**

Division	Purpose	Syntax
IDENTIFICATION (required)	Name a method.	"METHOD-ID paragraph for defining a class instance method" on page 534 (required) AUTHOR paragraph (optional) INSTALLATION paragraph (optional) DATE-WRITTEN paragraph (optional) DATE-COMPILED paragraph (optional)

Table 74. **Structure of instance method definitions** *(continued)*

Division	Purpose	Syntax
ENVIRONMENT (optional)	Relate the file-names used in a method to the corresponding file-names known to the operating system.	"INPUT-OUTPUT SECTION for defining a class instance method" on page 535 (optional)
DATA (optional)	Define external files. Allocate a copy of the data.	"DATA DIVISION for defining a class instance method" on page 535 (optional)
PROCEDURE (optional)	Code the executable statements to complete the service provided by the method.	"PROCEDURE DIVISION for defining a class instance method" on page 536 (optional)

Definition: The *signature* of a method consists of the name of the method and the number and type of its formal parameters. (You define the formal parameters of a COBOL method in the USING phrase of the method's PROCEDURE DIVISION header.)

Within a class definition, you do not need to make each method-name unique, but you do need to give each method a unique signature. (You *overload* methods by giving them the same name but a different signature.)

COBOL instance methods are equivalent to Java public nonstatic methods.

"Example: defining a method" on page 540

RELATED TASKS
"PROCEDURE DIVISION for defining a class instance method" on page 536
"Overloading an instance method" on page 538
"Overriding an instance method" on page 537
"Invoking methods (INVOKE)" on page 546
"Defining a subclass instance method" on page 555
"Defining a factory method" on page 558

METHOD-ID paragraph for defining a class instance method

Use the METHOD-ID paragraph to name an instance method. Immediately precede the METHOD-ID paragraph with an IDENTIFICATION DIVISION declaration to indicate the beginning of the method definition.

For example, the definition of the credit method in the Account class begins like this:

```
Identification Division.
Method-id. "credit".
```

Code the method-name as an alphanumeric or national literal. The method-name is processed in a case-sensitive manner and must conform to the rules of formation for a Java method-name.

Other Java or COBOL methods or programs (that is, clients) use the method-name to invoke a method.

RELATED TASKS
"Invoking methods (INVOKE)" on page 546
"Using national data (Unicode) in COBOL" on page 126

RELATED REFERENCES
Meaning of method names (*The Java Language Specification*)
Identifiers (*The Java Language Specification*)
METHOD-ID paragraph (*Enterprise COBOL Language Reference*)

INPUT-OUTPUT SECTION for defining a class instance method

The ENVIRONMENT DIVISION of an instance method can have only one section, the INPUT-OUTPUT SECTION. This section relates the file-names used in a method definition to the corresponding file-names as they are known to the operating system.

For example, if the Account class defined a method that read information from a file, the Account class might have an INPUT-OUTPUT SECTION that is coded like this:

```
Environment Division.
Input-Output Section.
File-Control.
    Select account-file Assign AcctFile.
```

The syntax for the INPUT-OUTPUT SECTION of a method is the same as the syntax for the INPUT-OUTPUT SECTION of a program.

RELATED TASKS
"Describing the computing environment" on page 7

RELATED REFERENCES
INPUT-OUTPUT section (*Enterprise COBOL Language Reference*)

DATA DIVISION for defining a class instance method

The DATA DIVISION of an instance method consists of any of the following four sections: FILE SECTION, LOCAL-STORAGE SECTION, WORKING-STORAGE SECTION, and LINKAGE SECTION.

FILE SECTION
The same as a program FILE SECTION, except that a method FILE SECTION can define EXTERNAL files only.

LOCAL-STORAGE SECTION
A separate copy of the LOCAL-STORAGE data is allocated for each invocation of the method, and is freed on return from the method. The method LOCAL-STORAGE SECTION is similar to a program LOCAL-STORAGE SECTION.

If you specify the VALUE clause on a data item, the item is initialized to that value on each invocation of the method.

WORKING-STORAGE SECTION
A single copy of the WORKING-STORAGE data is allocated. The data persists in its last-used state until the run unit ends. The same copy of the data is used whenever the method is invoked, regardless of the invoking object or thread. The method WORKING-STORAGE SECTION is similar to a program WORKING-STORAGE SECTION.

If you specify the VALUE clause on a data item, the item is initialized to that value on the first invocation of the method. You can specify the EXTERNAL clause for the data items.

`LINKAGE SECTION`
The same as a program `LINKAGE SECTION`.

If you define a data item with the same name in both the `DATA DIVISION` of an instance method and the `DATA DIVISION` of the `OBJECT` paragraph, a reference in the method to that data-name refers only to the method data item. The method `DATA DIVISION` takes precedence.

RELATED TASKS
"Describing the data" on page 12
"Sharing data by using the EXTERNAL clause" on page 461

RELATED REFERENCES
DATA DIVISION overview (*Enterprise COBOL Language Reference*)

PROCEDURE DIVISION for defining a class instance method

Code the executable statements to implement the service that an instance method provides in the `PROCEDURE DIVISION` of the instance method.

You can code most COBOL statements in the `PROCEDURE DIVISION` of a method that you can code in the `PROCEDURE DIVISION` of a program. You cannot, however, code the following statements in a method:

- `ENTRY`
- `EXIT PROGRAM`
- The following obsolete elements of Standard COBOL 85:
  - `ALTER`
  - `GOTO` without a specified procedure-name
  - `SEGMENT-LIMIT`
  - `USE FOR DEBUGGING`

Additionally, because you must compile all COBOL class definitions with the `THREAD` compiler option, you cannot use `SORT` or `MERGE` statements in a COBOL method.

You can code the `EXIT METHOD` or `GOBACK` statement in an instance method to return control to the invoking client. Both statements have the same effect. If you specify the `RETURNING` phrase upon invocation of the method, the `EXIT METHOD` or `GOBACK` statement returns the value of the data item to the invoking client.

An implicit `EXIT METHOD` is generated as the last statement in the `PROCEDURE DIVISION` of each method.

You can specify `STOP RUN` in a method; doing so terminates the entire run unit including all threads executing within it.

You must terminate a method definition with an `END METHOD` marker. For example, the following statement marks the end of the `credit` method:

```
End method "credit".
```

USING phrase for obtaining passed arguments: Specify the formal parameters to a method, if any, in the `USING` phrase of the method's `PROCEDURE DIVISION` header. You must specify that the arguments are passed `BY VALUE`. Define each parameter as a level-01 or level-77 item in the method's `LINKAGE SECTION`. The data type of each parameter must be one of the types that are interoperable with Java.

RETURNING phrase for returning a value: Specify the data item to be returned as the method result, if any, in the `RETURNING` phrase of the method's `PROCEDURE DIVISION` header. Define the data item as a level-01 or level-77 item in the method's `LINKAGE SECTION`. The data type of the return value must be one of the types that are interoperable with Java.

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 574
"Overriding an instance method"
"Overloading an instance method" on page 538
"Comparing and setting object references" on page 545
"Invoking methods (INVOKE)" on page 546
Chapter 16, "Compiling, linking, and running OO applications," on page 287

RELATED REFERENCES
"THREAD" on page 342
The procedure division header (*Enterprise COBOL Language Reference*)

Overriding an instance method

An instance method that is defined in a subclass is said to *override* an inherited instance method that would otherwise be accessible in the subclass if the two methods have the same signature.

To override a superclass instance method m1 in a COBOL subclass, define an instance method m1 in the subclass that has the same name and whose `PROCEDURE DIVISION USING` phrase (if any) has the same number and type of formal parameters as the superclass method has. (If the superclass method is implemented in Java, you must code formal parameters that are interoperable with the data types of the corresponding Java parameters.) When a client invokes m1 on an instance of the subclass, the subclass method rather than the superclass method is invoked.

For example, the Account class defines a method `debit` whose `LINKAGE SECTION` and `PROCEDURE DIVISION` header look like this:

```
Linkage section.
01 inDebit    pic S9(9) binary.
Procedure Division using by value inDebit.
```

If you define a CheckingAccount subclass and want it to have a `debit` method that overrides the `debit` method defined in the Account superclass, define the subclass method with exactly one input parameter also specified as `pic S9(9) binary`. If a client invokes `debit` using an object reference to a CheckingAccount instance, the CheckingAccount `debit` method (rather than the `debit` method in the Account superclass) is invoked.

The presence or absence of a method return value and the data type of the return value used in the `PROCEDURE DIVISION RETURNING` phrase (if any) must be identical in the subclass instance method and the overridden superclass instance method.

An instance method must not override a factory method in a COBOL superclass nor a static method in a Java superclass.

"Example: defining a method" on page 540

RELATED TASKS
"PROCEDURE DIVISION for defining a class instance method" on page 536

"Coding interoperable data types in COBOL and Java" on page 574
"Invoking methods (INVOKE)" on page 546
"Invoking overridden superclass methods" on page 549
"Defining a subclass" on page 553
"Hiding a factory or static method" on page 559

RELATED REFERENCES
Inheritance, overriding, and hiding (*The Java Language Specification*)

Overloading an instance method

Two methods that are supported in a class (whether defined in the class or inherited from a superclass) are said to be *overloaded* if they have the same name but different signatures.

You overload methods when you want to enable clients to invoke different versions of a method, for example, to initialize data using different sets of parameters.

To overload a method, define a method whose PROCEDURE DIVISION USING phrase (if any) has a different number or type of formal parameters than an identically named method that is supported in the same class. For example, the Account class defines an instance method init that has exactly one formal parameter. The LINKAGE SECTION and PROCEDURE DIVISION header of the init method look like this:

```
Linkage section.
01 inAccountNumber pic S9(9) binary.
Procedure Division using by value inAccountNumber.
```

Clients invoke this method to initialize an Account instance with a given account number (and a default account balance of zero) by passing exactly one argument that matches the data type of inAccountNumber.

But the Account class could define, for example, a second instance method init that has an additional formal parameter that allows the opening account balance to also be specified. The LINKAGE SECTION and PROCEDURE DIVISION header of this init method could look like this:

```
Linkage section.
01 inAccountNumber pic S9(9) binary.
01 inBalance       pic S9(9) binary.
Procedure Division using by value inAccountNumber
                                  inBalance.
```

Clients could invoke either init method by passing arguments that match the signature of the desired method.

The presence or absence of a method return value does not have to be consistent in overloaded methods, and the data type of the return value given in the PROCEDURE DIVISION RETURNING phrase (if any) does not have to be identical in overloaded methods.

You can overload factory methods in exactly the same way that you overload instance methods.

The rules for overloaded method definition and resolution of overloaded method invocations are based on the corresponding rules for Java.

RELATED TASKS
"Invoking methods (INVOKE)" on page 546
"Defining a factory method" on page 558

RELATED REFERENCES
Overloading (*The Java Language Specification*)

Coding attribute (get and set) methods

You can provide access to an instance variable `X` from outside the class in which `X` is defined by coding accessor (get) and mutator (set) methods for `X`.

Instance variables in COBOL are *private*. The class that defines instance variables fully encapsulates them, and only the instance methods defined in the same `OBJECT` paragraph can access them directly. Normally a well-designed object-oriented application does not need to access instance variables from outside the class.

COBOL does not directly support the concept of a *public* instance variable as defined in Java and other object-oriented languages, nor the concept of a class attribute as defined by CORBA. (A CORBA *attribute* is an instance variable that has an automatically generated get method for accessing the value of the variable, and an automatically generated set method for modifying the value of the variable if the variable is not read-only.)

"Example: coding a get method"

RELATED TASKS
"WORKING-STORAGE SECTION for defining class instance data" on page 532
"Processing the data" on page 19

Example: coding a get method

The following example shows the definition in the Account class of an instance method, getBalance, to return the value of the instance variable AccountBalance to a client. getBalance and AccountBalance are defined in the `OBJECT` paragraph of the Account class definition.

```
 Identification Division.
 Class-id. Account inherits Base.
*  (ENVIRONMENT DIVISION not shown)
*  (FACTORY paragraph not shown)
*
 Identification division.
 Object.
  Data division.
  Working-storage section.
  01 AccountBalance pic S9(9) value zero.
*   (Other instance data not shown)
*
  Procedure Division.
*
   Identification Division.
   Method-id. "getBalance".
   Data division.
   Linkage section.
   01 outBalance pic S9(9) binary.
*
   Procedure Division returning outBalance.
     Move AccountBalance to outBalance.
   End method "getBalance".
*
```

```
*  (Other instance methods not shown)
 End Object.
*
 End class Account.
```

Example: defining a method

The following example adds to the previous example the instance method definitions of the Account class, and shows the definition of the Java Check class.

(The previous example was "Example: defining a class" on page 533.)

Account class

```
 cbl dll,thread,pgmname(longmixed)
 Identification Division.
 Class-id. Account inherits Base.
 Environment Division.
 Configuration section.
 Repository.
     Class Base    is "java.lang.Object"
     Class Account is "Account".
*
*  (FACTORY paragraph not shown)
*
 Identification division.
 Object.
  Data division.
  Working-storage section.
  01 AccountNumber  pic 9(6).
  01 AccountBalance pic S9(9) value zero.
*
  Procedure Division.
*
*    init method to initialize the account:
   Identification Division.
   Method-id. "init".
   Data division.
   Linkage section.
   01 inAccountNumber pic S9(9) binary.
   Procedure Division using by value inAccountNumber.
     Move inAccountNumber to AccountNumber.
   End method "init".
*
*    getBalance method to return the account balance:
   Identification Division.
   Method-id. "getBalance".
   Data division.
   Linkage section.
   01 outBalance pic S9(9) binary.
   Procedure Division returning outBalance.
     Move AccountBalance to outBalance.
   End method "getBalance".
*
*    credit method to deposit to the account:
   Identification Division.
   Method-id. "credit".
   Data division.
   Linkage section.
   01 inCredit   pic S9(9) binary.
   Procedure Division using by value inCredit.
     Add inCredit to AccountBalance.
   End method "credit".
*
*    debit method to withdraw from the account:
   Identification Division.
   Method-id. "debit".
   Data division.
   Linkage section.
   01 inDebit    pic S9(9) binary.
   Procedure Division using by value inDebit.
     Subtract inDebit from AccountBalance.
   End method "debit".
*
*    print method to display formatted account number and balance:
   Identification Division.
   Method-id. "print".
   Data division.
   Local-storage section.
   01 PrintableAccountNumber  pic ZZZZZZ999999.
   01 PrintableAccountBalance pic $$$$,$$$,$$9CR.
   Procedure Division.
     Move AccountNumber  to PrintableAccountNumber
     Move AccountBalance to PrintableAccountBalance
     Display " Account: " PrintableAccountNumber
     Display " Balance: " PrintableAccountBalance.
   End method "print".
*
 End Object.
*
 End class Account.
```

Check class

```
/**
 * A Java class for check information
 */
public class Check {
  private CheckingAccount payer;
  private Account         payee;
  private int             amount;

  public Check(CheckingAccount inPayer, Account inPayee, int inAmount) {
    payer=inPayer;
    payee=inPayee;
    amount=inAmount;
  }

  public int getAmount() {
    return amount;
  }

  public Account getPayee() {
    return payee;
  }
}
```

RELATED TASKS
Chapter 16, "Compiling, linking, and running OO applications," on page 287

Defining a client

A program or method that requests services from one or more methods in a class is called a *client* of that class.

In a COBOL or Java client, you can:

- Create object instances of Java and COBOL classes.
- Invoke instance methods on Java and COBOL objects.
- Invoke COBOL factory methods and Java static methods.

In a COBOL client, you can also call services provided by the Java Native Interface (JNI).

A COBOL client program consists of the usual four divisions:

Table 75. **Structure of COBOL clients**

Division	Purpose	Syntax
IDENTIFICATION (required)	Name a client.	Code as usual, except that a client program must be: • Recursive (declared RECURSIVE in the PROGRAM-ID paragraph) • Thread-enabled (compiled with the THREAD option, and conforming to the coding guidelines for threaded applications)
ENVIRONMENT (required)	Describe the computing environment. Relate class-names used in the client to the corresponding external class-names known outside the compilation unit.	CONFIGURATION SECTION (required) "REPOSITORY paragraph for defining a client" on page 543 (required)
DATA (optional)	Describe the data that the client needs.	"DATA DIVISION for defining a client" on page 543 (optional)
PROCEDURE (optional)	Create instances of classes, manipulate object reference data items, and invoke methods.	Code using INVOKE, IF, and SET statements.

Because you must compile all COBOL programs that contain object-oriented syntax or that interoperate with Java with the THREAD compiler option, you cannot use the following language elements in a COBOL client:

- SORT or MERGE statements
- Nested programs

Any programs that you compile with the THREAD compiler option must be recursive. You must specify the RECURSIVE clause in the PROGRAM-ID paragraph of each OO COBOL client program.

"Example: defining a client" on page 552

RELATED TASKS
Chapter 16, "Compiling, linking, and running OO applications," on page 287
Chapter 27, "Preparing COBOL programs for multithreading," on page 477
Chapter 31, "Communicating with Java methods," on page 569
"Coding interoperable data types in COBOL and Java" on page 574
"Creating and initializing instances of classes" on page 550
"Comparing and setting object references" on page 545
"Invoking methods (INVOKE)" on page 546
"Invoking factory or static methods" on page 560

RELATED REFERENCES
"THREAD" on page 342

REPOSITORY paragraph for defining a client

Use the REPOSITORY paragraph to declare to the compiler that the specified words are class-names when you use them in a COBOL client, and to optionally relate the class-names to the corresponding external class-names (the class-names as they are known outside the compilation unit).

External class-names are case sensitive, and must conform to Java rules of formation. For example, in a client program that uses the Account and Check classes you might code this:

```
Environment division.      Required
Configuration section.     Required
  Source-Computer.  IBM-390.
  Object-Computer.  IBM-390.
Repository.                Required
    Class Account is "Account"
    Class Check   is "Check".
```

The REPOSITORY paragraph entries indicate that the external class-names of the classes referred to as Account and Check within the client are Account and Check, respectively.

In the REPOSITORY paragraph, you must code an entry for each class-name that you explicitly reference in the client. In a REPOSITORY paragraph entry, you must specify the external class-name if the name contains non-COBOL characters.

You must specify the external class-name for any referenced class that is part of a Java package. For such a class, specify the external class-name as the fully qualified name of the package, followed by period (.), followed by the simple name of the Java class.

An external class-name that you specify in the REPOSITORY paragraph must be an alphanumeric literal that conforms to the rules of formation for a fully qualified Java class-name.

If you do not include the external class-name in a REPOSITORY paragraph entry, the external class-name is formed from the class-name in the same manner as it is when an external class-name is not included in a REPOSITORY paragraph entry in a class definition. In the example above, class Account and class Check are known externally as Account and Check (in mixed case), respectively, because the external names are spelled using mixed case.

The SOURCE-COMPUTER, OBJECT-COMPUTER, and SPECIAL-NAMES paragraphs of the CONFIGURATION SECTION are optional.

RELATED TASKS
"REPOSITORY paragraph for defining a class" on page 530

RELATED REFERENCES
REPOSITORY paragraph (*Enterprise COBOL Language Reference*)
Identifiers (*The Java Language Specification*)
Packages (*The Java Language Specification*)

DATA DIVISION for defining a client

You can use any of the sections of the DATA DIVISION to describe the data that the client needs.

```
Data Division.
Local-storage section.
01  anAccount        usage object reference Account.
01  aCheckingAccount usage object reference CheckingAccount.
01  aCheck           usage object reference Check.
01  payee            usage object reference Account.
. . .
```

Because a client references classes, it needs one or more special data items called *object references*, that is, references to instances of those classes. All requests to instance methods require an object reference to an instance of a class in which the method is supported (that is, either defined or available by inheritance). You code object references to refer to instances of Java classes using the same syntax as you use to refer to instances of COBOL classes. In the example above, the phrase `usage object reference` indicates an object reference data item.

All four object references in the code above are called *typed* object references because a class-name appears after the `OBJECT REFERENCE` phrase. A typed object reference can refer only to an instance of the class named in the `OBJECT REFERENCE` phrase or to one of its subclasses. Thus anAccount can refer to instances of the Account class or one of its subclasses, but cannot refer to instances of any other class. Similarly, aCheck can refer only to instances of the Check class or any subclasses that it might have.

Another type of object reference, not shown above, does not have a class-name after the `OBJECT REFERENCE` phrase. Such a reference is called a *universal* object reference, which means that it can refer to instances of any class. Avoid coding universal object references, because they are interoperable with Java in only very limited circumstances (when used in the `RETURNING` phrase of the `INVOKE` *`class-name`* `NEW . . .` statement).

You must define, in the `REPOSITORY` paragraph of the `CONFIGURATION SECTION`, class-names that you use in the `OBJECT REFERENCE` phrase.

RELATED TASKS
"Choosing LOCAL-STORAGE or WORKING-STORAGE"
"Coding interoperable data types in COBOL and Java" on page 574
"Invoking methods (INVOKE)" on page 546
"REPOSITORY paragraph for defining a client" on page 543

RELATED REFERENCES
RETURNING phrase (*Enterprise COBOL Language Reference*)

Choosing LOCAL-STORAGE or WORKING-STORAGE

You can in general use the `WORKING-STORAGE SECTION` to define working data that a client program needs. However, if the program could simultaneously run on multiple threads, you might instead want to define the data in the `LOCAL-STORAGE SECTION`.

Each thread has access to a separate copy of `LOCAL-STORAGE` data but shares access to a single copy of `WORKING-STORAGE` data. If you define the data in the `WORKING-STORAGE SECTION`, you need to synchronize access to the data or ensure that no two threads can access it simultaneously.

RELATED TASKS
Chapter 27, "Preparing COBOL programs for multithreading," on page 477

Comparing and setting object references

You can compare object references by coding conditional statements or a call to the JNI service IsSameObject, and you can set object references by using the SET statement.

For example, code either IF statement below to check whether the object reference anAccount refers to no object instance:

```
If anAccount = Null . . .
If anAccount = Nulls . . .
```

You can code a call to IsSameObject to check whether two object references, object1 and object2, refer to the same object instance or whether each refers to no object instance. To ensure that the arguments and return value are interoperable with Java and to establish addressability to the callable service, code the following data definitions and statements before the call to IsSameObject:

```
Local-storage Section.
. . .
01  is-same Pic X.
    88 is-same-false Value X'00'.
    88 is-same-true  Value X'01' Through X'FF'.
Linkage Section.
    Copy JNI.
Procedure Division.
    Set Address Of JNIEnv To JNIEnvPtr
    Set Address Of JNINativeInterface To JNIEnv
    Call IsSameObject Using By Value JNIEnvPtr object1 object2
                      Returning is-same
    If is-same-true . . .
```

Within a method you can check whether an object reference refers to the object instance on which the method was invoked by coding a call to IsSameObject that compares the object reference and SELF.

You can instead invoke the Java `equals` method (inherited from java.lang.Object) to determine whether two object references refer to the same object instance.

You can make an object reference refer to no object instance by using the SET statement. For example:

```
Set anAccount To Null.
```

You can also make one object reference refer to the same instance as another object reference does by using the SET statement. For example:

```
Set anotherAccount To anAccount.
```

This SET statement causes anotherAccount to refer to the same object instance as anAccount does. If the receiver (anotherAccount) is a universal object reference, the sender (anAccount) can be either a universal or a typed object reference. If the receiver is a typed object reference, the sender must be a typed object reference bound to the same class as the receiver or to one of its subclasses.

Within a method you can make an object reference refer to the object instance on which the method was invoked by setting it to SELF. For example:

```
Set anAccount To Self.
```

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 574
"Accessing JNI services" on page 569

RELATED REFERENCES
IsSameObject (*The Java Native Interface*)

Invoking methods (INVOKE)

In a Java client, you can create object instances of classes that were implemented in COBOL and invoke methods on those objects using standard Java syntax. In a COBOL client, you can invoke methods that are defined in Java or COBOL classes by coding the INVOKE statement.

```
Invoke Account "createAccount"
   using by value 123456
   returning anAccount
Invoke anAccount "credit" using by value 500.
```

The first example INVOKE statement above uses the class-name Account to invoke a method called createAccount. This method must be either defined or inherited in the Account class, and must be one of the following types:

- A Java static method
- A COBOL factory method

The phrase using by value 123456 indicates that 123456 is an input argument to the method, and is passed by value. The input argument 123456 and the returned data item anAccount must conform to the definition of the formal parameters and return type, respectively, of the (possibly overloaded) createAccount method.

The second INVOKE statement uses the returned object reference anAccount to invoke the instance method credit, which is defined in the Account class. The input argument 500 must conform to the definition of the formal parameters of the (possibly overloaded) credit method.

Code the name of the method to be invoked either as a literal or as an identifier whose value at run time matches the method-name in the signature of the target
| method. The method-name must be an alphanumeric or national literal or a
| category alphabetic, alphanumeric, or national data item, and is interpreted in a case-sensitive manner.

When you code an INVOKE statement using an object reference (as in the second example statement above), the statement begins with one of the following two forms:

```
Invoke objRef "literal-name"  . . .
Invoke objRef identifier-name . . .
```

When the method-name is an identifier, you must define the object reference (objRef) as USAGE OBJECT REFERENCE with no specified type, that is, as a universal object reference.

If an invoked method is not supported in the class to which the object reference refers, a severity-3 Language Environment condition is raised at run time unless you code the ON EXCEPTION phrase in the INVOKE statement.

You can use the optional scope terminator END-INVOKE with the INVOKE statement.

The INVOKE statement does not set the RETURN-CODE special register.

RELATED TASKS
"USING phrase for passing arguments" on page 547

“RETURNING phrase for obtaining a returned value” on page 549
“PROCEDURE DIVISION for defining a class instance method” on page 536
“Coding interoperable data types in COBOL and Java” on page 574
“Invoking overridden superclass methods” on page 549
“Invoking factory or static methods” on page 560

RELATED REFERENCES
INVOKE statement (*Enterprise COBOL Language Reference*)

USING phrase for passing arguments

If you pass arguments to a method, specify the arguments in the USING phrase of the INVOKE statement. Code the data type of each argument so that it conforms to the type of the corresponding formal parameter in the intended target method.

Table 76. **Conformance of arguments in a COBOL client**

Programming language of the target method	Is the argument an object reference?	Then code the DATA DIVISION definition of the argument as:	Restriction
COBOL	No	The same as the definition of the corresponding formal parameter	
Java	No	Interoperable with the corresponding Java parameter	
COBOL or Java	Yes	An object reference that is typed to the same class as the corresponding parameter in the target method	In a COBOL client (unlike in a Java client), the class of an argument cannot be a subclass of the class of the corresponding parameter.

See the example referenced below for a way to make an object-reference argument conform to the type of a corresponding formal parameter by using the SET statement or the REDEFINES clause.

“Example: passing conforming object-reference arguments from a COBOL client” on page 548

If the target method is overloaded, the data types of the arguments are used to select from among the methods that have the same name.

You must specify that the arguments are passed BY VALUE. In other words, the arguments are not affected by any change to the corresponding formal parameters in the invoked method.

The data type of each argument must be one of the types that are interoperable with Java.

RELATED TASKS
“PROCEDURE DIVISION for defining a class instance method” on page 536
“Overloading an instance method” on page 538
“Coding interoperable data types in COBOL and Java” on page 574
“Passing data” on page 451

RELATED REFERENCES
INVOKE statement (*Enterprise COBOL Language Reference*)
SET statement (*Enterprise COBOL Language Reference*)
REDEFINES clause (*Enterprise COBOL Language Reference*)

Example: passing conforming object-reference arguments from a COBOL client

The following example shows a way to make an object-reference argument in a COBOL client conform to the expected class of the corresponding formal parameter in an invoked method.

Class C defines a method M that has one parameter, a reference to an object of class java.lang.Object:

```
 . . .
 Class-id. C inherits Base.
 . . .
 Repository.
     Class Base       is "java.lang.Object"
     Class JavaObject is "java.lang.Object".
 Identification division.
 Factory.
 . . .
  Procedure Division.
   Identification Division.
   Method-id. "M".
   Data division.
   Linkage section.
   01 obj object reference JavaObject.
   Procedure Division using by value obj.
   . . .
```

To invoke method M, a COBOL client must pass an argument that is a reference to an object of class java.lang.Object. The client below defines a data item aString, which cannot be passed as an argument to M because aString is a reference to an object of class java.lang.String. The client first uses a SET statement to assign aString to a data item, anObj, that is a reference to an object of class java.lang.Object. (This SET statement is legal because java.lang.String is a subclass of java.lang.Object.) The client then passes anObj as the argument to M.

```
 . . .
 Repository.
     Class jstring    is "java.lang.String"
     Class JavaObject is "java.lang.Object".
 Data division.
 Local-storage section.
 01  aString object reference jstring.
 01  anObj   object reference JavaObject.
*
 Procedure division.
     . . . (statements here assign a value to aString)
     Set anObj to aString
     Invoke C "M"
       using by value anObj
```

Instead of using a SET statement to obtain anObj as a reference to an object of class java.lang.Object, the client could define aString and anObj with the REDEFINES clause as follows:

```
 . . .
 01  aString object reference jstring.
 01  anObj   redefines aString object reference JavaObject.
```

| After the client assigns a value to data item aString (that is, a valid reference to an
| object of class java.lang.String), anObj can be passed as the argument to M. For an
| example of the use of the REDEFINES clause to obtain argument conformance, see
| the example referenced below.

"Example: J2EE client written in COBOL" on page 580

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 574
"PROCEDURE DIVISION for defining a class instance method" on page 536

RELATED REFERENCES
INVOKE statement (*Enterprise COBOL Language Reference*)
SET statement (*Enterprise COBOL Language Reference*)
REDEFINES clause (*Enterprise COBOL Language Reference*)

RETURNING phrase for obtaining a returned value

If a data item is to be returned as the method result, specify the item in the `RETURNING` phrase of the `INVOKE` statement. Define the returned item in the `DATA DIVISION` of the client.

The item that you specify in the `RETURNING` phrase of the `INVOKE` statement must conform to the type returned by the target method, as shown in the table below.

Table 77. **Conformance of the returned data item in a COBOL client**

Programming language of the target method	Is the returned item an object reference?	Then code the `DATA DIVISION` definition of the returned item as:
COBOL	No	The same as the definition of the `RETURNING` item in the target method
Java	No	Interoperable with the returned Java data item
COBOL or Java	Yes	An object reference that is typed to the same class as the object reference that is returned by the target method

In all cases, the data type of the returned value must be one of the types that are interoperable with Java.

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 574

RELATED REFERENCES
INVOKE statement (*Enterprise COBOL Language Reference*)

Invoking overridden superclass methods

Sometimes within a class you need to invoke an overridden superclass method instead of invoking a method that has the same signature and is defined in the current class.

For example, suppose that the CheckingAccount class overrides the `debit` instance method defined in its immediate superclass, Account. You could invoke the Account `debit` method within a method in the CheckingAccount class by coding this statement:

```
Invoke Super "debit" Using By Value amount.
```

You would define `amount` as `PIC S9(9) BINARY` to match the signature of the `debit` methods.

The CheckingAccount class overrides the `print` method that is defined in the Account class. Because the `print` method has no formal parameters, a method in the CheckingAccount class could invoke the superclass `print` method with this statement:

```
Invoke Super "print".
```

The keyword `SUPER` indicates that you want to invoke a superclass method rather than a method in the current class. (`SUPER` is an implicit reference to the object used in the invocation of the currently executing method.)

"Example: accounts" on page 526

RELATED TASKS
"Overriding an instance method" on page 537

RELATED REFERENCES
INVOKE statement (*Enterprise COBOL Language Reference*)

Creating and initializing instances of classes

Before you can use the instance methods that are defined in a Java or COBOL class, you must first create an instance of the class.

To create a new instance of class *class-name* and to obtain a reference *object-reference* to the created object, code a statement of the following form, where *object-reference* is defined in the `DATA DIVISION` of the client:

```
INVOKE class-name NEW . . . RETURNING object-reference
```

When you code the `INVOKE . . . NEW` statement within a method, and the use of the returned object reference is not limited to the duration of the method invocation, you must convert the returned object reference to a global reference by calling the JNI service NewGlobalRef:

```
Call NewGlobalRef using by value JNIEnvPtr object-reference
                  returning object-reference
```

If you do not call NewGlobalRef, the returned object reference is only a local reference, which means that it is automatically freed after the method returns.

RELATED TASKS
"Instantiating Java classes"
"Instantiating COBOL classes" on page 551
"Accessing JNI services" on page 569
"Managing local and global references" on page 572
"DATA DIVISION for defining a client" on page 543
"Invoking methods (INVOKE)" on page 546
"Coding interoperable data types in COBOL and Java" on page 574

RELATED REFERENCES
INVOKE statement (*Enterprise COBOL Language Reference*)

Instantiating Java classes

To instantiate a Java class, invoke any parameterized constructor that the class supports by coding the `USING` phrase in the `INVOKE . . . NEW` statement immediately before the `RETURNING` phrase, passing `BY VALUE` the number and types of arguments that match the signature of the constructor.

The data type of each argument must be one of the types that are interoperable with Java. To invoke the default (parameterless) constructor, omit the `USING` phrase.

For example, to create an instance of the Check class, initialize its instance data, and obtain reference aCheck to the Check instance created, you could code this statement in a COBOL client:

```
Invoke Check New
  using by value aCheckingAccount, payee, 125
  returning aCheck
```

RELATED TASKS
"Invoking methods (INVOKE)" on page 546
"Coding interoperable data types in COBOL and Java" on page 574

RELATED REFERENCES
VALUE clause (*Enterprise COBOL Language Reference*)
INVOKE statement (*Enterprise COBOL Language Reference*)

Instantiating COBOL classes

To instantiate a COBOL class, you can specify either a typed or universal object reference in the `RETURNING` phrase of the `INVOKE . . . NEW` statement. However, you cannot code the `USING` phrase: the instance data is initialized as specified in the `VALUE` clauses in the class definition.

Thus the `INVOKE . . . NEW` statement is useful for instantiating COBOL classes that have only simple instance data. For example, the following statement creates an instance of the Account class, initializes the instance data as specified in `VALUE` clauses in the `WORKING-STORAGE SECTION` of the `OBJECT` paragraph of the Account class definition, and provides reference outAccount to the new instance:

```
Invoke Account New returning outAccount
```

To make it possible to initialize COBOL instance data that cannot be initialized using `VALUE` clauses alone, when designing a COBOL class you must define a parameterized creation method in the `FACTORY` paragraph and a parameterized initialization method in the `OBJECT` paragraph:

1. In the parameterized factory creation method, do these steps:
   a. Code `INVOKE` *class-name* `NEW RETURNING` *objectRef* to create an instance of *class-name* and to give initial values to the instance data items that have `VALUE` clauses.
   b. Invoke the parameterized initialization method on the instance (*objectRef*), passing `BY VALUE` the arguments that were supplied to the factory method.
2. In the initialization method, code logic to complete the instance data initialization using the values supplied through the formal parameters.

To create an instance of the COBOL class and properly initialize it, the client invokes the parameterized factory method, passing `BY VALUE` the desired arguments. The object reference returned to the client is a local reference. If the client code is within a method, and the use of the returned object reference is not limited to the duration of that method, the client code must convert the returned object reference to a global reference by calling the JNI service NewGlobalRef.

"Example: defining a factory (with methods)" on page 560

RELATED TASKS
"Accessing JNI services" on page 569
"Managing local and global references" on page 572
"Invoking methods (INVOKE)" on page 546
"Defining a factory section" on page 557

RELATED REFERENCES
VALUE clause (*Enterprise COBOL Language Reference*)
INVOKE statement (*Enterprise COBOL Language Reference*)

Freeing instances of classes

You do not need to take any action to free individual object instances of any class. No syntax is available for doing so. The Java runtime system automatically performs *garbage collection*, that is, it reclaims the memory for objects that are no longer in use.

There could be times, however, when you need to explicitly free local or global references to objects within a native COBOL client in order to permit garbage collection of the referenced objects to occur.

RELATED TASKS
"Managing local and global references" on page 572

Example: defining a client

The following example shows a small client program of the Account class.

The program does this:

- Invokes a factory method createAccount to create an Account instance with a default balance of zero
- Invokes the instance method `credit` to deposit $500 to the new account
- Invokes the instance method `print` to display the account status

(The Account class was shown in "Example: defining a method" on page 540.)

```
 cbl dll,thread,pgmname(longmixed)
 Identification division.
 Program-id. "TestAccounts" recursive.
 Environment division.
 Configuration section.
 Repository.
     Class Account is "Account".
 Data Division.
*  Working data is declared in LOCAL-STORAGE instead of
*  WORKING-STORAGE so that each thread has its own copy:
 Local-storage section.
 01  anAccount usage object reference Account.
*
 Procedure division.
 Test-Account-section.
     Display "Test Account class"
*  Create account 123456 with 0 balance:
     Invoke Account "createAccount"
       using by value 123456
       returning anAccount
*  Deposit 500 to the account:
     Invoke anAccount "credit" using by value 500
     Invoke anAccount "print"
     Display space
*
     Stop Run.
 End program "TestAccounts".
```

"Example: defining a factory (with methods)" on page 560

RELATED TASKS
"Defining a factory method" on page 558
"Invoking factory or static methods" on page 560
Chapter 16, "Compiling, linking, and running OO applications," on page 287

Defining a subclass

You can make a class (called a *subclass*, derived class, or child class) a specialization of another class (called a *superclass*, base class, or parent class).

A subclass inherits the methods and instance data of its superclasses, and is related to its superclasses by an *is-a* relationship. For example, if subclass P inherits from superclass Q, and subclass Q inherits from superclass S, then an instance of P is an instance of Q and also (by transitivity) an instance of S. An instance of P inherits the methods and data of Q and S.

Using subclasses has several advantages:
- Reuse of code: Through inheritance, a subclass can reuse methods that already exist in a superclass.
- Specialization: In a subclass you can add new methods to handle cases that the superclass does not handle. You can also add new data items that the superclass does not need.
- Change in action: A subclass can override a method that it inherits from a superclass by defining a method of the same signature as that in the superclass. When you override a method, you might make only a few minor changes or completely change what the method does.

Restriction: You cannot use *multiple inheritance* in your COBOL programs. Each COBOL class that you define must have exactly one immediate superclass that is implemented in Java or COBOL, and each class must be derived directly or indirectly from java.lang.Object. The semantics of inheritance are as defined by Java.

The structure and syntax of a subclass definition are identical to those of a class definition: Define instance data and methods in the `DATA DIVISION` and `PROCEDURE DIVISION`, respectively, within the `OBJECT` paragraph of the subclass definition. In subclasses that require data and methods that are to be associated with the subclass itself rather than with individual object instances, define a separate `DATA DIVISION` and `PROCEDURE DIVISION` within the `FACTORY` paragraph of the subclass definition.

COBOL instance data is private. A subclass can access the instance data of a COBOL superclass only if the superclass defines attribute (get or set) instance methods for doing so.

"Example: accounts" on page 526
"Example: defining a subclass (with methods)" on page 555

RELATED TASKS
"Defining a class" on page 528
"Overriding an instance method" on page 537
"Coding attribute (get and set) methods" on page 539
"Defining a subclass instance method" on page 555
"Defining a factory section" on page 557

RELATED REFERENCES
Inheritance, overriding, and hiding (*The Java Language Specification*)
COBOL class definition structure (*Enterprise COBOL Language Reference*)

CLASS-ID paragraph for defining a subclass

Use the CLASS-ID paragraph to name the subclass and indicate from which immediate Java or COBOL superclass it inherits its characteristics.

```
Identification Division.                   Required
Class-id. CheckingAccount inherits Account. Required
```

In the example above, CheckingAccount is the subclass being defined. CheckingAccount inherits all the methods of the class known within the subclass definition as Account. CheckingAccount methods can access Account instance data only if the Account class provides attribute (get or set) methods for doing so.

You must specify the name of the immediate superclass in the REPOSITORY paragraph in the CONFIGURATION SECTION of the ENVIRONMENT DIVISION. You can optionally associate the superclass name with the name of the class as it is known externally. You can also specify the name of the subclass that you are defining (here, CheckingAccount) in the REPOSITORY paragraph and associate it with its corresponding external class-name.

RELATED TASKS
"CLASS-ID paragraph for defining a class" on page 530
"Coding attribute (get and set) methods" on page 539
"REPOSITORY paragraph for defining a subclass"

REPOSITORY paragraph for defining a subclass

Use the REPOSITORY paragraph to declare to the compiler that the specified words are class-names when you use them within a subclass definition, and to optionally relate the class-names to the corresponding external class-names (the class-names as they are known outside the compilation unit).

For example, in the CheckingAccount subclass definition, these REPOSITORY paragraph entries indicate that the external class-names of the classes referred to as CheckingAccount, Check, and Account within the subclass definition are CheckingAccount, Check, and Account, respectively.

```
Environment Division.                          Required
Configuration Section.                         Required
Repository.                                    Required
    Class CheckingAccount is "CheckingAccount" Optional
    Class Check           is "Check"           Required
    Class Account         is "Account".        Required
```

In the REPOSITORY paragraph, you must code an entry for each class-name that you explicitly reference in the subclass definition. For example:

- A user-defined superclass from which the subclass that you are defining inherits
- The classes that you reference in methods within the subclass definition

The rules for coding REPOSITORY paragraph entries in a subclass are identical to those for coding REPOSITORY paragraph entries in a class.

RELATED TASKS
"REPOSITORY paragraph for defining a class" on page 530

RELATED REFERENCES
REPOSITORY paragraph (*Enterprise COBOL Language Reference*)

WORKING-STORAGE SECTION for defining subclass instance data

Use the WORKING-STORAGE SECTION in the DATA DIVISION of the subclass OBJECT paragraph to describe any instance data that the subclass needs in addition to the instance data defined in its superclasses. Use the same syntax that you use to define instance data in a class.

For example, the definition of the instance data for the CheckingAccount subclass of the Account class might look like this:

```
Identification division.
Object.
 Data division.
 Working-storage section.
 01 CheckFee pic S9(9) value 1.
 . . .
End Object.
```

RELATED TASKS
"WORKING-STORAGE SECTION for defining class instance data" on page 532

Defining a subclass instance method

A subclass inherits the methods of its superclasses. In a subclass definition, you can override any instance method that the subclass inherits by defining an instance method with the same signature as the inherited method. You can also define new methods that the subclass needs.

The structure and syntax of a subclass instance method are identical to those of a class instance method. Define subclass instance methods in the PROCEDURE DIVISION of the OBJECT paragraph of the subclass definition.

"Example: defining a subclass (with methods)"

RELATED TASKS
"Defining a class instance method" on page 533
"Overriding an instance method" on page 537
"Overloading an instance method" on page 538

Example: defining a subclass (with methods)

The following example shows the instance method definitions for the CheckingAccount subclass of the Account class.

The processCheck method invokes the Java instance methods getAmount and getPayee of the Check class to get the check data. It invokes the `credit` and `debit` instance methods inherited from the Account class to credit the payee and debit the payer of the check.

The print method overrides the print instance method defined in the Account class. It invokes the overridden print method to display account status, and also displays the check fee. CheckFee is an instance data item defined in the subclass.

(The Account class was shown in “Example: defining a method” on page 540.)

CheckingAccount class (subclass of Account)

```
 cbl dll,thread,pgmname(longmixed)
 Identification Division.
 Class-id. CheckingAccount inherits Account.
 Environment Division.
 Configuration section.
 Repository.
     Class CheckingAccount is "CheckingAccount"
     Class Check           is "Check"
     Class Account         is "Account".
*
*  (FACTORY paragraph not shown)
*
 Identification division.
 Object.
  Data division.
  Working-storage section.
  01 CheckFee pic S9(9) value 1.
  Procedure Division.
*
*    processCheck method to get the check amount and payee,
*    add the check fee, and invoke inherited methods debit
*    to debit the payer and credit to credit the payee:
   Identification Division.
   Method-id. "processCheck".
   Data division.
   Local-storage section.
   01 amount pic S9(9) binary.
   01 payee usage object reference Account.
   Linkage section.
   01 aCheck usage object reference Check.
*
   Procedure Division using by value aCheck.
     Invoke aCheck "getAmount" returning amount
     Invoke aCheck "getPayee" returning payee
     Invoke payee  "credit" using by value amount
     Add checkFee to amount
     Invoke self   "debit"  using by value amount.
   End method "processCheck".
*
*    print method override to display account status:
   Identification Division.
   Method-id. "print".
   Data division.
   Local-storage section.
   01 printableFee pic $$,$$$,$$9.
   Procedure Division.
     Invoke super "print"
     Move CheckFee to printableFee
     Display " Check fee: " printableFee.
   End method "print".
*
 End Object.
*
 End class CheckingAccount.
```

RELATED TASKS
Chapter 16, “Compiling, linking, and running OO applications,” on page 287
“Invoking methods (INVOKE)” on page 546

"Overriding an instance method" on page 537
"Invoking overridden superclass methods" on page 549

Defining a factory section

Use the FACTORY paragraph in a class definition to define data and methods that are to be associated with the class itself rather than with individual object instances.

COBOL *factory data* is equivalent to Java private static data. A single copy of the data is instantiated for the class and is shared by all object instances of the class. You most commonly use factory data when you want to gather data from all the instances of a class. For example, you could define a factory data item to keep a running total of the number of instances of the class that are created.

COBOL *factory methods* are equivalent to Java public static methods. The methods are supported by the class independently of any object instance. You most commonly use factory methods to customize object creation when you cannot use VALUE clauses alone to initialize instance data.

By contrast, you use the OBJECT paragraph in a class definition to define data that is created for each object instance of the class, and methods that are supported for each object instance of the class.

A factory definition consists of three divisions, followed by an END FACTORY statement:

Table 78. **Structure of factory definitions**

Division	Purpose	Syntax
IDENTIFICATION (required)	Identify the start of the factory definition.	IDENTIFICATION DIVISION. FACTORY.
DATA (optional)	Describe data that is allocated once for the class (as opposed to data allocated for each instance of a class).	"WORKING-STORAGE SECTION for defining factory data" (optional)
PROCEDURE (optional)	Define factory methods.	Factory method definitions: "Defining a factory method" on page 558

"Example: defining a factory (with methods)" on page 560

RELATED TASKS
"Defining a class" on page 528
"Instantiating COBOL classes" on page 551
"Wrapping procedure-oriented COBOL programs" on page 566
"Structuring OO applications" on page 566

WORKING-STORAGE SECTION for defining factory data

Use the WORKING-STORAGE SECTION in the DATA DIVISION of the FACTORY paragraph to describe the *factory data* that a COBOL class needs, that is, statically allocated data to be shared by all object instances of the class.

The FACTORY keyword, which you must immediately precede with an IDENTIFICATION DIVISION declaration, indicates the beginning of the definitions of the factory data and factory methods for the class. For example, the definition of the factory data for the Account class might look like this:

```
Identification division.
Factory.
 Data division.
 Working-storage section.
 01 NumberOfAccounts pic 9(6) value zero.
 . . .
End Factory.
```

You can initialize simple factory data by using VALUE clauses as shown above.

COBOL factory data is equivalent to Java private static data. No other class or subclass (nor instance method in the same class, if any) can reference COBOL factory data directly. Factory data is global to all factory methods that the FACTORY paragraph defines. If you want to make factory data accessible from outside the FACTORY paragraph, define factory attribute (get or set) methods for doing so.

RELATED TASKS
"Coding attribute (get and set) methods" on page 539
"Instantiating COBOL classes" on page 551

Defining a factory method

Define COBOL *factory methods* in the PROCEDURE DIVISION of the FACTORY paragraph of a class definition. A factory method defines an operation that is supported by a class independently of any object instance of the class. COBOL factory methods are equivalent to Java public static methods.

You typically define factory methods for classes whose instances require complex initialization, that is, to values that you cannot assign by using VALUE clauses alone. Within a factory method you can invoke instance methods to initialize the instance data. A factory method cannot directly access instance data.

You can code factory attribute (get and set) methods to make factory data accessible from outside the FACTORY paragraph, for example, to make the data accessible from instance methods in the same class or from a client program. For example, the Account class could define a factory method getNumberOfAccounts to return the current tally of the number of accounts.

You can use factory methods to wrap procedure-oriented COBOL programs so that they are accessible from Java programs. You can code a factory method called main to enable you to run an OO application by using the java command, and to structure your applications in keeping with standard Java practice. See the related tasks for details.

In defining factory methods, you use the same syntax that you use to define instance methods. A COBOL factory method definition consists of four divisions (like a COBOL program), followed by an END METHOD marker:

Table 79. **Structure of factory method definitions**

Division	Purpose	Syntax
IDENTIFICATION (required)	Same as for a class instance method	Same as for a class instance method (required)
ENVIRONMENT (optional)	Same as for a class instance method	Same as for a class instance method

Table 79. **Structure of factory method definitions** *(continued)*

Division	Purpose	Syntax
`DATA` (optional)	Same as for a class instance method	Same as for a class instance method
`PROCEDURE` (optional)	Same as for a class instance method	Same as for a class instance method

Within a class definition, you do not need to make each factory method-name unique, but you do need to give each factory method a unique signature. You can overload factory methods in exactly the same way that you overload instance methods. For example, the CheckingAccount subclass provides two versions of the factory method createCheckingAccount: one that initializes the account to have a default balance of zero, and one that allows the opening balance to be passed in. Clients can invoke either createCheckingAccount method by passing arguments that match the signature of the intended method.

If you define a data item with the same name in both the `DATA DIVISION` of a factory method and the `DATA DIVISION` of the `FACTORY` paragraph, a reference in the method to that data-name refers only to the method data item. The method `DATA DIVISION` takes precedence.

"Example: defining a factory (with methods)" on page 560

RELATED TASKS
"Structuring OO applications" on page 566
"Wrapping procedure-oriented COBOL programs" on page 566
"Instantiating COBOL classes" on page 551
"Defining a class instance method" on page 533
"Coding attribute (get and set) methods" on page 539
"Overloading an instance method" on page 538
"Hiding a factory or static method"
"Invoking factory or static methods" on page 560
"Using object-oriented COBOL and Java under IMS" on page 418

Hiding a factory or static method

A factory method defined in a subclass is said to *hide* an inherited COBOL or Java method that would otherwise be accessible in the subclass if the two methods have the same signature.

To hide a superclass factory method `f1` in a COBOL subclass, define a factory method `f1` in the subclass that has the same name and whose `PROCEDURE DIVISION USING` phrase (if any) has the same number and type of formal parameters as the superclass method has. (If the superclass method is implemented in Java, you must code formal parameters that are interoperable with the data types of the corresponding Java parameters.) When a client invokes `f1` using the subclass name, the subclass method rather than the superclass method is invoked.

The presence or absence of a method return value and the data type of the return value used in the `PROCEDURE DIVISION RETURNING` phrase (if any) must be identical in the subclass factory method and the hidden superclass method.

A factory method must not hide an instance method in a Java or COBOL superclass.

"Example: defining a factory (with methods)"

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 574
"Overriding an instance method" on page 537
"Invoking methods (INVOKE)" on page 546

RELATED REFERENCES
Inheritance, overriding, and hiding (*The Java Language Specification*)
The procedure division header (*Enterprise COBOL Language Reference*)

Invoking factory or static methods

To invoke a COBOL factory method or Java static method in a COBOL method or client program, code the class-name as the first operand of the INVOKE statement.

For example, a client program could invoke one of the overloaded CheckingAccount factory methods called createCheckingAccount to create a checking account with account number 777777 and an opening balance of $300 by coding this statement:

```
Invoke CheckingAccount "createCheckingAccount"
  using by value 777777 300
  returning aCheckingAccount
```

To invoke a factory method from within the same class in which you define the factory method, you also use the class-name as the first operand in the INVOKE statement.

Code the name of the method to be invoked either as a literal or as an identifier whose value at run time is the method-name. The method-name must be an alphanumeric or national literal or a category alphabetic, alphanumeric, or national data item, and is interpreted in a case-sensitive manner.

If an invoked method is not supported in the class that you name in the INVOKE statement, a severity-3 Language Environment condition is raised at run time unless you code the ON EXCEPTION phrase in the INVOKE statement.

The conformance requirements for passing arguments to a COBOL factory method or Java static method in the USING phrase, and receiving a return value in the RETURNING phrase, are the same as those for invoking instance methods.

"Example: defining a factory (with methods)"

RELATED TASKS
"Invoking methods (INVOKE)" on page 546
"Using national data (Unicode) in COBOL" on page 126
"Coding interoperable data types in COBOL and Java" on page 574

RELATED REFERENCES
INVOKE statement (*Enterprise COBOL Language Reference*)

Example: defining a factory (with methods)

The following example updates the previous examples to show the definition of factory data and methods.

These updates are shown:

- The Account class adds factory data and a parameterized factory method, createAccount, which allows an Account instance to be created using an account number that is passed in.
- The CheckingAccount subclass adds factory data and an overloaded parameterized factory method, createCheckingAccount. One implementation of createCheckingAccount initializes the account with a default balance of zero, and the other allows the opening balance to be passed in. Clients can invoke either method by passing arguments that match the signature of the desired method.
- The TestAccounts client invokes the services provided by the factory methods of the Account and CheckingAccount classes, and instantiates the Java Check class.
- The output from the TestAccounts client program is shown.

(The previous examples were “Example: defining a method” on page 540, “Example: defining a client” on page 552, and “Example: defining a subclass (with methods)” on page 555.)

You can also find the complete source code for this example in the cobol/demo/oosample subdirectory in the HFS. Typically the complete path for the source is /usr/lpp/cobol/demo/oosample. You can use the makefile there to compile and link the code.

Account class

```
 cbl dll,thread,pgmname(longmixed),lib
 Identification Division.
 Class-id. Account inherits Base.
 Environment Division.
 Configuration section.
 Repository.
     Class Base    is "java.lang.Object"
     Class Account is "Account".
*
 Identification division.
 Factory.
  Data division.
  Working-storage section.
  01 NumberOfAccounts pic 9(6) value zero.
*
  Procedure Division.
*
*    createAccount method to create a new Account
*    instance, then invoke the OBJECT paragraph's init
*    method on the instance to initialize its instance data:
   Identification Division.
   Method-id. "createAccount".
   Data division.
   Linkage section.
   01 inAccountNumber  pic S9(6) binary.
   01 outAccount object reference Account.
*      Facilitate access to JNI services:
     Copy JNI.
   Procedure Division using by value inAccountNumber
       returning outAccount.
*      Establish addressability to JNI environment structure:
     Set address of JNIEnv to JNIEnvPtr
     Set address of JNINativeInterface to JNIEnv
     Invoke Account New returning outAccount
     Invoke outAccount "init" using by value inAccountNumber
     Add 1 to NumberOfAccounts.
   End method "createAccount".
*
 End Factory.
*
```

```
 Identification division.
 Object.
  Data division.
  Working-storage section.
  01 AccountNumber  pic 9(6).
  01 AccountBalance pic S9(9) value zero.
*
  Procedure Division.
*
*    init method to initialize the account:
   Identification Division.
   Method-id. "init".
   Data division.
   Linkage section.
   01 inAccountNumber pic S9(9) binary.
   Procedure Division using by value inAccountNumber.
     Move inAccountNumber to AccountNumber.
   End method "init".
*
*    getBalance method to return the account balance:
   Identification Division.
   Method-id. "getBalance".
   Data division.
   Linkage section.
   01 outBalance pic S9(9) binary.
   Procedure Division returning outBalance.
     Move AccountBalance to outBalance.
   End method "getBalance".
*
*    credit method to deposit to the account:
   Identification Division.
   Method-id. "credit".
   Data division.
   Linkage section.
   01 inCredit   pic S9(9) binary.
   Procedure Division using by value inCredit.
     Add inCredit to AccountBalance.
   End method "credit".
*
*    debit method to withdraw from the account:
   Identification Division.
   Method-id. "debit".
   Data division.
   Linkage section.
   01 inDebit    pic S9(9) binary.
   Procedure Division using by value inDebit.
     Subtract inDebit from AccountBalance.
   End method "debit".
*
*    print method to display formatted account number and balance:
   Identification Division.
   Method-id. "print".
   Data division.
   Local-storage section.
   01 PrintableAccountNumber  pic ZZZZZZ999999.
   01 PrintableAccountBalance pic $$$$,$$$,$$9CR.
   Procedure Division.
     Move AccountNumber  to PrintableAccountNumber
     Move AccountBalance to PrintableAccountBalance
     Display " Account: " PrintableAccountNumber
     Display " Balance: " PrintableAccountBalance.
   End method "print".
*
 End Object.
*
 End class Account.
```

CheckingAccount class (subclass of Account)

```
 cbl dll,thread,pgmname(longmixed),lib
 Identification Division.
 Class-id. CheckingAccount inherits Account.
 Environment Division.
 Configuration section.
 Repository.
     Class CheckingAccount is "CheckingAccount"
     Class Check           is "Check"
     Class Account         is "Account".
*
 Identification division.
 Factory.
  Data division.
  Working-storage section.
  01 NumberOfCheckingAccounts pic 9(6) value zero.
*
  Procedure Division.
*
*    createCheckingAccount overloaded method to create a new
*    CheckingAccount instance with a default balance, invoke
*    inherited instance method init to initialize the account
*    number, and increment factory data tally of checking accounts:
   Identification Division.
   Method-id. "createCheckingAccount".
   Data division.
   Linkage section.
   01 inAccountNumber  pic S9(6) binary.
   01 outCheckingAccount object reference CheckingAccount.
*      Facilitate access to JNI services:
     Copy JNI.
   Procedure Division using by value inAccountNumber
       returning outCheckingAccount.
*      Establish addressability to JNI environment structure:
     Set address of JNIEnv to JNIEnvPtr
     Set address of JNINativeInterface to JNIEnv
     Invoke CheckingAccount New returning outCheckingAccount
     Invoke outCheckingAccount "init"
       using by value inAccountNumber
     Add 1 to NumberOfCheckingAccounts.
   End method "createCheckingAccount".
*
*    createCheckingAccount overloaded method to create a new
*    CheckingAccount instance, invoke inherited instance methods
*    init to initialize the account number and credit to set the
*    balance, and increment factory data tally of checking accounts:
   Identification Division.
   Method-id. "createCheckingAccount".
   Data division.
   Linkage section.
   01 inAccountNumber  pic S9(6) binary.
   01 inInitialBalance pic S9(9) binary.
   01 outCheckingAccount object reference CheckingAccount.
     Copy JNI.
   Procedure Division using by value inAccountNumber
                                  inInitialBalance
       returning outCheckingAccount.
     Set address of JNIEnv to JNIEnvPtr
     Set address of JNINativeInterface to JNIEnv
     Invoke CheckingAccount New returning outCheckingAccount
     Invoke outCheckingAccount "init"
       using by value inAccountNumber
     Invoke outCheckingAccount "credit"
       using by value inInitialBalance
     Add 1 to NumberOfCheckingAccounts.
   End method "createCheckingAccount".
*
```

```
 End Factory.
*
 Identification division.
 Object.
  Data division.
  Working-storage section.
  01 CheckFee pic S9(9) value 1.
  Procedure Division.
*
*    processCheck method to get the check amount and payee,
*    add the check fee, and invoke inherited methods debit
*    to debit the payer and credit to credit the payee:
   Identification Division.
   Method-id. "processCheck".
   Data division.
   Local-storage section.
   01 amount pic S9(9) binary.
   01 payee usage object reference Account.
   Linkage section.
   01 aCheck usage object reference Check.
   Procedure Division using by value aCheck.
     Invoke aCheck "getAmount" returning amount
     Invoke aCheck "getPayee" returning payee
     Invoke payee  "credit" using by value amount
     Add checkFee to amount
     Invoke self   "debit"  using by value amount.
   End method "processCheck".
*
*    print method override to display account status:
   Identification Division.
   Method-id. "print".
   Data division.
   Local-storage section.
   01 printableFee pic $$,$$$,$$9.
   Procedure Division.
     Invoke super "print"
     Move CheckFee to printableFee
     Display " Check fee: " printableFee.
   End method "print".
*
 End Object.
*
 End class CheckingAccount.
```

Check class

```
/**
 * A Java class for check information
 */
public class Check {
  private CheckingAccount payer;
  private Account         payee;
  private int             amount;

  public Check(CheckingAccount inPayer, Account inPayee, int inAmount) {
    payer=inPayer;
    payee=inPayee;
    amount=inAmount;
  }

  public int getAmount() {
    return amount;
  }

  public Account getPayee() {
    return payee;
  }
}
```

TestAccounts client program

```
 cbl dll,thread,pgmname(longmixed)
 Identification division.
 Program-id. "TestAccounts" recursive.
 Environment division.
 Configuration section.
 Repository.
     Class Account         is "Account"
     Class CheckingAccount is "CheckingAccount"
     Class Check           is "Check".
 Data Division.
*  Working data is declared in Local-storage
*  so that each thread has its own copy:
 Local-storage section.
 01  anAccount        usage object reference Account.
 01  aCheckingAccount usage object reference CheckingAccount.
 01  aCheck           usage object reference Check.
 01  payee            usage object reference Account.
*
 Procedure division.
 Test-Account-section.
     Display "Test Account class"
*  Create account 123456 with 0 balance:
     Invoke Account "createAccount"
       using by value 123456
       returning anAccount
*  Deposit 500 to the account:
     Invoke anAccount "credit" using by value 500
     Invoke anAccount "print"
     Display space
*
     Display "Test CheckingAccount class"
*  Create checking account 777777 with balance of 300:
     Invoke CheckingAccount "createCheckingAccount"
       using by value 777777 300
       returning aCheckingAccount
*  Set account 123456 as the payee:
     Set payee to anAccount
*  Initialize check for 125 to be paid by account 777777 to payee:
     Invoke Check New
       using by value aCheckingAccount, payee, 125
       returning aCheck
*  Debit the payer, and credit the payee:
     Invoke aCheckingAccount "processCheck"
       using by value aCheck
     Invoke aCheckingAccount "print"
     Invoke anAccount "print"
*
     Stop Run.
 End program "TestAccounts".
```

Output produced by the TestAccounts client program

```
Test Account class
 Account:      123456
 Balance:        $500

Test CheckingAccount class
 Account:      777777
 Balance:        $174
 Check fee:        $1
 Account:      123456
 Balance:        $625
```

RELATED TASKS

"Creating and initializing instances of classes" on page 550

"Defining a factory method" on page 558
"Invoking factory or static methods" on page 560
Chapter 16, "Compiling, linking, and running OO applications," on page 287

Wrapping procedure-oriented COBOL programs

A *wrapper* is a class that provides an interface between object-oriented code and procedure-oriented code. Factory methods provide a convenient means for writing wrappers for existing procedural COBOL code to make it accessible from Java programs.

To wrap COBOL code, do these steps:

1. Create a simple COBOL class that contains a `FACTORY` paragraph.
2. In the `FACTORY` paragraph, code a factory method that uses a `CALL` statement to call the procedural program.

A Java program can invoke the factory method by using a static method invocation expression, thus invoking the COBOL procedural program.

RELATED TASKS
"Defining a class" on page 528
"Defining a factory section" on page 557
"Defining a factory method" on page 558

Structuring OO applications

You can structure applications that use object-oriented COBOL syntax in one of three ways.

An OO application can begin with:

- A COBOL program, which can have any name.

  Under UNIX, you can run the application by specifying the name of the linked module (which should match the program name) at the command prompt. You can also bind the program as a module in a PDSE and run it in JCL using the `EXEC PGM` statement.
- A Java class definition that contains a method called `main`. Declare `main` as `public`, `static`, and `void`, with a single parameter of type String[].

  You can run the application with the `java` command, specifying the name of the class that contains `main` and zero or more strings as command-line arguments.
- A COBOL class definition that contains a factory method called `main`. Declare `main` with no `RETURNING` phrase and a single `USING` parameter, an object reference to a class that is an array with elements of type java.lang.String. (Thus `main` is in effect public, static, and void, with a single parameter of type String[].)

  You can run the application with the `java` command, specifying the name of the class that contains `main` and zero or more strings as command-line arguments.

  Structure an OO application this way if you want to:
  - Run the application by using the `java` command.
  - Run the application in an environment where applications must start with the `main` method of a Java class file (such as an IMS Java dependent region).
  - Follow standard Java programming practice.

  "Examples: COBOL applications that you can run using the java command" on page 567

RELATED TASKS
Chapter 16, “Compiling, linking, and running OO applications,” on page 287
“Defining a factory method” on page 558
“Declaring arrays and strings for Java” on page 575
Chapter 22, “Developing COBOL programs for IMS,” on page 417

Examples: COBOL applications that you can run using the java command

The following examples show COBOL class definitions that contain a factory method called main.

In each case, main has no RETURNING phrase and has a single USING parameter, an object reference to a class that is an array with elements of type java.lang.String. You can run these applications by using the java command.

Displaying a message

```
 cbl dll,thread
 Identification Division.
 Class-id. CBLmain inherits Base.
 Environment Division.
 Configuration section.
 Repository.
     Class Base is "java.lang.Object"
     Class stringArray is "jobjectArray:java.lang.String"
     Class CBLmain is "CBLmain".
*
 Identification Division.
 Factory.
  Procedure division.
*
   Identification Division.
   Method-id. "main".
   Data division.
   Linkage section.
   01 SA usage object reference stringArray.
   Procedure division using by value SA.
     Display " >> COBOL main method entered"
     .
   End method "main".
 End factory.
 End class CBLmain.
```

Echoing the input strings

```
 cbl dll,thread,lib,pgmname(longmixed),ssrange
 Identification Division.
 Class-id. Echo inherits Base.
 Environment Division.
 Configuration section.
 Repository.
     Class Base is "java.lang.Object"
     Class stringArray is "jobjectArray:java.lang.String"
     Class jstring is "java.lang.String"
     Class Echo is "Echo".
*
 Identification Division.
 Factory.
  Procedure division.
*
   Identification Division.
   Method-id. "main".
   Data division.
   Local-storage section.
   01 SAlen        pic s9(9) binary.
   01 I            pic s9(9) binary.
   01 SAelement    object reference jstring.
   01 SAelementlen pic s9(9) binary.
   01 Sbuffer      pic X(65535).
   01 P            pointer.
   Linkage section.
   01 SA           object reference stringArray.
   Copy "JNI.cpy" suppress.
   Procedure division using by value SA.
     Set address of JNIEnv to JNIEnvPtr
     Set address of JNINativeInterface to JNIEnv
     Call GetArrayLength using by value JNIEnvPtr SA
       returning SAlen
     Display "Input string array length: " SAlen
     Display "Input strings:"
     Perform varying I from 0 by 1 until I = SAlen
       Call GetObjectArrayElement
         using by value JNIEnvPtr SA I
         returning SAelement
       Call "GetStringPlatformLength"
         using by value JNIEnvPtr
                        SAelement
                        address of SAelementlen
                        0
       Call "GetStringPlatform"
         using by value JNIEnvPtr
                        SAelement
                        address of Sbuffer
                        length of Sbuffer
                        0
       Display Sbuffer(1:SAelementlen)
     End-perform
     .
   End method "main".
 End factory.
 End class Echo.
```

RELATED TASKS

Chapter 16, "Compiling, linking, and running OO applications," on page 287
"Defining a factory method" on page 558
Chapter 31, "Communicating with Java methods," on page 569

Chapter 31. Communicating with Java methods

To achieve interlanguage interoperability with Java, you need to follow certain rules and guidelines for using services in the Java Native Interface (JNI), coding data types, and compiling COBOL programs.

You can invoke methods that are written in Java from COBOL programs, and you can invoke methods that are written in COBOL from Java programs. You need to code COBOL object-oriented language for basic Java object capabilities. For additional Java capabilities, you can call JNI services.

Because Java programs might be multithreaded and use asynchronous signals, compile COBOL programs with the `THREAD` option.

"Example: J2EE client written in COBOL" on page 580

RELATED TASKS
"Using national data (Unicode) in COBOL" on page 126
"Accessing JNI services"
"Sharing data with Java" on page 573
Chapter 30, "Writing object-oriented programs," on page 525
Chapter 16, "Compiling, linking, and running OO applications," on page 287
Chapter 27, "Preparing COBOL programs for multithreading," on page 477

RELATED REFERENCES
Java 2 Enterprise Edition Developer's Guide

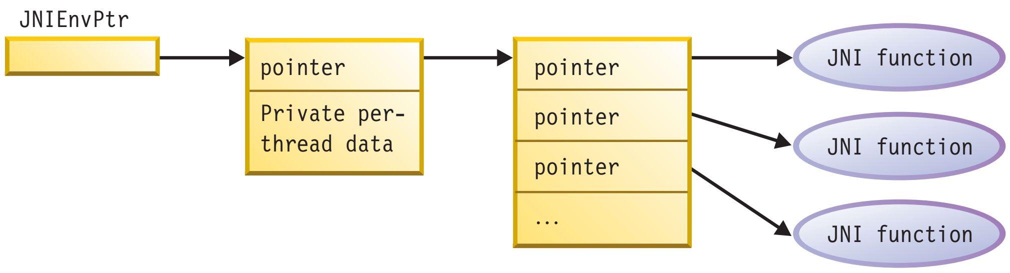
Accessing JNI services

The Java Native Interface (JNI) provides many callable services that you can use when you develop applications that mix COBOL and Java. To facilitate access to these services, copy JNI.cpy into the `LINKAGE SECTION` of your COBOL program.

The JNI.cpy copybook contains these definitions:

- COBOL data definitions that correspond to the Java JNI types
- JNINativeInterface, the JNI environment structure that contains function pointers for accessing the callable service functions

You obtain the JNI environment structure by two levels of indirection from the JNI environment pointer, as the following illustration shows:



Use the special register `JNIEnvPtr` to reference the JNI environment pointer to obtain the address for the JNI environment structure. `JNIEnvPtr` is implicitly defined as `USAGE POINTER`; do not use it as a receiving data item. Before you reference the contents of the JNI environment structure, you must code the following statements to establish its addressability:

```
Linkage section.
COPY JNI
. . .
Procedure division.
    Set address of JNIEnv to JNIEnvPtr
    Set address of JNINativeInterface to JNIEnv
    . . .
```

The code above sets the addresses of the following items:

- JNIEnv, a pointer data item that JNI.cpy provides. `JNIEnvPtr` is the COBOL special register that contains the environment pointer.
- JNINativeInterface, the COBOL group structure that JNI.cpy contains. This structure maps the JNI environment structure, which contains an array of function pointers for the JNI callable services.

After you code the statements above, you can access the JNI callable services with `CALL` statements that reference the function pointers. You can pass the `JNIEnvPtr` special register as the first argument to the services that require the environment pointer, as shown in the following example:

```
01 InputArrayObj usage object reference jlongArray.
01 ArrayLen pic S9(9) comp-5.
. . .
    Call GetArrayLength using by value JNIEnvPtr InputArrayObj
      returning ArrayLen
```

Important: Pass all arguments to the JNI callable services by value.

Some JNI callable services require a Java class-object reference as an argument. To obtain a reference to the class object that is associated with a class, use one of the following JNI callable services:

- GetObjectClass
- FindClass

Restriction: The JNI environment pointer is thread specific. Do not pass it from one thread to another.

RELATED TASKS
"Managing local and global references" on page 572
"Handling Java exceptions"
"Coding interoperable data types in COBOL and Java" on page 574
"Defining a client" on page 541

RELATED REFERENCES
Appendix F, "JNI.cpy," on page 693
The Java Native Interface

Handling Java exceptions

Use JNI services to throw and catch Java exceptions.

Throwing an exception: Use one of the following services to throw a Java exception from a COBOL method:

- Throw
- ThrowNew

You must make the thrown object an instance of a subclass of java.lang.Throwable.

The Java virtual machine (JVM) does not recognize and process the thrown exception until the method that contains the call has completed and returned to the JVM.

Catching an exception: After you invoke a method that might have thrown a Java exception, you can do these steps:

1. Test whether an exception occurred.
2. If an exception occurred, process the exception.
3. Clear the exception, if clearing is appropriate.

Use the following JNI services:

- ExceptionOccurred
- ExceptionCheck
- ExceptionDescribe
- ExceptionClear

To do error analysis, use the methods supported by the exception object that is returned. This object is an instance of the java.lang.Throwable class.

"Example: handling Java exceptions"

Example: handling Java exceptions

The following example shows the use of JNI services for catching an exception from Java and the use of the PrintStackTrace method of java.lang.Throwable for error analysis.

```
Repository.
    Class JavaException is "java.lang.Exception".
. . .
Local-storage section.
01 ex usage object reference JavaException.
Linkage section.
COPY "JNI.cpy".
. . .
Procedure division.
    Set address of JNIEnv to JNIEnvPtr
    Set address of JNINativeInterface to JNIEnv
    . . .
    Invoke anObj "someMethod"
    Perform ErrorCheck
. . .
ErrorCheck.
    Call ExceptionOccurred
        using by value JNIEnvPtr
        returning ex
    If ex not = null then
        Call ExceptionClear using by value JNIEnvPtr
        Display "Caught an unexpected exception"
        Invoke ex "printStackTrace"
        Stop run
    End-if
```

Managing local and global references

The Java virtual machine tracks the object references that you use in native methods, such as COBOL methods. This tracking ensures that the objects are not prematurely released during garbage collection.

There are two classes of such references:

Local references
: Local references are valid only while the method that you invoke runs. Automatic freeing of the local references occurs after the native method returns.

Global references
: Global references remain valid until you explicitly delete them. You can create global references from local references by using the JNI service NewGlobalRef.

The following object references are always local:

- Object references that are received as method parameters
- Object references that are returned as the method `RETURNING` value from a method invocation
- Object references that are returned by a call to a JNI function
- Object references that you create by using the `INVOKE . . . NEW` statement

You can pass either a local reference or a global reference as an object reference argument to a JNI service.

You can code methods to return either local or global references as `RETURNING` values. However, in either case, the reference that is received by the invoking program is a local reference.

You can pass either local or global references as `USING` arguments in a method invocation. However, in either case, the reference that is received by the invoked method is a local reference.

Local references are valid only in the thread in which you create them. Do not pass them from one thread to another.

RELATED TASKS
"Accessing JNI services" on page 569
"Deleting, saving, and freeing local references"

Deleting, saving, and freeing local references

You can manually delete local references at any point within a method. Save local references only in object references that you define in the `LOCAL-STORAGE SECTION` of a method.

Use a `SET` statement to convert a local reference to a global reference if you want to save a reference in any of these data items:

- An object instance variable
- A factory variable
- A data item in the `WORKING-STORAGE SECTION` of a method

Otherwise, an error occurs. These storage areas persist when a method returns; therefore a local reference is no longer valid.

In most cases you can rely on the automatic freeing of local references that occurs when a method returns. However, in some cases you should explicitly free a local reference within a method by using the JNI service DeleteLocalRef. Here are two situations where explicit freeing is appropriate:

- In a method you access a large object, thereby creating a local reference to the object. After extensive computations, the method returns. Free the large object if you do not need it for the additional computations, because the local reference prevents the object from being released during garbage collection.
- You create a large number of local references in a method, but do not use all of them at the same time. Because the Java virtual machine requires space to keep track of each local reference, you should free those that you no longer need. Freeing the local references helps prevent the system from running out of memory.

  For example, in a COBOL method you loop through a large array of objects, retrieve the elements as local references, and operate on one element at each iteration. You can free the local reference to the array element after each iteration.

Use the following callable services to manage local references and global references.

Table 80. **JNI services for local and global references**

Service	Input arguments	Return value	Purpose
NewGlobalRef	• The JNI environment pointer • A local or global object reference	The global reference, or `NULL` if the system is out of memory	To create a new global reference to the object that the input object reference refers to
DeleteGlobalRef	• The JNI environment pointer • A global object reference	None	To delete a global reference to the object that the input object reference refers to
DeleteLocalRef	• The JNI environment pointer • A local object reference	None	To delete a local reference to the object that the input object reference refers to

RELATED TASKS
"Accessing JNI services" on page 569

Java access controls

The Java access modifiers `protected` and `private` are not enforced when you use the Java Native Interface. Therefore a COBOL program could invoke a protected or private Java method that is not invocable from a Java client. This usage is not recommended.

Sharing data with Java

You can share the COBOL data types that have Java equivalents. (Some COBOL data types have Java equivalents, but others do not.)

Share data items with Java in these ways:

- Pass them as arguments in the `USING` phrase of an `INVOKE` statement.
- Receive them as parameters in the `USING` phrase from a Java method.

- Receive them as the `RETURNING` value in an `INVOKE` statement.
- Return them as the value in the `RETURNING` phrase of the `PROCEDURE DIVISION` header in a COBOL method.

To pass or receive arrays and strings, declare them as object references:
- Declare an array as an object reference that contains an instance of one of the special array classes.
- Declare a string as an object reference that contains an instance of the jstring class.

RELATED TASKS
"Coding interoperable data types in COBOL and Java"
"Declaring arrays and strings for Java" on page 575
"Manipulating Java arrays" on page 576
"Manipulating Java strings" on page 578
"Invoking methods (INVOKE)" on page 546
Chapter 25, "Sharing data," on page 451

Coding interoperable data types in COBOL and Java

Your COBOL program can use only certain data types when communicating with Java.

Table 81. **Interoperable data types in COBOL and Java**

Primitive Java data type	**Corresponding COBOL data type**

boolean[1]	`PIC X` followed by exactly two condition-names of this form:
	`level-number data-name PIC X.`
	`88          data-name-false value X'00'.`
	`88          data-name-true  value X'01' through X'FF'.`

byte[1]	Single-byte alphanumeric: `PIC X` or `PIC A`
short	`USAGE BINARY`, `COMP`, `COMP-4`, or `COMP-5`, with `PICTURE` clause of the form `S9(n)`, where 1<=*n*<=4
int	`USAGE BINARY`, `COMP`, `COMP-4`, or `COMP-5`, with `PICTURE` clause of the form `S9(n)`, where 5<=*n*<=9
long	`USAGE BINARY`, `COMP`, `COMP-4`, or `COMP-5`, with `PICTURE` clause of the form `S9(n)`, where 10<=*n*<=18
float[2]	`USAGE COMP-1`
double[2]	`USAGE COMP-2`
char	Single-character elementary national: `PIC N USAGE NATIONAL`. (Cannot be a national group.)
class types (object references)	`USAGE OBJECT REFERENCE` *class-name*

1. You must distinguish boolean from byte, because they each correspond to `PIC X`. `PIC X` is interpreted as boolean only when you define an argument or a parameter with the two condition-names as shown. Otherwise, a `PIC X` data item is interpreted as the Java byte type.
2. Java floating-point data is represented in IEEE floating point. Enterprise COBOL, however, uses hexadecimal floating-point representation. When you pass floating-point arguments by using an `INVOKE` statement or you receive floating-point data from a Java method, the arguments and data are automatically converted as needed.

RELATED TASKS
"Using national data (Unicode) in COBOL" on page 126

Declaring arrays and strings for Java

When you communicate with Java, declare arrays by using the special array classes, and declare strings by using jstring. Code the COBOL data types shown in the table below.

Table 82. **Interoperable arrays and strings in COBOL and Java**

Java data type	Corresponding COBOL data type
boolean[ ]	`object reference jbooleanArray`
byte[ ]	`object reference jbyteArray`
short[ ]	`object reference jshortArray`
int[ ]	`object reference jintArray`
long[ ]	`object reference jlongArray`
char[ ]	`object reference jcharArray`
Object[ ]	`object reference jobjectArray`
String	`object reference jstring`

To use one of these classes for interoperability with Java, you must code an entry in the `REPOSITORY` paragraph. For example:

```
Configuration section.
Repository.
    Class jbooleanArray is "jbooleanArray".
```

The `REPOSITORY` paragraph entry for an object array type must specify an external class-name in one of these forms:

```
"jobjectArray"
"jobjectArray:external-classname-2"
```

In the first case, the `REPOSITORY` entry specifies an array class in which the elements of the array are objects of type java.lang.Object. In the second case, the `REPOSITORY` entry specifies an array class in which the elements of the array are objects of type *external-classname-2*. Code a colon as the separator between the specification of the jobjectArray type and the external class-name of the array elements.

The following example shows both cases. In the example, `oa` defines an array of elements that are objects of type java.lang.Object. `aDepartment` defines an array of elements that are objects of type com.acme.Employee.

```
Environment Division.
Configuration Section.
Repository.
    Class jobjectArray is "jobjectArray"
    Class Employee     is "com.acme.Employee"
    Class Department   is "jobjectArray:com.acme.Employee".
. . .
Linkage section.
01 oa          usage object reference jobjectArray.
01 aDepartment usage object reference Department.
. . .
Procedure division using by value aDepartment.
. . .
```

"Examples: COBOL applications that you can run using the java command" on page 567

The following Java array types are currently not supported for interoperation with COBOL programs.

Table 83. **Noninteroperable array types in COBOL and Java**

Java data type	Corresponding COBOL data type
float[ ]	`object reference jfloatArray`
double[ ]	`object reference jdoubleArray`

RELATED TASKS
"REPOSITORY paragraph for defining a class" on page 530

Manipulating Java arrays

To represent an array in a COBOL program, code a group item that contains a single elementary item that is of the data type that corresponds to the Java type of the array. Specify an `OCCURS` or `OCCURS DEPENDING ON` clause that is appropriate for the array.

For example, the following code specifies a structure to receive 500 or fewer integer values from a jlongArray object:

```
01  longArray.
    02 X pic S9(10) comp-5 occurs 1 to 500 times depending on N.
```

To operate on objects of the special Java-array classes, call the services that the JNI provides. You can use services to access and set individual elements of an array and for the following purposes, using the services cited:

Table 84. **JNI array services**

Service	Input arguments	Return value	Purpose
GetArrayLength	• The JNI environment pointer • The array object reference	The array length as a binary fullword integer	To get the number of elements in a Java array object
NewBooleanArray, NewByteArray, NewCharArray, NewShortArray, NewIntArray, NewLongArray	• The JNI environment pointer • The number of elements in the array, as a binary fullword integer	The array object reference, or `NULL` if the array cannot be constructed	To create a new Java array object
GetBooleanArrayElements, GetByteArrayElements, GetCharArrayElements, GetShortArrayElements, GetIntArrayElements, GetLongArrayElements	• The JNI environment pointer • The array object reference • A pointer to a boolean item. If the pointer is not null, the boolean item is set to true if a copy of the array elements was made. If a copy was made, the corresponding Release*xxx*ArrayElements service must be called if changes are to be written back to the array object.	A pointer to the storage buffer	To extract the array elements from a Java array into a storage buffer. The services return a pointer to the storage buffer, which you can use as the address of a COBOL group data item defined in the `LINKAGE SECTION`.

Table 84. **JNI array services** *(continued)*

Service	Input arguments	Return value	Purpose
ReleaseBooleanArrayElements, ReleaseByteArrayElements, ReleaseCharArrayElements, ReleaseShortArrayElements, ReleaseIntArrayElements, ReleaseLongArrayElements	• The JNI environment pointer • The array object reference • A pointer to the storage buffer • The release mode, as a binary fullword integer. See Java JNI documentation for details. (Recommendation: Specify 0 to copy back the array content and free the storage buffer.)	None; the storage for the array is released.	To release the storage buffer that contains elements that have been extracted from a Java array, and conditionally map the updated array values back into the array object
NewObjectArray	• The JNI environment pointer • The number of elements in the array, as a binary fullword integer • An object reference for the array element class • An object reference for the initial element value. All array elements are set to this value.	The array object reference, or `NULL` if the array cannot be constructed[1]	To create a new Java object array
GetObjectArrayElement	• The JNI environment pointer • The array object reference • An array element index, as a binary fullword integer using origin zero	An object reference[2]	To return the element at a given index within an object array
SetObjectArrayElement	• The JNI environment pointer • The array object reference • The array element index, as a binary fullword integer using origin zero • The object reference for the new value	None[3]	To set an element within an object array

1. NewObjectArray throws an exception if the system runs out of memory.
2. GetObjectArrayElement throws an exception if the index is not valid.
3. SetObjectArrayElement throws an exception if the index is not valid or if the new value is not a subclass of the element class of the array.

"Examples: COBOL applications that you can run using the java command" on page 567
"Example: processing a Java int array"

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 574
"Declaring arrays and strings for Java" on page 575
"Accessing JNI services" on page 569

Example: processing a Java int array

The following example shows the use of the Java-array classes and JNI services to process a Java array in COBOL.

```
 cbl lib,thread,dll
 Identification division.
 Class-id. OOARRAY inherits Base.
 Environment division.
 Configuration section.
 Repository.
     Class Base is "java.lang.Object"
     Class jintArray is "jintArray".
 Identification division.
 Object.
 Procedure division.
   Identification division.
   Method-id. "ProcessArray".
   Data Division.
   Local-storage section.
   01 intArrayPtr pointer.
   01 intArrayLen pic S9(9) comp-5.
   Linkage section.
      COPY JNI.
   01 inIntArrayObj usage object reference jintArray.
   01 intArrayGroup.
      02 X pic S9(9) comp-5
          occurs 1 to 1000 times depending on intArrayLen.
   Procedure division using by value inIntArrayObj.
       Set address of JNIEnv to JNIEnvPtr
       Set address of JNINativeInterface to JNIEnv

       Call GetArrayLength
         using by value JNIEnvPtr inIntArrayObj
         returning intArrayLen
       Call GetIntArrayElements
         using by value JNIEnvPtr inIntArrayObj 0
         returning IntArrayPtr
       Set address of intArrayGroup to intArrayPtr

*  . . . process the array elements X(I) . . .

       Call ReleaseIntArrayElements
         using by value JNIEnvPtr inIntArrayObj intArrayPtr 0.
   End method "ProcessArray".
 End Object.
 End class OOARRAY.
```

Manipulating Java strings

COBOL represents Java String data in Unicode. To represent a Java String in a COBOL program, declare the string as an object reference of the jstring class. Then use JNI services to set or extract COBOL alphanumeric or national (Unicode) data from the object.

| **Services for Unicode:** Use the following standard services to convert between jstring object references and COBOL `USAGE NATIONAL` data items. Access these services by using function pointers in the JNINativeInterface environment structure.

Table 85. **Services that convert between jstring references and national data**

Service	Input arguments	Return value
NewString[1]	• The JNI environment pointer • A pointer to a Unicode string, such as a COBOL national data item • The number of characters in the string; binary fullword	jstring object reference

Table 85. **Services that convert between jstring references and national data** *(continued)*

Service	Input arguments	Return value
GetStringLength	• The JNI environment pointer • A jstring object reference	The number of Unicode characters in the jstring object reference; binary fullword
GetStringChars[1]	• The JNI environment pointer • A jstring object reference • A pointer to a boolean data item, or `NULL`	• A pointer to the array of Unicode characters extracted from the jstring object, or `NULL` if the operation fails. The pointer is valid until it is released with ReleaseStringChars. • When the pointer to the boolean data item is not null, the boolean value is set to true if a copy is made of the string and to false if no copy is made.
ReleaseStringChars	• The JNI environment pointer • A jstring object reference • A pointer to the array of Unicode characters that was returned from GetStringChars	None; the storage for the array is released.
1. This service throws an exception if the system runs out of memory.		

Services for EBCDIC: Use the following z/OS services, an extension of the JNI, to convert between jstring object references and COBOL alphanumeric data (`PIC X(`*n*`)`). Access these services by using function pointers in the JNI environment structure JNINativeInterface.

Table 86. **Services that convert between jstring references and alphanumeric data**

Service	Input arguments	Return value
NewStringPlatform	• The JNI environment pointer • Pointer to the null-terminated EBCDIC character string that you want to convert to a jstring object • Pointer to the jstring object reference in which you want the result • Pointer to the Java encoding name for the string, represented as a null-terminated EBCDIC character string[1]	Return code as a binary fullword integer: **0** Success. **-1** Malformed input or illegal input character. **-2** Unsupported encoding; the jstring object reference pointer is set to `NULL`.
GetStringPlatformLength	• The JNI environment pointer • jstring object reference for which you want the length • Pointer to a binary fullword integer for the result • Pointer to the Java encoding name for the string, represented as a null-terminated EBCDIC character string[1]	Return code as a binary fullword integer: **0** Success. **-1** Malformed input or illegal input character. **-2** Unsupported encoding; the jstring object reference pointer is set to `NULL`. Returns, in the third argument, the needed length in bytes of the output buffer to hold the converted Java string, including the terminating null byte referenced by the second argument.

Table 86. **Services that convert between jstring references and alphanumeric data** *(continued)*

Service	Input arguments	Return value
GetStringPlatform	• The JNI environment pointer • jstring object reference that you want to convert to a null-terminated string • Pointer to the output buffer in which you want the converted string • Length of the output buffer as a binary fullword integer • Pointer to the Java encoding name for the string, represented as a null-terminated EBCDIC character string[1]	Return code as a binary fullword integer: **0** Success. **-1** Malformed input or illegal input character. **-2** Unsupported encoding; the output string is set to a null string. **-3** Conversion buffer is full.

1. If the pointer is NULL, the encoding from the Java file.encoding property is used.

These EBCDIC services are packaged as a DLL that is part of your IBM Java 2 Software Development Kit. For details about the services, see jni_convert.h in the IBM Java 2 Software Development Kit.

Use CALL *literal* statements to call the services. The calls are resolved through the libjvm.x DLL side file, which you must include in the link step of any COBOL program that uses object-oriented language.

For example, the following code creates a Java String object from the EBCDIC string 'MyConverter'. (This code fragment is from the J2EE client program, which is shown in full in "Example: J2EE client written in COBOL.")

```
Move z"MyConverter" to stringBuf
Call "NewStringPlatform"
  using by value JNIEnvPtr
                 address of stringBuf
                 address of jstring1
                 0
  returning rc
```

If the EBCDIC services are the only JNI services that you call from a COBOL program, you do not need to copy the JNI.cpy copybook. You also do not need to establish addressability with the JNI environment pointer.

Services for UTF-8: The Java Native Interface also provides services for conversion between jstring object references and UTF-8 strings. These services are not recommended for use in COBOL programs due to the difficulty in handling UTF-8 character strings on the z/OS platform.

RELATED TASKS
"Accessing JNI services" on page 569
"Coding interoperable data types in COBOL and Java" on page 574
"Declaring arrays and strings for Java" on page 575
"Using national data (Unicode) in COBOL" on page 126
Chapter 16, "Compiling, linking, and running OO applications," on page 287

Example: J2EE client written in COBOL

The following example shows a COBOL client program that can access enterprise beans that run on a J2EE-compliant EJB server.

The COBOL client is equivalent to the J2EE client program in the Getting Started chapter of *Java 2 Enterprise Edition Developer's Guide*. For your convenience in comparing implementations, the second example shows the equivalent Java client from the guide. (The enterprise bean is the Java implementation of the simple currency-converter enterprise bean, and is in the same guide.)

COBOL client (ConverterClient.cbl)

```
 Process pgmname(longmixed),lib,dll,thread
*****************************************************************
* Demo J2EE client written in COBOL.                            *
*                                                               *
* Based on the sample J2EE client written in Java, which is     *
* given in the "Getting Started" chapter of "The Java(TM) 2     *
* Enterprise Edition Developer's Guide."                        *
*                                                               *
* The client:                                                   *
*   - Locates the home interface of a session enterprise bean   *
*       (a simple currency converter bean)                      *
*   - Creates an enterprise bean instance                       *
*   - Invokes a business method (currency conversion)           *
*****************************************************************
 Identification division.
 Program-id. "ConverterClient" is recursive.
 Environment Division.
 Configuration section.
 Repository.
     Class InitialContext is "javax.naming.InitialContext"
     Class PortableRemoteObject
                          is "javax.rmi.PortableRemoteObject"
     Class JavaObject     is "java.lang.Object"
     Class JavaClass      is "java.lang.Class"
     Class JavaException  is "java.lang.Exception"
     Class jstring        is "jstring"
     Class Converter      is "Converter"
     Class ConverterHome  is "ConverterHome".
 Data division.
 Working-storage section.
 01 initialCtx       object reference InitialContext.
 01 obj              object reference JavaObject.
 01 classObj         object reference JavaClass.
 01 ex               object reference JavaException.
 01 currencyConverter object reference Converter.
 01 home             object reference ConverterHome.
 01 homeObject redefines home object reference JavaObject.
 01 jstring1         object reference jstring.
 01 stringBuf        pic X(500) usage display.
 01 len              pic s9(9) comp-5.
 01 rc               pic s9(9) comp-5.
 01 amount           comp-2.
 Linkage section.
     Copy JNI.
 Procedure division.
     Set address of JNIenv to JNIEnvPtr
     Set address of JNINativeInterface to JNIenv

*****************************************************************
* Create JNDI naming context.                                   *
*****************************************************************
     Invoke InitialContext New returning initialCtx
     Perform JavaExceptionCheck

*****************************************************************
* Create a jstring object for the string "MyConverter" for use  *
* as argument to the lookup method.                             *
*****************************************************************
```

```
     Move z"MyConverter" to stringBuf
     Call "NewStringPlatform"
       using by value JNIEnvPtr
                      address of stringBuf
                      address of jstring1
                      0
       returning rc
     If rc not = zero then
        Display "Error occurred creating jstring object"
        Stop run
     End-if

*****************************************************************
* Use the lookup method to obtain a reference to the home       *
* object bound to the name "MyConverter".  (This is the JNDI    *
* name specified when deploying the J2EE application.)          *
*****************************************************************
     Invoke initialCtx "lookup" using by value jstring1
       returning obj
     Perform JavaExceptionCheck

*****************************************************************
* Narrow the home object to be of type ConverterHome.           *
* First obtain class object for the ConverterHome class, by     *
* passing the null-terminated ASCII string "ConverterHome" to   *
* the FindClass API.  Then use this class object as the         *
* argument to the static method "narrow".                       *
*****************************************************************
     Move z"ConverterHome" to stringBuf
     Call "__etoa"
       using by value address of stringBuf
       returning len
     If len = -1 then
        Display "Error occurred on ASCII conversion"
        Stop run
     End-if
     Call FindClass
       using by value JNIEnvPtr
                      address of stringBuf
       returning classObj
     If classObj = null
        Display "Error occurred locating ConverterHome class"
        Stop run
     End-if
     Invoke PortableRemoteObject "narrow"
       using by value obj
                      classObj
       returning homeObject
     Perform JavaExceptionCheck

*****************************************************************
* Create the ConverterEJB instance and obtain local object      *
* reference for its remote interface                            *
*****************************************************************
     Invoke home "create" returning currencyConverter
     Perform JavaExceptionCheck

*****************************************************************
* Invoke business methods                                       *
*****************************************************************
     Invoke currencyConverter "dollarToYen"
       using by value +100.00E+0
       returning amount
     Perform JavaExceptionCheck

     Display amount
     Invoke currencyConverter "yenToEuro"
       using by value +100.00E+0
       returning amount
     Perform JavaExceptionCheck

     Display amount

*****************************************************************
* Remove the object and return.                                 *
*****************************************************************
     Invoke currencyConverter "remove"
     Perform JavaExceptionCheck

     Goback
     .

*****************************************************************
* Check for thrown Java exceptions                              *
*****************************************************************
 JavaExceptionCheck.
     Call ExceptionOccurred using by value JNIEnvPtr
       returning ex
     If ex not = null then
        Call ExceptionClear using by value JNIEnvPtr
        Display "Caught an unexpected exception"
        Invoke ex "PrintStackTrace"
        Stop run
     End-if
     .
 End program "ConverterClient".
```

Java client (ConverterClient.java)

```
/*
 *
 * Copyright 2000 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.
 *
 */

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

import Converter;
import ConverterHome;

public class ConverterClient {

   public static void main(String[] args) {
       try {
           Context initial = new InitialContext();
           Object objref = initial.lookup("MyConverter");

           ConverterHome home =
               (ConverterHome)PortableRemoteObject.narrow(objref,
                                            ConverterHome.class);

           Converter currencyConverter = home.create();

           double amount = currencyConverter.dollarToYen(100.00);
           System.out.println(String.valueOf(amount));
           amount = currencyConverter.yenToEuro(100.00);
           System.out.println(String.valueOf(amount));
```

```
            currencyConverter.remove();

        } catch (Exception ex) {
            System.err.println("Caught an unexpected exception!");
            ex.printStackTrace();
        }
    }
}
```

RELATED TASKS

Chapter 16, “Compiling, linking, and running OO applications,” on page 287

WebSphere for z/OS: Applications

RELATED REFERENCES

Java 2 Enterprise Edition Developer’s Guide

Part 7. Specialized processing

Chapter 32. Interrupts and checkpoint/restart

When programs run for an extended period of time, interruptions might halt processing before the end of a job. The checkpoint/restart functions of z/OS allow an interrupted program to be restarted at the beginning of a job step or at a checkpoint that you have set.

Because the checkpoint/restart functions cause a lot of extra processing, use them only when you anticipate interruptions caused by machine malfunctions, input or output errors, or intentional operator intervention.

The checkpoint routine starts from the COBOL load module that contains your program. While your program is running, the checkpoint routine creates records at points that you have designated using the COBOL `RERUN` clause. A checkpoint record contains a snapshot of the information in the registers and main storage when the program reached the checkpoint.

The restart routine restarts an interrupted program. You can perform a restart at any time after the program was interrupted: either immediately (automatic restart), or later (deferred restart).

RELATED TASKS
"Setting checkpoints"
"Restarting programs" on page 590
"Resubmitting jobs for restart" on page 593
z/OS DFSMS: Checkpoint/Restart

RELATED REFERENCES
"DD statements for defining checkpoint data sets" on page 589
"Messages generated during checkpoint" on page 590
"Formats for requesting deferred restart" on page 592

Setting checkpoints

To set checkpoints, use job control statements and use the `RERUN` clause in the `ENVIRONMENT DIVISION`. Associate each `RERUN` clause with a particular COBOL file.

The `RERUN` clause indicates that a checkpoint record is to be written onto a checkpoint data set whenever a specified number of records in the COBOL file has been processed or when `END OF VOLUME` is reached. You cannot use the `RERUN` clause with files that have been defined with the `EXTERNAL` attribute.

You can write checkpoint records from several COBOL files onto one checkpoint data set, but you must use a separate data set exclusively for checkpoint records. You cannot embed checkpoint records in one of your program data sets.

Restrictions: A checkpoint data set must have sequential organization. You cannot write checkpoints on VSAM data sets or on data sets that are allocated to extended-format QSAM data sets. Also, a checkpoint cannot be taken if any program in the run unit has an extended-format QSAM data set that is open.

Checkpoint records are written on the checkpoint data set defined by a `DD` statement. In the `DD` statement, you also choose the checkpoint method:

Single (store single checkpoints)
Only one checkpoint record exists at any given time. After the first checkpoint record is written, any succeeding checkpoint record overlays the previous one.

This method is acceptable for most programs. You save space on the checkpoint data set, and you can restart your program at the latest checkpoint.

Multiple (store multiple contiguous checkpoints)
Checkpoints are recorded and numbered sequentially. Each checkpoint is saved.

Use this method when you want to restart a program at a checkpoint other than the latest one taken.

You must use the multiple checkpoint method for complete compliance to Standard COBOL 85.

Checkpoints during sort operations have the following requirements:

- If checkpoints are to be taken during a sort operation, add a `DD` statement for `SORTCKPT` in the job control procedure for execution.
- You can take checkpoint records on ASCII-collated sorts, but the *system-name* indicating the checkpoint data set must not specify an ASCII file.

RELATED TASKS
"Using checkpoint/restart with DFSORT" on page 231
"Designing checkpoints"
"Testing for a successful checkpoint"

RELATED REFERENCES
"DD statements for defining checkpoint data sets" on page 589

Designing checkpoints

Design your checkpoints at critical points in your program so that data can be easily reconstructed. Do not change the contents of files between the time of a checkpoint and the time of the restart.

In a program that uses disk files, design the program so that you can identify previously processed records. For example, consider a disk file that contains loan records that are periodically updated for interest due. If a checkpoint is taken, records are updated, and then the program is interrupted, you would want to test that the records that are updated after the last checkpoint are not updated again when the program is restarted. To do this, set up a date field in each record, and update the field each time the record is processed. Then, after the restart, test the field to determine whether the record was already processed.

For efficient repositioning of a print file, take checkpoints on the file only after printing the last line of a page.

Testing for a successful checkpoint

After each input or output statement that issues a checkpoint, the `RETURN-CODE` special register is updated with the return code from the checkpoint routine. Therefore, you can test whether the checkpoint was successful and decide whether conditions are right to allow a restart.

If the return code is greater than 4, an error has occurred in the checkpoint. Check the return code to prevent a restart that could cause incorrect output.

RELATED REFERENCES
Return codes (*z/OS DFSMS: Checkpoint/Restart*)

DD statements for defining checkpoint data sets

To define checkpoint data sets, use DD statements.

For tape:

```
//ddname  DD DSNAME=data-set-name,
//           [VOLUME=SER=volser,]UNIT=device-type,
//           DISP=({NEW|MOD},PASS)
```

For direct-access devices:

```
//ddname  DD DSNAME=data-set-name,
//           [VOLUME=(PRIVATE,RETAIN,SER=volser),]
//           UNIT=device-type,SPACE=(subparms),
//           DISP=({NEW|MOD},PASS,KEEP)
```

ddname
Provides a link to the DD statement. The same as the ddname portion of the *assignment-name* used in the COBOL RERUN clause.

data-set-name
Identifies the checkpoint data set to the restart procedure. The name given to the data set used to record checkpoint records.

volser Identifies the volume by serial number.

device-type
Identifies the device.

subparms
Specifies the amount of track space needed for the data set.

MOD Specifies the multiple contiguous checkpoint method.

NEW Specifies the single checkpoint method.

PASS Prevents deletion of the data set at successful completion of the job step, unless the job step is the last in the job. If it is the last step, the data set is deleted.

KEEP Keeps the data set if the job step abnormally ends.

"Examples: defining checkpoint data sets"

Examples: defining checkpoint data sets

The following examples show the JCL and COBOL coding you can use to define checkpoint data sets.

Writing single checkpoint records, using tape:

```
//CHECKPT  DD DSNAME=CHECK1,VOLUME=SER=ND0003,
//            UNIT=TAPE,DISP=(NEW,KEEP),LABEL=(,NL)
            . . .
       ENVIRONMENT DIVISION.
           . . .
           RERUN ON CHECKPT EVERY
             5000 RECORDS OF ACCT-FILE.
```

Writing single checkpoint records, using disk:

```
//CHEK     DD DSNAME=CHECK2,
//            VOLUME=(PRIVATE,RETAIN,SER=DB0030),
//            UNIT=3380,DISP=(NEW,KEEP),SPACE=(CYL,5)
              . . .
       ENVIRONMENT DIVISION.
              . . .
           RERUN ON CHEK EVERY
             20000 RECORDS OF PAYCODE.
           RERUN ON CHEK EVERY
             30000 RECORDS OF IN-FILE.
```

Writing multiple contiguous checkpoint records, using tape:

```
//CHEKPT   DD DSNAME=CHECK3,VOLUME=SER=111111,
//            UNIT=TAPE,DISP=(MOD,PASS),LABEL=(,NL)
              . . .
       ENVIRONMENT DIVISION.
              . . .
           RERUN ON CHEKPT EVERY
             10000 RECORDS OF PAY-FILE.
```

Messages generated during checkpoint

The system checkpoint routine advises the operator of the status of the checkpoints taken by displaying informative messages on the console.

Each time a checkpoint is successfully completed, a message is displayed that associates the jobname (*ddname*, *unit*, *volser*) with the checkpoint taken (*checkid*).

The control program assigns *checkid* as an eight-character string. The first character is the letter *C*, followed by a decimal number that indicates the checkpoint. For example, the following message indicates the fourth checkpoint taken in the job step:

```
checkid C0000004
```

Restarting programs

The system restart routine retrieves the information recorded in a checkpoint record, restores the contents of main storage and all registers, and restarts the program.

You can begin the restart routine in one of two ways:

- Automatically at the time an interruption stopped the program
- At a later time as a deferred restart

The RD parameter of the job control language determines the type of restart. You can use the RD parameter on either the JOB or the EXEC statement. If coded on the JOB statement, the parameter overrides any RD parameters on the EXEC statement.

To suppress both restart and writing checkpoints, code RD=NC.

Restriction: If you try to restart at a checkpoint taken by a COBOL program during a SORT or MERGE operation, an error message is issued and the restart is canceled. Only checkpoints taken by DFSORT are valid.

Data sets that have the SYSOUT parameter coded in their DD statements are handled in various ways depending on the type of restart.

If the checkpoint data set is multivolume, include in the VOLUME parameter the sequence number of the volume on which the checkpoint entry was written. If the checkpoint data set is on a 7-track tape with nonstandard labels or no labels, the SYSCHK DD statement must contain DCB=(TRTCH=C,. . .).

RELATED TASKS
"Using checkpoint/restart with DFSORT" on page 231
"Requesting automatic restart"
"Requesting deferred restart"

Requesting automatic restart

Automatic restart occurs only at the latest checkpoint taken. If no checkpoint was taken before interruption, automatic restart occurs at the beginning of the job step.

Whenever automatic restart is to occur, the system repositions all devices except unit-record devices.

If you want automatic restart, code RD=R or RD=RNC:

- RD=R indicates that restart is to occur at the latest checkpoint. Code the RERUN clause for at least one data set in the program in order to record checkpoints. If no checkpoint is taken before interruption, restart occurs at the beginning of the job step.
- RD=RNC indicates that no checkpoint is to be written, and that any restart is to occur at the beginning of the job step. In this case, RERUN clauses are unnecessary; if any are present, they are ignored.

If you omit the RD parameter, the CHKPT macro instruction remains active, and checkpoints can be taken during processing. If an interrupt occurs after the first checkpoint, automatic restart will occur.

To restart automatically, a program must satisfy the following conditions:

- In the program you must request restart by using the RD parameter or by taking a checkpoint.
- An abend that terminated the job must return a code that allows restart.
- The operator must authorize the restart.

"Example: requesting a step restart" on page 593

Requesting deferred restart

Deferred restart can occur at any checkpoint, not necessarily the latest one taken. You can restart your program at a checkpoint other than at the beginning of the job step.

When a deferred restart has been successfully completed, the system displays a message on the console stating that the job has been restarted. Control is then given to your program.

If you want deferred restart, code the RD parameter as RD=NR. This form of the parameter suppresses automatic restart but allows a checkpoint record to be written provided that a RERUN clause was coded.

Request a deferred restart by using the RESTART parameter on the JOB card and a SYSCHK DD statement to identify the checkpoint data set. If a SYSCHK DD statement is present in a job and the JOB statement does not contain the RESTART parameter, the

SYSCHK DD statement is ignored. If a RESTART parameter without the CHECKID subparameter is included in a job, a SYSCHK DD statement must not appear before the first EXEC statement for the job.

"Example: restarting a job at a specific checkpoint step" on page 593

RELATED REFERENCES
"Formats for requesting deferred restart"

Formats for requesting deferred restart

The formats for the RESTART parameter of the JOB statement and the SYSCHK DD statements are as shown below.

```
//jobname  JOB MSGLEVEL=1,RESTART=(request[,checkid])
//SYSCHK   DD  DSNAME=data-set-name,
//             DISP=OLD[,UNIT=device-type,
//             VOLUME=SER=volser]
```

MSGLEVEL=1 (or MSGLEVEL=(1,*y*))
MSGLEVEL is required.

RESTART=(*request*,[*checkid*])
Identifies the particular checkpoint at which restart is to occur.

request
Takes one of the following forms:

*****
Indicates restart at the beginning of the job.

stepname
Indicates restart at the beginning of a job step.

stepname.procstep
Indicates restart at a procedure step within the job step.

checkid
Identifies the checkpoint where restart is to occur.

SYSCHK
The ddname used to identify a checkpoint data set to the control program. The SYSCHK DD statement must immediately precede the first EXEC statement of the resubmitted job, and must follow any JOBLIB statement.

data-set-name
Identifies the checkpoint data set. It must be the same name that was used when the checkpoint was taken.

device-type* and *volser
Identify the device type and the serial number of the volume that contains the checkpoint data set.

"Example: requesting a deferred restart"

Example: requesting a deferred restart

This example shows JCL to restart the GO step of an IGYWCLG procedure at checkpoint identifier (CHECKID) C0000003.

```
//jobname  JOB MSGLEVEL=1,RESTART=(stepname.GO,C0000003)
//SYSCHK   DD  DSNAME=CHEKPT,
//             DISP=OLD[,UNIT=3380,VOLUME=SER=111111]
           . . .
```

Resubmitting jobs for restart

When you resubmit a job for restart, be careful with any DD statements that might affect the execution of the restarted job step. The restart routine uses information from DD statements in the resubmitted job to reset files for use after restart.

If you want a data set to be deleted at the end of a job step, give it a conditional disposition of PASS or KEEP (rather than DELETE). This disposition allows the data set to be available if an interruption forces a restart. If you want to restart a job at the beginning of a step, you must first discard any data set created (defined as NEW in a DD statement) in the previous run, or change the DD statement to mark the data set as OLD.

At restart time, position input data sets on cards as they were at the time of the checkpoint. The system automatically repositions input data sets on tape or disk.

"Example: resubmitting a job for a step restart"
"Example: resubmitting a job for a checkpoint restart" on page 594

Example: restarting a job at a specific checkpoint step

This example shows a sequence of job control statements for restarting a job at a specific step.

```
//PAYROLL  JOB  MSGLEVEL=1,REGION=80K,
//              RESTART=(STEP1,CHECKPT4)
//JOBLIB   DD   DSNAME=PRIV.LIB3,DISP=OLD
//SYSCHK   DD   DSNAME=CHKPTLIB,
//              [UNIT=TAPE,VOL=SER=456789,]
//              DISP=(OLD,KEEP)
//STEP1    EXEC PGM=PROG4,TIME=5
```

Example: requesting a step restart

This example shows the use of the RD parameter, which requests step restart for any abnormally terminated job step.

```
//J1234    JOB  386,SMITH,MSGLEVEL=1,RD=R
//S1       EXEC PGM=MYPROG
//INDATA   DD   DSNAME=INVENT[,UNIT=TAPE],DISP=OLD,
//              [VOLUME=SER=91468,]
//              LABEL=RETPD=14
//REPORT   DD   SYSOUT=A
//WORK     DD   DSNAME=T91468,DISP=(,,KEEP),
//              UNIT=SYSDA,SPACE=(3000,(5000,500)),
//              VOLUME=(PRIVATE,RETAIN,,6)
//DDCKPNT  DD   UNIT=TAPE,DISP=(MOD,PASS,CATLG),
//              DSNAME=C91468,LABEL=(,NL)
```

The DDCKPNT DD statement defines a checkpoint data set. For this step, after a RERUN clause is performed, only automatic checkpoint restart can occur unless a CHKPT cancel is issued.

Example: resubmitting a job for a step restart

This example shows the changes that you might make to the JCL before you resubmit a job for step restart.

```
//J3412    JOB  386,SMITH,MSGLEVEL=1,RD=R,RESTART=*
//S1       EXEC PGM=MYPROG
//INDATA   DD   DSNAME=INVENT[,UNIT=TAPE],DISP=OLD,
//              [VOLUME=SER=91468,]LABEL=RETPD=14
//REPORT   DD   SYSOUT=A
//WORK     DD   DSNAME=S91468,
//              DISP=(,,KEEP),UNIT=SYSDA,
//              SPACE=(3000,(5000,500)),
//              VOLUME=(PRIVATE,RETAIN,,6)
//DDCHKPNT DD   UNIT=TAPE,DISP=(MOD,PASS,CATLG),
//              DSNAME=R91468,LABEL=(,NL)
```

The following changes were made in the example above:

- The job name has been changed (from J1234 to J3412) to distinguish the original job from the restarted job.
- The RESTART parameter has been added to the JOB statement, and indicates that restart is to begin with the first job step.
- The WORK DD statement was originally assigned a conditional disposition of KEEP for this data set:
  - If the step terminated normally in the previous run of the job, the data set was deleted, and no changes need to be made to this statement.
  - If the step abnormally terminated, the data set was kept. In that case, define a new data set (S91468 instead of T91468, as shown), or change the status of the data set to OLD before resubmitting the job.
- A new data set (R91468 instead of C91468) has also been defined as the checkpoint data set.

"Example: requesting a step restart" on page 593

Example: resubmitting a job for a checkpoint restart

This example shows the changes that you might make to JCL before you resubmit a job for checkpoint restart.

```
//J3412    JOB  386,SMITH,MSGLEVEL=1,RD=R,
//              RESTART=(*,C0000002)
//SYSCHK   DD   DSNAME=C91468,DISP=OLD
//S1       EXEC PGM=MYPROG
//INDATA   DD   DSNAME=INVENT,UNIT=TAPE,DISP=OLD,
//              VOLUME=SER=91468,LABEL=RETPD=14
//REPORT   DD   SYSOUT=A
//WORK     DD   DSNAME=T91468,DISP=(,,KEEP),
//              UNIT=SYSDA,SPACE=(3000,(5000,500)),
//              VOLUME=(PRIVATE,RETAIN,,6)
//DDCKPNT  DD   UNIT=TAPE,DISP=(MOD,KEEP,CATLG),
//              DSNAME=C91468,LABEL=(,NL)
```

The following changes were made in the example above:

- The job name has been changed (from J1234 to J3412) to distinguish the original job from the restarted job.
- The RESTART parameter has been added to the JOB statement, and indicates that restart is to begin with the first step at the checkpoint entry named C0000002.
- The DD statement DDCKPNT was originally assigned a conditional disposition of CATLG for the checkpoint data set:
  - If the step terminated normally in the previous run of the job, the data set was kept. In that case, the SYSCHK DD statement must contain all of the information necessary for retrieving the checkpoint data set.

- – If the job abnormally terminated, the data set was cataloged. In that case, the only parameters required on the SYSCHK DD statement are DSNAME and DISP as shown.

If a checkpoint is taken in a job that is running when V=R is specified, the job cannot be restarted until adequate nonpageable dynamic storage becomes available.

Chapter 33. Processing two-digit-year dates

With the millennium language extensions (MLE), you can make simple changes in your COBOL programs to define date fields. The compiler recognizes and acts on these dates by using a century window to ensure consistency.

Use the following steps to implement automatic date recognition in a COBOL program:

1. Add the `DATE FORMAT` clause to the data description entries of the data items in the program that contain dates. You must identify all dates with `DATE FORMAT` clauses, even those that are not used in comparisons.
2. To expand dates, use `MOVE` or `COMPUTE` statements to copy the contents of windowed date fields to expanded date fields.
3. If necessary, use the `DATEVAL` and `UNDATE` intrinsic functions to convert between date fields and nondates.
4. Use the `YEARWINDOW` compiler option to set the century window as either a fixed window or a sliding window.
5. Compile the program with the `DATEPROC(FLAG)` compiler option, and review the diagnostic messages to see if date processing has produced any unexpected side effects.
6. When the compilation has only information-level diagnostic messages, you can recompile the program with the `DATEPROC(NOFLAG)` compiler option to produce a clean listing.

You can use certain programming techniques to take advantage of date processing and control the effects of using date fields such as when comparing dates, sorting and merging by date, and performing arithmetic operations involving dates. The millennium language extensions support year-first, year-only, and year-last date fields for the most common operations on date fields: comparisons, moving and storing, and incrementing and decrementing.

RELATED CONCEPTS
"Millennium language extensions (MLE)" on page 598

RELATED TASKS
"Resolving date-related logic problems" on page 599
"Using year-first, year-only, and year-last date fields" on page 604
"Manipulating literals as dates" on page 607
"Setting triggers and limits" on page 610
"Sorting and merging by date" on page 612
"Performing arithmetic on date fields" on page 613
"Controlling date processing explicitly" on page 615
"Analyzing and avoiding date-related diagnostic messages" on page 617
"Avoiding problems in processing dates" on page 619

RELATED REFERENCES
"DATEPROC" on page 308
"YEARWINDOW" on page 348
DATE FORMAT clause (*Enterprise COBOL Language Reference*)

Millennium language extensions (MLE)

The term *millennium language extensions* (MLE) refers to the features of Enterprise COBOL that the DATEPROC compiler option activates to help with logic problems that involve dates in the year 2000 and beyond.

When enabled, the extensions include:

- The DATE FORMAT clause. Add this clause to items in the DATA DIVISION to identify date fields and to specify the location of the year component within the date.

  There are several restrictions on use of the DATE FORMAT clause; for example, you cannot specify it for items that have USAGE NATIONAL. See the related references below for details.
- The reinterpretation as a date field of the function return value 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 nondates.
- The intrinsic function YEARWINDOW, which retrieves the starting year of the century window set by the YEARWINDOW compiler option.

The DATEPROC compiler option enables special date-oriented processing of identified date fields. The YEARWINDOW compiler option specifies the 100-year window (the century window) to use for interpreting two-digit windowed years.

RELATED CONCEPTS
"Principles and objectives of these extensions"

RELATED REFERENCES
"DATEPROC" on page 308
"YEARWINDOW" on page 348
Restrictions on using date fields (*Enterprise COBOL Language Reference*)

Principles and objectives of these extensions

To gain the most benefit from the millennium language extensions, you need to understand the reasons for their introduction into the COBOL language.

The millennium language extensions focus on a few key principles:

- Programs to be recompiled with date semantics are fully tested and valuable assets of the enterprise. Their only relevant limitation is that two-digit years in the programs are restricted to the range 1900-1999.

- No special processing is done for the nonyear part of dates. That is why the nonyear part of the supported date formats is denoted by `X`s. To do otherwise might change the meaning of existing programs. The only date-sensitive semantics that are provided involve automatically expanding (and contracting) the two-digit year part of dates with respect to the century window for the program.
- Dates with four-digit year parts are generally of interest only when used in combination with windowed dates. Otherwise there is little difference between four-digit year dates and nondates.

Based on these principles, the millennium language extensions are designed to meet several objectives. You should evaluate the objectives that you need to meet in order to resolve your date-processing problems, and compare them with the objectives of the millennium language extensions, to determine how your application can benefit from them. 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 objectives of the millennium language extensions are as follows:

- Extend the useful life of your application programs as they are currently specified.
- Keep source changes to a minimum, preferably limited to augmenting the declarations of date fields in the `DATA DIVISION`. To implement the century window solution, you should not need to change the program logic in the `PROCEDURE DIVISION`.
- Preserve the existing semantics of the programs when adding date fields. For example, when a date is expressed as a literal, as in the following statement, the literal is considered to be compatible (windowed or expanded) with the date field to which it is compared:

  ```
  If Expiry-Date Greater Than 980101 . . .
  ```

  Because the existing program assumes that two-digit-year dates expressed as literals are in the range 1900-1999, the extensions do not change this assumption.
- The windowing feature is not intended for long-term use. It can extend the useful life of applications as a start toward a long-term solution that can be implemented later.
- The expanded date field feature is intended for long-term use, as an aid for expanding date fields in files and databases.

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.

Resolving date-related logic problems

You can adopt any of three approaches to assist with date-processing problems: use a century window, internal bridging, or full field expansion.

Century window
: You define a century window and specify the fields that contain windowed dates. The compiler then interprets the two-digit years in these data fields according to the century window.

Internal bridging
: If your files and databases have not yet been converted to four-digit-year

dates, but you prefer to use four-digit expanded-year logic in your programs, you can use an internal bridging technique to process the dates as four-digit-year dates.

Full field expansion
: This solution involves explicitly expanding two-digit-year date fields to contain full four-digit years in your files and databases and then using these fields in expanded form in your programs. This is the only method that assures reliable date processing for all applications.

You can use the millennium language extensions with each approach to achieve a solution, but each has advantages and disadvantages, as shown below.

Table 87. **Advantages and disadvantages of Year 2000 solutions**

Aspect	Century window	Internal bridging	Full field expansion
Implementation	Fast and easy but might not suit all applications	Some risk of corrupting data	Must ensure that changes to databases, copybooks, and programs are synchronized
Testing	Less testing is required because no changes to program logic	Testing is easy because changes to program logic are straightforward	
Duration of fix	Programs can function beyond 2000, but not a long-term solution	Programs can function beyond 2000, but not a permanent solution	Permanent solution
Performance	Might degrade performance	Good performance	Best performance
Maintenance			Maintenance is easier.

"Example: century window" on page 601
"Example: internal bridging" on page 602
"Example: converting files to expanded date form" on page 603

RELATED TASKS
"Using a century window"
"Using internal bridging" on page 601
"Moving to full field expansion" on page 602

Using a century window

A *century window* is a 100-year interval, such as 1950-2049, within which any two-digit year is unique. For windowed date fields, you specify the century window start date by using the `YEARWINDOW` compiler option.

When the `DATEPROC` option is in effect, the compiler applies this window to two-digit date fields in the program. For example, with a century window of 1930-2029, COBOL interprets two-digit years 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.

To implement this century window, you use the `DATE FORMAT` clause to identify the date fields in your program and use the `YEARWINDOW` compiler option to define the century window as either a fixed window or a sliding window:

- For a fixed window, specify a four-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(-50)` defines a sliding window that starts 50 years before the year in which the program is running. So if the program is running in 2006, the century window is 1956-2055, and in 2007 it automatically becomes 1957-2056, and so on.

The compiler automatically applies the century window to operations on the date fields that you have identified. You do not need any extra program logic to implement the windowing.

"Example: century window"

RELATED REFERENCES
"DATEPROC" on page 308
"YEARWINDOW" on page 348
DATE FORMAT clause (*Enterprise COBOL Language Reference*)
Restrictions on using date fields (*Enterprise COBOL Language Reference*)

Example: century window

The following example shows (in bold) how to modify a program with the `DATE FORMAT` clause to use the automatic date windowing capability.

```
CBL LIB,QUOTE,NOOPT,DATEPROC(FLAG),YEARWINDOW(-60)
. . .
01  Loan-Record.
    05  Member-Number   Pic X(8).
    05  DVD-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 > Date-Due-Back Then
       Perform Fine-Member.
```

There are no changes to the `PROCEDURE DIVISION`. 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 `060102` (January 2, 2006) and `Date-Returned` contains `051231` (December 31, 2005), `Date-Returned` is less than (earlier than) `Date-Due-Back`, so the program does not perform the `Fine-Member` paragraph. (The program checks whether a DVD was returned on time.)

Using internal bridging

For internal bridging, you need to structure your program appropriately.

Do the following steps:

1. Read the input files with two-digit-year dates.
2. Declare these two-digit dates as windowed date fields and move them to expanded date fields, so that the compiler automatically expands them to four-digit-year dates.
3. In the main body of the program, use the four-digit-year dates for all date processing.
4. Window the dates back to two-digit years.
5. Write the two-digit-year dates to the output files.

This process provides a convenient migration path to a full expanded-date solution, and can have performance advantages over using windowed dates.

When you use this technique, your changes to the program logic are minimal. You simply add statements to expand and contract the dates, and change the statements that refer to dates to use the four-digit-year date fields in WORKING-STORAGE instead of the two-digit-year fields in the records.

Because you are converting the dates back to two-digit years for output, you should allow for the possibility that the year is outside the century window. For example, if a date field contains the year 2020, but the century window is 1920-2019, then the date is outside the window. Simply moving the year to a two-digit-year field will be incorrect. To protect against this problem, you can use a COMPUTE statement to store the date, with the ON SIZE ERROR phrase to detect whether the date is outside the century window.

"Example: internal bridging"

RELATED TASKS
"Using a century window" on page 600
"Performing arithmetic on date fields" on page 613
"Moving to full field expansion"

Example: internal bridging

The following example shows (in bold) how a program can be changed to implement internal bridging.

```
CBL  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.
```

Moving to full field expansion

Using the millennium language extensions, you can move gradually toward a solution that fully expands the date field.

Do the following steps:

1. Apply the century window solution, and use this solution until you have the resources to implement a more permanent solution.

2. Apply the internal bridging solution. This way you can use expanded dates in your programs while your files continue to hold dates in two-digit-year form. You can progress more easily to a full-field-expansion solution because there will be no further changes to the logic in the main body of the programs.
3. Change the file layouts and database definitions to use four-digit-year dates.
4. Change your COBOL copybooks to reflect these four-digit-year date fields.
5. Run a utility program (or special-purpose COBOL program) to copy files from the old format to the new format.
6. Recompile your programs and do regression testing and date testing.

After you have completed the first two steps, you can repeat the remaining steps 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 need to write special-purpose programs to convert your files to expanded-date form.

"Example: converting files to expanded date form"

Example: converting files to expanded date form

The following example shows a simple program that copies from one file to another while expanding the date fields. The record length of the output file is larger than that of the input file because the dates are expanded.

```
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.

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.
```

Notes

(1) The fields DUE-DATE and REMINDER-DATE in the input record are Gregorian dates with two-digit year components. They are defined with a DATE FORMAT clause so that the compiler recognizes them as windowed date fields.

(2) The output record contains the same two fields in expanded date format. They are defined with a DATE FORMAT clause so that the compiler treats them as four-digit-year date fields.

(3) The MOVE CORRESPONDING statement moves each item in INPUT-RECORD to its matching item in OUTPUT-RECORD. When the two windowed date fields are moved to the corresponding expanded date fields, the compiler expands the year values using the current century window.

Using year-first, year-only, and year-last date fields

A *year-first* date field is a date field whose DATE FORMAT specification consists of YY or YYYY, followed by one or more Xs. The date format of a *year-only* date field has just the YY or YYYY. A *year-last* date field is a date field whose DATE FORMAT clause specifies one or more Xs preceding YY or YYYY.

When you compare two date fields of either year-first or year-only types, the two dates must be compatible; that is, they must have the same number of nonyear characters. The number of digits for the year component need not be the same.

Year-last date formats are commonly used to display dates, but are less useful computationally because the year, which is the most significant part of the date, is in the least significant position of the date representation.

If your version of DFSORT (or equivalent) has the appropriate capabilities, year-last dates are supported as windowed keys in SORT or MERGE statements. Apart from sort and merge operations, functional support for year-last date fields is limited to equal or unequal comparisons and certain kinds of assignment. The operands must be either dates with identical (year-last) date formats, or a date and a nondate. The compiler does not provide automatic windowing for operations on year-last dates. When an unsupported usage (such as arithmetic on year-last dates) occurs, the compiler provides an error-level message.

If you need more general date-processing capability for year-last dates, you should isolate and operate on the year part of the date.

"Example: comparing year-first date fields" on page 606

RELATED CONCEPTS
"Compatible dates"

RELATED TASKS
"Sorting and merging by date" on page 612
"Using other date formats" on page 606

Compatible dates

The meaning of the term *compatible dates* depends on whether the usage occurs in the DATA DIVISION or the PROCEDURE DIVISION.

The DATA DIVISION usage deals with the declaration of date fields, and the rules that govern 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.
```

The PROCEDURE DIVISION usage deals with how date fields can be used together in operations such as comparisons, moves, and arithmetic expressions. For year-first and year-only date fields to be considered compatible, date fields must have the same number of nonyear 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.

Year-last date fields must have identical DATE FORMAT clauses. In particular, operations between windowed date fields and expanded year-last date fields are not allowed. For example, you can move a date field that has a date format of XXXXYY to another XXXXYY date field, but not to a date field that has a format of XXXXYYYY.

You can perform operations on date fields, or on a combination of date fields and nondates, provided that the date fields in the operation are compatible. For example, assume the following definitions:

```
01  Date-Gregorian-Win  Pic 9(6) Packed-Decimal Date Format yyxxxx.
01  Date-Julian-Win     Pic 9(5) Packed-Decimal Date Format yyxxx.
01  Date-Gregorian-Exp  Pic 9(8) Packed-Decimal Date Format yyyyxxxx.
```

The following statement is inconsistent because the number of nonyear digits is different between the two fields:

```
If Date-Gregorian-Win Less than Date-Julian-Win . . .
```

The following statement is accepted because the number of nonyear digits is the same for both fields:

```
If Date-Gregorian-Win Less than Date-Gregorian-Exp . . .
```

In this case the century window is applied to the windowed date field (`Date-Gregorian-Win`) to ensure that the comparison is meaningful.

When a nondate is used in conjunction with a date field, the nondate is either assumed to be compatible with the date field or is treated as a simple numeric value.

Example: comparing year-first date fields

The following example shows a windowed date field that is compared with an expanded date field.

```
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 > Todays-Date Then . . .
```

The century window is applied to `Date-Due-Back`. `Todays-Date` must have a `DATE FORMAT` clause to define it as an expanded date field. If it did not, it would be treated as a nondate 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 of 1900-1999, which would create an inconsistent comparison.

Using other date formats

To be eligible for automatic windowing, a date field should contain a two-digit year as the first or only part of the field. The remainder of the field, if present, must contain between one and four characters, but its content is not important.

If there are date fields in your application that do not fit these criteria, you might have to make some code changes to define just the year part of the date as a date field with the `DATE FORMAT` clause. Some examples of these types of date formats are:

- A seven-character field that consists of a two-digit year, three characters that contain an abbreviation of the month, and two digits for the day of the month. This format is not supported because date fields can have only one through four nonyear characters.
- A Gregorian date of the form `DDMMYY`. Automatic windowing is not provided because the year component is not the first part of the date. Year-last dates such as these are fully supported as windowed keys in `SORT` or `MERGE` statements, and are also supported in a limited number of other COBOL operations.

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.

"Example: isolating the year"

Example: isolating the year

The following example shows how you can isolate the year portion of a data field that is in 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 the code above, if Last-Review-Date contains 230105 (January 23, 2005), then Next-Review-Date will contain 230106 (January 23, 2006) after the ADD statement is executed. This is a simple method for setting the next date for an annual review. However, if Last-Review-Date contains 230199, then adding 1 yields 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 Date Format xxxxyy.
    05  Last-R-DDMM  Pic 9(4).
    05  Last-R-YY    Pic 99 Date Format yy.
03  Next-Review-Date Date Format xxxxyy.
    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.
```

Manipulating literals as dates

If a windowed date field has a level-88 condition-name associated with it, the literal in the VALUE clause is windowed against the century window of the compile unit rather than against the assumed century window of 1900-1999.

For example, suppose you have these data definitions:

```
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 are 051220 (representing December 20, 2005), then the first condition below evaluates to true, but the second condition evaluates to false:

```
If Date-Target. . .
If Date-Due = 051220
```

The literal 051220 is treated as a nondate; therefore it is windowed against the assumed century window of 1900-1999, and represents December 20, 1905. But where the literal is specified in the VALUE clause of an level-88 condition-name, the literal 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. The resulting date field will be treated as either a windowed date field or an expanded date field to ensure a consistent comparison. For example, using the above definitions, both of the following conditions evaluate to true:

```
If Date-Due = Function DATEVAL (051220 "YYXXXX")
If Date-Due = Function DATEVAL (20051220 "YYYYXXXX")
```

With a level-88 condition-name, you can specify the THRU option on the VALUE clause, but you must specify a fixed century window on the YEARWINDOW compiler option rather than a sliding window. 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 in the range must be greater than the windowed value of the first item. However, the compiler can verify this difference only if the YEARWINDOW compiler option specifies a fixed century window (for example, YEARWINDOW(1940) rather than YEARWINDOW(-60)).

The windowed order requirement does not apply to year-last date fields. If you specify a condition-name VALUE clause with the THROUGH phrase for a year-last date field, the two literals must follow normal COBOL rules. That is, the first literal must be less than the second literal.

RELATED CONCEPTS
"Assumed century window"
"Treatment of nondates" on page 609

RELATED TASKS
"Controlling date processing explicitly" on page 615

Assumed century window

When a program uses windowed date fields, the compiler applies the century window that is defined by the YEARWINDOW compiler option to the compilation unit. When a windowed date field is used in conjunction with a nondate, and the context demands that the nondate be treated as a windowed date, the compiler uses an assumed century window to resolve the nondate field.

The assumed century window is 1900-1999, which typically is not the same as the century window for the compilation unit.

In many cases, particularly for literal nondates, this assumed century window is the correct choice. In the following construct, 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:

```
01  Manufacturing-Record.
    03  Makers-Date Pic X(6) Date Format yyxxxx.
. . .
    If Makers-Date Greater than "720101" . . .
```

Even if the assumption is correct, it is better to make the year explicit and eliminate the warning-level diagnostic message (which results from applying the assumed century window) by using the DATEVAL intrinsic function:

```
If Makers-Date Greater than
    Function Dateval("19720101" "YYYYXXXX") . . .
```

In some cases, the assumption might not be correct. For the following 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:

```
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 . . .
```

In the example above, the following three conditions need to be true to make Date-Complete earlier than (less than) Date-Target:

- The century window is 1910-2009.
- Date-Complete is 991202 (Gregorian date: December 2, 1999).
- Date-Target is 000115 (Gregorian date: January 15, 2000).

However, because Date-Target does not have a DATE FORMAT clause, it is a nondate. Therefore, the century window applied to it is the assumed century window of 1900-1999, and 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") . . .
```

RELATED TASKS
"Controlling date processing explicitly" on page 615

Treatment of nondates

How the compiler treats a nondate depends upon its context.

The following items are nondates:

- A literal value.
- 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 nondate, whereas the sum of a date field and a nondate is a date field.
- The output from the UNDATE intrinsic function.

When you use a nondate in conjunction with a date field, the compiler interprets the nondate either as a date whose format is compatible with the date field or as a simple numeric value. This interpretation depends on the context in which the date field and nondate are used, as follows:

- Comparison

  When a date field is compared with a nondate, the nondate is considered to be compatible with the date field in the number of year and nonyear characters. In the following example, the nondate literal 971231 is compared with a windowed date field:

  ```
  01  Date-1    Pic 9(6) Date Format yyxxxx.
  . . .
      If Date-1 Greater than 971231 . . .
  ```

  The nondate literal 971231 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, nondate fields are treated as simple numeric values. In the following example, the numeric value 10000 is added to the Gregorian date in Date-2, effectively adding one year to the date:

```
01  Date-2    Pic 9(6) Date Format yyxxxx.
. . .
    Add 10000 to Date-2.
```

- MOVE statement

  Moving a date field to a nondate is not supported. However, you can use the UNDATE intrinsic function to do this.

  When you move a nondate to a date field, the sending field is assumed to be compatible with the receiving field in the number of year and nonyear characters. For example, when you move a nondate to a windowed date field, the nondate field is assumed to contain a compatible date with a two-digit year.

Setting triggers and limits

Triggers and limits are special values that never match valid dates because either their value is nonnumeric or the nonyear part of the value cannot occur in an actual date. Triggers and limits are recognized in date fields and also in nondates used in combination with date fields.

Type of date field	Special value
Alphanumeric windowed date or year fields	HIGH-VALUE, LOW-VALUE, and SPACE
Alphanumeric and numeric windowed date fields with at least one X in the DATE FORMAT clause (that is, date fields other than just a year)	All nines or all zeros

The difference between a trigger and a limit is not in the particular value, but in the way you use it. You can use any of the special values as either a trigger or a limit.

When used as triggers, special values can indicate a specific condition such as "date not initialized" or "account past due." When used as limits, special values are intended to act as dates earlier or later than any valid date. LOW-VALUE, SPACE and zeros are lower limits; HIGH-VALUE and nines are upper limits.

You activate trigger and limit support by specifying the TRIG suboption of the DATEPROC compiler option. If the DATEPROC(TRIG) compiler option is in effect, automatic expansion of windowed date fields (before their use as operands in comparisons, arithmetic, and so on) is sensitive to these special values.

The DATEPROC(TRIG) option results in slower-performing code when windowed dates are compared. The DATEPROC(NOTRIG) option is a performance option that assumes valid date values in all windowed date fields.

When an actual or assumed windowed date field contains a trigger, the compiler expands the trigger as if the value were propagated to the century part of the expanded date result, rather than inferring 19 or 20 as the century value as in normal windowing. In this way, your application can test for special values or use them as upper or lower date limits. Specifying DATEPROC(TRIG) also enables SORT and MERGE statement support of the DFSORT special indicators, which correspond to triggers and limits.

"Example: using limits"

RELATED TASKS
"Using sign conditions"

RELATED REFERENCES
"DATEPROC" on page 308

Example: using limits

This example shows how you can use an expiration date field to hold either a normal expiration date or else a high limit that allows an "everlasting" subscription.

Suppose that your application checks subscriptions for expiration, but you want some subscriptions to last indefinitely. Consider the following code fragment:

```
Process Dateproc(Flag,Trig). . .
. . .
01 SubscriptionRecord.
   03 ExpirationDate  PIC 9(6) Date Format yyxxxx.
. . .
77 TodaysDate  Pic 9(6) Date Format yyxxxx.
. . .
   If TodaysDate >= ExpirationDate
      Perform SubscriptionExpired
```

Suppose that the application encounters the following values:

- Today's date is January 4, 2006, represented in `TodaysDate` as 060104.
- One subscription record has a normal expiration date of December 31, 1999, represented in `ExpirationDate` as 991231.
- Another subscription record has a special expiration date coded in `ExpirationDate` as 999999.

Because both dates are windowed, the first subscription is tested as if 20060104 were compared with 19991231, and so the test succeeds. However, when the compiler detects the special value, it uses trigger expansion instead of windowing. Therefore, the test proceeds as if 20060104 were compared with 99999999. This test will always fail.

Using sign conditions

Some applications use special values such as zeros in date fields to act as a trigger, that is, to signify that some special processing is required.

For example, in an Orders file, a value of zero in `Order-Date` might signify that the record is a customer totals record rather than an order record. The program compares the date to zero, as follows:

```
01  Order-Record.
    05  Order-Date      Pic S9(5) Comp-3 Date Format yyxxx.
. . .
    If Order-Date Equal Zero Then . . .
```

However, if you are compiling with the `NOTRIG` suboption of the `DATEPROC` compiler option, this comparison is not valid because the literal value `Zero` is a nondate, and is therefore windowed against the assumed century window to give a value of 1900000.

Alternatively, you can use a sign condition instead of a literal comparison as follows. With a sign condition, `Order-Date` is treated as a nondate, and the century window is not considered.

```
If Order-Date Is Zero Then . . .
```

This approach applies only if the operand in the sign condition is a simple identifier rather than an arithmetic expression. If an expression is specified, the expression is evaluated first, with the century window being applied where appropriate. The sign condition is then compared with the results of the expression.

You could use the `UNDATE` intrinsic function instead or the `TRIG` suboption of the `DATEPROC` compiler option to achieve the same result.

RELATED CONCEPTS
"Treatment of nondates" on page 609

RELATED TASKS
"Setting triggers and limits" on page 610
"Controlling date processing explicitly" on page 615

RELATED REFERENCES
"DATEPROC" on page 308

Sorting and merging by date

If your sort product supports the `Y2PAST` option and the windowed year identifiers (`Y2B`, `Y2C`, `Y2D`, `Y2S`, and `Y2Z`), you can perform sort and merge operations using windowed date fields as sort keys. Virtually all date fields that can be specified with a `DATE FORMAT` clause are supported, including binary year fields and year-last date fields.

The fields are sorted in windowed year sequence according to the century window that you specify in the `YEARWINDOW` compiler option. If your sort product also supports the date field identifiers `Y2T`, `Y2U`, `Y2W`, `Y2X`, and `Y2Y`, you can use the `TRIG` suboption of the `DATEPROC` compiler option.

The special indicators that DFSORT recognizes match exactly those supported by COBOL: `LOW-VALUE`, `HIGH-VALUE`, and `SPACE` for alphanumeric date or year fields, and all zeros and all nines for numeric and alphanumeric date fields that have at least one nonyear digit.

DFSORT is the IBM licensed program for sorting and merging. Wherever DFSORT is mentioned here, you can use any equivalent product.

"Example: sorting by date and time" on page 613

RELATED TASKS
"Sorting on windowed date fields" on page 223
OPTION control statement (Y2PAST) (*DFSORT Application Programming Guide*)

RELATED REFERENCES
"DATEPROC" on page 308
"YEARWINDOW" on page 348
Restrictions on using date fields (*Enterprise COBOL Language Reference*)

Example: sorting by date and time

The following example shows a transaction file that has the transaction records sorted by date and time within account number. Trans-Date is a windowed Julian date field.

```
SD  Transaction-File
    Record Contains 29 Characters
    Data Record is Transaction-Record
01  Transaction-Record.
    05  Trans-Account  PIC 9(8).
    05  Trans-Type     PIC X.
    05  Trans-Date     PIC 9(5) Date Format yyxxx.
    05  Trans-Time     PIC 9(6).
    05  Trans-Amount   PIC 9(7)V99.
. . .
    Sort Transaction-File
         On Ascending Key  Trans-Account
                           Trans-Date
                           Trans-Time
         Using Input-File
         Giving Sorted-File.
```

COBOL passes the relevant information to DFSORT for it to perform the sort. In addition to the information COBOL always passes to DFSORT, COBOL also passes the following information, which DFSORT also uses:

- Century window as the Y2PAST sort option
- Windowed year field and date format of Trans-Date

Performing arithmetic on date fields

You can perform arithmetic operations on numeric date fields in the same manner as on any numeric data item. 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 and statements. Arithmetic operations that include date fields are restricted to:

- Adding a nondate to a date field
- Subtracting a nondate from a date field
- Subtracting a date field from a compatible date field to give a nondate result

The following arithmetic operations are not allowed:

- Any operation between incompatible date fields
- Adding two date fields
- Subtracting a date field from a nondate
- Unary minus applied to a date field
- Multiplication, division, or exponentiation of or by a date field
- Arithmetic expressions that specify a year-last date field
- Arithmetic expressions that specify a year-last date field, except as a receiving data item when the sending field is a nondate

Date semantics are provided for the year parts of date fields but not for the nonyear parts. For example, adding 1 to a windowed Gregorian date field that contains the value 980831 gives a result of 980832, not 980901.

RELATED TASKS
"Allowing for overflow from windowed date fields"
"Specifying the order of evaluation" on page 615

Allowing for overflow from windowed date fields

A (nonyear-last) windowed date field that participates in an arithmetic operation 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.

```
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.
```

In the example above, 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 result is stored in `Next-Review-Year` as 08.

However, the following statement would give a result of 2018:

```
Add 20 to Last-Review-Year Giving Next-Review-Year.
```

This result falls outside the range of the century window. If the result is stored in `Next-Review-Year`, it will be incorrect because later references to `Next-Review-Year` will 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:

- 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 consideration is important when you use internal bridging. When you contract a four-digit-year date field back to two digits to write it to the output file, you need to ensure that the date falls within the century window. Then the two-digit-year date will be represented correctly in the field.

To ensure appropriate calculations, use 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 Perform CenturyWindowOverflow.
```

`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.

If the `DATEPROC(TRIG)` compiler option is in effect, trigger values of zeros or nines in the result also cause the `SIZE ERROR` condition, even though the year part of the result (00 or 99, respectively) falls within the century window.

RELATED TASKS
"Using internal bridging" on page 601

Specifying the order of evaluation

Because of the restrictions on date fields in arithmetic expressions, you might find that programs that previously compiled successfully now produce diagnostic messages when some of the data items are changed to date fields.

```
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 the example above, the first arithmetic expression evaluated is:

```
Start-Year-2 + End-Year-1
```

However, the addition of two date fields is not permitted. To resolve these date fields, you should use parentheses to isolate the 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
```

The subtraction of one date field from another is permitted and gives a nondate result. This nondate result is then added to the date field `End-Year-1`, giving a date field result that is stored in `End-Year-2`.

Controlling date processing explicitly

There might 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 might contain two-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 recognized only by its true binary contents without further interpretation, the date in that field cannot be windowed. Such date fields include:

- A key in a VSAM file
- A search field in a database system such as DB2
- A key field in a CICS command

Conversely, there might be times when you want a date field to be treated as a nondate in specific parts of the program.

COBOL provides two intrinsic functions to deal with these conditions:

DATEVAL
: Converts a nondate to a date field

UNDATE Converts a date field to a nondate

RELATED TASKS
"Using DATEVAL" on page 616
"Using UNDATE" on page 616

Using DATEVAL

You can use the DATEVAL intrinsic function to convert a nondate to a date field, so that COBOL will apply the relevant date processing to the field.

The first argument in the function is the nondate 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.

In most cases, the compiler makes the correct assumption about the interpretation of a nondate but accompanies this assumption with a warning-level diagnostic message. This message typically happens when a windowed date is compared with 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.
```

“Example: DATEVAL”

Using UNDATE

You can use the UNDATE intrinsic function to convert a date field to a nondate so that it can be referenced without any date processing.

Attention: Avoid using UNDATE except as a last resort, because the compiler will lose the flow of date fields in your program. This problem could result in date comparisons not being windowed properly.

Use more DATE FORMAT clauses instead of function UNDATE for MOVE and COMPUTE.

“Example: UNDATE” on page 617

Example: DATEVAL

This example shows a case where it is better to leave a field as a nondate, and use the DATEVAL intrinsic function in a comparison statement.

Assume that a field Date-Copied is referenced many times in a program, but that most of the references just move the value between records or reformat it for printing. Only one reference relies on it to contain a date (for comparison with another date). In this case, it is better to leave the field as a nondate, 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, DATEVAL converts Date-Copied to a date field so that the comparison will be meaningful.

RELATED REFERENCES
DATEVAL (*Enterprise COBOL Language Reference*)

Example: UNDATE

The following example shows a case where you might want to convert a date field to a nondate.

The field Invoice-Date is a windowed Julian date. In some records, it contains the value 00999 to indicate that the record is not a true invoice record, but instead contains file-control information.

Invoice-Date has a DATE FORMAT clause because most of its references in the program are date-specific. However, when it is checked for the existence of a control record, the value 00 in the year component will lead to some confusion. A year value of 00 in Invoice-Date could represent either 1900 or 2000, depending on the century window. This is compared with a nondate (the literal 00999 in the example), which will always be windowed against the assumed century window and therefore always represents the year 1900.

To ensure a consistent comparison, you should use the UNDATE intrinsic function to convert Invoice-Date to a nondate. Therefore, if the IF statement is not comparing date fields, it does not need to apply windowing. For example:

```
01  Invoice-Record.
    03  Invoice-Date    Pic x(5) Date Format yyxxx.
. . .
    If FUNCTION UNDATE(Invoice-Date) Equal "00999" . . .
```

RELATED REFERENCES
UNDATE (*Enterprise COBOL Language Reference*)

Analyzing and avoiding date-related diagnostic 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.

As with all compiler-generated messages, each date-related message has one of the following severity levels:

- Information-level, to draw your attention to the definition or use of a date field.
- Warning-level, to indicate that the compiler has had to make an assumption about a date field or nondate because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.
- Error-level, to indicate that the usage of the date field is incorrect. Compilation continues, but runtime results are unpredictable.
- Severe-level, to indicate that the usage of the date field is incorrect. The statement that caused this error is discarded from the compilation.

The easiest way to use the MLE messages is to compile with a FLAG option setting that embeds the messages in the source listing after the line to which the messages refer. You can choose to see all MLE messages or just certain severities.

To see all MLE messages, specify the FLAG(I,I) and DATEPROC(FLAG) compiler options. Initially, you might want to see all of the messages to understand how

MLE is processing the date fields in your program. For example, if you want to do a static analysis of the date usage in a program by using the compile listing, use FLAG (I,I).

However, it is recommended that you specify FLAG(W,W) for MLE-specific compiles. You must resolve all severe-level (S-level) error messages, and you should resolve all error-level (E-level) messages as well. For the warning-level (W-level) messages, you need to examine each message and use the following guidelines to either eliminate the message or, for unavoidable messages, ensure that the compiler makes correct assumptions:

- The diagnostic messages might indicate some date data items that should have had a DATE FORMAT clause. Either add DATE FORMAT clauses to these items or use the DATEVAL intrinsic function in references to them.
- Pay particular attention to literals in relation conditions that involve date fields or in arithmetic expressions that include date fields. You can use the DATEVAL function on literals (as well as nondate data items) to specify a DATE FORMAT pattern to be used. As a last resort, you can use the UNDATE function to enable a date field to be used in a context where you do not want date-oriented behavior.
- With the REDEFINES and RENAMES clauses, the compiler might produce a warning-level diagnostic message if a date field and a nondate occupy the same storage location. You should check these cases carefully to confirm that all uses of the aliased data items are correct, and that none of the perceived nondate redefinitions actually is a date or can adversely affect the date logic in the program.

In some cases, a the W-level message might be acceptable, but you might want to change the code to get a compile with a return code of zero.

To avoid warning-level diagnostic messages, follow these guidelines:

- Add DATE FORMAT clauses to any data items that will contain dates, even if the
| items are not used in comparisons. But see the related references below about
| restrictions on using date fields. For example, you cannot use the DATE FORMAT
| clause on a data item that is described implicitly or explicitly as USAGE NATIONAL.
- Do not specify a date field in a context where a date field does not make sense, such as a FILE STATUS, PASSWORD, ASSIGN USING, LABEL RECORD, or LINAGE item. If you do, you will get a warning-level message and the date field will be treated as a nondate.
- Ensure that implicit or explicit aliases for date fields are compatible, such as in a group item that consists solely of a date field.
- 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 nondate treated as a date field, such as when moving a nondate to a date field or when comparing a windowed date with a nondate and you want a windowed date comparison. If you do not use DATEVAL, the compiler will make an assumption about the use of the nondate and produce a warning-level diagnostic message. Even if the assumption is correct, you might want to use DATEVAL to eliminate the message.
- Use the UNDATE intrinsic function if you want a date field treated as a nondate, such as moving a date field to a nondate, or comparing a nondate and a windowed date field when you do not want a windowed comparison.

RELATED TASKS
"Controlling date processing explicitly" on page 615
Analyzing date-related diagnostic messages (*COBOL Millennium Language Extensions Guide*)

RELATED REFERENCES
Restrictions on using date fields (*Enterprise COBOL Language Reference*)

Avoiding problems in processing dates

When you change a COBOL program to use the millennium language extensions, you might find that some parts of the program need special attention to resolve unforeseen changes in behavior. For example, you might need to avoid problems with packed-decimal fields and problems that occur if you move from expanded to windowed date fields.

RELATED TASKS
"Avoiding problems with packed-decimal fields"
"Moving from expanded to windowed date fields"

Avoiding problems with packed-decimal fields

`COMPUTATIONAL-3` fields (packed-decimal format) are often defined as having an odd number of digits even if the field will not be used to hold a number of that magnitude. The internal representation of packed-decimal numbers always allows for an odd number of digits.

A field that holds a six-digit Gregorian date, for example, can be declared as `PIC S9(6) COMP-3`. This declaration will reserve 4 bytes of storage. But a programmer might have declared the field as `PIC S9(7)`, knowing that this would reserve 4 bytes with the high-order digit always containing a zero.

If you add a `DATE FORMAT YYXXXX` clause to this field, the compiler will issue a diagnostic message because the number of digits in the `PICTURE` clause does not match the size of the date format specification. In this case, you need to carefully check each use of the field. If the high-order digit is never used, you can simply change the field definition to `PIC S9(6)`. If it is used (for example, if the same field can hold a value other than a date), you need to take some other action, such as:

- Using a `REDEFINES` clause to define the field as both a date and a nondate (this usage will also produce a warning-level diagnostic message)
- Defining another `WORKING-STORAGE` field to hold the date, and moving the numeric field to the new field
- Not adding a `DATE FORMAT` clause to the data item, and using the `DATEVAL` intrinsic function when referring to it as a date field

Moving from expanded to windowed date fields

When you move an expanded alphanumeric date field to a windowed date field, the move does not follow the normal COBOL conventions for alphanumeric moves. When both the sending and receiving fields are date fields, the move is right justified, not left justified as normal. For an expanded-to-windowed (contracting) move, the leading two digits of the year are truncated.

Depending on the contents of the sending field, the results of such a move might be incorrect. For example:

```
77  Year-Of-Birth-Exp  Pic x(4) Date Format yyyy.
77  Year-Of-Birth-Win  Pic xx   Date Format yy.
. . .
    Move Year-Of-Birth-Exp to Year-Of-Birth-Win.
```

If `Year-Of-Birth-Exp` contains '1925', `Year-Of-Birth-Win` will contain '25'. However, if the century window is 1930-2029, subsequent references to `Year-Of-Birth-Win` will treat it as 2025, which is incorrect.

Part 8. Improving performance and productivity

Chapter 34. Tuning your program

When a program is comprehensible, you can assess its performance. A program that has a tangled control flow is difficult to understand and maintain. The tangled control flow also inhibits the optimization of the code.

Therefore, before you try to improve the performance directly, you need to assess certain aspects of your program:

1. Examine the underlying algorithms for your program. For top performance, a sound algorithm is essential. For example, a sophisticated algorithm for sorting a million items can be hundreds of thousands times faster than a simple algorithm.
2. Look at the data structures. They should be appropriate for the algorithm. When your program frequently accesses data, reduce the number of steps needed to access the data wherever possible.
3. After you have improved the algorithm and data structures, look at other details of the COBOL source code that affect performance.

You can write programs that result in better generated code sequences and use system services better. These areas affect program performance:

- Coding techniques. These include using a programming style that helps the optimizer, choosing efficient data types, and handling tables efficiently.
- Optimization. You can optimize your code by using the `OPTIMIZE` compiler option.
- Compiler options and `USE FOR DEBUGGING ON ALL PROCEDURES`. Certain compiler options and language affect the efficiency of your program.
- Runtime environment. Carefully consider your choice of runtime options and other runtime considerations that control how your compiled program runs.
- Running under CICS, IMS, or using VSAM. Various tips can help make these programs run efficiently.

RELATED CONCEPTS
"Optimization" on page 631
Enterprise COBOL Version 3 Performance Tuning

RELATED TASKS
"Using an optimal programming style" on page 624
"Choosing efficient data types" on page 625
"Handling tables efficiently" on page 627
"Optimizing your code" on page 630
"Choosing compiler features to enhance performance" on page 633
"Running efficiently with CICS, IMS, or VSAM" on page 637
Specifying run-time options (*Language Environment Programming Guide*)

RELATED REFERENCES
"Performance-related compiler options" on page 633
Storage performance considerations (*Language Environment Programming Guide*)

Using an optimal programming style

The coding style you use can affect how the optimizer handles your code. You can improve optimization by using structured programming techniques, factoring expressions, using symbolic constants, and grouping constant and duplicate computations.

RELATED TASKS
"Using structured programming"
"Factoring expressions"
"Using symbolic constants"
"Grouping constant computations" on page 625
"Grouping duplicate computations" on page 625

Using structured programming

Using structured programming statements, such as EVALUATE and inline PERFORM, makes your program more comprehensible and generates a more linear control flow. As a result, the optimizer can operate over larger regions of the program, which gives you more efficient code.

Use top-down programming constructs. Out-of-line PERFORM statements are a natural means of doing top-down programming. Out-of-line PERFORM statements can often be as efficient as inline PERFORM statements, because the optimizer can simplify or remove the linkage code.

Avoid using the following constructs:
- ALTER statement
- Backward branches (except as needed for loops for which PERFORM is unsuitable)
- PERFORM procedures that involve irregular control flow (such as preventing control from passing to the end of the procedure and returning to the PERFORM statement)

Factoring expressions

By factoring expressions in your programs, you can potentially eliminate a lot of unnecessary computation.

For example, the first block of code below is more efficient than the second block of code:

```
MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
  COMPUTE TOTAL = TOTAL + ITEM(I)
END-PERFORM
COMPUTE TOTAL = TOTAL * DISCOUNT
```

```
MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
  COMPUTE TOTAL = TOTAL + ITEM(I) * DISCOUNT
END-PERFORM
```

The optimizer does not factor expressions.

Using symbolic constants

To have the optimizer recognize a data item as a constant throughout the program, initialize it with a VALUE clause and do not change it anywhere in the program.

If you pass a data item to a subprogram BY REFERENCE, the optimizer treats it as an external data item and assumes that it is changed at every subprogram call.

If you move a literal to a data item, the optimizer recognizes the data item as a constant only in a limited area of the program after the MOVE statement.

Grouping constant computations

When several items in an expression are constant, ensure that the optimizer is able to optimize them. The compiler is bound by the left-to-right evaluation rules of COBOL. Therefore, either move all the constants to the left side of the expression or group them inside parentheses.

For example, if V1, V2, and V3 are variables and C1, C2, and C3 are constants, the expressions on the left below are preferable to the corresponding expressions on the right:

More efficient	Less efficient
`V1 * V2 * V3 * (C1 * C2 * C3)`	`V1 * V2 * V3 * C1 * C2 * C3`
`C1 + C2 + C3 + V1 + V2 + V3`	`V1 + C1 + V2 + C2 + V3 + C3`

In production programming, there is often a tendency to place constant factors on the right-hand side of expressions. However, such placement can result in less efficient code because optimization is lost.

Grouping duplicate computations

When components of different expressions are duplicates, ensure that the compiler is able to optimize them. For arithmetic expressions, the compiler is bound by the left-to-right evaluation rules of COBOL. Therefore, either move all the duplicates to the left side of the expressions or group them inside parentheses.

If V1 through V5 are variables, the computation V2 * V3 * V4 is a duplicate (known as a common subexpression) in the following two statements:

```
COMPUTE A = V1 * (V2 * V3 * V4)
COMPUTE B = V2 * V3 * V4 * V5
```

In the following example, V2 + V3 is a common subexpression:

```
COMPUTE C = V1 + (V2 + V3)
COMPUTE D = V2 + V3 + V4
```

In the following example, there is no common subexpression:

```
COMPUTE A = V1 * V2 * V3 * V4
COMPUTE B = V2 * V3 * V4 * V5
COMPUTE C = V1 + (V2 + V3)
COMPUTE D = V4 + V2 + V3
```

The optimizer can eliminate duplicate computations. You do not need to introduce artificial temporary computations; a program is often more comprehensible without them.

Choosing efficient data types

Choosing the appropriate data type and PICTURE clause can produce more efficient code, as can avoiding USAGE DISPLAY and USAGE NATIONAL data items in areas that are heavily used for computations.

Consistent data types can reduce the need for conversions during operations on data items. You can also improve program performance by carefully determining when to use fixed-point and floating-point data types.

RELATED CONCEPTS
"Formats for numeric data" on page 49

RELATED TASKS
"Choosing efficient computational data items"
"Using consistent data types"
"Making arithmetic expressions efficient" on page 627
"Making exponentiations efficient" on page 627

Choosing efficient computational data items

When you use a data item mainly for arithmetic or as a subscript, code `USAGE BINARY` on the data description entry for the item. The operations for manipulating binary data are faster than those for manipulating decimal data.

However, if a fixed-point arithmetic statement has intermediate results with a large precision (number of significant digits), the compiler uses decimal arithmetic anyway, after converting the operands to packed-decimal form. For fixed-point arithmetic statements, the compiler normally uses binary arithmetic for simple computations with binary operands if the precision is eight or fewer digits. Above 18 digits, the compiler always uses decimal arithmetic. With a precision of nine to 18 digits, the compiler uses either form.

To produce the most efficient code for a `BINARY` data item, ensure that it has:

- A sign (an `S` in its `PICTURE` clause)
- Eight or fewer digits

| For a data item that is larger than eight digits or is used with `DISPLAY` or `NATIONAL` data items, use `PACKED-DECIMAL`. The code generated for `PACKED-DECIMAL` data items can be as fast as that for `BINARY` data items in some cases, especially if the statement is complicated or specifies rounding.

To produce the most efficient code for a `PACKED-DECIMAL` data item, ensure that it has:

- A sign (an `S` in its `PICTURE` clause)
- An odd number of digits (9s in the `PICTURE` clause), so that it occupies an exact number of bytes without a half byte left over
- 15 or fewer digits in the `PICTURE` specification to avoid using library routines for multiplication and division

Using consistent data types

In operations on operands of different types, one of the operands must be converted to the same type as the other. Each conversion requires several instructions. For example, one of the operands might need to be scaled to give it the appropriate number of decimal places.

You can largely avoid conversions by using consistent data types and by giving both operands the same usage and also appropriate `PICTURE` specifications. That is, you should ensure that two numbers to be compared, added, or subtracted not only have the same usage but also the same number of decimal places (9s after the `V` in the `PICTURE` clause).

Making arithmetic expressions efficient

Computation of arithmetic expressions that are evaluated in floating point is most efficient when the operands need little or no conversion. Use operands that are COMP-1 or COMP-2 to produce the most efficient code.

Declare integer items as BINARY or PACKED-DECIMAL with nine or fewer digits to afford quick conversion to floating-point data. Also, conversion from a COMP-1 or COMP-2 item to a fixed-point integer with nine or fewer digits, without SIZE ERROR in effect, is efficient when the value of the COMP-1 or COMP-2 item is less than 1,000,000,000.

Making exponentiations efficient

Use floating point for exponentiations for large exponents to achieve faster evaluation and more accurate results.

For example, the first statement below is computed more quickly and accurately than the second statement:

```
COMPUTE fixed-point1 = fixed-point2 ** 100000.E+00

COMPUTE fixed-point1 = fixed-point2 ** 100000
```

A floating-point exponent causes floating-point arithmetic to be used to compute the exponentiation.

Handling tables efficiently

You can use several techniques to improve the efficiency of table-handling operations, and to influence the optimizer. The return for your efforts can be significant, particularly when table-handling operations are a major part of an application.

The following two guidelines affect your choice of how to refer to table elements:

- Use indexing rather than subscripting.

  Although the compiler can eliminate duplicate indexes and subscripts, the original reference to a table element is more efficient with indexes (even if the subscripts were BINARY). The value of an index has the element size factored into it, whereas the value of a subscript must be multiplied by the element size when the subscript is used. The index already contains the displacement from the start of the table, and this value does not have to be calculated at run time. However, subscripting might be easier to understand and maintain.
- Use relative indexing.

  Relative index references (that is, references in which an unsigned numeric literal is added to or subtracted from the index-name) are executed at least as fast as direct index references, and sometimes faster. There is no merit in keeping alternative indexes with the offset factored in.

Whether you use indexes or subscripts, the following coding guidelines can help you get better performance:

- Put constant and duplicate indexes or subscripts on the left.

  You can reduce or eliminate runtime computations this way. Even when all the indexes or subscripts are variable, try to use your tables so that the rightmost subscript varies most often for references that occur close to each other in the program. This practice also improves the pattern of storage references and also paging. If all the indexes or subscripts are duplicates, then the entire index or subscript computation is a common subexpression.

- Specify the element length so that it matches that of related tables.

  When you index or subscript tables, it is most efficient if all the tables have the same element length. That way, the stride for the last dimension of the tables is the same, and the optimizer can reuse the rightmost index or subscript computed for one table. If both the element lengths and the number of occurrences in each dimension are equal, then the strides for dimensions other than the last are also equal, resulting in greater commonality between their subscript computations. The optimizer can then reuse indexes or subscripts other than the rightmost.
- Avoid errors in references by coding index and subscript checks into your program.

  If you need to validate indexes and subscripts, it might be faster to code your own checks than to use the `SSRANGE` compiler option.

You can also improve the efficiency of tables by using these guidelines:

- Use binary data items for all subscripts.

  When you use subscripts to address a table, use a `BINARY` signed data item with eight or fewer digits. In some cases, using four or fewer digits for the data item might also improve processing time.
- Use binary data items for variable-length table items.

  For tables with variable-length items, you can improve the code for `OCCURS DEPENDING ON` (ODO). To avoid unnecessary conversions each time the variable-length items are referenced, specify `BINARY` for `OCCURS . . . DEPENDING ON` objects.
- Use fixed-length data items whenever possible.

  Copying variable-length data items into a fixed-length data item before a period of high-frequency use can reduce some of the overhead associated with using variable-length data items.
- Organize tables according to the type of search method used.

  If the table is searched sequentially, put the data values most likely to satisfy the search criteria at the beginning of the table. If the table is searched using a binary search algorithm, put the data values in the table sorted alphabetically on the search key field.

RELATED CONCEPTS
"Optimization of table references"

RELATED TASKS
"Referring to an item in a table" on page 72
"Choosing efficient data types" on page 625

RELATED REFERENCES
"SSRANGE" on page 337

Optimization of table references

The COBOL compiler optimizes table references in several ways.

For the table element reference `ELEMENT(S1 S2 S3)`, where `S1`, `S2`, and `S3` are subscripts, the compiler evaluates the following expression:

```
comp_s1 * d1 + comp_s2 * d2 + comp_s3 * d3 + base_address
```

Here comp_s1 is the value of S1 after conversion to binary, comp-s2 is the value of S2 after conversion to binary, and so on. The strides for each dimension are d1, d2, and d3. The *stride* of a given dimension is the distance in bytes between table elements whose occurrence numbers in that dimension differ by 1 and whose other occurrence numbers are equal. For example, the stride d2 of the second dimension in the above example is the distance in bytes between ELEMENT(S1 1 S3) and ELEMENT(S1 2 S3).

Index computations are similar to subscript computations, except that no multiplication needs to be done. Index values have the stride factored into them. They involve loading the indexes into registers, and these data transfers can be optimized, much as the individual subscript computation terms are optimized.

Because the compiler evaluates expressions from left to right, the optimizer finds the most opportunities to eliminate computations when the constant or duplicate subscripts are the leftmost.

Optimization of constant and variable items

Assume that C1, C2, . . . are constant data items and that V1, V2, . . . are variable data items. Then, for the table element reference ELEMENT(V1 C1 C2) the compiler can eliminate only the individual terms comp_c1 * d2 and comp_c2 * d3 as constant from the expression:

```
comp_v1 * d1 + comp_c1 * d2 + comp_c2 * d3 + base_address
```

However, for the table element reference ELEMENT(C1 C2 V1) the compiler can eliminate the entire subexpression comp_c1 * d1 + comp_c2 * d2 as constant from the expression:

```
comp_c1 * d1 + comp_c2 * d2 + comp_v1 * d3 + base_address
```

In the table element reference ELEMENT(C1 C2 C3), all the subscripts are constant, and so no subscript computation is done at run time. The expression is:

```
comp_c1 * d1 + comp_c2 * d2 + comp_c3 * d3 + base_address
```

With the optimizer, this reference can be as efficient as a reference to a scalar (nontable) item.

Optimization of duplicate items

In the table element references ELEMENT(V1 V3 V4) and ELEMENT(V2 V3 V4) only the individual terms comp_v3 * d2 and comp_v4 * d3 are common subexpressions in the expressions needed to reference the table elements:

```
comp_v1 * d1 + comp_v3 * d2 + comp_v4 * d3 + base_address
comp_v2 * d1 + comp_v3 * d2 + comp_v4 * d3 + base_address
```

However, for the two table element references ELEMENT(V1 V2 V3) and ELEMENT(V1 V2 V4) the entire subexpression comp_v1 * d1 + comp_v2 * d2 is common between the two expressions needed to reference the table elements:

```
comp_v1 * d1 + comp_v2 * d2 + comp_v3 * d3 + base_address
comp_v1 * d1 + comp_v2 * d2 + comp_v4 * d3 + base_address
```

In the two references ELEMENT(V1 V2 V3) and ELEMENT(V1 V2 V3), the expressions are the same:

```
comp_v1 * d1 + comp_v2 * d2 + comp_v3 * d3 + base_address
comp_v1 * d1 + comp_v2 * d2 + comp_v3 * d3 + base_address
```

With the optimizer, the second (and any subsequent) reference to the same element can be as efficient as a reference to a scalar (nontable) item.

Optimization of variable-length items

A group item that contains a subordinate `OCCURS DEPENDING ON` data item has a variable length. The program must perform special code every time a variable-length data item is referenced.

Because this code is out-of-line, it might interrupt optimization. Furthermore, the code to manipulate variable-length data items is much less efficient than that for fixed-size data items and can significantly increase processing time. For instance, the code to compare or move a variable-length data item might involve calling a library routine and is much slower than the same code for fixed-length data items.

Comparison of direct and relative indexing

Relative index references are as fast as or faster than direct index references.

The direct indexing in `ELEMENT (I5, J3, K2)` requires this preprocessing:

```
SET I5 TO I
SET I5 UP BY 5
SET J3 TO J
SET J3 DOWN BY 3
SET K2 TO K
SET K2 UP BY 2
```

This processing makes the direct indexing less efficient than the relative indexing in `ELEMENT (I + 5, J - 3, K + 2)`.

RELATED CONCEPTS
"Optimization" on page 631

RELATED TASKS
"Handling tables efficiently" on page 627

Optimizing your code

When your program is ready for final testing, specify the `OPTIMIZE` compiler option so that the tested code and the production code are identical.

You might also want to use this compiler option during development if a program is used frequently without recompilation. However, the overhead for `OPTIMIZE` might outweigh its benefits if you recompile frequently, unless you are using the assembler language expansion (`LIST` compiler option) to fine-tune the program.

For unit-testing a program, you will probably find it easier to debug code that has not been optimized.

To see how the optimizer works on a program, compile it with and without the `OPTIMIZE` option and compare the generated code. (Use the `LIST` compiler option to request the assembler listing of the generated code.)

RELATED CONCEPTS
"Optimization" on page 631

RELATED REFERENCES
"LIST" on page 319
"OPTIMIZE" on page 327

Optimization

To improve the efficiency of the generated code, you can use the OPTIMIZE compiler option.

OPTIMIZE causes the COBOL optimizer to do the following optimizations:

- Eliminate unnecessary transfers of control and inefficient branches, including those generated by the compiler that are not evident from looking at the source program.
- Simplify the compiled code for both a PERFORM statement and a CALL statement to a contained (nested) program. Where possible, the optimizer places the statements inline, eliminating the need for linkage code. This optimization is known as *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.
- Delete from the program, and identify with a warning message, code that can never be performed (unreachable code elimination).
- Discard unreferenced data items from the DATA DIVISION, and suppress generation of code to initialize these data items to their VALUE clauses. (The optimizer takes this action only when you use the FULL suboption.)

Contained program procedure integration

In contained program procedure integration, the contained program code replaces a CALL to a contained program. The resulting program runs faster without the overhead of CALL linkage and with more linear control flow.

Program size: If several CALL statements call contained programs and these programs replace each such statement, the containing program can become large. The optimizer limits this increase to no more than 50 percent, after which it no longer integrates the programs. The optimizer then chooses the next best optimization for the CALL statement. The linkage overhead can be as few as two instructions.

Unreachable code: As a result of this 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.

PERFORM procedure integration

PERFORM procedure integration is the process whereby a PERFORM statement is replaced by its performed procedures. The advantage is that the resulting program runs faster without the overhead of PERFORM linkage and with more linear control flow.

Program size: If the performed procedures are invoked by several PERFORM statements and replace each such statement, the program could become large. The optimizer limits this increase to no more than 50 percent, after which it no longer integrates these procedures. If you are concerned about program size, you can prevent procedure integration in specific instances by using a priority number on section names.

If you do not want a PERFORM statement to be replaced by its performed procedures, put the PERFORM statement in one section and put the performed procedures in another section with a different priority number. The optimizer then chooses the next best optimization for the PERFORM statement. The linkage overhead can be as few as two instructions.

Unreachable code: Because of procedure integration, one PERFORM procedure might be repeated several times. As further optimization proceeds on each copy of the procedure, portions might be found to be unreachable, depending on the context into which the code was copied.

"Example: PERFORM procedure integration"

RELATED CONCEPTS
"Optimization of table references" on page 628

RELATED REFERENCES
"OPTIMIZE" on page 327

Example: PERFORM procedure integration

The following example shows code that will be transformed by procedure integration.

All the PERFORM statements in the following program will be transformed:

```
1   SECTION 5.
11. PERFORM 12
    STOP RUN.
12. PERFORM 21
    PERFORM 21.
2   SECTION 5.
21. IF A < 5 THEN
      ADD 1 TO A
      DISPLAY A
    END-IF.
```

The program will be compiled as if it had originally been written as follows:

```
1   SECTION 5.
11.
12. IF A < 5 THEN
      ADD 1 TO A
      DISPLAY A
    END-IF.
    IF A < 5 THEN
      ADD 1 TO A
      DISPLAY A
    END-IF.
    STOP RUN.
```

By contrast, in the following program only the first PERFORM statement, PERFORM 12, will be optimized by procedure integration:

```
1   SECTION.
11. PERFORM 12
    STOP RUN.
12. PERFORM 21
    PERFORM 21.
```

```
2   SECTION 5.
21. IF A < 5 THEN
      ADD 1 TO A
      DISPLAY A
    END-IF.
```

RELATED CONCEPTS
"Optimization of table references" on page 628

RELATED TASKS
"Optimizing your code" on page 630
Chapter 34, "Tuning your program," on page 623

Choosing compiler features to enhance performance

Your choice of performance-related compiler options and your use of the `USE FOR DEBUGGING ON ALL PROCEDURES` statement can affect how well your program is optimized.

You might have a customized system that requires certain options for optimum performance. Do these steps:

1. To see what your system defaults are, get a short listing for any program and review the listed option settings.
2. Determine which options are fixed as nonoverridable at your installation by checking with your system programmer.
3. For the options not fixed at installation, select performance-related options for compiling your programs.

   Important: Confer with your system programmer about how to tune COBOL programs. Doing so will ensure that the options you choose are appropriate for programs at your site.

Another compiler feature to consider is the `USE FOR DEBUGGING ON ALL PROCEDURES` statement. It can greatly affect the compiler optimizer. The `ON ALL PROCEDURES` option generates extra code at each transfer to a procedure name. Although very useful for debugging, it can make the program significantly larger and inhibit optimization substantially.

Although COBOL allows segmentation language, you will not improve storage allocation by using it, because COBOL does not perform overlay.

RELATED CONCEPTS
"Optimization" on page 631

RELATED TASKS
"Optimizing your code" on page 630
"Getting listings" on page 365

RELATED REFERENCES
"Performance-related compiler options"

Performance-related compiler options

In the table below you can see a brief description of the purpose of each option, its performance advantages and disadvantages, and usage notes where applicable.

Table 88. **Performance-related compiler options**

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
ARITH(EXTEND) See "ARITH" on page 302.	To increase the maximum number of digits allowed for decimal numbers	In general, none	ARITH(EXTEND) causes some degradation in performance for all decimal data types due to larger intermediate results.	The amount of degradation that you experience depends directly on the amount of decimal data that you use.
"AWO" on page 302	To get optimum use of buffer and device space	Can result in performance savings, because this option results in fewer calls to data management services to handle input and output	In general, none	When you use AWO, the APPLY WRITE-ONLY clause is in effect for all files in the program that are physical sequential with V-mode records.
DATA(31) (see "DATA" on page 307)	To have DFSMS allocate QSAM buffers above the 16-MB line (by using the RENT and DATA(31) compiler options)	Because extended-format QSAM data sets can require many buffers, allocating the buffers in unrestricted storage avoids virtual storage constraint problems.	In general, none	On a z/OS system with DFSMS, if your application processes striped extended-format QSAM data sets, use the RENT and DATA(31) compiler options to have the input-output buffers for your QSAM files allocated from storage above the 16-MB line.
"DYNAM" on page 313	To have subprograms (called through the CALL statement) dynamically loaded at run time	Subprograms are easier to maintain, because the application does not have to be link-edited again if a subprogram is changed.	There is a slight performance penalty, because the call must go through a Language Environment routine.	To free virtual storage that is no longer needed, issue the CANCEL statement.
"FASTSRT" on page 314	To specify that the IBM DFSORT product (or equivalent) will handle all of the input and output	Eliminates the overhead of returning to Enterprise COBOL after each record is processed	None	FASTSRT is recommended when direct work files are used for the sort work files. Not all sorts are eligible for this option.
NUMPROC(PFD) (see "NUMPROC" on page 325)	To have invalid sign processing bypassed for numeric operations	Generates significantly more efficient code for numeric comparisons	For most references to COMP-3 and DISPLAY numeric data items, NUMPROC(PFD) inhibits extra code from being generated to "fix up" signs. This extra code might also inhibit some other types of optimizations. The extra code is generated with NUMPROC(MIG) and NUMPROC(NOPFD).	When you use NUMPROC(PFD), the compiler assumes that the data has the correct sign and bypasses the sign "fix-up" process. Because not all external data files contain the proper sign for COMP-3 or DISPLAY signed numeric data, NUMPROC(PFD) might not be applicable for all programs. For performance-sensitive applications, NUMPROC(PFD) is recommended.

Table 88. **Performance-related compiler options** *(continued)*

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
OPTIMIZE(STD) (see “OPTIMIZE” on page 327)	To optimize generated code for better performance	Generally results in more efficient runtime code	Longer compile time: OPTIMIZE requires more processing time for compiles than NOOPTIMIZE.	NOOPTIMIZE is generally used during program development when frequent compiles are needed; it also allows for symbolic debugging. For production runs, OPTIMIZE is recommended.
OPTIMIZE(FULL) (see “OPTIMIZE” on page 327)	To optimize generated code for better performance and also optimize the DATA DIVISION	Generally results in more efficient runtime code and less storage usage	Longer compile time: OPTIMIZE requires more processing time for compiles than NOOPTIMIZE.	OPT(FULL) deletes unused data items, which might be undesirable in the case of time stamps or data items that are used only as markers for dump reading.
“RENT” on page 331	To generate a reentrant program	Enables the program to be placed in shared storage (LPA/ELPA) for faster execution	Generates additional code to ensure that the program is reentrant	
RMODE(ANY) (see “RMODE” on page 332)	To let the program be loaded anywhere	RMODE(ANY) with NORENT lets the program and its WORKING-STORAGE be located above the 16-MB line, relieving storage below the line.	In general, none	
NOSSRANGE (see “SSRANGE” on page 337)	To verify that all table references and reference modification expressions are in proper bounds	SSRANGE generates additional code for verifying table references. Using NOSSRANGE causes that code not to be generated.	None	In general, if you need to verify the table references only a few times instead of at every reference, coding your own checks might be faster than using SSRANGE. You can turn off SSRANGE at run time by using the CHECK(OFF) runtime option. For performance-sensitive applications, NOSSRANGE is recommended.

Table 88. **Performance-related compiler options** *(continued)*

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
TEST(NONE) or NOTEST (see "TEST" on page 338)	To avoid the additional object code that would be produced to take full advantage of Debug Tool, use TEST(NONE) or NOTEST. Additionally, when using TEST(NONE), you can use the SEPARATE suboption of TEST option to further reduce the size of your object code.	Because the TEST compiler option with any hook-location suboption other than NONE (that is, ALL, STMT, PATH, BLOCK) generates additional code, it can cause significant performance degradation when used in a production environment. The more compiled-in hooks you specify, the more additional code is generated and the greater performance degradation might be.	None	TEST without the hook-location suboption NONE in effect forces the NOOPTIMIZE compiler option into effect. For production runs, using NOTEST or TEST(NONE,SYM) with or without the SEPARATE suboption, is recommended. This results in overlay hooks rather than compiled-in hooks. If during the production run, you want a symbolic dump of the data items in a formatted dump when the program abends, compile with TEST(NONE,SYM) with or without the SEPARATE suboption.
"THREAD" on page 342	To enable programs for execution in a Language Environment enclave that has multiple POSIX threads or PL/I tasks	None	There is a slight performance penalty because of the overhead of serialization logic.	This is true for a threaded or a nonthreaded environment.
TRUNC(OPT) (see "TRUNC" on page 343)	To avoid having code generated to truncate the receiving fields of arithmetic operations	Does not generate extra code and generally improves performance	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, though its performance was improved in COBOL for OS/390 & VM V2R2.	TRUNC(STD) conforms to Standard COBOL 85, but TRUNC(BIN) and TRUNC(OPT) do not. With TRUNC(OPT), the compiler assumes that the data conforms to the PICTURE and USAGE specifications. TRUNC(OPT) is recommended where possible.

RELATED CONCEPTS
"Optimization" on page 631
"Storage and its addressability" on page 41

RELATED TASKS
"Generating a list of compiler error messages" on page 275
"Evaluating performance" on page 637
"Optimizing buffer and device space" on page 12
"Choosing compiler features to enhance performance" on page 633
"Improving sort performance with FASTSRT" on page 225
"Using striped extended-format QSAM data sets" on page 171
"Handling tables efficiently" on page 627

RELATED REFERENCES
"Sign representation of zoned and packed-decimal data" on page 55
"Allocation of buffers for QSAM files" on page 172

Evaluating performance

Fill in the following worksheet to help you evaluate the performance of your program. If you answer yes to each question, you are probably improving the performance.

In thinking about the performance tradeoff, be sure you understand the function of each option as well as the performance advantages and disadvantages. You might prefer function over increased performance in many instances.

Table 89. **Performance-tuning worksheet**

Compiler option	Consideration	Yes?
`AWO`	Do you use the `AWO` option when possible?	
`DATA`	When you use QSAM striped data sets, do you use the `DATA(31)` option?	
`DYNAM`	Do you use `NODYNAM`? Consider the performance tradeoffs.	
`FASTSRT`	When you use direct work files for the sort work files, did you use the `FASTSRT` option?	
`NUMPROC`	Do you use `NUMPROC(PFD)` when possible?	
`OPTIMIZE`	Do you use `OPTIMIZE` for production runs? Can you use `OPTIMIZE(FULL)`?	
`RENT`	Do you use `NORENT`? Consider the performance tradeoffs.	
`RMODE(ANY)`	Do you use `RMODE(ANY)` with your `NORENT` programs? Consider the performance tradeoffs with storage usage.	
`SSRANGE`	Do you use `NOSSRANGE` for production runs?	
`TEST`	Do you use `NOTEST`, `TEST(NONE,SYM)`, or `TEST(NONE,SYM,SEPARATE)` for production runs?	
`TRUNC`	Do you use `TRUNC(OPT)` when possible?	

RELATED TASKS
"Choosing compiler features to enhance performance" on page 633

RELATED REFERENCES
"Performance-related compiler options" on page 633

Running efficiently with CICS, IMS, or VSAM

You can improve performance for online programs running under CICS or IMS, or programs that use VSAM, by following these tips.

CICS: If your application runs under CICS, convert `EXEC CICS LINK` commands to COBOL `CALL` statements to improve transaction response time.

IMS: If your application runs under IMS, preloading the application program and the library routines can help reduce the overhead of loading and searching. It can also reduce the input-output activity.

For better system performance, use the RENT compiler option and preload the applications and library routines when possible. You can also use the Language Environment library routine retention (LRR) function to improve performance in IMS/TM regions.

VSAM: When you use VSAM files, increase the number of data buffers for sequential access or index buffers for random access. Also, select a control interval size (CISZ) that is appropriate for the application. A smaller CISZ results in faster retrieval for random processing at the expense of inserts. A larger CISZ is more efficient for sequential processing.

For better performance, access the records sequentially and avoid using multiple alternate indexes when possible. If you use alternate indexes, access method services builds them more efficiently than the AIXBLD runtime option.

RELATED TASKS
"Coding COBOL programs to run under CICS" on page 393
Chapter 22, "Developing COBOL programs for IMS," on page 417
"Improving VSAM performance" on page 204
Language Environment Customization

RELATED REFERENCES
Specifying run-time options (*Language Environment Programming Guide*)

Chapter 35. Simplifying coding

You can use coding techniques to improve your productivity. By using the COPY statement, COBOL intrinsic functions, and Language Environment callable services, you can avoid repetitive coding and having to code many arithmetic calculations or other complex tasks.

If your program contains frequently used code sequences (such as blocks of common data items, input-output routines, error routines, or even entire COBOL programs), write the code sequences once and put them in a COBOL copy library. You can use the COPY statement to retrieve these code sequences and have them included in your program at compile time. Using copybooks in this manner eliminates repetitive coding.

COBOL provides various capabilities for manipulating strings and numbers. These capabilities can help you simplify your coding.

The Language Environment date and time callable services store dates as fullword binary integers and store timestamps as long (64-bit) floating-point values. These formats let you do arithmetic calculations on date and time values simply and efficiently. You do not need to write special subroutines that use services outside the language library to perform such calculations.

RELATED TASKS
"Using numeric intrinsic functions" on page 59
"Using math-oriented callable services" on page 60
"Using date callable services" on page 62
"Eliminating repetitive coding"
"Converting data items (intrinsic functions)" on page 112
"Evaluating data items (intrinsic functions)" on page 115
"Using Language Environment callable services" on page 641

Eliminating repetitive coding

Use the COPY statement in any program division and at any code sequence level to include stored source statements in a program. You can nest COPY statements to any depth.

To specify more than one copy library, use either multiple system definitions or a combination of multiple definitions and the IN/OF phrase (IN/OF *library-name*):

z/OS batch
Use JCL to concatenate data sets in your SYSLIB DD statement. Alternatively, define multiple DD statements and use the IN/OF phrase of the COPY statement.

TSO
Use the ALLOCATE command to concatenate data sets for SYSLIB. Alternatively, issue multiple ALLOCATE statements and use the IN/OF phrase of the COPY statement.

UNIX
Use the SYSLIB environment variable to define multiple paths to your copybooks. Alternatively, use multiple environment variables and use the IN/OF phrase of the COPY statement.

For example:

```
COPY MEMBER1 OF COPYLIB
```

If you omit this qualifying phrase, the default is SYSLIB.

COPY and debugging line: In order for the text copied to be treated as debug lines, for example, as if there were a D inserted in column 7, put the D on the first line of the COPY statement. A COPY statement itself cannot be a debugging line; if it contains a D and WITH DEBUGGING mode is not specified, the COPY statement is nevertheless processed.

"Example: using the COPY statement"

RELATED REFERENCES
Chapter 18, "Compiler-directing statements," on page 351

Example: using the COPY statement

These examples show how you can use the COPY statement to include library text in a program.

Suppose the library entry CFILEA consists of the following FD entries:

```
    BLOCK CONTAINS 20 RECORDS
    RECORD CONTAINS 120 CHARACTERS
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS FILE-OUT.
01  FILE-OUT       PIC X(120).
```

You can retrieve the text-name CFILEA by using the COPY statement in a source program as follows:

```
FD FILEA
         COPY CFILEA.
```

The library entry is copied into your program, and the resulting program listing looks like this:

```
FD FILEA
         COPY CFILEA.
C    BLOCK CONTAINS 20 RECORDS
C    RECORD CONTAINS 120 CHARACTERS
C    LABEL RECORDS ARE STANDARD
C    DATA RECORD IS FILE-OUT.
C    01  FILE-OUT   PIC X(120).
```

In the compiler source listing, the COPY statement prints on a separate line. C precedes copied lines.

Assume that a copybook with the text-name DOWORK is stored by using the following statements:

```
COMPUTE QTY-ON-HAND = TOTAL-USED-NUMBER-ON-HAND
MOVE QTY-ON-HAND to PRINT-AREA
```

To retrieve the copybook identified as DOWORK, code:

```
paragraph-name.
    COPY DOWORK.
```

The statements that are in the DOWORK procedure will follow *paragraph-name*.

If you use the `EXIT` compiler option to provide a `LIBEXIT` module, your results might differ from those shown here.

RELATED TASKS
"Eliminating repetitive coding" on page 639

RELATED REFERENCES
Chapter 18, "Compiler-directing statements," on page 351

Using Language Environment callable services

Language Environment callable services make many types of programming tasks easier. You call them by using the `CALL` statement.

Language Environment services help you with the following tasks:

- Handling conditions

  The Language Environment condition-handling facilities enable COBOL applications to react to unexpected errors. You can use language constructs or runtime options to select the level at which to handle each condition. For example, you can handle a particular error in your COBOL program, let Language Environment take care of it, or have the operating system handle it.

  In support of Language Environment condition handling, COBOL provides procedure-pointer data items.
- Managing dynamic storage

  These services enable you to get, free, and reallocate storage. You can also create your own storage pools.
- Calculating dates and times

  With the date and time services, you can get the current local time and date in several formats, and perform date and time conversions. Two callable services, CEEQCEN and CEESCEN, provide a predictable way to handle two-digit years, such as 91 for 1991 or 06 for 2006.
- Making math calculations

  Calculations that are easy to perform with mathematical callable services include logarithmic, exponential, trigonometric, square root, and integer functions.

  COBOL also supports a set of intrinsic functions that include some of the same mathematical and date functions as those provided by the callable services. The Language Environment callable services and intrinsic functions provide equivalent results, with a few exceptions. You should be familiar with these differences before deciding which to use.
- Handling messages

  Message-handling services include services for getting, dispatching, and formatting messages. Messages for non-CICS applications can be directed to files or printers. CICS messages are directed to a CICS transient data queue. Language Environment splits messages to accommodate the record length of the destination, and presents messages in the correct national language such as Japanese or English.
- Supporting national languages

  These services make it easy for your applications to support the language desired by application users. You can set the language and country, and obtain default date, time, number, and currency formats. For example, you might want dates to appear as 23 June 06 or as 6,23,06.

- General services such as starting Debug Tool and obtaining a Language Environment formatted dump

  Debug Tool provides advanced debugging functions for COBOL applications, including both batch and interactive debugging of COBOL-CICS programs. Debug Tool enables you to debug a COBOL application from the host or, in conjunction with the Debug Perspective of WebSphere Developer for System z, from a Windows-based workstation.

  Depending on the options that you select, the Language Environment formatted dump might contain the names and values of data items, and information about conditions, program tracebacks, control blocks, storage, and files. All Language Environment dumps have a common, well-labeled, easy-to-read format.

"Example: Language Environment callable services" on page 644

RELATED CONCEPTS
"Sample list of Language Environment callable services"

RELATED TASKS
"Using numeric intrinsic functions" on page 59
"Using math-oriented callable services" on page 60
"Using date callable services" on page 62
"Calling Language Environment services" on page 643
"Using procedure and function pointers" on page 447

Sample list of Language Environment callable services

The following table shows some examples of the callable services that are available with Language Environment. Many more services are available than those listed.

Table 90. **Language Environment callable services**

Function type	Service	Purpose
Condition handling	CEEHDLR	To register a user condition handler
	CEESGL	To raise or signal a condition
	CEEMRCR	To indicate where the program will resume running after the condition handler has finished
Dynamic storage	CEEGTST	To get storage
	CEECZST	To change the size of a previously allocated storage block
	CEEFRST	To free storage
Date and time	CEECBLDY	To convert a string that represents a date into COBOL integer date format, which represents a date as the number of days since 31 December 1600
	CEEQCEN, CEESCEN	To query and set the Language Environment century window, which is valuable when a program uses two digits to express a year
	CEEGMTO	To calculate the difference between the local system time and Greenwich Mean Time
	CEELOCT	To get the current local time in your choice of three formats
Math	CEESIABS	To calculate the absolute value of an integer
	CEESSNWN	To calculate the nearest whole number for a single-precision floating-point number
	CEESSCOS	To calculate the cosine of an angle

Table 90. **Language Environment callable services** *(continued)*

Function type	Service	Purpose
Message handling	CEEMOUT	To dispatch a message
	CEEMGET	To retrieve a message
National language support	CEE3LNG	To change or query the current national language
	CEE3CTY	To change or query the current national country
	CEE3MCS	To obtain the default currency symbol for a given country
General	CEE3DMP	To obtain a Language Environment formatted dump
	CEETEST	To start a debugging tool, such as Debug Tool

RELATED REFERENCES
Language Environment Programming Reference

Calling Language Environment services

To invoke a Language Environment service, use a `CALL` statement with the correct parameters for that service. Define the variables for the `CALL` statement in the `DATA DIVISION` with the definitions that are required by that service.

```
77  argument        comp-1.
77  feedback-code   pic  x(12)  display.
77  result          comp-1.
. . .
CALL "CEESSSQT" using argument, feedback-code, result
```

In the example above, Language Environment service CEESSSQT calculates the value of the square root of the variable `argument` and returns this value in the variable `result`.

You can choose whether to specify the feedback code parameter. If you specify it, the value returned in `feedback-code` indicates whether the service completed successfully. If you specify `OMITTED` instead of the feedback code, and the service is not successful, a Language Environment condition is automatically signaled to the Language Environment condition manager. You can handle such a condition by recovery logic implemented in a user-written condition handler, or allow the default Language Environment processing for unhandled conditions to occur. In either case, you avoid having to write logic to check the feedback code explicitly after each call.

If you call a Language Environment callable service and specify `OMITTED` for the feedback code, the `RETURN-CODE` special register is set to 0 if the service is successful. It is not altered if the service is unsuccessful. If you do not specify `OMITTED` for the feedback code, the `RETURN-CODE` special register is always set to 0 regardless of whether the service completed successfully.

"Example: Language Environment callable services" on page 644

RELATED CONCEPTS
General callable services (*Language Environment Programming Guide*)

RELATED REFERENCES
General callable services (*Language Environment Programming Reference*)
CALL statement (*Enterprise COBOL Language Reference*)

Example: Language Environment callable services

This example shows a COBOL program that uses Language Environment services CEEDAYS and CEEDATE to format and display a date from the results of a COBOL ACCEPT statement.

Using CEEDAYS and CEEDATE reduces the coding that would be required without Language Environment.

```
 ID DIVISION.
 PROGRAM-ID. HOHOHO.
************************************************************
* FUNCTION:  DISPLAY TODAY'S DATE IN THE FOLLOWING FORMAT: *
*            WWWWWWWWW, MMMMMMMM DD, YYYY                  *
*                                                          *
*            For example: TUESDAY, AUGUST 15, 2006         *
*                                                          *
************************************************************
 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 LANGUAGE ENVIRONMENT 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.
```

Part 9. Appendixes

Appendix A. Intermediate results and arithmetic precision

The compiler handles arithmetic statements as a succession of operations performed according to operator precedence, and sets up intermediate fields to contain the results of those operations. The compiler uses algorithms to determine the number of integer and decimal places to reserve.

Intermediate results are possible in the following cases:
- In an `ADD` or `SUBTRACT` statement that contains more than one operand immediately after the verb
- In a `COMPUTE` statement that specifies a series of arithmetic operations or multiple result fields
- In an arithmetic expression contained in a conditional statement or in a reference-modification specification
- In an `ADD`, `SUBTRACT`, `MULTIPLY`, or `DIVIDE` statement that uses the `GIVING` option and multiple result fields
- In a statement that uses an intrinsic function as an operand

"Example: calculation of intermediate results" on page 649

The precision of intermediate results depends on whether you compile using the default option `ARITH(COMPAT)` (referred to as *compatibility mode*) or using `ARITH(EXTEND)` (referred to as *extended mode*).

In compatibility mode, evaluation of arithmetic operations is unchanged from that in releases of IBM COBOL before COBOL for OS/390 & VM Version 2 Release 2:
- A maximum of 30 digits is used for fixed-point intermediate results.
- Floating-point intrinsic functions return long-precision (64-bit) floating-point results.
- Expressions that contain floating-point operands, fractional exponents, or floating-point intrinsic functions are evaluated as if all operands that are not in floating point are converted to long-precision floating point and floating-point operations are used to evaluate the expression.
- Floating-point literals and external floating-point data items are converted to long-precision floating point for processing.

In extended mode, evaluation of arithmetic operations has the following characteristics:
- A maximum of 31 digits is used for fixed-point intermediate results.
- Floating-point intrinsic functions return extended-precision (128-bit) floating-point results.
- Expressions that contain floating-point operands, fractional exponents, or floating-point intrinsic functions are evaluated as if all operands that are not in floating point are converted to extended-precision floating point and floating-point operations are used to evaluate the expression.
- Floating-point literals and external floating-point data items are converted to extended-precision floating point for processing.

RELATED CONCEPTS
"Formats for numeric data" on page 49
"Fixed-point contrasted with floating-point arithmetic" on page 64

RELATED REFERENCES
"Fixed-point data and intermediate results" on page 649
"Floating-point data and intermediate results" on page 654
"Arithmetic expressions in nonarithmetic statements" on page 655
"ARITH" on page 302

Terminology used for intermediate results

To understand this information about intermediate results, you need to understand the following terminology.

i The number of integer places carried for an intermediate result. (If you use the `ROUNDED` phrase, one more integer place might be carried for accuracy if necessary.)

d The number of decimal places carried for an intermediate result. (If you use the `ROUNDED` phrase, one more decimal place might be carried for accuracy if necessary.)

dmax In a particular statement, the largest of the following items:
- The number of decimal places needed for the final result field or fields
- The maximum number of decimal places defined for any operand, except divisors or exponents
- The *outer-dmax* for any function operand

inner-dmax
In reference to a function, the largest of the following items:
- 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 of decimal places that 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 (in division, the divisor).

op2 The second operand in a generated arithmetic statement (in division, the dividend).

i1, i2 The number of integer places in *op1* and *op2*, respectively.

d1, d2 The number of decimal places in *op1* and *op2*, respectively.

ir The intermediate result when a generated arithmetic statement or operation is performed. (Intermediate results are generated either in registers or storage locations.)

ir1, ir2 Successive intermediate results. (Successive intermediate results might have the same storage location.)

RELATED REFERENCES
ROUNDED phrase (*Enterprise COBOL Language Reference*)

Example: calculation of intermediate results

The following example shows how the compiler performs an arithmetic statement as a succession of operations, storing intermediate results as needed.

```
COMPUTE Y = A + B * C - D / E + F ** G
```

The result is calculated in the following order:

1. Exponentiate `F` by `G` yielding *ir1*.
2. Multiply `B` by `C` yielding *ir2*.
3. Divide `E` into `D` yielding *ir3*.
4. Add `A` to `ir2` yielding *ir4*.
5. Subtract *ir3* from *ir4* yielding *ir5*.
6. Add *ir5* to *ir1* yielding `Y`.

RELATED TASKS
"Using arithmetic expressions" on page 58

RELATED REFERENCES
"Terminology used for intermediate results" on page 648

Fixed-point data and intermediate results

The compiler determines the number of integer and decimal places in an intermediate result.

Addition, subtraction, multiplication, and division

The following table shows the precision theoretically possible as the result of addition, subtraction, multiplication, or division.

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 obtain the accuracy you want in the final result.

The following table shows the number of places the compiler carries for fixed-point intermediate results of arithmetic operations that involve addition, subtraction, multiplication, or division in *compatibility mode* (that is, when the default compiler option `ARITH(COMPAT)` is in effect):

Value of *i* + *d*	**Value of *d***	**Value of *i* + *dmax***	**Number of places carried for *ir***
<30 or =30	Any value	Any value	*i* integer and *d* decimal places
>30	<*dmax* or =*dmax*	Any value	30-*d* integer and *d* decimal places
	>*dmax*	<30 or =30	*i* integer and 30-*i* decimal places
		>30	30-*dmax* integer and *dmax* decimal places

The following table shows the number of places the compiler carries for fixed-point intermediate results of arithmetic operations that involve addition, subtraction, multiplication, or division in *extended mode* (that is, when the compiler option `ARITH(EXTEND)` is in effect):

Value of *i* **+** *d*	**Value of** *d*	**Value of** *i* **+** *dmax*	**Number of places carried for** *ir*
<31 or =31	Any value	Any value	*i* integer and *d* decimal places
>31	<*dmax* or =*dmax*	Any value	31-*d* integer and *d* decimal places
	>*dmax*	<31 or =31	*i* integer and 31-*i* decimal places
		>31	31-*dmax* integer and *dmax* decimal places

Exponentiation

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 instructions are used.
- When *op2* is an integral literal or constant, the value *d* is computed as

  `d = d1 * |op2|`

  and the value *i* is computed based on the characteristics of *op1*:
  - When *op1* is a data-name or variable,

    `i = i1 * |op2|`
  - When *op1* is a literal or constant, *i* is set equal to the number of integers in the value of *op1* ** |*op2*|.

  In compatibility mode (compilation using `ARITH(COMPAT)`), the compiler having calculated *i* and *d* takes the action indicated in the table below to handle the intermediate results *ir* of the 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.	Same action as when *op2* is an integral data-name or variable (shown below). Exception: for a 30-digit integer raised to the power of literal 1, *i* integer and *d* decimal places are carried for *ir*.
>30	Any	Same action as when *op2* is an integral data-name or variable (shown below)

  In extended mode (compilation using `ARITH(EXTEND)`), the compiler having calculated *i* and *d* takes the action indicated in the table below to handle the intermediate results *ir* of the exponentiation.

Value of *i* **+** *d*	**Other conditions**	**Action taken**
<31	Any	*i* integer and *d* decimal places are carried for *ir*.
=31 or >31	Any	Same action as when *op2* is an integral data-name or variable (shown below). Exception: for a 31-digit integer raised to the power of literal 1, *i* integer and *d* decimal places are carried for *ir*.

If *op2* is negative, the value of 1 is then divided by the result produced by the preliminary computation. The values of *i* and *d* that are used are calculated following the division rules for fixed-point data already shown above.

- When *op2* is an integral data-name or variable, *dmax* decimal places and 30-*dmax* (compatibility mode) or 31-*dmax* (extended mode) integer places are used. *op1* is multiplied by itself (|*op2*| - 1) times for nonzero *op2*.

  If *op2* is equal to 0, the result is 1. Division-by-0 and exponentiation `SIZE ERROR` conditions apply.

Fixed-point exponents with more than nine significant digits are always truncated to nine 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.

"Example: exponentiation in fixed-point arithmetic"

RELATED REFERENCES
"Terminology used for intermediate results" on page 648
"Truncated intermediate results"
"Binary data and intermediate results" on page 652
"Floating-point data and intermediate results" on page 654
"Intrinsic functions evaluated in fixed-point arithmetic" on page 652
"ARITH" on page 302
SIZE ERROR phrases (*Enterprise COBOL Language Reference*)

Example: exponentiation in fixed-point arithmetic

The following example shows how the compiler performs an exponentiation to a nonzero integer power as a succession of multiplications, storing intermediate results as needed.

```
COMPUTE Y = A ** B
```

If `B` is equal to 4, the result is computed as shown below. The values of *i* and *d* that are used are calculated according to the multiplication rules for fixed-point data and intermediate results (referred to below).

1. Multiply `A` by `A` yielding *ir1*.
2. Multiply *ir1* by `A` yielding *ir2*.
3. Multiply *ir2* by `A` yielding *ir3*.
4. Move *ir3* to *ir4*.

   ir4 has *dmax* decimal places. Because `B` is positive, *ir4* is moved to `Y`. If `B` were equal to -4, however, an additional fifth step would be performed:
5. Divide *ir4* into 1 yielding *ir5*.

ir5 has *dmax* decimal places, and would then be moved to `Y`.

RELATED REFERENCES
"Terminology used for intermediate results" on page 648
"Fixed-point data and intermediate results" on page 649

Truncated intermediate results

Whenever the number of digits in an intermediate result exceeds 30 in compatibility mode or 31 in extended mode, the compiler truncates to 30 (compatibility mode) or 31 (extended mode) digits and issues a warning. If truncation occurs at run time, a message is issued and the program continues running.

If you want to avoid the truncation of intermediate results that can occur in fixed-point calculations, use floating-point operands (`COMP-1` or `COMP-2`) instead.

RELATED CONCEPTS
"Formats for numeric data" on page 49

RELATED REFERENCES
"Fixed-point data and intermediate results" on page 649
"ARITH" on page 302

Binary data and intermediate results

If an operation that involves binary operands requires intermediate results longer than 18 digits, the compiler converts the operands to internal decimal before performing the operation. If the result field is binary, the compiler converts the result from internal decimal to binary.

Binary operands are most efficient when intermediate results will not exceed nine digits.

RELATED REFERENCES
"Fixed-point data and intermediate results" on page 649
"ARITH" on page 302

Intrinsic functions evaluated in fixed-point arithmetic

The compiler determines the *inner-dmax* and *outer-dmax* values for an intrinsic function from the characteristics of the function.

Integer functions

Integer intrinsic functions return an integer; thus their *outer-dmax* is always zero. For those integer functions whose arguments must all be integers, the *inner-dmax* is thus also always zero.

The following table summarizes the *inner-dmax* and the precision of the function result.

Function	*Inner-dmax*	**Digit precision of function result**
`DATE-OF-INTEGER`	0	8
`DATE-TO-YYYYMMDD`	0	8
`DAY-OF-INTEGER`	0	7
`DAY-TO-YYYYDDD`	0	7
`FACTORIAL`	0	30 in compatibility mode, 31 in extended mode
`INTEGER-OF-DATE`	0	7
`INTEGER-OF-DAY`	0	7
`LENGTH`	n/a	9
`MOD`	0	`min(`*`i1 i2`*`)`
`ORD`	n/a	3
`ORD-MAX`		9
`ORD-MIN`		9
`YEAR-TO-YYYY`	0	4

Function	*Inner-dmax*	**Digit precision of function result**
`INTEGER`		For a fixed-point argument: one more digit than in the argument. For a floating-point argument: 30 in compatibility mode, 31 in extended mode.
`INTEGER-PART`		For a fixed-point argument: same number of digits as in the argument. For a floating-point argument: 30 in compatibility mode, 31 in extended mode.

Mixed functions

A *mixed* intrinsic function is a function whose result type depends on the type of its arguments. A mixed function is fixed point if all of its arguments are numeric and none of its arguments is floating point. (If any argument of a mixed function is floating point, the function is evaluated with floating-point instructions and returns a floating-point result.) When a mixed function is evaluated with fixed-point arithmetic, the result is integer if all of the arguments are integer; otherwise, the result is fixed point.

For the mixed functions `MAX`, `MIN`, `RANGE`, `REM`, and `SUM`, the *outer-dmax* is always equal to the *inner-dmax* (and both are thus zero if all the arguments are integer). To determine the precision of the result returned for these functions, apply the rules for fixed-point arithmetic and intermediate results (as referred to below) to each step in the algorithm.

`MAX`

1. Assign the first argument to the function result.
2. For each remaining argument, do the following steps:
   a. Compare the algebraic value of the function result with the argument.
   b. Assign the greater of the two to the function result.

`MIN`

1. Assign the first argument to the function result.
2. For each remaining argument, do the following steps:
   a. Compare the algebraic value of the function result with the argument.
   b. Assign the lesser of the two to the function result.

`RANGE`

1. Use the steps for `MAX` to select the maximum argument.
2. Use the steps for `MIN` to select the minimum argument.
3. Subtract the minimum argument from the maximum.
4. Assign the difference to the function result.

`REM`

1. Divide argument one by argument two.
2. Remove all noninteger digits from the result of step 1.
3. Multiply the result of step 2 by argument two.
4. Subtract the result of step 3 from argument one.
5. Assign the difference to the function result.

`SUM`

1. Assign the value 0 to the function result.

2. For each argument, do the following steps:
   a. Add the argument to the function result.
   b. Assign the sum to the function result.

RELATED REFERENCES
"Terminology used for intermediate results" on page 648
"Fixed-point data and intermediate results" on page 649
"Floating-point data and intermediate results"
"ARITH" on page 302

Floating-point data and intermediate results

If any operation in an arithmetic expression is computed in floating-point arithmetic, the entire expression is computed as if all operands were converted to floating point and the operations were performed using floating-point instructions.

Floating-point instructions are used to compute an arithmetic expression if any of the following conditions is true of the expression:

- A receiver or operand is `COMP-1`, `COMP-2`, external floating point, or a floating-point literal.
- An exponent contains decimal places.
- An exponent is an expression that contains an exponentiation or division operator, and *dmax* is greater than zero.
- An intrinsic function is a floating-point function.

In compatibility mode, if an expression is computed in floating-point arithmetic, the precision used to evaluate the arithmetic operations is determined as follows:

- Single precision is used if all receivers and operands are `COMP-1` data items and the expression contains no multiplication or exponentiation operations.
- In all other cases, long precision is used.

Whenever long-precision floating point is used for one operation in an arithmetic expression, all operations in the expression are computed as if long floating-point instructions were used.

In extended mode, if an expression is computed in floating-point arithmetic, the precision used to evaluate the arithmetic operations is determined as follows:

- Single precision is used if all receivers and operands are `COMP-1` data items and the expression contains no multiplication or exponentiation operations.
- Long precision is used if all receivers and operands are `COMP-1` or `COMP-2` data items, at least one receiver or operand is a `COMP-2` data item, and the expression contains no multiplication or exponentiation operations.
- In all other cases, extended precision is used.

Whenever extended-precision floating point is used for one operation in an arithmetic expression, all operations in the expression are computed as if extended-precision floating-point instructions were used.

Alert: If a floating-point operation has an intermediate result field in which exponent overflow occurs, the job is abnormally terminated.

Exponentiations evaluated in floating-point arithmetic

In compatibility mode, floating-point exponentiations are always evaluated using long floating-point arithmetic. In extended mode, floating-point exponentiations are always evaluated using extended-precision floating-point arithmetic.

The value of a negative number raised to a fractional power is undefined in COBOL. For example, (-2) ** 3 is equal to -8, but (-2) ** (3.000001) is undefined. When an exponentiation is evaluated in floating point and there is a possibility that the result is undefined, the exponent is evaluated at run time to determine if it has an integral value. If not, a diagnostic message is issued.

Intrinsic functions evaluated in floating-point arithmetic

In compatibility mode, floating-point intrinsic functions always return a long (64-bit) floating-point value. In extended mode, floating-point intrinsic functions always return an extended-precision (128-bit) floating-point value.

Mixed functions that have at least one floating-point argument are evaluated using floating-point arithmetic.

RELATED REFERENCES
"Terminology used for intermediate results" on page 648
"ARITH" on page 302

Arithmetic expressions in nonarithmetic statements

Arithmetic expressions can appear in contexts other than arithmetic statements. For example, you can use an arithmetic expression with the `IF` or `EVALUATE` statement.

In such statements, the rules for intermediate results with fixed-point data and for intermediate results with floating-point data apply, with the following changes:

- Abbreviated `IF` statements are handled as though the statements were not abbreviated.
- In an explicit relation condition where at least one of the comparands is an arithmetic expression, *dmax* is the maximum number of decimal places for any operand of either comparand, excluding divisors and exponents. The rules for floating-point arithmetic apply if any of the following conditions is true:
  - Any operand in either comparand is `COMP-1`, `COMP-2`, external floating point, or a floating-point literal.
  - An exponent contains decimal places.
  - An exponent is an expression that contains an exponentiation or division operator, and *dmax* is greater than zero.

  For example:

  ```
  IF operand-1 = expression-1 THEN . . .
  ```

  If *operand-1* is a data-name defined to be `COMP-2`, the rules for floating-point arithmetic apply to *expression-1* even if it contains only fixed-point operands, because it is being compared to a floating-point operand.
- When the comparison between an arithmetic expression and another data item or arithmetic expression does not use a relational operator (that is, there is no explicit relation condition), the arithmetic expression is evaluated without regard to the attributes of its comparand. For example:

```
EVALUATE expression-1
  WHEN expression-2 THRU expression-3
  WHEN expression-4
  . . .
END-EVALUATE
```

In the statement above, each arithmetic expression is evaluated in fixed-point or floating-point arithmetic based on its own characteristics.

RELATED CONCEPTS
"Fixed-point contrasted with floating-point arithmetic" on page 64

RELATED REFERENCES
"Terminology used for intermediate results" on page 648
"Fixed-point data and intermediate results" on page 649
"Floating-point data and intermediate results" on page 654
IF statement (*Enterprise COBOL Language Reference*)
EVALUATE statement (*Enterprise COBOL Language Reference*)
Conditional expressions (*Enterprise COBOL Language Reference*)

Appendix B. Complex OCCURS DEPENDING ON

Several types of complex OCCURS DEPENDING ON (*complex ODO*) are possible. Complex ODO is supported as an extension to Standard COBOL 85.

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

- Variably located item or group: A data item described by an OCCURS clause with the DEPENDING ON phrase is followed by a nonsubordinate elementary or group data item.
- Variably located table: A data item described by an OCCURS clause with the DEPENDING ON phrase is followed by a nonsubordinate data item described by an OCCURS clause.
- Table that has variable-length elements: A data item described by an OCCURS clause contains a subordinate data item described by an OCCURS clause with the DEPENDING ON phrase.
- Index name for a table that has variable-length elements.
- Element of a table that has variable-length elements.

"Example: complex ODO"

RELATED TASKS
"Preventing index errors when changing ODO object value" on page 659
"Preventing overlay when adding elements to a variable table" on page 659

RELATED REFERENCES
"Effects of change in ODO object value" on page 658
OCCURS DEPENDING ON clause (*Enterprise COBOL Language Reference*)

Example: complex ODO

The following example illustrates the possible types of occurrence of complex ODO.

```
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)(3)
            INDEXED BY INDX.                      (4)
       03 TABLE-ITEM                    PIC 99.   (5)
       03 RECORD-2 OCCURS 1 TO 3 TIMES
                DEPENDING ON COUNTER-2.
          04 DATA-NUM                   PIC S99.
```

Definition: In the example, COUNTER-1 is an *ODO object*, that is, it is the object of the DEPENDING ON clause of RECORD-1. RECORD-1 is said to be an *ODO subject*. Similarly, COUNTER-2 is the ODO object of the corresponding ODO subject, RECORD-2.

The types of complex ODO occurrences shown in the example above are as follows:

(1) A variably located item: EMPLOYEE-NUMBER is a data item that follows, but is not subordinate to, a variable-length table in the same level-01 record.

(2) A variably located table: TABLE-2 is a table that follows, but is not subordinate to, a variable-length table in the same level-01 record.

(3) A table with variable-length elements: TABLE-2 is a table that contains a subordinate data item, RECORD-2, whose number of occurrences varies depending on the content of its ODO object.

(4) An index-name, INDX, for a table that has variable-length elements.

(5) An element, TABLE-ITEM, of a table that has variable-length elements.

How length is calculated

The length of the variable portion of each record is the product of its ODO object and the length of its ODO subject. For example, whenever a reference is made to one of the complex ODO items shown above, the actual length, if used, is computed as follows:

- The length of TABLE-1 is calculated by multiplying the contents of COUNTER-1 (the number of occurrences of RECORD-1) by 3 (the length of RECORD-1).
- The length of TABLE-2 is calculated by multiplying the contents of COUNTER-2 (the number of occurrences of RECORD-2) by 2 (the length of RECORD-2), and adding the length of TABLE-ITEM.
- The length of FIELD-A is calculated by adding the lengths of COUNTER-1, COUNTER-2, TABLE-1, EMPLOYEE-NUMBER, and TABLE-2 times 5.

Setting values of ODO objects

You must set *every* ODO object in a group item before you reference any complex ODO item in the group. For example, before you refer to EMPLOYEE-NUMBER in the code above, you must set COUNTER-1 and COUNTER-2 even though EMPLOYEE-NUMBER does not directly depend on either ODO object for its value.

Restriction: An ODO object cannot be variably located.

Effects of change in ODO object value

If a data item that is described by an OCCURS clause with the DEPENDING ON phrase is followed in the same group by one or more nonsubordinate data items (a form of complex ODO), any change in value of the ODO object affects subsequent references to complex ODO items in the record.

For example:

- The size of any group that contains the relevant ODO clause reflects the new value of the ODO object.
- A MOVE to a group that contains the ODO subject is made based on the new value of the ODO object.
- The location of any nonsubordinate items that follow the item described with the ODO clause is affected by the new value of the ODO object. (To preserve the contents of the nonsubordinate items, move them to a work area before the value of the ODO object changes, then move them back.)

The value of an ODO object can change when you move data to the ODO object or to the group in which it is contained. The value can also change if the ODO object is contained in a record that is the target of a READ statement.

RELATED TASKS
"Preventing index errors when changing ODO object value"
"Preventing overlay when adding elements to a variable table"

Preventing index errors when changing ODO object value

Be careful if you reference a complex-ODO index-name, that is, an index-name for a table that has variable-length elements, after having changed the value of the ODO object for a subordinate data item in the table.

When you change the value of an ODO object, the byte offset in an associated complex-ODO index is no longer valid because the table length has changed. Unless you take precautions, you will have unexpected results if you then code a reference to the index-name such as:

- A reference to an element of the table
- A SET statement of the form SET *integer-data-item* TO *index-name* (format 1)
- A SET statement of the form SET *index-name* UP|DOWN BY *integer* (format 2)

To avoid this type of error, do these steps:

1. Save the index in an integer data item. (Doing so causes an implicit conversion: the integer item receives the table element occurrence number that corresponds to the offset in the index.)
2. Change the value of the ODO object.
3. Immediately restore the index from the integer data item. (Doing so causes an implicit conversion: the index-name receives the offset that corresponds to the table element occurrence number in the integer item. The offset is computed according to the table length then in effect.)

The following code shows how to save and restore the index-name (shown in "Example: complex ODO" on page 657) when the ODO object COUNTER-2 changes.

```
 77  INTEGER-DATA-ITEM-1      PIC 99.
 . . .
     SET INDX TO 5.
*         INDX is valid at this point.
     SET INTEGER-DATA-ITEM-1 TO INDX.
*         INTEGER-DATA-ITEM-1 now has the
*         occurrence number that corresponds 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, containing the offset
*         that corresponds to INTEGER-DATA-ITEM-1, and
*         can be used with the expected results.
```

RELATED REFERENCES
SET statement (*Enterprise COBOL Language Reference*)

Preventing overlay when adding elements to a variable table

Be careful if you increase the number of elements in a variable-occurrence table that is followed by one or more nonsubordinate data items in the same group. When you increment the value of the ODO object and add an element to a table, you can inadvertently overlay the variably located data items that follow the table.

To avoid this type of error, do the following:

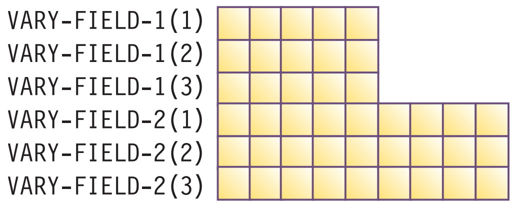
1. Save the variably located data items that follow the table in another data area.
2. Increment the value of the ODO object.

3. Move data into the new table element (if needed).
4. Restore the variably located data items from the data area where you saved them.

In the following example, suppose you want to add an element to the table VARY-FIELD-1, whose number of elements depends on the ODO object CONTROL-1. VARY-FIELD-1 is followed by the nonsubordinate variably located data item GROUP-ITEM-1, whose elements could potentially be overlaid.

```
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).
```

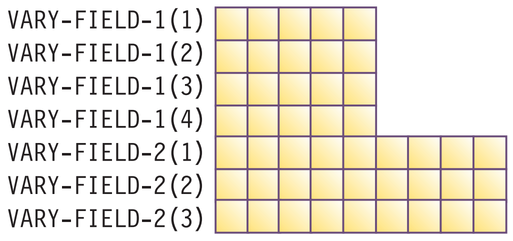
Each element of VARY-FIELD-1 has 5 bytes, and each element of VARY-FIELD-2 has 9 bytes. If CONTROL-1 and CONTROL-2 both contain the value 3, you can picture storage for VARY-FIELD-1 and VARY-FIELD-2 as follows:



To add a fourth element to VARY-FIELD-1, code as follows to prevent overlaying the first 5 bytes of VARY-FIELD-2. (GROUP-ITEM-2 serves as temporary storage for the variably located GROUP-ITEM-1.)

```
MOVE GROUP-ITEM-1 TO GROUP-ITEM-2.
ADD 1 TO CONTROL-1.
MOVE five-byte-field TO
  VARY-FIELD-1 (CONTROL-1).
MOVE GROUP-ITEM-2 TO GROUP-ITEM-1.
```

You can picture the updated storage for VARY-FIELD-1 and VARY-FIELD-2 as follows:



Note that the fourth element of `VARY-FIELD-1` did not overlay the first element of `VARY-FIELD-2`.

Appendix C. Converting double-byte character set (DBCS) data

The Language Environment service routines IGZCA2D and IGZCD2A were intended for converting alphanumeric data items that contain DBCS data to and from pure DBCS data items in order to reliably perform operations such as `STRING`, `UNSTRING`, and reference modification.

These service routines continue to be provided for compatibility; however, using national data items and the national conversion operations is now recommended instead for this purpose.

The service routines do not support a code-page argument and are not sensitive to the code page specified by the `CODEPAGE` compiler option. The `DBCS` compiler option does not affect their operation.

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 133
"Processing alphanumeric data items that contain DBCS data" on page 143

RELATED REFERENCES
"DBCS notation"
"Alphanumeric to DBCS data conversion (IGZCA2D)"
"DBCS to alphanumeric data conversion (IGZCD2A)" on page 666
"CODEPAGE" on page 305

DBCS notation

The symbols shown below are used in the DBCS data conversion examples to describe DBCS items.

Symbols	**Meaning**
< and >	Shift-out (SO) and shift-in (SI), respectively
D0, D1, D2, . . ., D*n*	Any DBCS character except for double-byte EBCDIC characters that correspond to single-byte EBCDIC characters
.A, .B, .C, . . .	Any double-byte EBCDIC character that corresponds to a single-byte EBCDIC character. The period (.) represents the value X'42'.
A single letter, such as A, B, or s	Any single-byte EBCDIC character

Alphanumeric to DBCS data conversion (IGZCA2D)

The Language Environment IGZCA2D service routine converts alphanumeric data that contains double-byte characters to pure DBCS data.

IGZCA2D syntax

To use the IGZCA2D service routine, pass the following four parameters to the routine by using the `CALL` statement:

parameter-1
The sending field for the conversion, handled as an alphanumeric data item.

parameter-2
The receiving field for the conversion, handled as a DBCS data item.

You cannot use reference modification with *parameter-2*.

parameter-3
The number of bytes in *parameter-1* to be converted.

It can be the `LENGTH OF` special register of *parameter-1*, or a 4-byte `USAGE IS BINARY` data item containing the number of bytes of *parameter-1* to be converted. Shift codes count as 1 byte each.

parameter-4
The number of bytes in *parameter-2* that will receive the converted data.

It can be the `LENGTH OF` special register of *parameter-2*, or a 4-byte `USAGE IS BINARY` data item containing the number of bytes of *parameter-2* to receive the converted data.

Usage notes

- You can pass *parameter-1*, *parameter-3*, and *parameter-4* to the routine `BY REFERENCE` or `BY CONTENT`, but you must pass *parameter-2* `BY REFERENCE`.
- The compiler does not perform syntax checking on these parameters. Ensure that the parameters are correctly set and passed in the `CALL` statement to the conversion routine. Otherwise, results are unpredictable.
- When creating *parameter-2* from *parameter-1*, IGZCA2D makes these changes:
  - Removes the shift codes, leaving the DBCS data unchanged
  - Converts the single-byte (nonspace) EBCDIC character X'*nn*' to a character represented by X'42*nn*'
  - Converts the single-byte space (X'40') to DBCS space (X'4040'), instead of X'4240'
- IGZCA2D does not change the contents of *parameter-1*, *parameter-3*, or *parameter-4*.
- | The valid range for the contents of *parameter-3* and for the contents of | *parameter-4* is 1 to 134,217,727.

"Example: IGZCA2D" on page 665

RELATED REFERENCES
"IGZCA2D return codes"

IGZCA2D return codes

`IGZCA2D` sets the `RETURN-CODE` special register to reflect the status of the conversion.

Table 91. **IGZCA2D return codes**

Return code	Explanation
0	*parameter-1* was converted and the results were placed in *parameter-2*.
2	*parameter-1* was converted and the results were placed in *parameter-2*. *parameter-2* was padded on the right with DBCS spaces.
4	*parameter-1* was converted and the results were placed in *parameter-2*. The DBCS data placed in *parameter-2* was truncated on the right.

Table 91. **IGZCA2D return codes** *(continued)*

Return code	Explanation
6	*parameter-1* was converted and the results were placed in *parameter-2*. A single-byte character in the range X′00′ to X′3F′ or X′FF′ was encountered. The valid single-byte character was converted into an out-of-range DBCS character.
8	*parameter-1* was converted and the results were placed in *parameter-2*. A single-byte character in the range X′00′ to X′3F′ or X′FF′ was encountered. The valid single-byte character was converted into an out-of-range DBCS character. *parameter-2* was padded on the right with DBCS spaces.
10	*parameter-1* was converted and the results were placed in *parameter-2*. A single-byte character in the range X′00′ to X′3F′ or X′FF′ was encountered. The valid single-byte character was converted into an out-of-range DBCS character. The DBCS data in *parameter-2* was truncated on the right.
12	An odd number of bytes was found between paired shift codes in *parameter-1*. No conversion occurred.
13	Unpaired or nested shift codes were found in *parameter-1*. No conversion occurred.
14	*parameter-1* and *parameter-2* were overlapping. No conversion occurred.
15	The value provided for *parameter-3* or *parameter-4* was out of range. No conversion occurred.
16	An odd number of bytes was coded in *parameter-4*. No conversion occurred.

Example: IGZCA2D

This example `CALL` statement converts the alphanumeric data in `alpha-item` to DBCS data. The results of the conversion are placed in `dbcs-item`.

```
CALL "IGZCA2D" USING BY REFERENCE alpha-item dbcs-item
    BY CONTENT LENGTH OF alpha-item LENGTH OF dbcs-item
```

Suppose the contents of `alpha-item` and `dbcs-item` and the lengths before the conversion are:

```
alpha-item = AB<D1D2D3>CD
dbcs-item  = D4D5D6D7D8D9D0
```

```
LENGTH OF alpha-item = 12
LENGTH OF dbcs-item  = 14
```

Then after the conversion, `alpha-item` and `dbcs-item` will contain:

```
alpha-item = AB<D1D2D3>CD
dbcs-item  = .A.BD1D2D3.C.D
```

The content of the `RETURN-CODE` register is 0.

RELATED REFERENCES
"DBCS notation" on page 663

DBCS to alphanumeric data conversion (IGZCD2A)

The Language Environment IGZCD2A routine converts pure DBCS data to alphanumeric data that can contain double-byte characters.

IGZCD2A syntax

To use the IGZCD2A service routine, pass the following four parameters to the routine using the `CALL` statement:

parameter-1
: The sending field for the conversion, handled as a DBCS data item.

parameter-2
: The receiving field for the conversion, handled as an alphanumeric data item.

parameter-3
: The number of bytes in *parameter-1* to be converted.

: It can be the `LENGTH OF` special register of *parameter-1*, or a 4-byte `USAGE IS BINARY` data item containing the number of bytes of *parameter-1* to be converted.

parameter-4
: The number of bytes in *parameter-2* that will receive the converted data.

: It can be the `LENGTH OF` special register of *parameter-2*, or a 4-byte `USAGE IS BINARY` data item containing the number of bytes of *parameter-2* to receive the converted data. Shift codes count as 1 byte each.

Usage notes

- You can pass *parameter-1*, *parameter-3*, and *parameter-4* to the routine `BY REFERENCE` or `BY CONTENT`, but you must pass *parameter-2* `BY REFERENCE`.
- The compiler does not perform syntax checking on these parameters. Ensure that the parameters are correctly set and passed to the conversion routine. Otherwise, results are unpredictable.
- When creating *parameter-2* from *parameter-1*, IGZCD2A makes these changes:
  - Inserts shift codes around DBCS characters that do not correspond to single-byte EBCDIC characters
  - Converts DBCS characters to single-byte characters when the DBCS characters correspond to single-byte EBCDIC characters
  - Converts the DBCS space (X'4040') to a single-byte space (X'40')
- IGZCD2A does not change the contents of *parameter-1*, *parameter-3*, or *parameter-4*.
- If the converted data contains double-byte characters, shift codes are counted in the length of *parameter-2*.
- The valid range for the contents of *parameter-3* and for the contents of *parameter-4* is 1 to 134,217,727.

"Example: IGZCD2A" on page 667

RELATED REFERENCES
"IGZCD2A return codes"

IGZCD2A return codes

`IGZCD2A` sets the `RETURN-CODE` special register to reflect the status of the conversion.

Table 92. **IGZCD2A return codes**

Return code	Explanation
0	*parameter-1* was converted and the results were placed in *parameter-2*.
2	*parameter-1* was converted and the results were placed in *parameter-2*. *parameter-2* was padded on the right with single-byte spaces.
4	*parameter-1* was converted and the results were placed in *parameter-2*. *parameter-2* was truncated on the right.[1]
14	*parameter-1* and *parameter-2* were overlapping. No conversion occurred.
15	The value of *parameter-3* or *parameter-4* was out of range. No conversion occurred.
16	An odd number of bytes was coded in *parameter-3*. No conversion occurred.

1. If a truncation occurs within the DBCS characters, the truncation is on an even-byte boundary and a shift-in (SI) is inserted. If necessary, the alphanumeric data is padded with a single-byte space after the shift-in.

Example: IGZCD2A

This example `CALL` statement converts the DBCS data in `dbcs-item` to alphanumeric data with double-byte characters. The results of the conversion are placed in `alpha-item`.

```
CALL "IGZCD2A" USING BY REFERENCE dbcs-item alpha-item
    BY CONTENT LENGTH OF dbcs-item LENGTH OF alpha-item
```

Suppose the contents of `dbcs-item` and `alpha-item` and the lengths before the conversion are:

```
dbcs-item  = .A.BD1D2D3.C.D
alpha-item = ssssssssssss

LENGTH OF dbcs-item  = 14
LENGTH OF alpha-item = 12
```

Then after the conversion, `dbcs-item` and `alpha-item` will contain:

```
dbcs-item  = .A.BD1D2D3.C.D
alpha-item = AB<D1D2D3>CD
```

The content of the `RETURN-CODE` register is 0.

RELATED REFERENCES
"DBCS notation" on page 663

Appendix D. XML reference material

This information describes the XML exception codes that the XML parser and the `XML GENERATE` statement return in special register `XML-CODE`. This information also documents which well-formedness constraints from the *XML specification* that the parser checks.

RELATED REFERENCES
"XML PARSE exceptions that allow continuation"
"XML PARSE exceptions that do not allow continuation" on page 672
"XML GENERATE exceptions" on page 677
"XML conformance" on page 675
XML specification

XML PARSE exceptions that allow continuation

The following table shows the exception codes that are associated with the XML `EXCEPTION` event and that the XML parser returns in special register `XML-CODE` when the parser can continue processing the XML data.

That is, the code is within one of the following ranges:

- 1-99
- 100,001-165,535

The table describes each exception and the actions that the parser takes when you request that it continue after the exception. Some of the descriptions use the following terms:

- *Basic document encoding*
- *Document encoding declaration*

For definitions of the terms, see the related task below about understanding the encoding of XML documents.

Table 93. **XML PARSE exceptions that allow continuation**

Code	Description	Parser action on continuation
1	The parser found an invalid character while scanning white space outside element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
2	The parser found an invalid start of a processing instruction, element, comment, or document type declaration outside element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.

Table 93. **XML PARSE exceptions that allow continuation** *(continued)*

Code	Description	Parser action on continuation
3	The parser found a duplicate attribute name.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
4	The parser found the markup character '<' in an attribute value.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
5	The start and end tag names of an element did not match.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
6	The parser found an invalid character in element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
7	The parser found an invalid start of an element, comment, processing instruction, or CDATA section in element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
8	The parser found in element content the CDATA closing character sequence ']]>' without the matching opening character sequence '<![CDATA['.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
9	The parser found an invalid character in a comment.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
10	The parser found in a comment the character sequence '--' (two hyphens) not followed by '>'.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
11	The parser found an invalid character in a processing instruction data segment.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.

Table 93. **XML PARSE exceptions that allow continuation** *(continued)*

Code	Description	Parser action on continuation
12	A processing instruction target name was 'xml' in lowercase, uppercase, or mixed case.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
13	The parser found an invalid digit in a hexadecimal character reference (of the form &#x*dddd*;).	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
14	The parser found an invalid digit in a decimal character reference (of the form &#*dddd*;).	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
15	The encoding declaration value in the XML declaration did not begin with lowercase or uppercase A through Z.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
16	A character reference did not refer to a legal XML character.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
17	The parser found an invalid character in an entity reference name.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
18	The parser found an invalid character in an attribute value.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
70	The basic document encoding was EBCDIC, and the `CODEPAGE` compiler option specified a supported EBCDIC code page, but the document encoding declaration did not specify a supported EBCDIC code page.	The parser uses the encoding specified by the `CODEPAGE` compiler option.
71	The basic document encoding was EBCDIC, and the document encoding declaration specified a supported EBCDIC encoding, but the `CODEPAGE` compiler option did not specify a supported EBCDIC code page.	The parser uses the encoding specified by the document encoding declaration.

Table 93. **XML PARSE exceptions that allow continuation** *(continued)*

Code	Description	Parser action on continuation
72	The basic document encoding was EBCDIC, the CODEPAGE compiler option did not specify a supported EBCDIC code page, and the document did not contain an encoding declaration.	The parser uses EBCDIC code page 1140 (USA, Canada, . . . Euro Country Extended Code Page).
73	The basic document encoding was EBCDIC, but neither the CODEPAGE compiler option nor the document encoding declaration specified a supported EBCDIC code page.	The parser uses EBCDIC code page 1140 (USA, Canada, . . . Euro Country Extended Code Page).
82	The basic document encoding was ASCII, but the document did not contain an encoding declaration.	The parser uses ASCII code page 819 (ISO-8859-1 Latin 1/Open Systems).
83	The basic document encoding was ASCII, but the document encoding declaration did not specify code page 813, 819, or 920.	The parser uses ASCII code page 819 (ISO-8859-1 Latin 1/Open Systems).
92	The document data item was alphanumeric, but the basic document encoding was Unicode UTF-16.	The parser uses code page 1200 (Unicode UTF-16).
100,001 - 165,535	The CODEPAGE compiler option and the document encoding declaration specified different supported EBCDIC code pages. XML-CODE contains the code page CCSID for the encoding declaration plus 100,000.	If you set XML-CODE to zero before returning from the EXCEPTION event, the parser uses the encoding specified by the CODEPAGE compiler option. If you set XML-CODE to the CCSID for the document encoding declaration (by subtracting 100,000), the parser uses this encoding.

RELATED CONCEPTS
"The content of XML-CODE" on page 499

RELATED TASKS
"Understanding the encoding of XML documents" on page 502
"Handling exceptions that the XML parser finds" on page 505

XML PARSE exceptions that do not allow continuation

With these XML exceptions, no further events are returned from the parser even if you set XML-CODE to zero and return control to the parser after processing the exception.

Control is passed to the statement that you specify in the ON EXCEPTION phrase, or to the end of the XML PARSE statement if you did not code an ON EXCEPTION phrase.

Table 94. **XML PARSE exceptions that do not allow continuation**

Code	Description
100	The parser reached the end of the document while scanning the start of the XML declaration.
101	The parser reached the end of the document while looking for the end of the XML declaration.

Table 94. **XML PARSE exceptions that do not allow continuation** *(continued)*

Code	Description
102	The parser reached the end of the document while looking for the root element.
103	The parser reached the end of the document while looking for the version information in the XML declaration.
104	The parser reached the end of the document while looking for the version information value in the XML declaration.
106	The parser reached the end of the document while looking for the encoding declaration value in the XML declaration.
108	The parser reached the end of the document while looking for the `standalone` declaration value in the XML declaration.
109	The parser reached the end of the document while scanning an attribute name.
110	The parser reached the end of the document while scanning an attribute value.
111	The parser reached the end of the document while scanning a character reference or entity reference in an attribute value.
112	The parser reached the end of the document while scanning an empty element tag.
113	The parser reached the end of the document while scanning the root element name.
114	The parser reached the end of the document while scanning an element name.
115	The parser reached the end of the document while scanning character data in element content.
116	The parser reached the end of the document while scanning a processing instruction in element content.
117	The parser reached the end of the document while scanning a comment or CDATA section in element content.
118	The parser reached the end of the document while scanning a comment in element content.
119	The parser reached the end of the document while scanning a CDATA section in element content.
120	The parser reached the end of the document while scanning a character reference or entity reference in element content.
121	The parser reached the end of the document while scanning after the close of the root element.
122	The parser found a possible invalid start of a document type declaration.
123	The parser found a second document type declaration.
124	The first character of the root element name was not a letter, ’_’, or ’:’.
125	The first character of the first attribute name of an element was not a letter, ’_’, or ’:’.
126	The parser found an invalid character either in or following an element name.
127	The parser found a character other than ’=’ following an attribute name.
128	The parser found an invalid attribute value delimiter.
130	The first character of an attribute name was not a letter, ’_’, or ’:’.

Table 94. **XML PARSE exceptions that do not allow continuation** *(continued)*

Code	Description
131	The parser found an invalid character either in or following an attribute name.
132	An empty element tag was not terminated by a '>' following the '/'.
133	The first character of an element end tag name was not a letter, '_', or ':'.
134	An element end tag name was not terminated by a '>'.
135	The first character of an element name was not a letter, '_', or ':'.
136	The parser found an invalid start of a comment or CDATA section in element content.
137	The parser found an invalid start of a comment.
138	The first character of a processing instruction target name was not a letter, '_', or ':'.
139	The parser found an invalid character in or following a processing instruction target name.
140	A processing instruction was not terminated by the closing character sequence '?>'.
141	The parser found an invalid character following '&' in a character reference or entity reference.
142	The version information was not present in the XML declaration.
143	'version' in the XML declaration was not followed by '='.
144	The version declaration value in the XML declaration is either missing or improperly delimited.
145	The version information value in the XML declaration specified a bad character, or the start and end delimiters did not match.
146	The parser found an invalid character following the version information value closing delimiter in the XML declaration.
147	The parser found an invalid attribute instead of the optional encoding declaration in the XML declaration.
148	'encoding' in the XML declaration was not followed by '='.
149	The encoding declaration value in the XML declaration is either missing or improperly delimited.
150	The encoding declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
151	The parser found an invalid character following the encoding declaration value closing delimiter in the XML declaration.
152	The parser found an invalid attribute instead of the optional `standalone` declaration in the XML declaration.
153	`standalone` in the XML declaration was not followed by `=`.
154	The `standalone` declaration value in the XML declaration is either missing or improperly delimited.
155	The `standalone` declaration value was neither 'yes' nor 'no' only.
156	The `standalone` declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
157	The parser found an invalid character following the `standalone` declaration value closing delimiter in the XML declaration.

Table 94. **XML PARSE exceptions that do not allow continuation** *(continued)*

Code	Description
158	The XML declaration was not terminated by the proper character sequence '?>', or contained an invalid attribute.
159	The parser found the start of a document type declaration after the end of the root element.
160	The parser found the start of an element after the end of the root element.
315	The basic document encoding was UTF-16 little-endian, which the parser does not support on this platform.
316	The basic document encoding was UCS4, which the parser does not support.
317	The parser cannot determine the document encoding. The document might be damaged.
318	The basic document encoding was UTF-8, which the parser does not support.
320	The document data item was national, but the basic document encoding was EBCDIC.
321	The document data item was national, but the basic document encoding was ASCII.
500-599	Internal error. Please report the error to your service representative.

RELATED CONCEPTS
"The content of XML-CODE" on page 499

RELATED TASKS
"Handling exceptions that the XML parser finds" on page 505

XML conformance

The XML parser that is included in Enterprise COBOL is not a conforming XML processor according to the definition in the *XML specification*. It does not validate the XML documents that you parse. Although it does check for many well-formedness errors, it does not perform all of the actions required of a nonvalidating XML processor.

In particular, it does not process the internal document type definition (DTD internal subset). Thus it does not supply default attribute values, does not normalize attribute values, and does not include the replacement text of internal entities except for the predefined entities. Instead, it passes the entire document type declaration as the contents of `XML-TEXT` or `XML-NTEXT` for the `DOCUMENT-TYPE-DESCRIPTOR` XML event, which allows the application to perform these actions if required.

The parser optionally allows programs to continue processing an XML document after some errors. The purpose of allowing processing to continue is to facilitate the debugging of XML documents and processing procedures.

Recapitulating the definition in the *XML specification*, a textual object is a *well-formed* XML document if:

- Taken as a whole, it conforms to the grammar for XML documents.
- It meets all the explicit well-formedness constraints given in the *XML specification*.

- Each parsed entity (piece of text) that is referenced directly or indirectly within the document is well formed.

The COBOL XML parser does check that documents conform to the XML grammar, except for any document type declaration. The declaration is supplied in its entirety, unchecked, to your application.

The *XML specification* also contains twelve explicit well-formedness constraints. The constraints that the COBOL XML parser checks partly or completely are shown in **bold** type:

1. Parameter Entities (PEs) in Internal Subset: “In the internal DTD subset, parameter-entity references can occur only where markup declarations can occur, not within markup declarations. (This does not apply to references that occur in external parameter entities or to the external subset.)”

   The parser does not process the internal DTD subset, so it does not enforce this constraint.
2. External Subset: “The external subset, if any, must match the production for extSubset.”

   The parser does not process the external subset, so it does not enforce this constraint.
3. Parameter Entity Between Declarations: “The replacement text of a parameter entity reference in a DeclSep must match the production extSubsetDecl.”

   The parser does not process the internal DTD subset, so it does not enforce this constraint.
4. **Element Type Match:** “The Name in an element’s end-tag must match the element type in the start-tag.”

   The parser enforces this constraint.
5. **Unique Attribute Specification:** “No attribute name may appear more than once in the same start-tag or empty-element tag.”

   The parser partly supports this constraint by checking up to 10 attribute names in a given element for uniqueness. The application can check any attribute names beyond this limit.
6. No External Entity References: “Attribute values cannot contain direct or indirect entity references to external entities.”

   The parser does not enforce this constraint.
7. No ’<’ in Attribute Values: “The replacement text of any entity referred to directly or indirectly in an attribute value must not contain a ’<’.”

   The parser does not enforce this constraint.
8. **Legal Character:** “Characters referred to using character references must match the production for Char.”

   The parser enforces this constraint.
9. Entity Declared: “In a document without any DTD, a document with only an internal DTD subset which contains no parameter entity references, or a document with standalone=’yes’, for an entity reference that does not occur

within the external subset or a parameter entity, the Name given in the entity reference must match that in an entity declaration that does not occur within the external subset or a parameter entity, except that well-formed documents need not declare any of the following entities: amp, lt, gt, apos, quot. The declaration of a general entity must precede any reference to it which appears in a default value in an attribute-list declaration.

Note that if entities are declared in the external subset or in external parameter entities, a non-validating processor is not obligated to read and process their declarations; for such documents, the rule that an entity must be declared is a well-formedness constraint only if standalone='yes'."

The parser does not enforce this constraint.

10. Parsed Entity: "An entity reference must not contain the name of an unparsed entity. Unparsed entities may be referred to only in attribute values declared to be of type ENTITY or ENTITIES."

    The parser does not enforce this constraint.
11. No Recursion: "A parsed entity must not contain a recursive reference to itself, either directly or indirectly."

    The parser does not enforce this constraint.
12. In DTD: "Parameter-entity references may only appear in the DTD."

    The parser does not enforce this constraint, because the error cannot occur.

The preceding material is an annotation from the *XML specification*. The W3C is not responsible for any content not found at the original URL (www.w3.org/TR/REC-xml); all these annotations are non-normative. This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. The normative version of the specification is the English version found at the W3C site; any translated document may contain errors from the translation.

RELATED CONCEPTS
"XML parser in COBOL" on page 487

RELATED REFERENCES
Extensible Markup Language (XML)
2.8 Prolog and document type declaration (*XML specification*)

XML GENERATE exceptions

One of several exception codes might be returned in special register `XML-CODE` during XML generation. If one of these exceptions occurs, control is passed to the statement in the `ON EXCEPTION` phrase, or to the end of the `XML GENERATE` statement if you did not code an `ON EXCEPTION` phrase.

Table 95. **XML GENERATE exceptions**

Code	Description
400	The receiver was too small to contain the generated XML document. The `COUNT IN` data item, if specified, contains the count of character positions that were actually generated.
401	A DBCS data-name contained a character that, when converted to Unicode, was not valid in an XML element name.
402	The first character of a DBCS data-name, when converted to Unicode, was not valid as the first character of an XML element name.

Table 95. **XML GENERATE exceptions** *(continued)*

Code	Description
403	The value of an `OCCURS DEPENDING ON` variable exceeded 16,777,215.
410	The code page specified by the `CODEPAGE` compiler option is not supported for conversion to Unicode.
411	The code page specified by the `CODEPAGE` compiler option is not a supported single-byte EBCDIC code page.
600-699	Internal error. Report the error to your service representative.

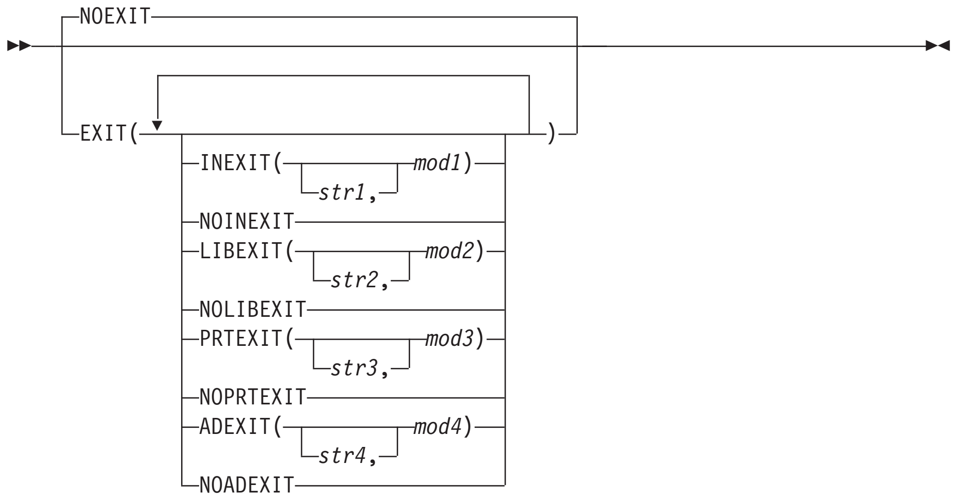
RELATED TASKS
"Handling errors in generating XML output" on page 522

Appendix E. 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.

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.

EXIT option syntax



Default is: NOEXIT

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. You can specify the suboptions in any order and separate them 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 once, the last one specified takes effect.

You can specify the EXIT option only at invocation in the JCL PARM field (under TSO/E, in a command argument) or at installation time. Do not specify the EXIT option in a PROCESS (CBL) statement.

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 copybooks 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. They can be up to 64 characters in length, and you must enclose them in single quotation marks. Any character is allowed, but included single quotation marks must be doubled. Lowercase characters are folded to uppercase.

If one of *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) that contains the length of the string.

"Example: INEXIT user exit" on page 689

RELATED TASKS
"Using the user-exit work area"
"Calling from exit modules" on page 681
"Using the EXIT compiler option with CICS and SQL statements" on page 688

RELATED REFERENCES
"Processing of INEXIT" on page 681
"Processing of LIBEXIT" on page 682
"Processing of PRTEXIT" on page 685
"Processing of ADEXIT" on page 686
"Error handling for exit modules" on page 687

Using the user-exit work area

When you use an exit, the compiler provides a user-exit work area where you can save the address of `GETMAIN` storage obtained by the exit module. This work area allows the module to be reentrant.

The user-exit work area is 4 fullwords residing on a fullword boundary. These fullwords are initialized to binary zeros 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.

You 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, the PRTEXIT module uses the third word, and the ADEXIT module uses the fourth word.

RELATED REFERENCES
"Processing of INEXIT"
"Processing of LIBEXIT" on page 682
"Processing of PRTEXIT" on page 685
"Processing of ADEXIT" on page 686

Calling from exit modules

Use COBOL standard linkage in your exit modules to call COBOL programs or library routines. You need to be aware of these conventions in order to trace the call chain correctly.

When a call is made to a program or to a routine, the registers are set up as follows:

R1	Points to the parameter list passed to the called program or library routine
R13	Points to the register save area provided by the calling program or routine
R14	Holds the return address of the calling program or routine
R15	Holds the address of the called program or routine

Exit modules must have the `RMODE` attribute of `24` and the `AMODE` attribute of `ANY`.

RELATED CONCEPTS
"Storage and its addressability" on page 41

Processing of INEXIT

The exit module is used to read source code from a user-supplied load module in place of SYSIN.

Table 96. **INEXIT processing**

Action by compiler	Resulting action by exit module
Loads the exit module (*mod1*) during initialization	
Calls the exit module with an OPEN operation code (op code)	Prepares its source for processing. Passes the status of the OPEN request back to the compiler.
Calls the exit module with a GET op code when a source statement is needed	Returns either the address and length of the next statement or the end-of-data indication (if no more source statements exist)
Calls the exit module with a CLOSE op code when the end-of-data is presented	Releases any resources that are related to its output

INEXIT parameters

The compiler uses a parameter list to communicate with the exit module. The parameter list consists of 10 fullwords containing addresses, and register 1 contains the address of the parameter list. 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. The following table describes the contents of the parameter list.

Table 97. **INEXIT parameters**

Offset	Contains address of	Description of item
00	User-exit type	Halfword identifying which user exit is to perform the operation. 1=INEXIT
04	Operation code	Halfword indicating the type of operation. 0=OPEN; 1=CLOSE; 2=GET
08	Return code	Fullword, placed by the exit module, indicating the success of the requested operation. 0=Operation was successful; 4=End-of-data; 12=Operation failed
12	User-exit work area	Four-fullword work area provided by the compiler, for use by the user-exit module
16	Data length	Fullword, placed by the exit module, specifying the length of the record being returned by the GET operation (must be 80)
20	Data or *str1*	Fullword, placed by the exit module, containing the address of the record in a user-owned buffer, for the GET operation. *str1* applies only to OPEN. The first halfword (on a halfword boundary) contains the length of the string, followed by the string.
24	Not used	(Used only by LIBEXIT)
28	Not used	(Used only by LIBEXIT)
32	Not used	(Used only by LIBEXIT)
36	Not used	(Used only by LIBEXIT)

"Example: INEXIT user exit" on page 689

RELATED TASKS
"Using the EXIT compiler option with CICS and SQL statements" on page 688

RELATED REFERENCES
"Processing of LIBEXIT"

Processing of LIBEXIT

The exit module is used in place of the SYSLIB, or *library-name*, data set. Calls are made to the module by the compiler to obtain copybooks whenever `COPY` or `BASIS` statements are encountered.

If `LIBEXIT` is specified, the `LIB` compiler option must be in effect.

Table 98. **LIBEXIT processing**

Action by compiler	Resulting action by exit module
Loads the exit module (*mod2*) during initialization	

Table 98. **LIBEXIT processing** *(continued)*

Action by compiler	Resulting action by exit module
Calls the exit module with an OPEN operation code (op code)	Prepares the specified *library-name* for processing. Passes the status of the OPEN request to the compiler.
Calls the exit module with a FIND op code if the *library-name* was successfully opened	Establishes positioning at the requested *text-name* (or *basis-name*) in the specified *library-name*; this place becomes the active copybook. Passes an appropriate return code to the compiler when positioning is complete.
Calls the exit module with a GET op code	Passes the compiler either the length and address of the record to be copied from the active copybook or the end-of-data indicator
Calls the exit module with a CLOSE op code when the end-of-data is presented	Releases any resources that are related to its input

Processing of LIBEXIT with nested COPY statements

Any record from the active copybook 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.)

The compiler does not allow recursive calls to *text-name*. That is, a copybook can be named only once in a set of nested `COPY` statements until the end-of-data for that copybook is reached.

The following table shows how the processing of `LIBEXIT` changes when there are one or more valid `COPY` statements that are not nested:

Table 99. **LIBEXIT processing with nonnested COPY statements**

Action by compiler	Resulting action by exit module
Loads the exit module (*mod2*) during initialization	
Calls the exit module with an OPEN operation code (op code)	Prepares the specified *library-name* for processing. Passes the status of the OPEN request to the compiler.
Calls the exit module with a FIND op code if the *library-name* was successfully opened	Establishes positioning at the requested *text-name* (or *basis-name*) in the specified *library-name*; this place becomes the active copybook. Passes an appropriate return code to the compiler when positioning is complete.
Calls the exit module with a FIND op code if the *library-name* was successfully opened	Reestablishes positioning at the previous active copybook. Passes an appropriate return code to the compiler when positioning is complete.
Calls the exit module with a GET op code. Verifies that the same record was passed.	Passes the compiler the same record as was passed previously from this copybook. After verification, passes either the length and address of the record to be copied from the active copybook or the end-of-data indicator.
Calls the exit module with a CLOSE op code when the end-of-data is presented	Releases any resources that are related to its input

The following table shows how the processing of `LIBEXIT` changes when the compiler encounters a valid nested `COPY` statement.

Table 100. **LIBEXIT processing with nested COPY statements**

Action by compiler	Resulting action by exit module
If the requested *library-name* from the nested `COPY` statement was not previously opened, calls the exit module with an OPEN op code	Pushes its control information about the active copybook onto a stack. Completes the requested action (OPEN). The newly requested *text-name* (or *basis-name*) becomes the active copybook.
Calls the exit module with a FIND op code for the requested new *text-name*	Pushes its control information about the active copybook onto a stack. Completes the requested action (FIND). The newly requested *text-name* (or *basis-name*) becomes the active copybook.
Calls the exit module with a GET op code	Passes the compiler either the length and address of the record to be copied from the active copybook or the end-of-data indicator. At end-of-data, pops its control information from the stack.

LIBEXIT parameters

The compiler uses a parameter list to communicate with the exit module. The parameter list consists of 10 fullwords that contain addresses, and register 1 contains the address of the parameter list. 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.

Table 101. **LIBEXIT parameters**

Offset	Contains address of	Description of item
00	User-exit type	Halfword identifying which user exit is to perform the operation. 2=LIBEXIT
04	Operation code	Halfword indicating the type of operation. 0=OPEN; 1=CLOSE; 2=GET; 4=FIND
08	Return code	Fullword, placed by the exit module, indicating the success of the requested operation. 0=Operation was successful; 4=End-of-data; 12=Operation failed
12	User-exit work area	Four-fullword work area provided by the compiler for use by the user-exit module
16	Data length	Fullword, placed by the exit module, specifying the length of the record being returned by the GET operation (must be 80)
20	Data or *str2*	Fullword, placed by the exit module, containing the address of the record in a user-owned buffer, for the `GET` operation. *str2* applies only to OPEN. The first halfword (on a halfword boundary) contains the length of the string, followed by the string.

Table 101. **LIBEXIT parameters** *(continued)*

Offset	Contains address of	Description of item
24	System *library-name*	Eight-character area containing the *library-name* from the COPY statement. Processing and conversion rules for a program-name are applied. Padded with blanks if required. Applies to OPEN, CLOSE, and FIND.
28	System *text-name*	Eight-character area containing the *text-name* from the COPY statement (*basis-name* from BASIS statement). Processing and conversion rules for a *program name* are applied. Padded with blanks if required. Applies only to FIND.
32	Library-name	Thirty-character area containing the full *library-name* from the COPY statement. Padded with blanks if required, and used as-is (not folded to uppercase). Applies to OPEN, CLOSE, and FIND.
36	Text-name	Thirty-character area containing the full *text-name* from the COPY statement. Padded with blanks if required, and used as-is (not folded to uppercase). Applies only to FIND.

RELATED TASKS
"Using the EXIT compiler option with CICS and SQL statements" on page 688

Processing of PRTEXIT

The exit module is used in place of the SYSPRINT data set.

Table 102. **PRTEXIT processing**

Action by compiler	Resulting action by exit module
Loads the exit module (*mod3*) during initialization	
Calls the exit module with an OPEN operation code (op code)	Prepares its output destination for processing. Passes the status of the OPEN request to the compiler.
Calls the exit modules with a PUT op code when a line is to be printed, supplying the address and length of the record that is to be printed	Passes 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.
Calls the exit module with a CLOSE op code when the end-of-data is presented	Releases any resources that are related to its output destination

PRTEXIT parameters

The compiler uses a parameter list to communicate with the exit module. The parameter list consists of 10 fullwords that contain addresses, and register 1 contains the address of the parameter list. The return code, data length, and data buffer 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.

Table 103. **PRTEXIT parameters**

Offset	Contains address of	Description of item
00	User-exit type	Halfword identifying which user exit is to perform the operation. 3=PRTEXIT
04	Operation code	Halfword indicating the type of operation. 0=OPEN; 1=CLOSE; 3=PUT
08	Return code	Fullword, placed by the exit module, indicating the success of the requested operation. 0=Operation was successful; 12=Operation failed
12	User-exit work area	Four-fullword work area provided by the compiler, for use by the user-exit module
16	Data length	Fullword specifying the length of the record being supplied by the PUT operation (the compiler sets this value to 133)
20	Data buffer or *str3*	Fullword containing the address of the data buffer where the compiler has placed the record to be printed by the PUT operation. *str3* applies only to OPEN. The first halfword (on a halfword boundary) contains the length of the string, followed by the string.
24	Not used	(Used only by LIBEXIT)
28	Not used	(Used only by LIBEXIT)
32	Not used	(Used only by LIBEXIT)
36	Not used	(Used only by LIBEXIT)

RELATED TASKS
"Using the EXIT compiler option with CICS and SQL statements" on page 688

RELATED REFERENCES
"Processing of LIBEXIT" on page 682

Processing of ADEXIT

Use of the ADEXIT module requires the compiler option `ADATA` to produce SYSADATA output, and the `DD` statement `SYSADATA`.

Table 104. **ADEXIT processing**

Action by compiler	Resulting action by exit module
Loads the exit module (*mod4*) during initialization	
Calls the exit module with an OPEN operation code (op code)	Prepares its output destination for processing. Passes the status of the OPEN request to the compiler.
Calls the exit modules with a PUT op code when the compiler has written a SYSADATA record, supplying the address and length of the SYSADATA record	Passes the status of the PUT request to the compiler by a return code

Table 104. **ADEXIT processing** *(continued)*

Action by compiler	Resulting action by exit module
Calls the exit module with a CLOSE op code when the end-of-data is presented	Releases any resources

ADEXIT parameters

The compiler uses a parameter list to communicate with the exit module. The parameter list consists of 10 fullwords that contain addresses, and register 1 contains the address of the parameter list. The return code, data length, and data buffer 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.

Table 105. **ADEXIT parameters**

Offset	Contains address of	Description of item
00	User-exit type	Halfword identifying which user exit is to perform the operation. 4=ADEXIT
04	Operation code	Halfword indicating the type of operation. 0=OPEN; 1=CLOSE; 3=PUT
08	Return code	Fullword, placed by the exit module, indicating the success of the requested operation. 0=Operation was successful; 12=Operation failed
12	User-exit work area	Four-fullword work area provided by the compiler, for use by the user-exit module
16	Data length	Fullword specifying the length of the record being supplied by the PUT operation
20	Data buffer or *str4*	Fullword containing the address of the data buffer where the compiler has placed the record to be printed by the PUT operation. *str4* applies only to OPEN. The first halfword (on a halfword boundary) contains the length of the string, followed by the string.
24	Not used	(Used only by LIBEXIT)
28	Not used	(Used only by LIBEXIT)
32	Not used	(Used only by LIBEXIT)
36	Not used	(Used only by LIBEXIT)

RELATED TASKS
"Using the EXIT compiler option with CICS and SQL statements" on page 688

RELATED REFERENCES
"Processing of LIBEXIT" on page 682

Error handling for exit modules

The compiler reports an error message whenever an exit module cannot be loaded or an exit module returns an "operation failed" message or nonzero return code.

Message IGYSI5008 is written to the operator and the compiler terminates with return code 16 when any of the following events occurs:

- An exit module cannot be loaded.
- A nonzero return code is received from INEXIT during an OPEN request.
- A nonzero return code is received from PRTEXIT during an OPEN request.

The exit type and operation (OPEN or LOAD) are identified in the message. Any other error from INEXIT or PRTEXIT causes the compiler to terminate.

The compiler detects and reports the following conditions:

5203 PUT request to SYSPRINT user exit failed with return code *nn*.

5204 Record address not set by *exit-name* user exit.

5205 GET request from SYSIN user exit failed with return code *nn*.

5206 Record length not set by *exit-name* user exit.

Using the EXIT compiler option with CICS and SQL statements

When you compile using suboptions of the `EXIT` compiler option and you need to translate CICS or SQL statements, the actions that you can take in the exit modules depend on whether you use the separate CICS translator and DB2 precompiler or the integrated CICS translator and DB2 coprocessor.

When you use the integrated translators, you can process `EXEC CICS` and `EXEC SQL` statements in the exit modules. The following table shows your alternatives for the four exit modules.

Table 106. **Actions allowed on CICS and SQL statements in exit modules**

Compile with suboption	Translate with integrated CICS translator and DB2 coprocessor	Translate with separate CICS translator and DB2 coprocessor	Actions allowed in module	Comments
`INEXIT`	Yes	No	Can process `EXEC CICS` and `EXEC SQL` statements in the INEXIT module	The INEXIT module does not get control of the COBOL statements that are generated for the `EXEC` statements.
	No	Yes	Can process the COBOL statements that are generated for the `EXEC` statements in the INEXIT module	You can change the generated statements in the INEXIT module, but doing so is not supported by IBM.
`LIBEXIT`	Yes	No	Can process in the LIBEXIT module the statements that are brought in by the `EXEC SQL INCLUDE` statements. Can process `EXEC CICS` source statements in the LIBEXIT module.	`EXEC SQL INCLUDE` statements are processed like COBOL `COPY` statements.
	No	Yes	Can process the COBOL statements that are generated for the `EXEC CICS` statements in the LIBEXIT module	You can process the input statements that are brought in by the `EXEC SQL INCLUDE` statements only by using the `INEXIT` suboption.

Table 106. **Actions allowed on CICS and SQL statements in exit modules** *(continued)*

Compile with suboption	Translate with integrated CICS translator and DB2 coprocessor	Translate with separate CICS translator and DB2 coprocessor	Actions allowed in module	Comments
PRTEXIT	Yes	No	Can process the EXEC CICS and EXEC SQL source statements from the SOURCE listing in the PRTEXIT module	The PRTEXIT module does not have access to the COBOL source statements that are generated.
	No	Yes	Can process the COBOL SOURCE listing statements that are generated for the EXEC statements in the PRTEXIT module	
ADEXIT	Yes	No	Can process the EXEC CICS and EXEC SQL source statements in the ADEXIT module	The ADEXIT module does not have access to the COBOL source statements that are generated.
	No	Yes	Can process the COBOL SYSADATA source statements that are generated for the EXEC statements in the ADEXIT module	

RELATED CONCEPTS
"DB2 coprocessor" on page 405
"Integrated CICS translator" on page 399

RELATED TASKS
"Compiling with the SQL option" on page 409
"Compiling with the CICS option" on page 397

RELATED REFERENCES
"Processing of INEXIT" on page 681
"Processing of LIBEXIT" on page 682
"Processing of PRTEXIT" on page 685
"Processing of ADEXIT" on page 686

Example: INEXIT user exit

The following example shows an INEXIT user-exit module in COBOL.

```
************************************************************
*                                                          *
* Name:  SKELINX                                           *
*                                                          *
* Function:  Example of an INEXIT user exit written        *
*            in the COBOL language.                        *
*                                                          *
************************************************************

 Identification Division.
    Program-ID.  Skelinx.

 Environment Division.
 Data Division.

   WORKING-STORAGE Section.

*   *****************************************************
*   *                                                   *
*   *  Local variables.                                 *
*   *                                                   *
*   *****************************************************

    01  Record-Variable    Pic X(80).

*   *****************************************************
*   *                                                   *
*   *  Definition of the User-Exit Parameter List, which  *
*   *  is passed from the COBOL compiler to the user exit *
*   *  module.                                          *
*   *                                                   *
*   *****************************************************

   Linkage Section.
    01  Exit-Type         Pic 9(4)   Binary.
    01  Exit-Operation    Pic 9(4)   Binary.
    01  Exit-ReturnCode   Pic 9(9)   Binary.
    01  Exit-WorkArea.
        05  INEXIT-Slot   Pic 9(9)   Binary.
        05  LIBEXIT-Slot  Pic 9(9)   Binary.
        05  PRTEXIT-Slot  Pic 9(9)   Binary.
        05  Reserved-Slot Pic 9(9)   Binary.
    01  Exit-DataLength   Pic 9(9)   Binary.
    01  Exit-DataArea     Pointer.
    01  Exit-Open-Parm    Redefines  Exit-DataArea.
        05  String-Len    Pic 9(4)   Binary.
        05  Open-String   Pic X(64).
    01  Exit-Print-Line   Redefines  Exit-DataArea  Pic X(133).
    01  Exit-LIBEXIT      Pic X(8).
    01  Exit-Systext      Pic X(8).
    01  Exit-CBLLibrary   Pic X(30).
    01  Exit-CBLText      Pic X(30).

*********************************************************
*                                                       *
*  Begin PROCEDURE DIVISION                             *
*                                                       *
*  Invoke the section to handle the exit.               *
*                                                       *
*********************************************************

 Procedure Division Using Exit-Type       Exit-Operation
                          Exit-ReturnCode Exit-WorkArea
                          Exit-DataLength Exit-DataArea
                          Exit-LIBEXIT    Exit-Systext
                          Exit-CBLLibrary Exit-CBLText.

     Evaluate Exit-type
       When (1) Perform Handle-INEXIT
       When (2) Perform Handle-LIBEXIT
       When (3) Perform Handle-PRTEXIT
     End-Evaluate
     Move 16 To Exit-ReturnCode
     Goback.

***********************************************
*   I N E X I T   E X I T   P R O C E S S O R *
***********************************************
 Handle-INEXIT.
```

```
     Evaluate Exit-Operation
       When (0) Perform INEXIT-Open
       When (1) Perform INEXIT-Close
       When (2) Perform INEXIT-Get
     End-Evaluate

     Move 16 To Exit-ReturnCode
     Goback.

 INEXIT-Open.
*    -----------------------------------------------------
*    Prepare for reading source
*    -----------------------------------------------------
     Goback.

 INEXIT-Close.
*    -----------------------------------------------------
*    Release resources
*    -----------------------------------------------------
     Goback.

 INEXIT-Get.
*    -----------------------------------------------------
*    Retrieve next source record
*    -----------------------------------------------------

*    -----------------------------------------------------
*    Return the address of the record to the compiler.
*    -----------------------------------------------------
     Set Exit-DataArea to Address of Record-Variable

*    -----------------------------------------------------
*    Set length of record in User-Exit Parameter List
*    -----------------------------------------------------
     Move 80 To Exit-DataLength

     Goback.

****************************************************
*    L I B E X I T   P R O C E S S O R            *
****************************************************
 Handle-LIBEXIT.
     Display "**** This module for INEXIT only"
     Move 16 To Exit-ReturnCode
     Goback.

********************************************************
*    P R I N T  E X I T   P R O C E S S O R          *
********************************************************
 Handle-PRTEXIT.
     Display "**** This module for INEXIT only"
     Move 16 To Exit-ReturnCode
     Goback.
********************************************************

 End Program Skelinx.
```

Appendix F. JNI.cpy

This listing shows the copybook JNI.cpy, which you can use to access the Java Native Interface (JNI) services from your COBOL programs.

JNI.cpy contains sample COBOL data definitions that correspond to the Java JNI types, and contains JNINativeInterface, the JNI environment structure that contains function pointers for accessing the JNI callable services.

JNI.cpy is in the HFS in the `include` subdirectory of the COBOL install directory (typically /usr/lpp/cobol/include). JNI.cpy is analogous to the header file jni.h that C programmers use to access the JNI.

```
*****************************************************************
* COBOL declarations for Java native method interoperation      *
*                                                               *
* To use the Java Native Interface callable services from a     *
* COBOL program:                                                *
* 1) Use a COPY statement to include this file into the         *
*    the Linkage Section of the program, e.g.                   *
*      Linkage Section.                                         *
*      Copy JNI                                                 *
* 2) Code the following statements at the beginning of the      *
*    Procedure Division:                                        *
*      Set address of JNIEnv to JNIEnvPtr                       *
*      Set address of JNINativeInterface to JNIEnv              *
*****************************************************************
*
* Sample JNI type definitions in COBOL
*
*01 jboolean1 pic X.
*  88 jboolean1-true  value X'01' through X'FF'.
*  88 jboolean1-false value X'00'.
*
*01 jbyte1 pic X.
*
*01 jchar1 pic N usage national.
*
*01 jshort1 pic s9(4)  comp-5.
*01 jint1   pic s9(9)  comp-5.
*01 jlong1  pic s9(18) comp-5.
*
*01 jfloat1  comp-1.
*01 jdouble1 comp-2.
*
*01 jobject1 object reference.
*01 jclass1  object reference.
*01 jstring1 object reference jstring.
*01 jarray1  object reference jarray.
*
*01 jbooleanArray1 object reference jbooleanArray.
*01 jbyteArray1    object reference jbyteArray.
*01 jcharArray1    object reference jcharArray.
*01 jshortArray1   object reference jshortArray.
*01 jintArray1     object reference jintArray.
*01 jlongArray1    object reference jlongArray.
*01 floatArray1    object reference floatArray.
*01 jdoubleArray1  object reference jdoubleArray.
*01 jobjectArray1  object reference jobjectArray.


* Possible return values for JNI functions.
```

```
 01 JNI-RC pic S9(9) comp-5.
* success
   88 JNI-OK        value  0.
* unknown error
   88 JNI-ERR       value -1.
* thread detached from the VM
   88 JNI-EDETACHED value -2.
* JNI version error
   88 JNI-EVERSION  value -3.
* not enough memory
   88 JNI-ENOMEM    value -4.
* VM already created
   88 JNI-EEXIST    value -5.
* invalid arguments
   88 JNI-EINVAL    value -6.

* Used in ReleaseScalarArrayElements
 01 releaseMode pic s9(9) comp-5.
   88 JNI-COMMIT value 1.
   88 JNI-ABORT  value 2.

 01 JNIenv pointer.

* JNI Native Method Interface - environment structure.
 01 JNINativeInterface.
   02 pointer.
   02 pointer.
   02 pointer.
   02 pointer.
   02 GetVersion                       function-pointer.
   02 DefineClass                      function-pointer.
   02 FindClass                        function-pointer.
   02 FromReflectedMethod              function-pointer.
   02 FromReflectedField               function-pointer.
   02 ToReflectedMethod                function-pointer.
   02 GetSuperclass                    function-pointer.
   02 IsAssignableFrom                 function-pointer.
   02 ToReflectedField                 function-pointer.
   02 Throw                            function-pointer.
   02 ThrowNew                         function-pointer.
   02 ExceptionOccurred                function-pointer.
   02 ExceptionDescribe                function-pointer.
   02 ExceptionClear                   function-pointer.
   02 FatalError                       function-pointer.
   02 PushLocalFrame                   function-pointer.
   02 PopLocalFrame                    function-pointer.
   02 NewGlobalRef                     function-pointer.
   02 DeleteGlobalRef                  function-pointer.
   02 DeleteLocalRef                   function-pointer.
   02 IsSameObject                     function-pointer.
   02 NewLocalRef                      function-pointer.
   02 EnsureLocalCapacity              function-pointer.
   02 AllocObject                      function-pointer.
   02 NewObject                        function-pointer.
   02 NewObjectV                       function-pointer.
   02 NewObjectA                       function-pointer.
   02 GetObjectClass                   function-pointer.
   02 IsInstanceOf                     function-pointer.
   02 GetMethodID                      function-pointer.
   02 CallObjectMethod                 function-pointer.
   02 CallObjectMethodV                function-pointer.
   02 CallObjectMethodA                function-pointer.
   02 CallBooleanMethod                function-pointer.
   02 CallBooleanMethodV               function-pointer.
   02 CallBooleanMethodA               function-pointer.
   02 CallByteMethod                   function-pointer.
   02 CallByteMethodV                  function-pointer.
```

```
02 CallByteMethodA                    function-pointer.
02 CallCharMethod                     function-pointer.
02 CallCharMethodV                    function-pointer.
02 CallCharMethodA                    function-pointer.
02 CallShortMethod                    function-pointer.
02 CallShortMethodV                   function-pointer.
02 CallShortMethodA                   function-pointer.
02 CallIntMethod                      function-pointer.
02 CallIntMethodV                     function-pointer.
02 CallIntMethodA                     function-pointer.
02 CallLongMethod                     function-pointer.
02 CallLongMethodV                    function-pointer.
02 CallLongMethodA                    function-pointer.
02 CallFloatMethod                    function-pointer.
02 CallFloatMethodV                   function-pointer.
02 CallFloatMethodA                   function-pointer.
02 CallDoubleMethod                   function-pointer.
02 CallDoubleMethodV                  function-pointer.
02 CallDoubleMethodA                  function-pointer.
02 CallVoidMethod                     function-pointer.
02 CallVoidMethodV                    function-pointer.
02 CallVoidMethodA                    function-pointer.
02 CallNonvirtualObjectMethod         function-pointer.
02 CallNonvirtualObjectMethodV        function-pointer.
02 CallNonvirtualObjectMethodA        function-pointer.
02 CallNonvirtualBooleanMethod        function-pointer.
02 CallNonvirtualBooleanMethodV       function-pointer.
02 CallNonvirtualBooleanMethodA       function-pointer.
02 CallNonvirtualByteMethod           function-pointer.
02 CallNonvirtualByteMethodV          function-pointer.
02 CallNonvirtualByteMethodA          function-pointer.
02 CallNonvirtualCharMethod           function-pointer.
02 CallNonvirtualCharMethodV          function-pointer.
02 CallNonvirtualCharMethodA          function-pointer.
02 CallNonvirtualShortMethod          function-pointer.
02 CallNonvirtualShortMethodV         function-pointer.
02 CallNonvirtualShortMethodA         function-pointer.
02 CallNonvirtualIntMethod            function-pointer.
02 CallNonvirtualIntMethodV           function-pointer.
02 CallNonvirtualIntMethodA           function-pointer.
02 CallNonvirtualLongMethod           function-pointer.
02 CallNonvirtualLongMethodV          function-pointer.
02 CallNonvirtualLongMethodA          function-pointer.
02 CallNonvirtualFloatMethod          function-pointer.
02 CallNonvirtualFloatMethodV         function-pointer.
02 CallNonvirtualFloatMethodA         function-pointer.
02 CallNonvirtualDoubleMethod         function-pointer.
02 CallNonvirtualDoubleMethodV        function-pointer.
02 CallNonvirtualDoubleMethodA        function-pointer.
02 CallNonvirtualVoidMethod           function-pointer.
02 CallNonvirtualVoidMethodV          function-pointer.
02 CallNonvirtualVoidMethodA          function-pointer.
02 GetFieldID                         function-pointer.
02 GetObjectField                     function-pointer.
02 GetBooleanField                    function-pointer.
02 GetByteField                       function-pointer.
02 GetCharField                       function-pointer.
02 GetShortField                      function-pointer.
02 GetIntField                        function-pointer.
02 GetLongField                       function-pointer.
02 GetFloatField                      function-pointer.
02 GetDoubleField                     function-pointer.
02 SetObjectField                     function-pointer.
02 SetBooleanField                    function-pointer.
02 SetByteField                       function-pointer.
02 SetCharField                       function-pointer.
02 SetShortField                      function-pointer.
```

```
           02 SetIntField                     function-pointer.
           02 SetLongField                    function-pointer.
           02 SetFloatField                   function-pointer.
           02 SetDoubleField                  function-pointer.
           02 GetStaticMethodID               function-pointer.
           02 CallStaticObjectMethod          function-pointer.
           02 CallStaticObjectMethodV         function-pointer.
           02 CallStaticObjectMethodA         function-pointer.
           02 CallStaticBooleanMethod         function-pointer.
           02 CallStaticBooleanMethodV        function-pointer.
           02 CallStaticBooleanMethodA        function-pointer.
           02 CallStaticByteMethod            function-pointer.
           02 CallStaticByteMethodV           function-pointer.
           02 CallStaticByteMethodA           function-pointer.
           02 CallStaticCharMethod            function-pointer.
           02 CallStaticCharMethodV           function-pointer.
           02 CallStaticCharMethodA           function-pointer.
           02 CallStaticShortMethod           function-pointer.
           02 CallStaticShortMethodV          function-pointer.
           02 CallStaticShortMethodA          function-pointer.
           02 CallStaticIntMethod             function-pointer.
           02 CallStaticIntMethodV            function-pointer.
           02 CallStaticIntMethodA            function-pointer.
           02 CallStaticLongMethod            function-pointer.
           02 CallStaticLongMethodV           function-pointer.
           02 CallStaticLongMethodA           function-pointer.
           02 CallStaticFloatMethod           function-pointer.
           02 CallStaticFloatMethodV          function-pointer.
           02 CallStaticFloatMethodA          function-pointer.
           02 CallStaticDoubleMethod          function-pointer.
           02 CallStaticDoubleMethodV         function-pointer.
           02 CallStaticDoubleMethodA         function-pointer.
           02 CallStaticVoidMethod            function-pointer.
           02 CallStaticVoidMethodV           function-pointer.
           02 CallStaticVoidMethodA           function-pointer.
           02 GetStaticFieldID                function-pointer.
           02 GetStaticObjectField            function-pointer.
           02 GetStaticBooleanField           function-pointer.
           02 GetStaticByteField              function-pointer.
           02 GetStaticCharField              function-pointer.
           02 GetStaticShortField             function-pointer.
           02 GetStaticIntField               function-pointer.
           02 GetStaticLongField              function-pointer.
           02 GetStaticFloatField             function-pointer.
           02 GetStaticDoubleField            function-pointer.
           02 SetStaticObjectField            function-pointer.
           02 SetStaticBooleanField           function-pointer.
           02 SetStaticByteField              function-pointer.
           02 SetStaticCharField              function-pointer.
           02 SetStaticShortField             function-pointer.
           02 SetStaticIntField               function-pointer.
           02 SetStaticLongField              function-pointer.
           02 SetStaticFloatField             function-pointer.
           02 SetStaticDoubleField            function-pointer.
           02 NewString                       function-pointer.
           02 GetStringLength                 function-pointer.
           02 GetStringChars                  function-pointer.
           02 ReleaseStringChars              function-pointer.
           02 NewStringUTF                    function-pointer.
           02 GetStringUTFLength              function-pointer.
           02 GetStringUTFChars               function-pointer.
           02 ReleaseStringUTFChars           function-pointer.
           02 GetArrayLength                  function-pointer.
           02 NewObjectArray                  function-pointer.
           02 GetObjectArrayElement           function-pointer.
           02 SetObjectArrayElement           function-pointer.
           02 NewBooleanArray                 function-pointer.
```

```
   02 NewByteArray                     function-pointer.
   02 NewCharArray                     function-pointer.
   02 NewShortArray                    function-pointer.
   02 NewIntArray                      function-pointer.
   02 NewLongArray                     function-pointer.
   02 NewFloatArray                    function-pointer.
   02 NewDoubleArray                   function-pointer.
   02 GetBooleanArrayElements          function-pointer.
   02 GetByteArrayElements             function-pointer.
   02 GetCharArrayElements             function-pointer.
   02 GetShortArrayElements            function-pointer.
   02 GetIntArrayElements              function-pointer.
   02 GetLongArrayElements             function-pointer.
   02 GetFloatArrayElements            function-pointer.
   02 GetDoubleArrayElements           function-pointer.
   02 ReleaseBooleanArrayElements      function-pointer.
   02 ReleaseByteArrayElements         function-pointer.
   02 ReleaseCharArrayElements         function-pointer.
   02 ReleaseShortArrayElements        function-pointer.
   02 ReleaseIntArrayElements          function-pointer.
   02 ReleaseLongArrayElements         function-pointer.
   02 ReleaseFloatArrayElements        function-pointer.
   02 ReleaseDoubleArrayElements       function-pointer.
   02 GetBooleanArrayRegion            function-pointer.
   02 GetByteArrayRegion               function-pointer.
   02 GetCharArrayRegion               function-pointer.
   02 GetShortArrayRegion              function-pointer.
   02 GetIntArrayRegion                function-pointer.
   02 GetLongArrayRegion               function-pointer.
   02 GetFloatArrayRegion              function-pointer.
   02 GetDoubleArrayRegion             function-pointer.
   02 SetBooleanArrayRegion            function-pointer.
   02 SetByteArrayRegion               function-pointer.
   02 SetCharArrayRegion               function-pointer.
   02 SetShortArrayRegion              function-pointer.
   02 SetIntArrayRegion                function-pointer.
   02 SetLongArrayRegion               function-pointer.
   02 SetFloatArrayRegion              function-pointer.
   02 SetDoubleArrayRegion             function-pointer.
   02 RegisterNatives                  function-pointer.
   02 UnregisterNatives                function-pointer.
   02 MonitorEnter                     function-pointer.
   02 MonitorExit                      function-pointer.
   02 GetJavaVM                        function-pointer.
   02 GetStringRegion                  function-pointer.
   02 GetStringUTFRegion               function-pointer.
   02 GetPrimitiveArrayCritical        function-pointer.
   02 ReleasePrimitiveArrayCritical    function-pointer.
   02 GetStringCritical                function-pointer.
   02 ReleaseStringCritical            function-pointer.
   02 NewWeakGlobalRef                 function-pointer.
   02 DeleteWeakGlobalRef              function-pointer.
   02 ExceptionCheck                   function-pointer.
```

RELATED TASKS
"Compiling OO applications under UNIX" on page 287
"Accessing JNI services" on page 569

Appendix G. COBOL SYSADATA file contents

When you use the `ADATA` compiler option, the compiler produces a file that contains program data. You can use this file instead of the compiler listing to extract information about the program. For example, you can extract information about the program for symbolic debugging tools or cross-reference tools.

"Example: SYSADATA" on page 701

RELATED REFERENCES
"ADATA" on page 301
"Existing compiler options affecting the SYSADATA file"
"SYSADATA record types" on page 700
"SYSADATA record descriptions" on page 702

Existing compiler options affecting the SYSADATA file

Several compiler options could affect the contents of the SYSADATA file.

`COMPILE`
`NOCOMPILE(W|E|S)` might stop compilation prematurely, resulting in the loss of specific messages.

`EVENTS` (For compatibility) `EVENTS` has the same result as the `ADATA` option (that is, when either `ADATA` or `EVENTS` is in effect, the SYSADATA file is produced).

`EXIT` `INEXIT` prohibits identification of the compilation source file.

`LANGUAGE`
`LANGUAGE` controls the message text (Uppercase English, Mixed-Case English, or Japanese). Selection of Japanese could result in DBCS characters written to Error Identification records.

`TEST` `TEST` causes additional object text records to be created that also affect the contents of the SYSADATA file.

`NUM` `NUM` causes the compiler to use the contents of columns 1-6 in the source records for line numbering, rather than using generated sequence numbers. Any invalid (nonnumeric) or out-of-sequence numbers are replaced with a number one higher than that of the previous record.

The following SYSADATA fields contain line numbers whose contents differ depending on the `NUM|NONUM` setting:

Type	Field	Record
0020	AE_LINE	External Symbol record
0030	ATOK_LINE	Token record
0032	AF_STMT	Source Error record
0038	AS_STMT	Source record
0039	AS_REP_EXP_SLIN	COPY REPLACING record
0039	AS_REP_EXP_ELIN	COPY REPLACING record
0042	ASY_STMT	Symbol record
0044	AX_DEFN	Symbol Cross Reference record

Type	Field	Record
0044	AX_STMT	Symbol Cross Reference record
0046	AN_STMT	Nested Program record

The Type 0038 Source record contains two fields that relate to line numbers and record numbers:

- AS_STMT contains the compiler line number in both the `NUM` and `NONUM` cases.
- AS_CUR_REC# contains the physical source record number.

These two fields can always be used to correlate the compiler line numbers, used in all the above fields, with physical source record numbers.

The remaining compiler options have no direct effect on the SYSADATA file, but might trigger generation of additional error messages associated with the specific option, such as `FLAGSAA`, `FLAGSTD`, or `SSRANGE`.

"Example: SYSADATA" on page 701

RELATED REFERENCES
"SYSADATA record types"
"COMPILE" on page 305
"LANGUAGE" on page 318
"NUMBER" on page 324
"TEST" on page 338

SYSADATA record types

The SYSADATA file contains records classified into different record types. Each type of record provides information about the COBOL program being compiled.

Each record consists of two parts:

- A 12-byte header section, which has the same structure for all record types, and contains the record code that identifies the type of record
- A variable-length data section, which varies by record type

Table 107. **SYSADATA record types**

Record type	What it does
"Job identification record - X'0000'" on page 704	Provides information about the environment used to process the source data
"ADATA identification record - X'0001'" on page 705	Provides common information about the records in the SYSADATA file
"Compilation unit start|end record - X'0002'" on page 705	Marks the beginning and ending of compilation units in a source file
"Options record - X'0010'" on page 706	Describes the compiler options used for the compilation
"External symbol record - X'0020'" on page 715	Describes all external names in the program, definitions, and references
"Parse tree record - X'0024'" on page 716	Defines a node in the parse tree of the program
"Token record - X'0030'" on page 731	Defines a source token

Table 107. **SYSADATA record types** *(continued)*

Record type	**What it does**
"Source error record - X'0032'" on page 744	Describes errors in source program statements
"Source record - X'0038'" on page 745	Describes a single source line
"COPY REPLACING record - X'0039'" on page 745	Describes an instance of text replacement as a result of a match of `COPY. . .REPLACING` *operand-1* with text in the copybook
"Symbol record - X'0042'" on page 746	Describes a single symbol defined in the program. There is one symbol record for each symbol defined in the program.
"Symbol cross-reference record - X'0044'" on page 759	Describes references to a single symbol
"Nested program record - X'0046'" on page 760	Describes the name and nesting level of a program
"Library record - X'0060'" on page 761	Describes the library files and members used from each library
"Statistics record - X'0090'" on page 761	Describes the statistics about the compilation
"EVENTS record - X'0120'" on page 762	EVENTS records provide compatibility with COBOL/370(TM). The record format is identical with that in COBOL/370, with the addition of the standard ADATA header at the beginning of the record and a field indicating the length of the EVENTS record data.

"Example: SYSADATA"

Example: SYSADATA

The following sample shows part of the listing of a COBOL program. If this COBOL program were compiled with the ADATA option, the records produced in the associated data file would be in the sequence shown in the table below.

```
000001        IDENTIFICATION DIVISION.                              AD000020
000002        PROGRAM-ID. AD04202.                                  AD000030
000003        ENVIRONMENT DIVISION.                                 AD000040
000004        DATA DIVISION.                                        AD000050
000005        WORKING-STORAGE SECTION.                              AD000060
000006        77   COMP3-FLD2  pic  S9(3)v9.                        AD000070
000007        PROCEDURE DIVISION.                                   AD000080
000008            STOP RUN.
```

Type	**Description**
X'0120'	EVENTS Timestamp record
X'0120'	EVENTS Processor record
X'0120'	EVENTS File-ID record
X'0120'	EVENTS Program record
X'0001'	ADATA Identification record
X'0000'	Job Identification record
X'0010'	Options record
X'0038'	Source record for statement 1

Type	Description
X'0038'	Source record for statement 2
X'0038'	Source record for statement 3
X'0038'	Source record for statement 4
X'0038'	Source record for statement 5
X'0038'	Source record for statement 6
X'0038'	Source record for statement 7
X'0038'	Source record for statement 8
X'0020'	External Symbol record for AD04202
X'0044'	Symbol Cross Reference record for STOP
X'0044'	Symbol Cross Reference record for COMP3-FLD2
X'0044'	Symbol Cross Reference record for AD04202
X'0042'	Symbol record for AD04202
X'0042'	Symbol record for COMP3-FLD2
X'0090'	Statistics record
X'0120'	EVENTS FileEnd record

RELATED REFERENCES
"SYSADATA record descriptions"

SYSADATA record descriptions

The formats of the records written to the associated data file are shown in the related references below.

In the fields described in each of the record types, these symbols occur:

C	Indicates character (EBCDIC or ASCII) data
H	Indicates 2-byte binary integer data
F	Indicates 4-byte binary integer data
A	Indicates 4-byte binary integer address and offset data
X	Indicates hexadecimal (bit) data or 1-byte binary integer data

No boundary alignments are implied by any data type, and the implied lengths above might be changed by the presence of a length indicator (L*n*). All integer data is in big-endian or little-endian format depending on the indicator bit in the header flag byte. *Big-endian* format means that bit 0 is always the most significant bit and bit *n* is the least significant bit. *Little-endian* refers to "byte-reversed" integers as seen on Intel(R) processors.

All undefined fields and unused values are reserved.

RELATED REFERENCES
"Common header section" on page 703
"Job identification record - X'0000'" on page 704
"ADATA identification record - X'0001'" on page 705
"Compilation unit start|end record - X'0002'" on page 705
"Options record - X'0010'" on page 706

“External symbol record - X’0020’” on page 715
“Parse tree record - X’0024’” on page 716
“Token record - X’0030’” on page 731
“Source error record - X’0032’” on page 744
“Source record - X’0038’” on page 745
“COPY REPLACING record - X’0039’” on page 745
“Symbol record - X’0042’” on page 746
“Symbol cross-reference record - X’0044’” on page 759
“Nested program record - X’0046’” on page 760
“Library record - X’0060’” on page 761
“Statistics record - X’0090’” on page 761
“EVENTS record - X’0120’” on page 762

Common header section

The table below shows the format of the header section that is common for all record types. For MVS and VSE, each record is preceded by a 4-byte RDW (record-descriptor word) that is normally used only by access methods and stripped off by download utilities.

Table 108. **SYSADATA common header section**

Field	Size	Description
Language code	XL1	**16** High Level Assembler **17** COBOL on all platforms **40** PL/I on supported platforms
Record type	HL2	The record type, which can be one of the following: **X’0000’** Job Identification record[1] **X’0001’** ADATA Identification record **X’0002’** Compilation unit start/end record **X’0010’** Options record[1] **X’0020’** External Symbol record **X’0024’** Parse Tree record **X’0030’** Token record **X’0032’** Source Error record **X’0038’** Source record **X’0039’** COPY REPLACING record **X’0042’** Symbol record **X’0044’** Symbol Cross-Reference record **X’0046’** Nested Program record **X’0060’** Library record **X’0090’** Statistics record[1] **X’0120’** EVENTS record
Associated data architecture level	XL1	**3** Definition level for the header structure

Table 108. **SYSADATA common header section** *(continued)*

Field	Size	Description
Flag	XL1	**.... ..1.** ADATA record integers are in little-endian (Intel) format **.... ...1** This record is continued in the next record **1111 11..** Reserved for future use
Associated data record edition level	XL1	Used to indicate a new format for a specific record type, usually 0
Reserved	CL4	Reserved for future use
Associated data field length	HL2	The length in bytes of the data following the header
1. When a batch compilation (sequence of programs) is run with the ADATA option, there will be multiple Job Identification, Options, and Statistics records for each compilation.		

The mapping of the 12-byte header does not include the area used for the variable-length record-descriptor word required by the access method on MVS and VSE.

Job identification record - X'0000'

The following table shows the contents of the job identification record.

Table 109. **SYSADATA job identification record**

Field	Size	Description
Date	CL8	The date of the compilation in the format YYYYMMDD
Time	CL4	The time of the compilation in the format HHMM
Product number	CL8	The product number of the compiler that produced the associated data file
Product version	CL8	The version number of the product that produced the associated data file, in the form V.R.M
PTF level	CL8	The PTF level number of the product that produced the associated data file. (This field is blank if the PTF number is not available.)
System ID	CL24	The system identification of the system on which the compilation was run
Job name	CL8	The MVS job name of the compilation job
Step name	CL8	The MVS step name of the compilation step
Proc step	CL8	The MVS procedure step name of the compilation procedure
Number of input files[1]	HL2	The number of input files recorded in this record. The following group of seven fields will occur *n* times depending on the value in this field.
...Input file number	HL2	The assigned sequence number of the file
...Input file name length	HL2	The length of the following input file name

Table 109. **SYSADATA job identification record** *(continued)*

Field	Size	Description
...Volume serial number length	HL2	The length of the volume serial number
...Member name length	HL2	The length of the member name
...Input file name	CL(*n*)	The name of the input file for the compilation
...Volume serial number	CL(*n*)	The volume serial number of the (first) volume on which the input file resides
...Member name	CL(*n*)	Where applicable, the name of the member in the input file
1. Where the number of input files would exceed the record size for the associated data file, the record is continued on the next record. The current number of input files (for that record) is stored in the record, and the record is written to the associated data file. The next record contains the rest of the input files. The count of the number of input files is a count for the current record.		

ADATA identification record - X'0001'

The following table shows the contents of the ADATA identification record.

Table 110. **ADATA identification record**

Field	Size	Description
Time (binary)	XL8	Universal Time (UT) as a binary number of microseconds since midnight Greenwich Mean Time, with the low-order bit representing 1 microsecond. This time can be used as a time-zone-independent time stamp. On Windows and AIX systems, only bytes 5-8 of the field are used as a fullword binary field that contains the time.
CCSID[1]	XL2	Coded Character Set Identifier
Character-set flags	XL1	**X'80'** EBCDIC (IBM-037) **X'40'** ASCII (IBM-1252)
Code-page name length	XL2	Length of the code-page name that follows
Code-page name	CL(*n*)	Name of the code page
1. The appropriate CCS flag will always be set. If the CCSID is set to nonzero, the code-page name length will be zero. If the CCSID is set to zero, the code-page name length will be nonzero and the code-page name will be present.		

Compilation unit start|end record - X'0002'

The following table shows the contents of the compilation unit start|end record.

Table 111. **SYSADATA compilation unit start|end record**

Field	Size	Description
Type	HL2	Compilation unit type, which can be one of the following: **X'0000'** Start compilation unit **X'0001'** End compilation unit
Reserved	CL2	Reserved for future use
Reserved	FL4	Reserved for future use

Options record - X'0010'

The following table shows the contents of the options record.

Table 112. **SYSADATA options record**

Field	Size	Description
Option byte 0	XL1	**1111 1111** Reserved for future use
Option byte 1	XL1	**1... ....** Bit 1 = DECK, Bit 0 = NODECK **.1.. ....** Bit 1 = ADATA, Bit 0 = NOADATA **..1. ....** Bit 1 = COLLSEQ(EBCDIC), Bit 0 = COLLSEQ(LOCALE|BINARY) (Windows and AIX only) **...1 ....** Bit 1 = SEPOBJ, Bit 0 = NOSEPOBJ (Windows and AIX only) **.... 1...** Bit 1 = NAME, Bit 0 = NONAME **.... .1..** Bit 1 = OBJECT, Bit 0 = NOOBJECT **.... ..1.** Bit 1 = SQL, Bit 0 = NOSQL **.... ...1** Bit 1 = CICS, Bit 0 = NOCICS

Table 112. **SYSADATA options record** *(continued)*

Field	Size	Description
Option byte 2	XL1	`1... ....` Bit 1 = OFFSET, Bit 0 = NOOFFSET (host only) `.1.. ....` Bit 1 = MAP, Bit 0 = NOMAP `..1. ....` Bit 1 = LIST, Bit 0 = NOLIST `...1 ....` Bit 1 = DBCSXREF, Bit 0 = NODBCSXREF `.... 1...` Bit 1 = XREF(SHORT), Bit 0 = not XREF(SHORT). This flag should be used in combination with the flag at bit 7. XREF(FULL) is indicated by this flag being off and the flag at bit 7 being on. `.... .1..` Bit 1 = SOURCE, Bit 0 = NOSOURCE `.... ..1.` Bit 1 = VBREF, Bit 0 = NOVBREF `.... ...1` Bit 1 = XREF, Bit 0 = not XREF. See also flag at bit 4 above.
Option byte 3	XL1	`1... ....` Bit 1 = FLAG imbedded diagnostics level specified (a value *y* is specified as in FLAG(*x*,*y*)) `.1.. ....` Bit 1 = FLAGSTD, Bit 0 = NOFLAGSTD `..1. ....` Bit 1 = NUM, Bit 0 = NONUM `...1 ....` Bit 1 = SEQUENCE, Bit 0 = NOSEQUENCE `.... 1...` Bit 1 = SOSI, Bit 0 = NOSOSI (Windows and AIX only) `.... .1..` Bit 1 = NSYMBOL(NATIONAL), Bit 0 = NSYMBOL(DBCS) `.... ..1.` Bit 1 = PROFILE, Bit 0 = NOPROFILE (Windows and AIX only) `.... ...1` Bit 1 = WORD, Bit 0 = NOWORD

Table 112. **SYSADATA options record** *(continued)*

Field	Size	Description
Option byte 4	XL1	`1... ....` Bit 1 = ADV, Bit 0 = NOADV `.1.. ....` Bit 1 = APOST, Bit 0 = QUOTE `..1. ....` Bit 1 = DYNAM, Bit 0 = NODYNAM `...1 ....` Bit 1 = AWO, Bit 0 = NOAWO `.... 1...` Bit 1 = RMODE specified, Bit 0 = RMODE(AUTO) `.... .1..` Bit 1 = RENT, Bit 0 = NORENT `.... ..1.` Bit 1 = RES: this flag will always be set on for COBOL. `.... ...1` Bit 1 = RMODE(24), Bit 0 = RMODE(ANY)
Option byte 5	XL1	`1... ....` Reserved for compatibility `.1.. ....` Bit 1 = OPT, Bit 0 = NOOPT `..1. ....` Bit 1 = LIB, Bit 0 = NOLIB `...1 ....` Bit 1 = DBCS, Bit 0 = NODBCS `.... 1...` Bit 1 = OPT(FULL), Bit 0 = not OPT(FULL) `.... .1..` Bit 1 = SSRANGE, Bit 0 = NOSSRANGE `.... ..1.` Bit 1 = TEST, Bit 0 = NOTEST `.... ...1` Bit 1 = PROBE, Bit 0 = NOPROBE (Windows only)

Table 112. **SYSADATA options record** *(continued)*

Field	Size	Description
Option byte 6	XL1	`..1. ....` Bit 1 = NUMPROC(PFD), Bit 0 = NUMPROC(NOPFD) `...1 ....` Bit 1 = NUMCLS(ALT), Bit 0 = NUMCLS(PRIM) `.... .1..` Bit 1 = BINARY(S390), Bit 0 = BINARY(NATIVE) (Windows and AIX only) `.... ..1.` Bit 1 = TRUNC(STD), Bit 0 = TRUNC(OPT) `.... ...1` Bit 1 = ZWB, Bit 0 = NOZWB `11.. 1...` Reserved for future use
Option byte 7	XL1	`1... ....` Bit 1 = ALOWCBL, Bit 0 = NOALOWCBL (Windows and AIX only) `.1.. ....` Bit 1 = TERM, Bit 0 = NOTERM `..1. ....` Bit 1 = DUMP, Bit 0 = NODUMP `...1 11..` Reserved `.... ..1.` Bit 1 = CURRENCY, Bit 0 = NOCURRENCY `.... ...1` Reserved
Option byte 8	XL1	`1111 1111` Reserved for future use
Option byte 9	XL1	`1... ....` Bit 1 = DATA(24), Bit 0 = DATA(31) `.1.. ....` Bit 1 = FASTSRT, Bit 0 = NOFASTSRT `..1. ....` Bit 1 = SIZE(MAX), Bit 0 = SIZE(*nnnn*) or SIZE(*nnnn*K) `.... .1..` Bit 1 = THREAD, Bit 0 = NOTHREAD `...1 1.11` Reserved for future use
Option byte A	XL1	`1111 1111` Reserved for future use
Option byte B	XL1	`1111 1111` Reserved for future use

Table 112. **SYSADATA options record** *(continued)*

Field	Size	Description
Option byte C	XL1	`1... ....` Bit 1 = NCOLLSEQ(LOCALE) (Windows and AIX only) `.1.. ....` Reserved for future use `..1. ....` Bit 1 = INTDATE(LILIAN), Bit 0 = INTDATE(ANSI) `...1 ....` Bit 1 = NCOLLSEQ(BINARY) (Windows and AIX only) `.... 1...` Bit 1 = CHAR(EBCDIC), Bit 0 = CHAR(NATIVE) (Windows and AIX only) `.... .1..` Bit 1 = FLOAT(HEX), Bit 0 = FLOAT(NATIVE) (Windows and AIX only) `.... ..1.` Bit 1 = COLLSEQ(BINARY) (Windows and AIX only) `.... ...1` Bit 1 = COLLSEQ(LOCALE) (Windows and AIX only)
Option byte D	XL1	`1... ....` Bit 1 = DLL Bit 0 = NODLL (host only) `.1.. ....` Bit 1 = EXPORTALL, Bit 0 = NOEXPORTALL (host only) `..1. ....` Bit 1 = CODEPAGE (host only) `...1 ....` Bit 1 = DATEPROC, Bit 0 = NODATEPROC `.... 1...` Bit 1 = DATEPROC(FLAG), Bit 0 = DATEPROC(NOFLAG) `.... .1..` Bit 1 = YEARWINDOW `.... ..1.` Bit 1 = WSCLEAR, Bit 0 = NOWSCLEAR (Windows and AIX only) `.... ...1` Bit 1 = BEOPT, Bit 0 = NOBEOPT (Windows and AIX only)

Table 112. **SYSADATA options record** *(continued)*

Field	Size	Description
Option byte E	XL1	**1... ....** Bit 1 = DATEPROC(TRIG), Bit 0 = DATEPROC(NOTRIG) **.1.. ....** Bit 1 = DIAGTRUNC, Bit 0 = NODIAGTRUNC **.... .1..** Bit 1 = LSTFILE(UTF-8), Bit 0 = LSTFILE(LOCALE) (Windows and AIX only) **..11 1.11** Reserved for future use
Option byte F	XL1	**1111 1111** Reserved for future use
Flag level	XL1	**X'00'** Flag(I) **X'04'** Flag(W) **X'08'** Flag(E) **X'0C'** Flag(S) **X'10'** Flag(U) **X'FF'** Noflag
Imbedded diagnostic level	XL1	**X'00'** Flag(I) **X'04'** Flag(W) **X'08'** Flag(E) **X'0C'** Flag(S) **X'10'** Flag(U) **X'FF'** Noflag
FLAGSTD (FIPS) specification	XL1	**1... ....** Minimum **.1.. ....** Intermediate **..1. ....** High **...1 ....** IBM extensions **.... 1...** Level-1 segmentation **.... .1..** Level-2 segmentation **.... ..1.** Debugging **.... ...1** Obsolete

Table 112. **SYSADATA options record** *(continued)*

Field	Size	Description
Reserved for flagging	XL1	**1111 1111** Reserved for future use
Compiler mode	XL1	**X'00'** Unconditional Nocompile, Nocompile(I) **X'04'** Nocompile(W) **X'08'** Nocompile(E) **X'0C'** Nocompile(S) **X'FF'** Compile
Space value	CL1	
Data for 3-valued options	XL1	**1... ....** NAME(ALIAS) specified **.1.. ....** NUMPROC(MIG) specified **..1. ....** TRUNC(BIN) specified **...1 1111** Reserved for future use
TEST(hook,sym,sep) suboptions (host only)	XL1	**1... ....** TEST(ALL,x) **.1.. ....** TEST(NONE,x) **..1. ....** TEST(STMT,x) **...1 ....** TEST(PATH,x) **.... 1...** TEST(BLOCK,x) **.... .1..** TEST(x,SYM) **.... ..1.** Bit 1 = SEPARATE, Bit 0 = NOSEPARATE **.... ...1** Reserved for TEST suboptions
OUTDD name length	HL2	Length of OUTDD name
RWT ID Length	HL2	Length of Reserved Word Table identifier
LVLINFO	CL4	User-specified LVLINFO data

Table 112. **SYSADATA options record** *(continued)*

Field	Size	Description
PGMNAME suboptions	XL1	**1... ....** Bit 1 = PGMNAME(COMPAT) **.1.. ....** Bit 1 = PGMNAME(LONGUPPER) **..1. ....** Bit 1 = PGMNAME(LONGMIXED) **...1 1111** Reserved for future use
Entry interface suboptions	XL1	**1... ....** Bit 1 = EntryInterface(System) (Windows only) **.1.. ....** Bit 1 = EntryInterface(OptLink) (Windows only) **..11 1111** Reserved for future use
CallInterface suboptions	XL1	**1... ....** Bit 1 = CallInterface(System) (Windows and AIX only) **.1.. ....** Bit 1 = CallInterface(OptLink) (Windows only) **...1 ....** Bit 1 = CallInterface(Cdecl) (Windows only) **.... 1...** Bit 1 = CallInterface(System(Desc)) (Windows and AIX only) **..1. .111** Reserved for future use
ARITH suboption	XL1	**1... ....** Bit 1 = ARITH(COMPAT) **.1.. ....** Bit 1 = ARITH(EXTEND) **11 1111** Reserved for future use
DBCS Req	FL4	DBCS XREF storage requirement
DBCS ORDPGM length	HL2	Length of name of DBCS Ordering Program
DBCS ENCTBL length	HL2	Length of name of DBCS Encode Table
DBCS ORD TYPE	CL2	DBCS Ordering type
Reserved	CL6	Reserved for future use
Converted SO	CL1	Converted SO hexadecimal value
Converted SI	CL1	Converted SI hexadecimal value
Language id	CL2	This field holds the two-character abbreviation (one of EN, UE, JA, or JP) from the LANGUAGE option.
Reserved	CL8	Reserved for future use

Table 112. **SYSADATA options record** *(continued)*

Field	Size	Description
INEXIT name length	HL2	Length of SYSIN user-exit name
PRTEXIT name length	HL2	Length of SYSPRINT user-exit name
LIBEXIT name length	HL2	Length of 'Library' user-exit name
ADEXIT name length	HL2	Length of ADATA user-exit name
CURROPT	CL5	CURRENCY option value
Reserved	CL1	Reserved for future use
YEARWINDOW	HL2	YEARWINDOW option value
CODEPAGE	HL2	CODEPAGE CCSID option value
Reserved	CL50	Reserved for future use
LINECNT	HL2	LINECOUNT value
Reserved	CL2	Reserved for future use
BUFSIZE	FL4	BUFSIZE option value
Size value	FL4	SIZE option value
Reserved	FL4	Reserved for future use
Phase residence bits byte 1	XL1	`1... ....` Bit 1 = IGYCLIBR in user region `.1.. ....` Bit 1 = IGYCSCAN in user region `..1. ....` Bit 1 = IGYCDSCN in user region `...1 ....` Bit 1 = IGYCGROU in user region `.... 1...` Bit 1 = IGYCPSCN in user region `.... .1..` Bit 1 = IGYCPANA in user region `.... ..1.` Bit 1 = IGYCFGEN in user region `.... ...1` Bit 1 = IGYCPGEN in user region

Table 112. **SYSADATA options record** *(continued)*

Field	Size	Description
Phase residence bits byte 2	XL1	`1... ....` Bit 1 = IGYCOPTM in user region `.1.. ....` Bit 1 = IGYCLSTR in user region `..1. ....` Bit 1 = IGYCXREF in user region `...1 ....` Bit 1 = IGYCDMAP in user region `.... 1...` Bit 1 = IGYCASM1 in user region `.... .1..` Bit 1 = IGYCASM2 in user region `.... ..1.` Bit 1 = IGYCDIAG in user region `.... ...1` Reserved for future use
Phase residence bits bytes 3 and 4	XL2	Reserved
Reserved	CL8	Reserved for future use
OUTDD name	CL(*n*)	OUTDD name
RWT	CL(*n*)	Reserved word table identifier
DBCS ORDPGM	CL(*n*)	DBCS Ordering program name
DBCS ENCTBL	CL(*n*)	DBCS Encode table name
INEXIT name	CL(*n*)	SYSIN user-exit name
PRTEXIT name	CL(*n*)	SYSPRINT user-exit name
LIBEXIT name	CL(*n*)	'Library' user-exit name
ADEXIT name	CL(*n*)	ADATA user-exit name

External symbol record - X'0020'

The following table shows the contents of the external symbol record.

Table 113. **SYSADATA external symbol record**

Field	Size	Description
Section type	XL1	**X'00'** `PROGRAM-ID` name (main entry point name) **X'01'** `ENTRY` name (secondary entry point name) **X'02'** External reference (referenced external entry point) **X'04'** N/A for COBOL **X'05'** N/A for COBOL **X'06'** N/A for COBOL **X'0A'** N/A for COBOL **X'12'** Internal reference (referenced internal subprogram) **X'C0'** External class-name (OO COBOL class definition) **X'C1'** `METHOD-ID` name (OO COBOL method definition) **X'C6'** Method reference (OO COBOL method reference) **X'FF'** N/A for COBOL Types X'12', X'C0', X'C1' and X'C6X' are for COBOL only.
Flags	XL1	N/A for COBOL
Reserved	HL2	Reserved for future use
Symbol-ID	FL4	Symbol-ID of program that contains the reference (only for types x'02' and x'12')
Line number	FL4	Line number of statement that contains the reference (only for types x'02' and x'12')
Section length	FL4	N/A for COBOL
LD ID	FL4	N/A for COBOL
Reserved	CL8	Reserved for future use
External name length	HL2	Number of characters in the external name
Alias name length	HL2	N/A for COBOL
External name	CL(*n*)	The external name
Alias section name	CL(*n*)	N/A for COBOL

Parse tree record - X'0024'

The following table shows the contents of the parse tree record.

Table 114. **SYSADATA parse tree record**

Field	Size	Description
Node number	FL4	The node number generated by the compiler, starting at 1

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
Node type	HL2	The type of the node: **001** Program **002** Class **003** Method
		101 Identification Division **102** Environment Division **103** Data Division **104** Procedure Division **105** End Program/Method/Class
		201 Declaratives body **202** Nondeclaratives body
		301 Section **302** Procedure section
		401 Paragraph **402** Procedure paragraph
		501 Sentence **502** File definition **503** Sort file definition **504** Program-name **505** Program attribute **508** ENVIRONMENT DIVISION clause **509** CLASS attribute **510** METHOD attribute **511** USE statement
		601 Statement **602** Data description clause **603** Data entry **604** File description clause **605** Data entry name **606** Data entry level **607** EXEC entry

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		701 EVALUATE subject phrase **702** EVALUATE WHEN phrase **703** EVALUATE WHEN OTHER phrase **704** SEARCH WHEN phrase **705** INSPECT CONVERTING phrase **706** INSPECT REPLACING phrase **707** INSPECT TALLYING phrase **708** PERFORM UNTIL phrase **709** PERFORM VARYING phrase **710** PERFORM AFTER phrase **711** Statement block **712** Scope terminator **713** INITIALIZE REPLACING phrase **714** EXEC CICS Command **720** DATA DIVISION phrase
		801 Phrase **802** ON phrase **803** NOT phrase **804** THEN phrase **805** ELSE phrase **806** Condition **807** Expression **808** Relative indexing **809** EXEC CICS Option **810** Reserved word **811** INITIALIZE REPLACING category

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		901 Section or paragraph name **902** Identifier **903** Alphabet-name **904** Class-name **905** Condition-name **906** File-name **907** Index-name **908** Mnemonic-name **910** Symbolic-character **911** Literal **912** Function identifier **913** Data-name **914** Special register **915** Procedure reference **916** Arithmetic operator **917** All Procedures **918** INITIALIZE literal (no tokens) **919** ALL literal or figcon **920** Keyword class test name **921** Reserved word at identifier level **922** Unary operator **923** Relational operator
		1001 Subscript **1002** Reference modification
Node subtype	HL2	The subtype of the node. For Section type: **0001** CONFIGURATION Section **0002** INPUT-OUTPUT Section **0003** FILE Section **0004** WORKING-STORAGE Section **0005** LINKAGE Section **0006** LOCAL-STORAGE Section **0007** REPOSITORY Section

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For Paragraph type: **0001** PROGRAM-ID paragraph **0002** AUTHOR paragraph **0003** INSTALLATION paragraph **0004** DATE-WRITTEN paragraph **0005** SECURITY paragraph **0006** SOURCE-COMPUTER paragraph **0007** OBJECT-COMPUTER paragraph **0008** SPECIAL-NAMES paragraph **0009** FILE-CONTROL paragraph **0010** I-O-CONTROL paragraph **0011** DATE-COMPILED paragraph **0012** CLASS-ID paragraph **0013** METHOD-ID paragraph **0014** REPOSITORY paragraph
		For Environment Division clause type: **0001** WITH DEBUGGING MODE **0002** MEMORY-SIZE **0003** SEGMENT-LIMIT **0004** CURRENCY-SIGN **0005** DECIMAL POINT **0006** PROGRAM COLLATING SEQUENCE **0007** ALPHABET **0008** SYMBOLIC-CHARACTER **0009** CLASS **0010** ENVIRONMENT NAME **0011** SELECT

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For Data description clause type: `0001` BLANK WHEN ZERO `0002` DATA-NAME OR FILLER `0003` JUSTIFIED `0004` OCCURS `0005` PICTURE `0006` REDEFINES `0007` RENAMES `0008` SIGN `0009` SYNCHRONIZED `0010` USAGE `0011` VALUE `0023` GLOBAL `0024` EXTERNAL

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For File description clause type: **0001** FILE STATUS **0002** ORGANIZATION **0003** ACCESS MODE **0004** RECORD KEY **0005** ASSIGN **0006** RELATIVE KEY **0007** PASSWORD **0008** PROCESSING MODE **0009** RECORD DELIMITER **0010** PADDING CHARACTER **0011** BLOCK CONTAINS **0012** RECORD CONTAINS **0013** LABEL RECORDS **0014** VALUE OF **0015** DATA RECORDS **0016** LINAGE **0017** ALTERNATE KEY **0018** LINES AT TOP **0019** LINES AT BOTTOM **0020** CODE-SET **0021** RECORDING MODE **0022** RESERVE **0023** GLOBAL **0024** EXTERNAL **0025** LOCK

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For Statement type: **0002** NEXT SENTENCE **0003** ACCEPT **0004** ADD **0005** ALTER **0006** CALL **0007** CANCEL **0008** CLOSE **0009** COMPUTE **0010** CONTINUE **0011** DELETE **0012** DISPLAY **0013** DIVIDE (INTO) **0113** DIVIDE (BY) **0014** ENTER **0015** ENTRY **0016** EVALUATE **0017** EXIT **0018** GO **0019** GOBACK **0020** IF **0021** INITIALIZE **0022** INSPECT

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		0023 INVOKE **0024** MERGE **0025** MOVE **0026** MULTIPLY **0027** OPEN **0028** PERFORM **0029** READ **0030** READY **0031** RELEASE **0032** RESET **0033** RETURN **0034** REWRITE **0035** SEARCH **0036** SERVICE **0037** SET **0038** SORT **0039** START **0040** STOP **0041** STRING **0042** SUBTRACT **0043** UNSTRING **0044** EXEC SQL **0144** EXEC CICS **0045** WRITE **0046** XML

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For Phrase type:
		0001 INTO
		0002 DELIMITED
		0003 INITIALIZE. . .REPLACING
		0004 INSPECT. . .ALL
		0005 INSPECT. . .LEADING
		0006 SET. . .TO
		0007 SET. . .UP
		0008 SET. . .DOWN
		0009 PERFORM. . .TIMES
		0010 DIVIDE. . .REMAINDER
		0011 INSPECT. . .FIRST
		0012 SEARCH. . .VARYING
		0013 MORE-LABELS
		0014 SEARCH ALL
		0015 SEARCH. . .AT END
		0016 SEARCH. . .TEST INDEX
		0017 GLOBAL
		0018 LABEL
		0019 DEBUGGING
		0020 SEQUENCE
		0021 Reserved for future use
		0022 Reserved for future use
		0023 Reserved for future use
		0024 TALLYING
		0025 Reserved for future use
		0026 ON SIZE ERROR
		0027 ON OVERFLOW
		0028 ON ERROR
		0029 AT END
		0030 INVALID KEY

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		0031 END-OF-PAGE **0032** USING **0033** BEFORE **0034** AFTER **0035** EXCEPTION **0036** CORRESPONDING **0037** Reserved for future use **0038** RETURNING **0039** GIVING **0040** THROUGH **0041** KEY **0042** DELIMITER **0043** POINTER **0044** COUNT **0045** METHOD **0046** PROGRAM **0047** INPUT **0048** OUTPUT **0049** I-O **0050** EXTEND **0051** RELOAD **0052** ASCENDING **0053** DESCENDING **0054** DUPLICATES **0055** NATIVE (USAGE) **0056** INDEXED **0057** FROM **0058** FOOTING **0059** LINES AT BOTTOM **0060** LINES AT TOP

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For Function identifier type:
		`0001` COS
		`0002` LOG
		`0003` MAX
		`0004` MIN
		`0005` MOD
		`0006` ORD
		`0007` REM
		`0008` SIN
		`0009` SUM
		`0010` TAN
		`0011` ACOS
		`0012` ASIN
		`0013` ATAN
		`0014` CHAR
		`0015` MEAN
		`0016` SQRT
		`0017` LOG10
		`0018` RANGE
		`0019` LENGTH
		`0020` MEDIAN
		`0021` NUMVAL
		`0022` RANDOM
		`0023` ANNUITY
		`0024` INTEGER
		`0025` ORD-MAX
		`0026` ORD-MIN
		`0027` REVERSE
		`0028` MIDRANGE
		`0029` NUMVAL-C
		`0030` VARIANCE
		`0031` FACTORIAL
		`0032` LOWER-CASE

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		0033 UPPER-CASE **0034** CURRENT-DATE **0035** INTEGER-PART **0036** PRESENT-VALUE **0037** WHEN-COMPILED **0038** DAY-OF-INTEGER **0039** INTEGER-OF-DAY **0040** DATE-OF-INTEGER **0041** INTEGER-OF-DATE **0042** STANDARD-DEVIATION **0043** YEAR-TO-YYYY **0044** DAY-TO-YYYYDDD **0045** DATE-TO-YYYYMMDD **0046** UNDATE **0047** DATEVAL **0048** YEARWINDOW **0049** DISPLAY-OF **0050** NATIONAL-OF
		For Special Register type: **0001** ADDRESS OF **0002** LENGTH OF
		For Keyword Class Test Name type: **0001** ALPHABETIC **0002** ALPHABETIC-LOWER **0003** ALPHABETIC-UPPER **0004** DBCS **0005** KANJI **0006** NUMERIC **0007** NEGATIVE **0008** POSITIVE **0009** ZERO
		For Reserved Word type: **0001** TRUE **0002** FALSE **0003** ANY **0004** THRU

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For Identifier, Data-name, Index-name, Condition-name or Mnemonic-name type: **0001** REFERENCED **0002** CHANGED **0003** REFERENCED & CHANGED
		For Initialize literal type: **0001** ALPHABETIC **0002** ALPHANUMERIC **0003** NUMERIC **0004** ALPHANUMERIC-EDITED **0005** NUMERIC-EDITED **0006** DBCS/EGCS **0007** NATIONAL **0008** NATIONAL-EDITED
		For Procedure-name type: **0001** SECTION **0002** PARAGRAPH
		For Reserved word at identifier level type: **0001** ROUNDED **0002** TRUE **0003** ON **0004** OFF **0005** SIZE **0006** DATE **0007** DAY **0008** DAY-OF-WEEK **0009** TIME **0010** WHEN-COMPILED **0011** PAGE **0012** DATE YYYYMMDD **0013** DAY YYYYDDD

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For Arithmetic Operator type: **0001** PLUS **0002** MINUS **0003** TIMES **0004** DIVIDE **0005** DIVIDE REMAINDER **0006** EXPONENTIATE **0007** NEGATE
		For Relational Operator type: **0008** LESS **0009** LESS OR EQUAL **0010** EQUAL **0011** NOT EQUAL **0012** GREATER **0013** GREATER OR EQUAL **0014** AND **0015** OR **0016** CLASS CONDITION **0017** NOT CLASS CONDITION
Parent node number	FL4	The node number of the parent of the node
Left sibling node number	FL4	The node number of the left sibling of the node, if any. If none, the value is zero.
Symbol Id	FL4	The Symbol Id of the node, if it is a user-name of one of the following types: • Data entry • Identifier • File-name • Index-name • Procedure-name • Condition-name • Mnemonic-name This value corresponds to the Symbol ID in a Symbol (Type 42) record, except for procedure-names where it corresponds to the Paragraph ID. For all other node types this value is zero.
Section Symbol Id	FL4	The Symbol Id of the section containing the node, if it is a qualified paragraph-name reference. This value corresponds to the Section ID in a Symbol (Type 42) record. For all other node types this value is zero.
First token number	FL4	The number of the first token associated with the node

Table 114. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
Last token number	FL4	The number of the last token associated with the node
Reserved	FL4	Reserved for future use
Flags	CL1	Information about the node: **X'80'** Reserved **X'40'** Generated node, no tokens
Reserved	CL3	Reserved for future use

Token record - X'0030'

The compiler does not generate token records for any lines that are treated as comment lines, which include, but are not limited to, items in the following list.

- Comment lines, which are lines that have an asterisk (*) or a slash (/) in column 7
- The following compiler-directing statements:
  - `*CBL (*CONTROL)`
  - `BASIS`
  - `COPY`
  - `DELETE`
  - `EJECT`
  - `INSERT`
  - `REPLACE`
  - `SKIP1`
  - `SKIP2`
  - `SKIP3`
  - `TITLE`
- Debugging lines, which are lines that have a `D` in column 7, if `WITH DEBUGGING MODE` is not specified

Table 115. **SYSADATA token record**

Field	Size	Description
Token number	FL4	The token number within the source file generated by the compiler, starting at 1. Any copybooks have already been included in the source.

Table 115. **SYSADATA token record** *(continued)*

Field	Size	Description
Token code	HL2	The type of token (user-name, literal, reserved word, and so forth). For reserved words, the compiler reserved-word table values are used. For PICTURE strings, the special code 0000 is used. For each piece (other than the last) of a continued token, the special code 3333 is used. Otherwise, the following codes are used: **0001** ACCEPT **0002** ADD **0003** ALTER **0004** CALL **0005** CANCEL **0007** CLOSE **0009** COMPUTE **0011** DELETE **0013** DISPLAY **0014** DIVIDE **0017** READY **0018** END-PERFORM **0019** ENTER **0020** ENTRY **0021** EXIT **0022** EXEC EXECUTE **0023** GO **0024** IF **0025** INITIALIZE **0026** INVOKE **0027** INSPECT **0028** MERGE **0029** MOVE

Table 115. **SYSADATA token record** *(continued)*

Field	Size	Description
		0030 MULTIPLY
		0031 OPEN
		0032 PERFORM
		0033 READ
		0035 RELEASE
		0036 RETURN
		0037 REWRITE
		0038 SEARCH
		0040 SET
		0041 SORT
		0042 START
		0043 STOP
		0044 STRING
		0045 SUBTRACT
		0048 UNSTRING
		0049 USE
		0050 WRITE
		0051 CONTINUE
		0052 END-ADD
		0053 END-CALL
		0054 END-COMPUTE
		0055 END-DELETE
		0056 END-DIVIDE
		0057 END-EVALUATE
		0058 END-IF
		0059 END-MULTIPLY
		0060 END-READ
		0061 END-RETURN
		0062 END-REWRITE
		0063 END-SEARCH
		0064 END-START
		0065 END-STRING
		0066 END-SUBTRACT
		0067 END-UNSTRING
		0068 END-WRITE
		0069 GOBACK

Table 115. **SYSADATA token record** *(continued)*

Field	Size	Description
		0070 EVALUATE
		0071 RESET
		0072 SERVICE
		0073 END-INVOKE
		0074 END-EXEC
		0075 XML
		0076 END-XML
		0099 FOREIGN-VERB
		0101 DATA-NAME
		0105 DASHED-NUM
		0106 DECIMAL
		0107 DIV-SIGN
		0108 EQ
		0109 EXPONENTIATION
		0110 GT
		0111 INTEGER
		0112 LT
		0113 LPAREN
		0114 MINUS-SIGN
		0115 MULT-SIGN
		0116 NONUMLIT
		0117 PERIOD
		0118 PLUS-SIGN
		0121 RPAREN
		0122 SIGNED-INTEGER
		0123 QUID
		0124 COLON
		0125 IEOF
		0126 EGCS-LIT
		0127 COMMA-SPACE
		0128 SEMICOLON-SPACE
		0129 PROCEDURE-NAME
		0130 FLT-POINT-LIT
		0131 Language Environment

Table 115. **SYSADATA token record** *(continued)*

Field	Size	Description
		0132 GE
		0133 IDREF
		0134 EXPREF
		0136 CICS
		0137 NEW
		0138 NATIONAL-LIT
		0200 ADDRESS
		0201 ADVANCING
		0202 AFTER
		0203 ALL
		0204 ALPHABETIC
		0205 ALPHANUMERIC
		0206 ANY
		0207 AND
		0208 ALPHANUMERIC-EDITED
		0209 BEFORE
		0210 BEGINNING
		0211 FUNCTION
		0212 CONTENT
		0213 CORR
		CORRESPONDING
		0214 DAY
		0215 DATE
		0216 DEBUG-CONTENTS
		0217 DEBUG-ITEM
		0218 DEBUG-LINE
		0219 DEBUG-NAME
		0220 DEBUG-SUB-1
		0221 DEBUG-SUB-2
		0222 DEBUG-SUB-3
		0223 DELIMITED
		0224 DELIMITER
		0225 DOWN

Table 115. **SYSADATA token record** *(continued)*

Field	Size	Description	
		0226	NUMERIC-EDITED
		0227	XML-EVENT
		0228	END-OF-PAGE
			EOP
		0229	EQUAL
		0230	ERROR
		0231	XML-NTEXT
		0232	EXCEPTION
		0233	EXTEND
		0234	FIRST
		0235	FROM
		0236	GIVING
		0237	GREATER
		0238	I-O
		0239	IN
		0240	INITIAL
		0241	INTO
		0242	INVALID
		0243	SQL
		0244	LESS
		0245	LINAGE-COUNTER
		0246	XML-TEXT
		0247	LOCK
		0248	GENERATE
		0249	NEGATIVE
		0250	NEXT
		0251	NO
		0252	NOT
		0253	NUMERIC
		0254	KANJI
		0255	OR
		0256	OTHER
		0257	OVERFLOW
		0258	PAGE
		0259	CONVERTING

Table 115. **SYSADATA token record** (continued)

Field	Size	Description
		0260 POINTER
		0261 POSITIVE
		0262 DBCS
		0263 PROCEDURES
		0264 PROCEED
		0265 REFERENCES
		0266 DAY-OF-WEEK
		0267 REMAINDER
		0268 REMOVAL
		0269 REPLACING
		0270 REVERSED
		0271 REWIND
		0272 ROUNDED
		0273 RUN
		0274 SENTENCE
		0275 STANDARD
		0276 RETURN-CODE
		SORT-CORE-SIZE
		SORT-FILE-SIZE
		SORT-MESSAGE
		SORT-MODE-SIZE
		SORT-RETURN
		TALLY
		XML-CODE
		0277 TALLYING
		0278 SUM
		0279 TEST
		0280 THAN
		0281 UNTIL
		0282 UP
		0283 UPON
		0284 VARYING
		0285 RELOAD
		0286 TRUE

Table 115. **SYSADATA token record** *(continued)*

Field	Size	Description	
		0287	THEN
		0288	RETURNING
		0289	ELSE
		0290	SELF
		0291	SUPER
		0292	WHEN-COMPILED
		0293	ENDING
		0294	FALSE
		0295	REFERENCE
		0296	NATIONAL-EDITED
		0297	COM-REG
		0298	ALPHABETIC-LOWER
		0299	ALPHABETIC-UPPER
		0301	REDEFINES
		0302	OCCURS
		0303	SYNC
			SYNCHRONIZED
		0304	MORE-LABELS
		0305	JUST
			JUSTIFIED
		0306	SHIFT-IN
		0307	BLANK
		0308	VALUE
		0309	COMP
			COMPUTATIONAL
		0310	COMP-1
			COMPUTATIONAL-1
		0311	COMP-3
			COMPUTATIONAL-3
		0312	COMP-2
			COMPUTATIONAL-2
		0313	COMP-4
			COMPUTATIONAL-4
		0314	DISPLAY-1
		0315	SHIFT-OUT

Table 115. **SYSADATA token record** *(continued)*

Field	Size	Description
		0316 INDEX
		0317 USAGE
		0318 SIGN
		0319 LEADING
		0320 SEPARATE
		0321 INDEXED
		0322 LEFT
		0323 RIGHT
		0324 PIC
		PICTURE
		0325 VALUES
		0326 GLOBAL
		0327 EXTERNAL
		0328 BINARY
		0329 PACKED-DECIMAL
		0330 EGCS
		0331 PROCEDURE-POINTER
		0332 COMP-5
		COMPUTATIONAL-5
		0333 FUNCTION-POINTER
		0334 TYPE
		0335 JNIENVPTR
		0336 NATIONAL
		0337 GROUP-USAGE
		0401 HIGH-VALUE
		HIGH-VALUES
		0402 LOW-VALUE
		LOW-VALUES
		0403 QUOTE
		QUOTES
		0404 SPACE
		SPACES
		0405 ZERO

Table 115. **SYSADATA token record** *(continued)*

Field	Size	Description
		0406 ZEROES
		ZEROS
		0407 NULL
		NULLS
		0501 BLOCK
		0502 BOTTOM
		0505 CHARACTER
		0506 CODE
		0507 CODE-SET
		0514 FILLER
		0516 FOOTING
		0520 LABEL
		0521 LENGTH
		0524 LINAGE
		0526 OMITTED
		0531 RENAMES
		0543 TOP
		0545 TRAILING
		0549 RECORDING
		0601 INHERITS
		0603 RECURSIVE
		0701 ACCESS
		0702 ALSO
		0703 ALTERNATE
		0704 AREA
		AREAS
		0705 ASSIGN
		0707 COLLATING
		0708 COMMA
		0709 CURRENCY
		0710 CLASS
		0711 DECIMAL-POINT
		0712 DUPLICATES
		0713 DYNAMIC
		0714 EVERY

Table 115. **SYSADATA token record** *(continued)*

Field	Size	Description
		0716 MEMORY
		0717 MODE
		0718 MODULES
		0719 MULTIPLE
		0720 NATIVE
		0721 OFF
		0722 OPTIONAL
		0723 ORGANIZATION
		0724 POSITION
		0725 PROGRAM
		0726 RANDOM
		0727 RELATIVE
		0728 RERUN
		0729 RESERVE
		0730 SAME
		0731 SEGMENT-LIMIT
		0732 SELECT
		0733 SEQUENCE
		0734 SEQUENTIAL
		0736 SORT-MERGE
		0737 STANDARD-1
		0738 TAPE
		0739 WORDS
		0740 PROCESSING
		0741 APPLY
		0742 WRITE-ONLY
		0743 COMMON
		0744 ALPHABET
		0745 PADDING
		0746 SYMBOLIC
		0747 STANDARD-2
		0748 OVERRIDE
		0750 PASSWORD

Table 115. **SYSADATA token record** *(continued)*

Field	Size	Description	
		0801	ARE
			IS
		0802	ASCENDING
		0803	AT
		0804	BY
		0805	CHARACTERS
		0806	CONTAINS
		0808	COUNT
		0809	DEBUGGING
		0810	DEPENDING
		0811	DESCENDING
		0812	DIVISION
		0814	FOR
		0815	ORDER
		0816	INPUT
		0817	REPLACE
		0818	KEY
		0819	LINE
			LINES
		0821	OF
		0822	ON
		0823	OUTPUT
		0825	RECORD
		0826	RECORDS
		0827	REEL
		0828	SECTION
		0829	SIZE
		0830	STATUS
		0831	THROUGH
			THRU
		0832	TIME
		0833	TIMES
		0834	TO
		0836	UNIT

Table 115. **SYSADATA token record** *(continued)*

Field	Size	Description
		0837 USING
		0838 WHEN
		0839 WITH
		0901 PROCEDURE
		0902 DECLARATIVES
		0903 END
		1001 DATA
		1002 FILE
		1003 FD
		1004 SD
		1005 WORKING-STORAGE
		1006 LOCAL-STORAGE
		1007 LINKAGE
		1101 ENVIRONMENT
		1102 CONFIGURATION
		1103 SOURCE-COMPUTER
		1104 OBJECT-COMPUTER
		1105 SPECIAL-NAMES
		1106 REPOSITORY
		1107 INPUT-OUTPUT
		1108 FILE-CONTROL
		1109 I-O-CONTROL
		1201 ID
		IDENTIFICATION
		1202 PROGRAM-ID
		1203 AUTHOR
		1204 INSTALLATION
		1205 DATE-WRITTEN
		1206 DATE-COMPILED
		1207 SECURITY
		1208 CLASS-ID
		1209 METHOD-ID
		1210 METHOD
		1211 FACTORY

Table 115. **SYSADATA token record** *(continued)*

Field	Size	Description
		1212 OBJECT **2020** TRACE **3000** DATADEF **3001** F-NAME **3002** UPSI-SWITCH **3003** CONDNAME **3004** CONDVAR **3005** BLOB **3006** CLOB **3007** DBCLOB **3008** BLOB-LOCATOR **3009** CLOB-LOCATOR **3010** DBCLOB-LOCATOR **3011** BLOB-FILE **3012** CLOB-FILE **3013** DBCLOB-FILE **3014** DFHRESP **5001** PARSE **5002** AUTOMATIC **5003** PREVIOUS **9999** COBOL
Token length	HL2	The length of the token
Token column	FL4	The starting column number of the token on the source line
Token line	FL4	The line number of the token
Flags	CL1	Information about the token: **X'80'** Token is continued **X'40'** Last piece of continued token Note that for `PICTURE` strings, even if the source token is continued, there will be only one Token record generated. It will have a token code of 0000, the token column and line of the first piece, the length of the complete string, no continuation flags set, and the token text of the complete string.
Reserved	CL7	Reserved for future use
Token text	CL(*n*)	The actual token string

Source error record - X'0032'

The following table shows the contents of the source error record.

Table 116. **SYSADATA source error record**

Field	Size	Description
Statement number	FL4	The statement number of the statement in error
Error identifier	CL16	The error message identifier (left-justified and padded with blanks)
Error severity	HL2	The severity of the error
Error message length	HL2	The length of the error message text
Line position	XL1	The line position indicator provided in FIPS messages
Reserved	CL7	Reserved for future use
Error message	CL(*n*)	The error message text

Source record - X'0038'

The following table shows the contents of the source record.

Table 117. **SYSADATA source record**

Field	Size	Description
Line number	FL4	The listing line number of the source record
Input record number	FL4	The input source record number in the current input file
Primary file number	HL2	The input file's assigned sequence number if this record is from the primary input file. (Refer to the Input file *n* field in the Job identification record).
Library file number	HL2	The library input file's assigned sequence number if this record is from a COPY|BASIS input file. (Refer to the Member File ID *n* field in the Library record.)
Reserved	CL8	Reserved for future use
Parent record number	FL4	The parent source record number. This will be the record number of the COPY|BASIS statement.
Parent primary file number	HL2	The parent file's assigned sequence number if the parent of this record is from the primary input file. (Refer to the Input file *n* field in the Job Identification Record.)
Parent library assigned file number	HL2	The parent library file's assigned sequence number if this record's parent is from a COPY|BASIS input file. (Refer to the COPY/BASIS Member File ID *n* field in the Library record.)
Reserved	CL8	Reserved for future use
Length of source record	HL2	The length of the actual source record following
Reserved	CL10	Reserved for future use
Source record	CL(*n*)	

COPY REPLACING record - X'0039'

One COPY REPLACING type record will be emitted each time a REPLACING action takes place. That is, whenever *operand-1* of the REPLACING phrase is matched with text in the copybook, a COPY REPLACING TEXT record will be written.

The following table shows the contents of the COPY REPLACING record.

Table 118. **SYSADATA COPY REPLACING record**

Field	Size	Description
Starting line number of replaced string	FL4	The listing line number of the start of the text that resulted from REPLACING
Starting column number of replaced string	FL4	The listing column number of the start of the text that resulted from REPLACING
Ending line number of replaced string	FL4	The listing line number of the end of the text that resulted from REPLACING
Ending column number of replaced string	FL4	The listing column number of the end of the text that resulted from REPLACING
Starting line number of original string	FL4	The source file line number of the start of the text that was changed by REPLACING
Starting column number of original string	FL4	The source file column number of the start of the text that was changed by REPLACING
Ending line number of original string	FL4	The source file line number of the end of the text that was changed by REPLACING
Ending column number of original string	FL4	The source file column number of the end of the text that was changed by REPLACING

Symbol record - X'0042'

The following table shows the contents of the symbol record.

Table 119. **SYSADATA symbol record**

Field	Size	Description
Symbol ID	FL4	Unique ID of symbol
Line number	FL4	The listing line number of the source record in which the symbol is defined or declared
Level	XL1	True level-number of symbol (or relative level-number of a data item within a structure). For COBOL, this can be in the range 01-49, 66 (for RENAMES items), 77, or 88 (for condition items).
Qualification indicator	XL1	**X'00'** Unique name; no qualification needed. **X'01'** This data item needs qualification. The name is not unique within the program. This field applies only when this data item is *not* the level-01 name.

Table 119. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Symbol type	XL1	**X'68'** Class-name (Class-ID) **X'58'** Method-name **X'40'** Data-name **X'20'** Procedure-name **X'10'** Mnemonic-name **X'08'** Program-name **X'81'** Reserved The following are ORed into the above types, when applicable: **X'04'** External **X'02'** Global
Symbol attribute	XL1	**X'01'** Numeric **X'02'** Elementary character of one of these classes: • Alphabetic • Alphanumeric • DBCS • National **X'03'** Group **X'04'** Pointer **X'05'** Index data item **X'06'** Index-name **X'07'** Condition **X'0F'** File **X'10'** Sort file **X'17'** Class-name (repository) **X'18'** Object reference

Table 119. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Clauses	XL1	Clauses specified in symbol definition. For symbols that have a symbol attribute of Numeric (`X'01'`), Elementary character (`X'02'`), Group (`X'03'`), Pointer (`X'04'`), Index data item (`X'05'`), or Object reference (`X'18'`): **`1... ....`** Value **`.1.. ....`** Indexed **`..1. ....`** Redefines **`...1 ....`** Renames **`.... 1...`** Occurs **`.... .1..`** Has Occurs keys **`.... ..1.`** Occurs Depending On **`.... ...1`** Occurs in parent For both file types: **`1... ....`** Select **`.1.. ....`** Assign **`..1. ....`** Rerun **`...1 ....`** Same area **`.... 1...`** Same record area **`.... .1..`** Recording mode **`.... ..1.`** Reserved **`.... ...1`** Record

Table 119. **SYSADATA symbol record** *(continued)*

Field	Size	Description
		For mnemonic-name symbols: **01** CSP **02** C01 **03** C02 **04** C03 **05** C04 **06** C05 **07** C06 **08** C07 **09** C08 **10** C09 **11** C10 **12** C11 **13** C12 **14** S01 **15** S02 **16** S03 **17** S04 **18** S05 **19** CONSOLE **20** SYSIN|SYSIPT **22** SYSOUT|SYSLST|SYSLIST **24** SYSPUNCH|SYSPCH **26** UPSI-0 **27** UPSI-1 **28** UPSI-2 **29** UPSI-3 **30** UPSI-4 **31** UPSI-5 **32** UPSI-6 **33** UPSI-7 **34** AFP-5A

Table 119. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Data flags 1	XL1	For both file types, and for symbols that have a symbol attribute of Numeric (X'01'), Elementary character (X'02'), Group (X'03'), Pointer (X'04'), Index data item (X'05'), or Object reference (X'18'): **1... ....** Redefined **.1.. ....** Renamed **..1. ....** Synchronized **...1 ....** Implicitly redefined **.... 1...** Date field **.... .1..** Implicit redefines **.... ..1.** FILLER **.... ...1** Level 77

Table 119. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Data flags 2	XL1	For symbols that have a symbol attribute of Numeric (`X'01'`): **`1... ....`** Binary **`.1.. ....`** External floating point (of `USAGE DISPLAY` or `USAGE NATIONAL`) **`..1. ....`** Internal floating point **`...1 ....`** Packed **`.... 1...`** External decimal (of `USAGE DISPLAY` or `USAGE NATIONAL`) **`.... .1..`** Scaled negative **`.... ..1.`** Numeric edited (of `USAGE DISPLAY` or `USAGE NATIONAL`) **`.... ...1`** Reserved for future use For symbols that have a symbol attribute of Elementary character (`X'02'`) or Group (`X'03'`): **`1... ....`** Alphabetic **`.1.. ....`** Alphanumeric **`..1. ....`** Alphanumeric edited **`...1 ....`** Group contains its own ODO object **`.... 1...`** DBCS item **`.... .1..`** Group variable length **`.... ..1.`** EGCS item **`.... ...1`** EGCS edited

Table 119. **SYSADATA symbol record** *(continued)*

Field	Size	Description
		For both file types: `1... ....` Object of ODO in record `.1.. ....` Subject of ODO in record `..1. ....` Sequential access `...1 ....` Random access `.... 1...` Dynamic access `.... .1..` Locate mode `.... ..1.` Record area `.... ...1` Reserved for future use Field will be zero for all other data types.
Data flags 3	XL1	For both file types: `1... ....` All records are the same length `.1.. ....` Fixed length `..1. ....` Variable length `...1 ....` Undefined `.... 1...` Spanned `.... .1..` Blocked `.... ..1.` Apply write only `.... ...1` Same sort merge area Field will be zero for all other data types.

Table 119. **SYSADATA symbol record** *(continued)*

Field	Size	Description
File organization	XL1	For both file types: **1... ....** QSAM **.1.. ....** ASCII **..1. ....** Standard label **...1 ....** User label **.... 1...** VSAM sequential **.... .1..** VSAM indexed **.... ..1.** VSAM relative **.... ...1** Line sequential Field will be zero for all other data types.
USAGE clause	FL1	**X'00'** USAGE IS DISPLAY **X'01'** USAGE IS COMP-1 **X'02'** USAGE IS COMP-2 **X'03'** USAGE IS PACKED-DECIMAL or USAGE IS COMP-3 **X'04'** USAGE IS BINARY, USAGE IS COMP, or USAGE IS COMP-4 **X'05'** USAGE IS DISPLAY-1 **X'06'** USAGE IS POINTER **X'07'** USAGE IS INDEX **X'08'** USAGE IS PROCEDURE-POINTER **X'09'** USAGE IS OBJECT-REFERENCE **X'0B'** NATIONAL **X'0A'** FUNCTION-POINTER
Sign clause	FL1	**X'00'** No SIGN clause **X'01'** SIGN IS LEADING **X'02'** SIGN IS LEADING SEPARATE CHARACTER **X'03'** SIGN IS TRAILING **X'04'** SIGN IS TRAILING SEPARATE CHARACTER
Indicators	FL1	**X'01'** Has JUSTIFIED clause. Right-justified attribute is in effect. **X'02'** Has BLANK WHEN ZERO clause.

Table 119. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Size	FL4	The size of this data item. The actual number of bytes this item occupies in storage. If a DBCS item, the number is in bytes, not characters. For variable-length items, this field will reflect the maximum size of storage reserved for this item by the compiler. Also known as the "Length attribute."
Precision	FL1	The precision of a fixed or float data item
Scale	FL1	The scale factor of a fixed data item. This is the number of digits to the right of the decimal point.

Table 119. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Base locator type	FL1	For host:
		01 Base Locator File
		02 Base Locator Working-Storage
		03 Base Locator Linkage Section
		05 Base Locator special regs
		07 Indexed by variable
		09 COMREG special reg
		10 UPSI switch
		13 Base Locator for Varloc items
		14 Base Locator for Extern data
		15 Base Locator alphanumeric FUNC
		16 Base Locator alphanumeric EVAL
		17 Base Locator for Object data
		19 Base Locator for Local-Storage
		20 Factory data
		21 `XML-TEXT` and `XML-NTEXT`
		For Windows and AIX:
		01 Base Locator File
		02 Base Locator Linkage Section
		03 Base Locator for Varloc items
		04 Base Locator for Extern data
		05 Base Locator for Object data
		06 `XML-TEXT` and `XML-NTEXT`
		10 Base Locator Working-Storage
		11 Base Locator special regs
		12 Base Locator alphanumeric FUNC
		13 Base Locator alphanumeric EVAL
		14 Indexed by variable
		16 COMREG special reg
		17 UPSI switch
		18 Factory data
		22 Base Locator for Local-Storage

Table 119. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Date format	FL1	Date format: **01** YY **02** YYXX **03** YYXXXX **04** YYXXX **05** YYYY **06** YYYYXX **07** YYYYXXXX **08** YYYYXXX **09** YYX **10** YYYYX **22** XXYY **23** XXXXYY **24** XXXYY **26** XXYYYY **27** XXXXYYYY **28** XXXYYYY **29** XYY **30** XYYYY
Data flags 4	XL1	For symbols that have a symbol attribute of Numeric (X'01'): **1... ....** Numeric national For symbols that have a symbol attribute of Elementary character (X'02'): **1... ....** National **.1.. ....** National edited For symbols that have a symbol attribute of Group (X'03'): **1... ....** Group-Usage National
Reserved	FL3	Reserved for future use

Table 119. **SYSADATA symbol record** *(continued)*

Field	**Size**	**Description**
Addressing information	FL4	For host, the Base Locator number and displacement: **Bits 0-4** Unused **Bits 5-19** Base Locator (BL) number **Bits 20-31** Displacement off Base Locator For Windows and AIX, the W-code SymId.
Structure displacement	AL4	Offset of symbol within structure. This offset is set to 0 for variably located items.
Parent displacement	AL4	Byte offset from immediate parent of the item being defined.
Parent ID	FL4	The symbol ID of the immediate parent of the item being defined.
Redefined ID	FL4	The symbol ID of the data item that this item redefines, if applicable.
Start-renamed ID	FL4	If this item is a level-66 item, the symbol ID of the starting COBOL data item that this item renames. If not a level-66 item, this field is set to 0.
End-renamed ID	FL4	If this item is a level-66 item, the symbol ID of the ending COBOL data item that this item renames. If not a level-66 item, this field is set to 0.
Program-name symbol ID	FL4	ID of the program-name of the program or the class-name of the class where this symbol is defined.
`OCCURS` minimum Paragraph ID	FL4	Minimum value for `OCCURS` Proc-name ID for a paragraph-name
`OCCURS` maximum Section ID	FL4	Maximum value for `OCCURS` Proc-name ID for a section-name
Dimensions	FL4	Number of dimensions
Reserved	CL12	Reserved for future use
Value pairs count	HL2	Count of value pairs
Symbol name length	HL2	Number of characters in the symbol name
Picture data length for data-name or Assignment-name length for file-name	HL2	Number of characters in the picture data; zero if symbol has no associated `PICTURE` clause. (Length of the `PICTURE` field.) Length represents the field as it is found in the source input. This length does not represent the expanded field for `PICTURE` items that contain a replication factor. The maximum COBOL length for a `PICTURE` string is 50 bytes. Zero in this field indicates no `PICTURE` specified. Number of characters in the external file-name if this is a file-name. This is the `DD` name part of the assignment-name. Zero if file-name and `ASSIGN USING` specified.

Table 119. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Initial Value length for data-name External class-name length for CLASS-ID	HL2	Number of characters in the symbol value; zero if symbol has no initial value Number of characters in the external class-name for CLASS-ID
ODO symbol name ID for data-name ID of ASSIGN data-name if file-name	FL4	If data-name, ID of the ODO symbol name; zero if ODO not specified If file-name, Symbol-ID for ASSIGN USING data-name; zero if ASSIGN TO specified
Keys count	HL2	The number of keys defined
Index count	HL2	Count of Index symbol IDs; zero if none specified
Symbol name	CL(*n*)	
Picture data string for data-name or Assignment-name for file-name	CL(*n*)	The PICTURE character string *exactly* as the user types it in. The character string includes all symbols, parentheses, and replication factor. The external file-name if this is a file-name. This is the DD name part of the assignment-name.
Index ID list	(*n*)FL4	ID of each index symbol name
Keys	(*n*)XL8	This field contains data describing keys specified for an array. The following three fields are repeated as many times as specified in the 'Keys count' field.
...Key Sequence	FL1	Ascending or descending indicator. **X'00'** DESCENDING **X'01'** ASCENDING
...Filler	CL3	Reserved
...Key ID	FL4	The symbol ID of the data item that is the key field in the array
Initial Value data for data-name External class-name for CLASS-ID	CL(*n*)	This field contains the data specified in the INITIAL VALUE clause for this symbol. The following four subfields are repeated according to the count in the 'Value pairs count' field. The total length of the data in this field is contained in the 'Initial value length' field. The external class-name for CLASS-ID.
...1st value length	HL2	Length of first value
...1st value data	CL(*n*)	1st value. This field contains the literal (or figurative constant) as it is specified in the VALUE clause in the source file. It includes any beginning and ending delimiters, embedded quotation marks, and SHIFT IN and SHIFT OUT characters. If the literal spans multiple lines, the lines are concatenated into one long string. If a figurative constant is specified, this field contains the actual reserved word, not the value associated with that word.
...2nd value length	HL2	Length of second value, zero if not a THRU value pair

Table 119. **SYSADATA symbol record** *(continued)*

Field	Size	Description
...2nd value data	CL(*n*)	2nd value. This field contains the literal (or figurative constant) as it is specified in the VALUE clause in the source file. It includes any beginning and ending delimiters, embedded quotation marks, and SHIFT IN and SHIFT OUT characters. If the literal spans multiple lines, the lines are concatenated into one long string. If a figurative constant is specified, this field contains the actual reserved word, not the value associated with that word.

Symbol cross-reference record - X'0044'

The following table shows the contents of the symbol cross-reference record.

Table 120. **SYSADATA symbol cross-reference record**

Field	Size	Description
Symbol length	HL2	The length of the symbol
Statement definition	FL4	The statement number where the symbol is defined or declared **For VERB XREF only:** Verb count - total number of references to this verb.
Number of references[1]	HL2	The number of references in this record to the symbol following
Cross-reference type	XL1	**X'01'** Program **X'02'** Procedure **X'03'** Verb **X'04'** Symbol or data-name **X'05'** Method **X'06'** Class
Reserved	CL7	Reserved for future use
Symbol name	CL(*n*)	The symbol. Variable length.

Table 120. **SYSADATA symbol cross-reference record** *(continued)*

Field	Size	Description
...Reference flag	CL1	For symbol or data-name references: **C' '** Blank means reference only **C'M'** Modification reference flag For Procedure type symbol references: **C'A'** `ALTER` (procedure-name) **C'D'** `GO TO` (procedure-name) `DEPENDING ON` **C'E'** End of range of (`PERFORM`) through (procedure-name) **C'G'** `GO TO` (procedure-name) **C'P'** `PERFORM` (procedure-name) **C'T'** (`ALTER`) `TO PROCEED TO` (procedure-name) **C'U'** Use for debugging (procedure-name)
...Statement number	XL4	The statement number on which the symbol or verb is referenced
1. The reference flag field and the statement number field occur as many times as the number of references field dictates. For example, if there is a value of 10 in the number of references field, there will be 10 occurrences of the reference flag and statement number pair for data-name, procedure, or program symbols, or 10 occurrences of the statement number for verbs. Where the number of references would exceed the record size for the SYSADATA file, the record is continued on the next record. The continuation flag is set in the common header section of the record.		

Nested program record - X'0046'

The following table shows the contents of the nested program record.

Table 121. **SYSADATA nested program record**

Field	Size	Description
Statement definition	FL4	The statement number where the symbol is defined or declared
Nesting level	XL1	Program nesting level
Program attributes	XL1	**1... ....** Initial **.1.. ....** Common **..1. ....** `PROCEDURE DIVISION using` **...1 1111** Reserved for future use
Reserved	XL1	Reserved for future use
Program-name length	XL1	Length of the following field
Program-name	CL(*n*)	The program-name

Library record - X'0060'

The following table shows the contents of the SYSADATA library record.

Table 122. **SYSADATA library record**

Field	Size	Description
Number of members[1]	HL2	Count of the number of COPY/INCLUDE code members described in this record
Library name length	HL2	The length of the library name
Library volume length	HL2	The length of the library volume ID
Concatenation number	XL2	Concatenation number of the library
Library ddname length	HL2	The length of the library ddname
Reserved	CL4	Reserved for future use
Library name	CL(*n*)	The name of the library from which the COPY/INCLUDE member was retrieved
Library volume	CL(*n*)	The volume identification of the volume where the library resides
Library ddname	CL(*n*)	The ddname (or equivalent) used for this library
...COPY/BASIS member file ID[2]	HL2	The library file ID of the name following
...COPY/BASIS name length	HL2	The length of the name following
...COPY/BASIS name	CL(*n*)	The name of the COPY/BASIS member that has been used

1. If 10 COPY members are retrieved from a library, the "Number of members" field will contain 10 and there will be 10 occurrences of the "COPY/BASIS member file ID" field, the "COPY/BASIS name length" field, and the "COPY/BASIS name" field.
2. If COPY/BASIS members are retrieved from different libraries, a library record is written to the SYSADATA file for each unique library.

Statistics record - X'0090'

The following table shows the contents of the statistics record.

Table 123. **SYSADATA statistics record**

Field	Size	Description
Source records	FL4	The number of source records processed
DATA DIVISION statements	FL4	The number of data division statements processed
PROCEDURE DIVISION statements	FL4	The number of procedure division statements processed
Compilation number	HL2	Batch compilation number
Error severity	XL1	The highest error message severity

Table 123. **SYSADATA statistics record** *(continued)*

Field	Size	Description
Flags	XL1	**1... ....** End of Job indicator **.1.. ....** Class definition indicator **..11 1111** Reserved for future use
EOJ severity	XL1	The maximum return code for the compile job
Program-name length	XL1	The length of the program-name
Program-name	CL(*n*)	Program-name

EVENTS record - X'0120'

Events records are included in the ADATA file to provide compatibility with previous levels of the compiler.

Events records are of the following types:
- Timestamp
- Processor
- File end
- Program
- File ID
- Error

Table 124. **SYSADATA EVENTS TIMESTAMP record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type TIMESTAMP record	CL12	C'TIMESTAMP'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Date	XL8	YYYYMMDD
Hour	XL2	HH
Minutes	XL2	MI
Seconds	XL2	SS

Table 125. **SYSADATA EVENTS PROCESSOR record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)

Table 125. **SYSADATA EVENTS PROCESSOR record layout** *(continued)*

Field	Size	Description
EVENTS record type PROCESSOR record	CL9	C'PROCESSOR'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Output file ID	XL1	
Blank separator	CL1	
Line-class indicator	XL1	

Table 126. **SYSADATA EVENTS FILE END record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type FILE END record	CL7	C'FILEEND'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Input file ID	XL1	
Blank separator	CL1	
Expansion indicator	XL1	

Table 127. **SYSADATA EVENTS PROGRAM record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type PROGRAM record	CL7	C'PROGRAM'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Output file ID	XL1	
Blank separator	CL1	
Program input record number	XL1	

Table 128. **SYSADATA EVENTS FILE ID record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header

Table 128. **SYSADATA EVENTS FILE ID record layout** *(continued)*

Field	Size	Description
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type FILE ID record	CL7	C'FILEID'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Input source file ID	XL1	File ID of source file
Blank separator	CL1	
Reference indicator	XL1	
Blank separator	CL1	
Source file name length	H2	
Blank separator	CL1	
Source file name	CL(n)	

Table 129. **SYSADATA EVENTS ERROR record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type ERROR record	CL5	C'ERROR'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Input source file ID	XL1	File ID of source file
Blank separator	CL1	
Annot class	XL1	Annot-class message placement
Blank separator	CL1	
Error input record number	XL10	
Blank separator	CL1	
Error start line number	XL10	
Blank separator	CL1	
Error token start number	XL1	Column number of error token start
Blank separator	CL1	
Error end line number	XL10	
Blank separator	CL1	

Table 129. **SYSADATA EVENTS ERROR record layout** *(continued)*

Field	Size	Description
Error token end number	XL1	Column number of error token end
Blank separator	CL1	
Error message ID number	XL9	
Blank separator	CL1	
Error message severity code	XL1	
Blank separator	CL1	
Error message severity level number	XL2	
Blank separator	CL1	
Error message length	HL3	
Blank separator	CL1	
Error message text	CL(n)	

Appendix H. Sample programs

The sample programs, which are included on your product tape, demonstrate many language elements and concepts of COBOL.

This information contains the following items:
- Overview of the programs, including program charts for two of the samples
- Format and sample of the input data
- Sample of reports produced
- Information about how to run the programs
- List of the language elements and concepts that are illustrated

Pseudocode and other comments about the programs are included in the program prologue, which you can obtain in a program listing.

There are three sample programs:
- IGYTCARA is an example of using QSAM files and VSAM indexed files, and shows how to use many COBOL intrinsic functions.
- IGYTCARB is an example of using IBM Interactive System Product Facility (ISPF).
- IGYTSALE is an example of using several of the features of the Language Environment callable services.

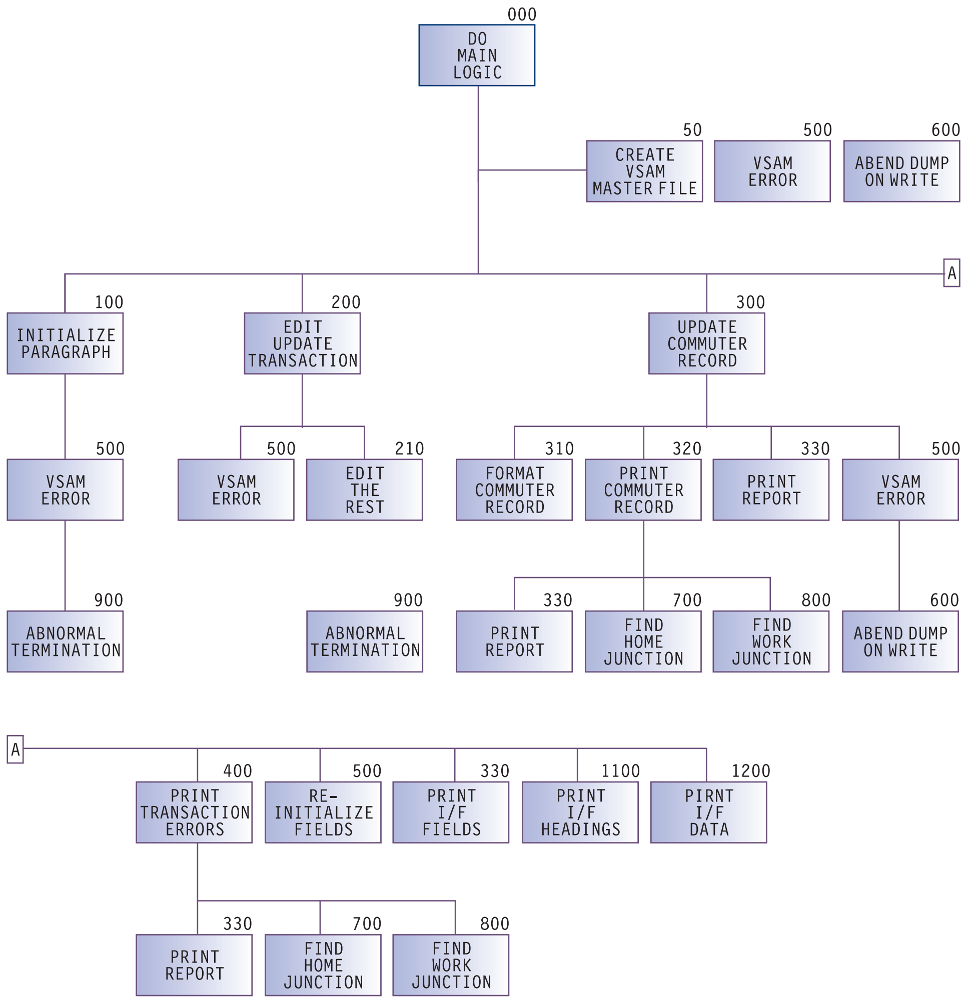
RELATED CONCEPTS
"IGYTCARA: batch application"
"IGYTCARB: interactive program" on page 771
"IGYTSALE: nested program application" on page 774

IGYTCARA: batch application

A company that has several local offices wants to establish employee carpools. Application IGYTCARA validates the transaction-file entries (QSAM sequential file processing) and updates a master file (VSAM indexed file processing).

This batch application does two tasks:
- Produces reports of employees who can share rides from the same home location to the same work location
- Updates the carpool data:
  - Adds data for new employees
  - Changes information for participating employees
  - Deletes employee records
  - Lists update requests that are not valid

The following diagram shows the parts of the application and how they are organized:

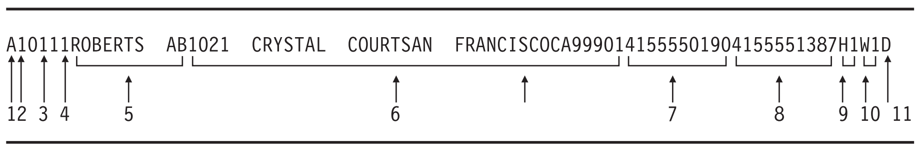


RELATED TASKS
"Preparing to run IGYTCARA" on page 770

RELATED REFERENCES
"Input data for IGYTCARA"
"Report produced by IGYTCARA" on page 769
"Language elements and concepts that are illustrated" on page 782

Input data for IGYTCARA

As input to the program, the company collected information from interested employees, coded the information, and produced an input file. Here is an example of the format of the input file (spaces between fields are left out, as they would be in your input file) with an explanation of each item.



```
A10111ROBERTS   AB1021   CRYSTAL   COURTSAN   FRANCISCOCA99901415555019041555513 87H1W1D
```

1. Transaction code
2. Shift

3. Home code
4. Work code
5. Commuter name
6. Home address
7. Home phone
8. Work phone
9. Home location code
10. Work location code
11. Driving status code

This sample below shows a section of the input file:

```
A10111ROBERTS  AB1021 CRYSTAL COURTSAN FRANCISCOCA99901415555019041555513 87H1W1D
A20212KAHN     DE789 EMILY LANE    SAN FRANCISCOCA99921415555189041555525 89H2W2D
P48899                                          99ASDFG0005557890123ASDFGHJ    T
R10111ROBERTS  AB1221 CRYSTAL COURTSAN FRANCISCOCA99901415555019041555513 87H1W1D
A20212KAHN     DE789 EMILY LANE    SAN FRANCISCOCA99921415555189041555525 89H2W2D
D20212KAHN     DE
D20212KAHN     DE
A20212KAHN     DE789 EMILY LANE    SAN FRANCISCOCA99921415555189041555525 89H2W2D
A10111BONNICK  FD1025 FIFTH AVENUE SAN FRANCISCOCA99905415555959041555578 95H8W3
A10111PETERSON SW435 THIRD AVENUE  SAN FRANCISCOCA99905415555469041555537 17H3W4
   . . .
```

Report produced by IGYTCARA

The following sample shows the first page of the output report produced by IGYTCARA. Your actual output might vary slightly in appearance, depending on your system.

```
1REPORT  #: IGYTCAR1                               COMMUTER FILE UPDATE LIST                                         PAGE #:    1
-PROGRAM #: IGYTCAR1       RUN TIME: 01:40                                                RUN DATE: 11/24/2003
-=========================================================================================================================
      | RE-|  SHIFT  |              |                        |              |                         |STA-|
 TRANS|CORD|HOME CODE|   COMMUTER   |         HOME           | HOME PHONE   | HOME LOCATION JUNCTION  |TUS | TRANS. ERROR
 CODE |TYPE|WORK CODE|     NAME     |        ADDRESS         | WORK PHONE   | WORK LOCATION JUNCTION  |CODE|
 =========================================================================================================================
  A    NEW   1 01 11  ROBERTS    AB 1021 CRYSTAL COURT       (415) 555-0190 RODNEY/CRYSTAL              D
                                    SAN FRANCISCO   CA 99901 (415) 555-1387 BAYFAIR PLAZA
 -------------------------------------------------------------------------------------------------------------------------
  A    NEW   2 02 12  KAHN       DE 789 EMILY LANE           (415) 555-1890 COYOTE                      D
                                    SAN FRANCISCO   CA 99921 (415) 555-2589 14TH STREET/166TH AVENUE
 -------------------------------------------------------------------------------------------------------------------------
  P          4 88 99                                         (000) 555-7890 HOME CODE '  ' NOT FOUND.   T
                                                    99 ASDFG (123) ASD-FGHJ WORK CODE '  ' NOT FOUND.       TRANSACT. CODE
                                                                                                            SHIFT CODE
                                                                                                            HOME LOC. CODE
                                                                                                            WORK LOC. CODE
                                                                                                            LAST NAME
                                                                                                            INITIALS
                                                                                                            ADDRESS
                                                                                                            CITY
                                                                                                            STATE CODE
                                                                                                            ZIPCODE
                                                                                                            HOME PHONE
                                                                                                            WORK PHONE
                                                                                                            HOME JUNCTION
                                                                                                            WORK JUNCTION
                                                                                                            DRIVING STATUS
 -------------------------------------------------------------------------------------------------------------------------
  R    OLD   1 01 11  ROBERTS    AB 1021 CRYSTAL COURT       (415) 555-0190 RODNEY/CRYSTAL              D
                                    SAN FRANCISCO   CA 99901 (415) 555-1387 BAYFAIR PLAZA
         NEW   1 01 11  ROBERTS    AB 1221 CRYSTAL COURT       (415) 555-0190 RODNEY/CRYSTAL              D
                                    SAN FRANCISCO   CA 99901 (415) 555-1387 BAYFAIR PLAZA
 -------------------------------------------------------------------------------------------------------------------------
  A          2 02 12  KAHN       DE 789 EMILY LANE           (415) 555-1890 COYOTE                      D
                                    SAN FRANCISCO   CA 99921 (415) 555-2589 14TH STREET/166TH AVENUE        DUPLICATE REC.
 -------------------------------------------------------------------------------------------------------------------------
  D    OLD   2 02 12  KAHN       DE 789 EMILY LANE           (415) 555-1890 COYOTE                      D
                                    SAN FRANCISCO   CA 99921 (415) 555-2589 14TH STREET/166TH AVENUE
 -------------------------------------------------------------------------------------------------------------------------
  D          2 02 12  KAHN       DE                                                                         REC. NOT FOUND
 -------------------------------------------------------------------------------------------------------------------------
  A    NEW   2 02 12  KAHN       DE 789 EMILY LANE           (415) 555-1890 COYOTE                      D
                                    SAN FRANCISCO   CA 99921 (415) 555-2589 14TH STREET/166TH AVENUE
 -------------------------------------------------------------------------------------------------------------------------
  A    NEW   1 01 11  BONNICK    FD 1025 FIFTH AVENUE        (415) 555-9590 RODNEY
                                    SAN FRANCISCO   CA 99905 (415) 555-7895 17TH FREEWAY SAN LEANDRO
 -------------------------------------------------------------------------------------------------------------------------
  A    NEW   1 01 11  PETERSON   SW 435 THIRD AVENUE         (415) 555-4690 RODNEY/THIRD AVENUE
```

Preparing to run IGYTCARA

All files required by the IGYTCARA program (IGYTCARA, IGYTCODE, and IGYTRANX) are supplied on the product installation tape. These files are located in the IGY.V3R4M0.SIGYSAMP data set.

Data set and procedure names might be changed at installation time. You should check with your system programmer to verify these names.

Do not change these options on the `CBL` statement in the source file for IGYTCARA:

- `NOADV`
- `NODYNAM`
- `NONAME`
- `NONUMBER`
- `QUOTE`
- `SEQUENCE`

With these options in effect, the program will not cause any diagnostic messages to be issued. You can use the sequence number string in the source file to search for the language elements used.

RELATED CONCEPTS
"IGYTCARA: batch application" on page 767

RELATED TASKS
"Running IGYTCARA"

RELATED REFERENCES
"Input data for IGYTCARA" on page 768
"Report produced by IGYTCARA" on page 769
"Language elements and concepts that are illustrated" on page 782

Running IGYTCARA

The following procedure compiles, link-edits, and runs the IGYTCARA program. If you want only to compile or only to compile and link-edit the program, you need to change the IGYWCLG cataloged procedure.

To run IGYTCARA under z/OS, use JCL to define a VSAM cluster and compile the program. Insert the information specific to your system and installation in the fields that are shown in lowercase letters (accounting information, volume serial number, unit name, cluster prefix). These examples use the name IGYTCAR.MASTFILE; you can use another name if you want to.

1. Use this JCL to create the required VSAM cluster:

```
//CREATE   JOB (acct-info),'IGYTCAR CREATE VSAM',MSGLEVEL=(1,1),
// TIME=(0,29)
//CREATE   EXEC PGM=IDCAMS
//VOL1     DD VOL=SER=your-volume-serial,UNIT=your-unit,DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSIN    DD *
 DELETE your-prefix.IGYTCAR.MASTFILE -
     FILE(VOL1) -
     PURGE
 DEFINE CLUSTER -
    (NAME(your-prefix.IGYTCAR.MASTFILE) -
     VOLUME(your-volume-serial) -
     FILE(VOL1) -
     INDEXED -
     RECSZ(80 80) -
     KEYS(16 0) -
     CYLINDERS(2))
/*
```

To remove any existing cluster, a DELETE is issued before the VSAM cluster is created.

2. Use the following JCL to compile, link-edit, and run the IGYTCARA program:

```
//IGYTCARA JOB (acct-info),'IGYTCAR',MSGLEVEL=(1,1),TIME=(0,29)
//TEST  EXEC IGYWCLG
//COBOL.SYSLIB  DD DSN=IGY.V3R4M0.SIGYSAMP,DISP=SHR
//COBOL.SYSIN   DD DSN=IGY.V3R4M0.SIGYSAMP(IGYTCARA),DISP=SHR
//GO.SYSOUT     DD SYSOUT=A
//GO.COMMUTR    DD DSN=your-prefix.IGYTCAR.MASTFILE,DISP=SHR
//GO.LOCCODE    DD DSN=IGY.V3R4M0.SIGYSAMP(IGYTCODE),DISP=SHR
//GO.UPDTRANS   DD DSN=IGY.V3R4M0.SIGYSAMP(IGYTRANX),DISP=SHR
//GO.UPDPRINT   DD SYSOUT=A,DCB=BLKSIZE=133
//
```

RELATED TASKS
Chapter 10, "Processing VSAM files," on page 179

RELATED REFERENCES
"Compile, link-edit, and run procedure (IGYWCLG)" on page 251

IGYTCARB: interactive program

IGYTCARB contains an interactive program for entering carpool data by using IBM Interactive System Productivity Facility (ISPF) to invoke Dialog Manager and Enterprise COBOL. IGYTCARB creates a file that can be used as input for a carpool listing or matching program such as IGYTCARA.

The input data for IGYTCARB is the same as that for IGYTCARA. IGYTCARB lets you append to the information in your input file by using an ISPF panel. An example of the panel used by IGYTCARB is shown below:

```
--------------------------- CARPOOL DATA ENTRY --------------------------------
              New Data Entry                              Previous Entry
Type =======> -                             A, R, or D    A
Shift ======> -                             1, 2, or 3    1
Home Code ==> --                            2  Chars      01
Work Code ==> --                            2  Chars      11
Name =======> ---------                     9  Chars      POPOWICH
Initials ===> --                            2  Chars      AD
Address ====> ------------------            18 Chars      134 SIXTH AVENUE
City =======> -------------                 13 Chars      SAN FRANCISCO
State ======> --                            2  Chars      CA
Zip Code ===> -----                         5  Chars      99903
Home Phone => ----------                    10 Chars      4155553390
Work Phone => ----------                    10 Chars      4155557855
Home Jnc code > --                          2  Chars      H3
Work Jnc Code > --                          2  Chars      W7
Commuter Stat > -                           D, R or blank
```

RELATED TASKS
"Preparing to run IGYTCARB" on page 772

Preparing to run IGYTCARB

Run the IGYTCARB program under Interactive System Productivity Facility (ISPF). All files required by IGYTCARB (IGYTCARB, IGYTRANB, and IGYTPNL) are supplied on the product installation tape in the IGY.V3R4M0.SIGYSAMP data set.

Data set names and procedure-names might be changed at installation time. Check with your system programmer to verify the names.

Do not change the following options on the `CBL` card in the source file for IGYTCARB:

- `NONUMBER`
- `QUOTE`
- `SEQUENCE`

With these options in effect, the program will not cause any diagnostic messages to be issued. You can use the sequence number string in the source file to search for language elements.

RELATED CONCEPTS
"IGYTCARB: interactive program" on page 771

RELATED TASKS
"Running IGYTCARB"

RELATED REFERENCES
"Language elements and concepts that are illustrated" on page 782

Running IGYTCARB

The following procedure compiles, link-edits, and runs the IGYTCARB program. If you want only to compile or only to compile and link-edit the program, you need to change the procedure.

To run IGYTCARB under z/OS, do the following steps:

1. Using the ISPF editor, change the ISPF/PDF Primary Option Panel (ISR@PRIM) or some other panel to include the IGYTCARB invocation. Panel ISR@PRIM is in your site's PDF panel data set (normally ISRPLIB).

   The following example shows an ISR@PRIM panel modified, in two identified locations, to include the IGYTCARB invocation. If you add or change an option in the upper portion of the panel definition, you must also add or change the corresponding line on the lower portion of the panel.

```
%---------------------- ISPF/PDF PRIMARY OPTION PANEL ------------------------
%OPTION  ===>_ZCMD                                                           +
%                                                         +USERID   - &ZUSER
%   0 +ISPF PARMS  - Specify terminal and user parameters   +TIME     - &ZTIME
%   1 +BROWSE      - Display source data or output listings +TERMINAL - &ZTERM
%   2 +EDIT        - Create or change source data           +PF KEYS  - &ZKEYS
%   3 +UTILITIES   - Perform utility functions
%   4 +FOREGROUND  - Invoke language processors in foreground
%   5 +BATCH       - Submit to batch for language processing
%   6 +COMMAND     - Enter TSO or Workstation commands
%   7 +DIALOG TEST - Perform dialog testing
%   8 +LM UTILITIES- Perform library management utility functions
%   C +IGYTCARB    - Run IGYTCARB UPDATE TRANSACTION PROGRAM        (1)
%   T +TUTORIAL    - Display information about ISPF/PDF
%   X +EXIT        - Terminate using console, log, and list defaults
%
%
```

```
+Enter%END+command to terminate ISPF.
%
)INIT
  .HELP = ISR00003
  &ZPRIM = YES         /* ALWAYS A PRIMARY OPTION MENU */
  &ZHTOP = ISR00003    /* TUTORIAL TABLE OF CONTENTS   */
  &ZHINDEX = ISR91000  /* TUTORIAL INDEX - 1ST PAGE    */
  VPUT (ZHTOP,ZHINDEX) PROFILE
)PROC
  &Z1 = TRUNC(&ZCMD,1)
  IF (&Z1 &notsym.= '.')
    &ZSEL = TRANS( TRUNC (&ZCMD,'.')
                 0,'PANEL(ISPOPTA)'
                 1,'PGM(ISRBRO)  PARM(ISRBRO01)'
                 2,'PGM(ISREDIT) PARM(P,ISREDM01)'
                 3,'PANEL(ISRUTIL)'
                 4,'PANEL(ISRFPA)'
                 5,'PGM(ISRJB1) PARM(ISRJPA) NOCHECK'
                 6,'PGM(ISRPCC)'
                 7,'PGM(ISRYXDR) NOCHECK'
                 8,'PANEL(ISRLPRIM)'
                 C,'PGM(IGYTCARB)'                         (2)
                 T,'PGM(ISPTUTOR) PARM(ISR00000)'
               ' ',' '
                 X,'EXIT'
                 *,'?' )
    &ZTRAIL = .TRAIL
  IF (&Z1 = '.') .msg = ISPD141
)END
```

As indicated by **(1)** in this example, you add IGYTCARB to the upper portion of the panel by entering:

```
%   C +IGYTCARB    - Run IGYTCARB UPDATE TRANSACTION PROGRAM
```

You add the corresponding line on the lower portion of the panel, indicated by **(2)**, by entering:

```
C,'PGM(IGYTCARB)'
```

2. Place ISR@PRIM (or your other modified panel) and IGYTPNL in a library and make this library the first library in the ISPPLIB concatenation.
3. Comment sequence line IB2200 and uncomment sequence line IB2210 in IGYTCARB. (The `OPEN EXTEND` verb is supported under z/OS.)
4. Compile and link-edit IGYTCARB and place the resulting load module in your LOADLIB.
5. Allocate ISPLLIB using the following command:

   ```
   ALLOCATE FILE(ISPLLIB) DATASET(DSN1, SYS1.COBLIB, DSN2) SHR REUSE
   ```

   Here *DSN1* is the library name of the LOADLIB from step 4. *DSN2* is your installed ISPLLIB.
6. Allocate the input and output data sets using the following command:

   ```
   ALLOCATE FILE(UPDTRANS) DA('IGY.V3R4M0.SIGYSAMP(IGYTRANB)) SHR REUSE
   ```
7. Allocate ISPPLIB using the following command:

   ```
   ALLOCATE FILE(ISPPLIB) DATASET(DSN3, DSN4) SHR REUSE
   ```

   Here *DSN3* is the library containing the modified panels. *DSN4* is the ISPF panel library.
8. Invoke IGYTCARB using your modified panel.

RELATED REFERENCES

ISPF Dialog Developer's Guide and Reference

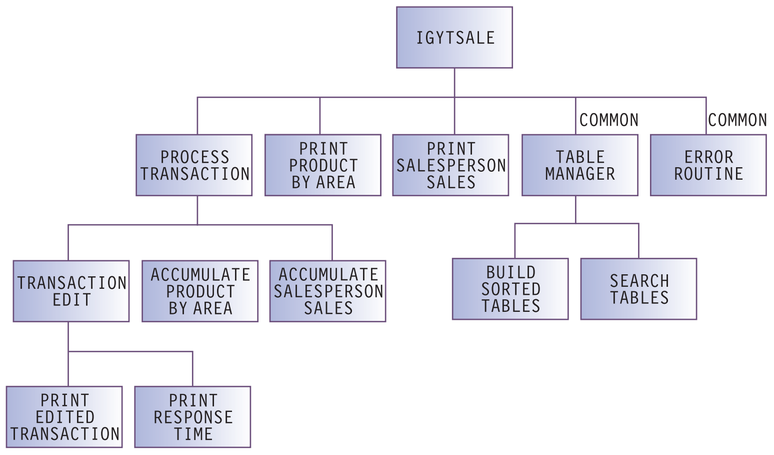
IGYTSALE: nested program application

Application IGYTSALE tracks product sales and sales commissions for a sporting-goods distributor.

This nested program application does the following tasks:

1. Keeps a record of the product line, customers, and number of salespeople. This data is stored in a file called IGYTABLE.
2. Maintains a file that records valid transactions and transaction errors. All transactions that are not valid are flagged, and the results are printed in a report. Transactions to be processed are in a file called IGYTRANA.
3. Processes transactions and report sales by location.
4. Records an individual's sales performance and commission, and prints the results in a report.
5. Reports the sale and shipment dates in local time and UTC (Universal Time Coordinate), and calculates the response time.

The following diagram shows the parts of the application as a hierarchy:



The following diagram shows how the parts are nested:

```
IGYTSALE
    Process-transactions.
        Transaction-edit is Initial.
            Print-edited-Transactions and Response-Time.
            End Program Print-edited-transactions.
        End Program transaction-edit.
        Accumulate-product-by-area.
        End Program Accumulate-product-by-area.
        Accumulate-salesperson-sales.
        End Accumulate-salesperson-sales.
    End Program Process-transactions.
    Print-product-by-area.
    End Program Print-product-by-area.
    Print-salesperson-sales.
    End Program Print-salesperson-sales.
    Table-manager is common.
        Build-sorted-tables.
        End Program Build-sorted-tables.
        Search-tables.
        End Program Search-tables.
    End Program Table-manager.
    Error-routine is Common.
    End Program Error-routine is Common.
End Program IGYTSALE.
```

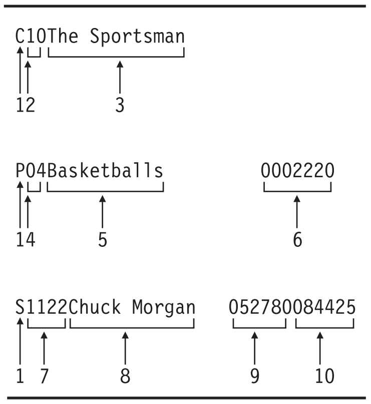
RELATED TASKS
"Preparing to run IGYTSALE" on page 781

RELATED REFERENCES
"Input data for IGYTSALE"
"Reports produced by IGYTSALE" on page 777
"Language elements and concepts that are illustrated" on page 782

Input data for IGYTSALE

As input to our program, the distributor collected information about its customers, salespeople, and products, coded the information, and produced an input file.

This input file, called IGYTABLE, is loaded into three separate tables for use during transaction processing. The format of the file is as follows, with an explanation of the items below:


```
C10The Sportsman

P04Basketballs          0002220

S1122Chuck Morgan     052780084425
```

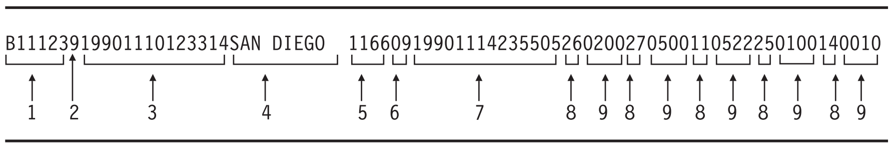
1. Record type
2. Customer code
3. Customer name

4. Product code
5. Product description
6. Product unit price
7. Salesperson number
8. Salesperson name
9. Date of hire
10. Commission rate

The value of field 1 (C, P, or S) determines the format of the input record. The following sample shows a section of IGYTABLE:

```
S1111Edyth Phillips 062484042327
S1122Chuck Morgan   052780084425
S1133Art Tung       022882061728
S1144Billy Jim Bob  010272121150
S1155Chris Preston  122083053377
S1166Al Willie Roz  111276100000
P01Footballs           0000620
P02Football Equipment  0032080
P03Football Uniform    0004910
P04Basketballs         0002220
P05Basketball Rim/Board0008830
P06Basketball Uniform  0004220
C01L. A. Sports
C02Gear Up
C03Play Outdoors
C04Sports 4 You
C05Sports R US
C06Stay Active
C07Sport Shop
C08Stay Sporty
C09Hot Sports
C10The Sportsman
C11Playing Ball
C12Sports Play
. . .
```

In addition, the distributor collected information about sales transactions. Each transaction represents an individual salesperson's sales to a particular customer. The customer can purchase from one to five items during each transaction. The transaction information is coded and put into an input file, called IGYTRANA. The format of this file is as follows, with an explanation of the items below:


B1112391990111010123314SAN DIEGO   11660919901114235505260200270500110522250100140010
1
2
3
4
5
6
7
8
9
8
9
8
9
8
9
8
9

1. Sales order number
2. Invoiced items (number of different items ordered)
3. Date of sale (year month day hour minutes seconds)
4. Sales area
5. Salesperson number
6. Customer code
7. Date of shipment (year month day hour minutes seconds)
8. Product code

9. Quantity sold

Fields 8 and 9 occur one to eight times depending on the number of different items ordered (field 2). The following sample shows a section of IGYTRANA:

```
A00001119900227010101CNTRL VALLEY11442019900228259999
A00004119900310100530CNTRL VALLEY11441019900403150099
A00005119900418222409CNTRL VALLEY11441219900419059900
A00006119900523151010CNTRL VALLEY11442019900623250004
      419990324591515SAN DIEGO    11615        60200132200110522045100
B11114419901111003301SAN DIEGO   11661519901114260200132200110522041100
A00007119901115003205CNTRL VALLEY11332019901117120023
C00125419900118101527SF BAY AREA 11331519900120160200112200250522145111
B11116419901201132013SF BAY AREA 11331519901203060200102200110522045102
B11117319901201070833SAN Diego   1165661990120333020o132200120522041100
B11118419901221191544SAN DIEGO   11661419901223160200142200130522040300
B11119419901210211544SAN DIEGO   11221219901214060200152200160522050500
B11120419901212000816SAN DIEGO   11220419901213150200052200160522040100
B11121419901201131544SAN DIEGO   11330219901203120200112200140522250100
B11122419901112073312SAN DIEGO   11221019901113100200162200260522250100
B11123919901110123314SAN DIEGO   11660919901114260200270500110522250100140010
B11124219901313510000SAN DIEGO   116611        1 0200042200120a22141100
B11125419901215012510SAN DIEGO   11661519901216110200162200130522141111
B11126119901111000034SAN DIEGO   11331619901113260022
B11127119901110154100SAN DIEGO   11221219901113122000
B11128419901110175001SAN DIEGO   11661519901113260200132200160521041104
. . .
```

Reports produced by IGYTSALE

The figures referenced below are samples of IGYTSALE output.

The program records the following data in reports:

- Transaction errors
- Sales by product and area
- Individual sales performance and commissions
- Response time between the sale date and the date the sold products are shipped

Your output might vary slightly in appearance, depending on your system.

"Example: IGYTSALE transaction errors"
"Example: IGYTSALE sales analysis by product by area" on page 779
"Example: IGYTSALE sales and commissions" on page 780
"Example: IGYTSALE response time from sale to ship" on page 781

Example: IGYTSALE transaction errors

The following sample of IGYTSALE output shows transaction errors in the last column.

```
Day of Report: Monday             C O B O L   S P O R T S        11/24/2003    03:12    Page:   1
                                                Invalid Edited Transactions
  Sales  Inv.  Sales           Sales        Sales  Cust. Product And Quantity Sold       Ship
  Order  Items Time Stamp      Area         Pers   Code                                  Date Stamp
  -----  ----- --------------  -----------  -----  ----- --------------------------       ------------
             4 19990324591515  SAN DIEGO      116     15 60200132200110522045100                   Error Descriptions
                                                                                                 -Sales order number is missing
                                                                                                 -Date of sale time stamp is invalid
                                                                                                 -Salesperson number not numeric
                                                                                                 -Product code not in product-table
                                                                                                 -Date of ship time stamp is invalid
  B11117     3 19901201070833  SAN Diego     1165     66 33020o132200120522041100        19901203 Error Descriptions
                                                                                                 -Sales area not in area-table
                                                                                                 -Salesperson not in sales-per-table
                                                                                                 -Customer code not in customer-table
                                                                                                 -Product code not in product-table
                                                                                                 -Quantity sold not numeric
  B11123     9 19901110123314  SAN DIEGO     1166     09 260200270500110522250100140010  19901114 Error Descriptions
                                                                                                 -Invoiced items is invalid
                                                                                                 -Product and quantity not checked
                                                                                                 -Date of ship time stamp is invalid
  B11124     2 19901313510000  SAN DIEGO     1166     11 1 0200042200120a22141100                  Error Descriptions
                                                                                                 -Date of sale time stamp is invalid
                                                                                                 -Product code is invalid
                                                                                                 -Date of ship time stamp is invalid
     133          81119110000  LOS ANGELES   1166     10 040112110210160321251104                  Error Descriptions
                                                                                                 -Sales order number is invalid
                                                                                                 -Invoiced items is invalid
                                                                                                 -Date of sale time stamp is invalid
                                                                                                 -Product and quantity not checked
                                                                                                 -Date of ship time stamp is invalid
  C11133     4 1990111944                    1166     10 040112110210160321251104                  Error Descriptions
                                                                                                 -Date of sale time stamp is invalid
                                                                                                 -Sales area is missing
                                                                                                 -Date of ship time stamp is invalid
  C11138     4 19901117091530  LOS ANGELES   1155        113200102010260321250004        19901119 Error Descriptions
                                                                                                 -Customer code is invalid
  D00009     9 19901201222222  CNTRL COAST   115      19 141  1131221                    19901202 Error Descriptions
                                                                                                 -Invoiced items is invalid
```

Example: IGYTSALE sales analysis by product by area

The following sample of IGYTSALE output shows sales by product and area.

```
Day of Report: Monday             C O B O L   S P O R T S        11/24/2003    03:12    Page:   1
                                             Sales Analysis By Product By Area
                                                      Areas of Sale
  |                    | CNTRL COAST  | CNTRL VALLEY | LOS ANGELES  | NORTH COAST  | SAN DIEGO    | SF BAY AREA  |                  |
  |   Product   Codes  |              |              |              |              |              |              | Product Totals   |
  ==================================================================================================================================
  |Product Number  04  |              |              |              |              |              |              |                  |
  |Basketballs         |              |              |              |              |              |              |                  |
  |   Units Sold       |              |              |          433 |              |         2604 |         5102 |             8139 |
  |   Unit Price       |              |              |        22.20 |              |        22.20 |        22.20 |                  |
  |   Amount of Sale   |              |              |    $9,612.60 |              |   $57,808.80 |  $113,264.40 |      $180,685.80 |
  ----------------------------------------------------------------------------------------------------------------------------------
  |Product Number  05  |              |              |              |              |              |              |                  |
  |Basketball Rim/Board|              |              |              |              |              |              |                  |
  |   Units Sold       |              |         9900 |         2120 |           11 |         2700 |              |            14731 |
  |   Unit Price       |              |        88.30 |        88.30 |        88.30 |        88.30 |              |                  |
  |   Amount of Sale   |              |  $874,170.00 |  $187,196.00 |      $971.30 |  $238,410.00 |              |    $1,300,747.30 |
  ----------------------------------------------------------------------------------------------------------------------------------
  |Product Number  06  |              |              |              |              |              |              |                  |
  |Basketball Uniform  |              |              |              |              |              |              |                  |
  |   Units Sold       |              |              |              |          990 |          200 |          200 |             1390 |
  |   Unit Price       |              |              |              |        42.20 |        42.20 |        42.20 |                  |
  |   Amount of Sale   |              |              |              |   $41,778.00 |    $8,440.00 |    $8,440.00 |       $58,658.00 |
  ----------------------------------------------------------------------------------------------------------------------------------
  |Product Number  10  |              |              |              |              |              |              |                  |
  |Baseball Cage       |              |              |              |              |              |              |                  |
  |   Units Sold       |           45 |              |         3450 |           16 |          200 |         3320 |             7031 |
  |   Unit Price       |       890.00 |              |       890.00 |       890.00 |       890.00 |       890.00 |                  |
  |   Amount of Sale   |   $40,050.00 |              |$3,070,500.00 |   $14,240.00 |  $178,000.00 |$2,954,800.00 |    $6,257,590.00 |
  ----------------------------------------------------------------------------------------------------------------------------------
  |Product Number  11  |              |              |              |              |              |              |                  |
  |Baseball Uniform    |              |              |              |              |              |              |                  |
  |   Units Sold       |        10003 |              |         3578 |              |         2922 |         2746 |            19249 |
  |   Unit Price       |        45.70 |              |        45.70 |              |        45.70 |        45.70 |                  |
  |   Amount of Sale   |  $457,137.10 |              |  $163,514.60 |              |  $133,535.40 |  $125,492.20 |      $879,679.30 |
  ----------------------------------------------------------------------------------------------------------------------------------
  |Product Number  12  |              |              |              |              |              |              |                  |
  |Softballs           |              |              |              |              |              |              |                  |
  |   Units Sold       |           10 |          137 |         2564 |           13 |         2200 |           22 |             4946 |
  |   Unit Price       |         1.40 |         1.40 |         1.40 |         1.40 |         1.40 |         1.40 |                  |
  |   Amount of Sale   |       $14.00 |      $191.80 |    $3,589.60 |       $18.20 |    $3,080.00 |       $30.80 |        $6,924.40 |
  ----------------------------------------------------------------------------------------------------------------------------------
  |Product Number  13  |              |              |              |              |              |              |                  |
  |Softball Bats       |              |              |              |              |              |              |                  |
  |   Units Sold       |         3227 |              |         3300 |         1998 |         5444 |           99 |            14068 |
  |   Unit Price       |        12.60 |              |        12.60 |        12.60 |        12.60 |        12.60 |                  |
  |   Amount of Sale   |   $40,660.20 |              |   $41,580.00 |   $25,174.80 |   $68,594.40 |    $1,247.40 |      $177,256.80 |
  ----------------------------------------------------------------------------------------------------------------------------------
  |Product Number  14  |              |              |              |              |              |              |                  |
  |Softball Gloves     |              |              |              |              |              |              |                  |
  |   Units Sold       |         1155 |              |          136 |         3119 |         3833 |         5152 |            13395 |
  |   Unit Price       |        12.00 |              |        12.00 |        12.00 |        12.00 |        12.00 |                  |
  |   Amount of Sale   |   $13,860.00 |              |    $1,632.00 |   $37,428.00 |   $45,996.00 |   $61,824.00 |      $160,740.00 |
  ----------------------------------------------------------------------------------------------------------------------------------
  |Product Number  15  |              |              |              |              |              |              |                  |
  |Softball Cage       |              |              |              |              |              |              |                  |
  |   Units Sold       |          997 |           99 |         2000 |              |         2400 |              |             5496 |
  |   Unit Price       |       890.00 |       890.00 |       890.00 |              |       890.00 |              |                  |
  |   Amount of Sale   |  $887,330.00 |   $88,110.00 |$1,780,000.00 |              |$2,136,000.00 |              |    $4,891,440.00 |
  ----------------------------------------------------------------------------------------------------------------------------------
  |Product Number  16  |              |              |              |              |              |              |                  |
  |Softball Uniform    |              |              |              |              |              |              |                  |
  |   Units Sold       |           44 |              |          465 |           16 |         6165 |          200 |             6890 |
  |   Unit Price       |        45.70 |              |        45.70 |        45.70 |        45.70 |        45.70 |                  |
  |   Amount of Sale   |    $2,010.80 |              |   $21,250.50 |      $731.20 |  $281,740.50 |    $9,140.00 |      $314,873.00 |
  ----------------------------------------------------------------------------------------------------------------------------------
  |Product Number  25  |              |              |              |              |              |              |                  |
  |RacketBalls         |              |              |              |              |              |              |                  |
  |   Units Sold       |         1001 |        10003 |         1108 |         8989 |          200 |          522 |            21823 |
  |   Unit Price       |         0.60 |         0.60 |         0.60 |         0.60 |         0.60 |         0.60 |                  |
  |   Amount of Sale   |      $600.60 |    $6,001.80 |      $664.80 |    $5,393.40 |      $120.00 |      $313.20 |       $13,093.80 |
  ----------------------------------------------------------------------------------------------------------------------------------
  |Product Number  26  |              |              |              |              |              |              |                  |
  |Racketball Rackets  |              |              |              |              |              |              |                  |
  |   Units Sold       |           21 |              |          862 |          194 |          944 |           31 |             2052 |
  |   Unit Price       |        12.70 |              |        12.70 |        12.70 |        12.70 |        12.70 |                  |
  |   Amount of Sale   |      $266.70 |              |   $10,947.40 |    $2,463.80 |   $11,988.80 |      $393.70 |       $26,060.40 |
  ----------------------------------------------------------------------------------------------------------------------------------
  ==================================================================================================================================
  |  Total Units Sold  |        16503 |        20139 |        20016 |        15346 |        29812 |        17394 *           119210 *
  |  Total Sales       |$1,441,929.40 |  $968,473.60 |$5,290,487.50 |  $128,198.70 |$3,163,713.90 |$3,274,945.70 *   $14,267,748.80 *
```

Example: IGYTSALE sales and commissions

The following sample of IGYTSALE output shows sales performance and commissions by salesperson.

```
Day of Report: Monday             C O B O L   S P O R T S        11/24/2003    03:12    Page:   1
                                                Sales and Commission Report
  Salesperson: Billy Jim Bob
  Customers:           Number of    Products      Total for       Discount     Discount      Commission
                        Orders      Ordered         Order         (if any)      Amount         Earned
  -------------------- ---------    --------    --------------    --------   -----------     -----------
  Sports Stop              3         10117         $6,161.40       2.25%         $138.63         $746.45
  The Sportsman            1            99        $88,110.00       5.06%       $4,458.36      $10,674.52
  Sports Play              1          9900       $874,170.00       7.59%      $66,349.50     $105,905.69
                       ---------    --------    --------------               -----------     -----------
  Totals:                  5         20116       $968,441.40                  $70,946.49     $117,326.66
  Salesperson: Willie Al Roz
  Customers:           Number of    Products      Total for       Discount     Discount      Commission
                        Orders      Ordered         Order         (if any)      Amount         Earned
  -------------------- ---------    --------    --------------    --------   -----------     -----------
  Winners Club             4         13998     $1,572,775.90       7.59%     $119,373.69     $157,277.59
  Winning Sports           1          3222        $48,777.20       3.38%       $1,648.66       $4,877.72
  The Sportsman            1          1747        $27,415.50       3.38%         $926.64       $2,741.55
  Play Outdoors            1          2510        $18,579.60       3.38%         $627.99       $1,857.96
                       ---------    --------    --------------               -----------     -----------
  Totals:                  7         21477     $1,667,548.20                 $122,576.98     $166,754.82
  Salesperson: Art Tung
  Customers:           Number of    Products      Total for       Discount     Discount      Commission
                        Orders      Ordered         Order         (if any)      Amount         Earned
  -------------------- ---------    --------    --------------    --------   -----------     -----------
  Sports Stop              1            23            $32.20       2.25%            $.72           $1.98
  Winners Club             2         16057     $2,274,885.00       7.59%     $172,663.77     $140,424.10
  Gear Up                  1          3022       $107,144.00       7.59%       $8,132.22       $6,613.78
  Sports Club              1            22           $279.40       2.25%           $6.28          $17.24
  Sports Fans Shop         1          1044        $20,447.30       3.38%         $691.11       $1,262.17
  L. A. Sports             1          1163       $979,198.10       7.59%      $74,321.13      $60,443.94
                       ---------    --------    --------------               -----------     -----------
  Totals:                  7         21331     $3,381,986.00                 $255,815.23     $208,763.21
  Salesperson: Chuck Morgan
  Customers:           Number of    Products      Total for       Discount     Discount      Commission
                        Orders      Ordered         Order         (if any)      Amount         Earned
  -------------------- ---------    --------    --------------    --------   -----------     -----------
  Sports Play              3          7422     $3,817,245.40       7.59%     $289,728.92     $322,270.94
  Sports 4 You             1          3022       $398,335.40       7.59%      $30,233.65      $33,629.46
  The Sportsman            1          3022       $285,229.40       7.59%      $21,648.91      $24,080.49
  Sports 4 Winners         1          1100        $68,509.40       5.06%       $3,466.57       $5,783.90
  Sports Club              1         12027     $1,324,256.10       7.59%     $100,511.03     $111,800.32
                       ---------    --------    --------------               -----------     -----------
  Totals:                  7         26593     $5,893,575.70                 $445,589.08     $497,565.11
  Salesperson: Chris Preston
  Customers:           Number of    Products      Total for       Discount     Discount      Commission
                        Orders      Ordered         Order         (if any)      Amount         Earned
  -------------------- ---------    --------    --------------    --------   -----------     -----------
  Playing Ball             1          5535     $1,939,219.10       7.59%     $147,186.72     $103,509.69
  Play Sports              1          5675       $225,130.80       7.59%      $17,087.42      $12,016.80
  Winners Club             1           631        $14,069.70       2.25%         $316.56         $750.99
  The Jock Shop            1          2332        $28,716.60       3.38%         $970.62       $1,532.80
                       ---------    --------    --------------               -----------     -----------
  Totals:                  4         14173     $2,207,136.20                 $165,561.32     $117,810.28
  Salesperson: Edyth Phillips
  Customers:           Number of    Products      Total for       Discount     Discount      Commission
                        Orders      Ordered         Order         (if any)      Amount         Earned
  -------------------- ---------    --------    --------------    --------   -----------     -----------
  Sports Play              2          3575        $92,409.90       5.06%       $4,675.94       $3,911.43
  Winning Sports           1         11945        $56,651.40       5.06%       $2,866.56       $2,397.88
                       ---------    --------    --------------               -----------     -----------
  Totals:                  3         15520       $149,061.30                   $7,542.50       $6,309.31
    Grand Totals:           33        119210    $14,267,748.80               $1,068,031.60   $1,114,529.39
```

Example: IGYTSALE response time from sale to ship

The following sample of IGYTSALE output shows response time between the sale date in the United States and the date the sold products are shipped to Europe.

```
Day of Report: Monday   COBOL SPORTS       11/24/2003    03:12    Page:   1
                     Response Time from USA Sale to European Ship
  Prod    Units    Sale Date/Time(PST)      Ship Date     Ship  Response Time
  Code    Sold     YYYYMMDD  HHMMSS         YYYYMMDD      Day    Days
  ----    -----    --------  ------         --------      ----  -------------
  25      9999     19900226  010101         19900228      WED        .95
  15        99     19900310  100530         19900403      TUE      23.57
  05      9900     19900418  222409         19900419      THU        .06
  25         4     19900523  151010         19900623      SAT      30.36
  04      1100     19901110  003301         19901114      WED       2.97
  12        23     19901114  003205         19901117      SAT       1.97
  14      5111     19900118  101527         19900120      SAT       1.57
  04      5102     19901201  132013         19901203      MON       1.44
  04       300     19901221  191544         19901223      SUN       1.19
  05       500     19901210  211544         19901214      FRI       3.11
  04       100     19901211  000816         19901213      THU        .99
  25       100     19901201  131544         19901203      MON       1.44
  25       100     19901112  073312         19901113      TUE        .68
  14      1111     19901214  012510         19901216      SUN        .94
  26        22     19901110  000034         19901113      TUE       1.99
  12      2000     19901110  154100         19901113      TUE       2.34
  04      1104     19901110  175001         19901113      TUE       2.25
  12       114     19901229  115522         19901230      SUN        .50
  15      2000     19901110  190113         19901114      WED       3.20
  10      1440     19901112  001500         19901115      THU       1.98
  25      1104     19901118  120101         19901119      MON        .49
  25         4     19901118  110030         19901119      MON        .54
  12       144     19901114  010510         19901119      MON       3.95
  14       112     19901119  010101         19901122      THU       1.95
  26       321     19901117  173945         19901119      MON       1.26
  13      1221     19901101  135133         19901102      FRI        .42
  10        22     19901029  210000         19901030      TUE        .12
  14        35     19901130  160500         19901201      SAT        .32
  11      9005     19901211  050505         19901212      WED        .78
  06       990     19900511  214409         19900515      TUE       3.09
  13      1998     19900712  150100         19900716      MON       3.37
  26        31     19901010  185559         19901011      THU        .21
  14        30     19901210  195500         19901212      WED       1.17
```

Preparing to run IGYTSALE

All files required by the IGYTSALE program (IGYTSALE, IGYTCRC, IGYTPRC, IGYTSRC, IGYTABLE, and IGYTRANA) are on the product installation tape in the IGY.V3R4M0.SIGYSAMP data set.

You can change data set names and procedure-names at installation time. Check with your system programmer to verify these names.

Do not change these options on the `CBL` card in the source file for IGYTSALE:

- `LIB`
- `NONUMBER`
- `SEQUENCE`
- `NONUMBER`
- `QUOTE`

With these options in effect, the program might not cause any diagnostic messages to be issued. You can use the sequence number string in the source file to search for the language elements used.

When you run IGYTSALE, the following messages are printed to the SYSOUT data set:

```
Program IGYTSALE Begins
There were 00041 records processed in this program
Program IGYTSALE Normal End
```

RELATED CONCEPTS
"IGYTSALE: nested program application" on page 774

RELATED TASKS
"Running IGYTSALE"

RELATED REFERENCES
"Input data for IGYTSALE" on page 775
"Reports produced by IGYTSALE" on page 777
"Language elements and concepts that are illustrated"

Running IGYTSALE

Use the following JCL to compile, link-edit, and run the IGYTSALE program. If you want only to compile or only to compile and link-edit the program, change the IGYWCLG cataloged procedure.

Insert the information for your system or installation in the fields that are shown in lowercase letters (accounting information).

```
//IGYTSALE JOB (acct-info),'IGYTSALE',MSGLEVEL=(1,1),TIME=(0,29)
//TEST  EXEC IGYWCLG
//COBOL.SYSLIB  DD DSN=IGY.V3R4M0.SIGYSAMP,DISP=SHR
//COBOL.SYSIN   DD DSN=IGY.V3R4M0.SIGYSAMP(IGYTSALE),DISP=SHR
//GO.SYSOUT     DD SYSOUT=A
//GO.IGYTABLE   DD DSN=IGY.V3R4M0.SIGYSAMP(IGYTABLE),DISP=SHR
//GO.IGYTRANS   DD DSN=IGY.V3R4M0.SIGYSAMP(IGYTRANA),DISP=SHR
//GO.IGYPRINT   DD SYSOUT=A,DCB=BLKSIZE=133
//GO.IGYPRT2    DD SYSOUT=A,DCB=BLKSIZE=133
//
```

Language elements and concepts that are illustrated

The sample programs illustrate several COBOL language elements and concepts.

To find the applicable language element for a sample program, locate the abbreviation for that program in the sequence string:

Sample program	Abbreviation
IGYTCARA	IA
IGYTCARB	IB
IGYTSALE	IS

The following table lists the language elements and programming concepts that the sample programs illustrate. The language element or concept is described, and the sequence string is shown. The sequence string is the special character string that appears in the sequence field of the source file. You can use this string as a search argument for locating the elements in the listing.

Language element or concept	Sequence string
`ACCEPT . . . FROM DAY-OF-WEEK`	IS0900

Language element or concept	Sequence string
ACCEPT . . . FROM DATE	IS0901
ACCEPT . . . FROM TIME	IS0902
ADD . . . TO	IS4550
AFTER ADVANCING	IS2700
AFTER PAGE	IS2600
ALL	IS4200
ASSIGN	IS1101
AUTHOR	IA0040
CALL	IS0800
Callable services (Language Environment): 1. CEEDATM: format date or time output 2. CEEDCOD: feedback code check 3. CEEGMTO: UTC offset from local time 4. CEELOCT: local date and time 5. CEESECS: convert timestamp to seconds	 1. IS0875, IS2575 2. IS0905 3. IS0904 4. IS0850 5. IS2350, IS2550
CLOSE files	IS1900
Comma, semicolon, and space interchangeable	IS3500, IS3600
COMMON statement for nested programs	IS4600
Complex OCCURS DEPENDING ON	IS0700, IS3700
COMPUTE	IS4501
COMPUTE ROUNDED	IS4500
CONFIGURATION SECTION	IA0970
CONFIGURATION SECTION (optional)	IS0200
CONTINUE statement	IA5310, IA5380
COPY statement	IS0500
DATA DIVISION (optional)	IS5100
Data validation	IA5130-6190
Do-until (PERFORM . . . TEST AFTER)	IA4900-5010, IA7690-7770
Do-while (PERFORM . . . TEST BEFORE)	IS1660
END-ADD	IS2900
END-COMPUTE	IS4510
END-EVALUATE	IA6590, IS2450
END-IF	IS1680
END-MULTIPLY	IS3100
END-PERFORM	IS1700
END PROGRAM	IA9990
END-READ	IS1800
END-SEARCH	IS3400
ENVIRONMENT DIVISION (optional)	IS0200
Error handling, termination of program	IA4620, IA5080, IA7800-7980
EVALUATE statement	IA6270-6590

Language element or concept	Sequence string
EVALUATE . . . ALSO	IS2400
EXIT PROGRAM not only statement in paragraph	IS2000
Exponentiation	IS4500
EXTERNAL clause	IS1200
FILE-CONTROL entry for sequential file	IA1190-1300
FILE-CONTROL entry for VSAM indexed file	IA1070-1180
FILE SECTION (optional)	IS0200
FILE STATUS code check	IA4600-4630, IA4760-4790
FILLER (optional)	IS0400
Flags, level-88, definition	IA1730-1800, IA2440-2480, IA2710
Flags, level-88, testing	IA4430, IA5200-5250
FLOATING POINT	IS4400
GLOBAL statement	IS0300
INITIAL statement for nested programs	IS2300
INITIALIZE	IS2500
Initializing a table in the DATA DIVISION	IA2920-4260
Inline PERFORM statement	IA4410-4520
I-O-CONTROL paragraphs (optional)	IS0200
INPUT-OUTPUT SECTION (optional)	IS0200
Intrinsic functions: 1. CURRENT-DATE 2. MAX 3. MEAN 4. MEDIAN 5. MIN 6. STANDARD-DEVIATION 7. UPPER-CASE 8. VARIANCE 9. WHEN-COMPILED	 1. IA9005 2. IA9235 3. IA9215 4. IA9220 5. IA9240 6. IA9230 7. IA9015 8. IA9225 9. IA9000
IS (optional in all clauses)	IS0700
LABEL RECORDS (optional)	IS1150
LINKAGE SECTION	IS4900
Mixing of indexes and subscripts	IS3500
Mnemonic names	IA1000
MOVE	IS0903
MOVE CORRESPONDING statement	IA4810, IA4830
MULTIPLY . . . GIVING	IS3000
Nested IF statement, using END-IF	IA5460-5830
Nested program	IS1000
NEXT SENTENCE	IS4300
NOT AT END	IS1600
NULL	IS4800

Language element or concept	Sequence string
OBJECT-COMPUTER (optional)	IS0200
OCCURS DEPENDING ON	IS0710
ODO uses maximum length for receiving item	IS1550
OPEN EXTEND	IB2210
OPEN INPUT	IS1400
OPEN OUTPUT	IS1500
ORGANIZATION (optional)	IS1100
Page eject	IA7180-7210
Parenthesis in abbreviated conditions	IS4850
PERFORM . . . WITH TEST AFTER (Do-until)	IA4900-5010, IA7690-7770
PERFORM . . . WITH TEST BEFORE (Do-while)	IS1660
PERFORM . . . UNTIL	IS5000
PERFORM . . . VARYING statement	IA7690-7770
POINTER function	IS4700
Print file FD entry	IA1570-1620
Print report	IA7100-7360
PROCEDURE DIVISION . . . USING	IB1320-IB1650
PROGRAM-ID (30 characters allowed)	IS0120
READ . . . INTO . . . AT END	IS1550
REDEFINES statement	IA1940, IA2060, IA2890, IA3320
Reference modification	IS2425
Relational operator <= (less than or equal)	IS4400
Relational operator >= (greater than or equal)	IS2425
Relative subscripting	IS4000
REPLACE	IS4100
SEARCH statement	IS3300
SELECT	IS1100
Sequence number can contain any character	IA, IB, IS
Sequential file processing	IA4480-4510, IA4840-4870
Sequential table search, using PERFORM	IA7690-7770
Sequential table search, using SEARCH	IA5270-5320, IA5340-5390
SET INDEX	IS3200
SET . . . TO TRUE statement	IA4390, IA4500, IA4860, IA4980
SOURCE-COMPUTER (optional)	IS0200
SPECIAL-NAMES paragraph (optional)	IS0200
STRING statement	IA6950, IA7050
Support for lowercase letters	IS0100
TALLY	IS1650
TITLE statement for nested programs	IS0100
Update commuter record	IA6200-6610
Update transaction work value spaces	IB0790-IB1000

Language element or concept	Sequence string
`USAGE BINARY`	IS1300
`USAGE PACKED-DECIMAL`	IS1301
Validate elements	IB0810, IB0860, IB1000
`VALUE` with `OCCURS`	IS0600
`VALUE SPACE (S)`	IS0601
`VALUE ZERO (S) (ES)`	IS0600
Variable-length table control variable	IA5100
Variable-length table definition	IA2090-2210
Variable-length table loading	IA4840-4990
`VSAM` indexed file key definition	IA1170
`VSAM` return-code display	IA7800-7900
`WORKING-STORAGE SECTION`	IS0250

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. 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 IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described 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:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

If you are viewing this information in softcopy, the photographs and color illustrations may not appear.

Trademarks

Glossary

The terms in this glossary are defined in accordance with their meaning in COBOL. These terms might or might not have the same meaning in other languages.

This glossary includes terms and definitions from the following publications:

- *ANSI INCITS 23-1985, Programming languages - COBOL*, as amended by *ANSI INCITS 23a-1989, Programming Languages - COBOL - Intrinsic Function Module for COBOL*, and *ANSI INCITS 23b-1993, Programming Languages - Correction Amendment for COBOL*
- *ANSI X3.172-2002, American National Standard Dictionary for Information Systems*

American National Standard definitions are preceded by an asterisk (*).

This glossary includes definitions developed by Sun Microsystems, Inc. for their Java and J2EE glossaries. When Sun is the source of a definition, that is indicated.

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 a program.

above the 16-MB line. Storage above the so-called 16-MB line (or boundary) but below the 2-GB bar. This storage is addressable only in 31-bit mode. Before IBM introduced the MVS/XA$^{(TM)}$ architecture in the 1980s, the virtual storage for a program was limited to 16 MB. Programs that have been compiled with a 24-bit mode can address only 16 MB of space, as though they were kept under an imaginary storage line. Since VS COBOL II, a program that has been compiled with a 31-bit mode can be above the 16-MB line.

* **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 or collating sequence or both.

* **alphabetic character.** A letter or a space character.

alphabetic data item. A data item that is described with a `PICTURE` character string that contains only the symbol `A`. An alphabetic data item has `USAGE DISPLAY`.

* **alphanumeric character.** Any character in the single-byte character set of the computer.

alphanumeric data item. A general reference to a data item that is described implicitly or explicitly as `USAGE DISPLAY`, and that has category alphanumeric, alphanumeric-edited, or numeric-edited.

alphanumeric-edited data item. A data item that is described by a `PICTURE` character string that contains at least one instance of the symbol `A` or `X` and at least one of the simple insertion symbols `B`, `0`, or `/`. An alphanumeric-edited data item has `USAGE DISPLAY`.

* **alphanumeric function.** A function whose value is composed of a string of one or more characters from the alphanumeric character set of the computer.

alphanumeric group item. A group item that is defined without a `GROUP-USAGE NATIONAL` clause. For operations such as `INSPECT`, `STRING`, and `UNSTRING`, an alphanumeric group item is processed as though all its content were described as `USAGE DISPLAY` regardless of the actual content of the group. For operations that require processing of the elementary items within a group, such as `MOVE CORRESPONDING`, `ADD CORRESPONDING`, or `INITIALIZE`, an alphanumeric group item is processed using group semantics.

alphanumeric literal. A literal that has an opening delimiter from the following set: `'`, `"`, `X'`, `X"`, `Z'`, or `Z"`. The string of characters can include any character in the character set of the computer.

* **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 that consists of producers, consumers, and general-interest groups and establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

argument. (1) An identifier, a literal, an arithmetic expression, or a function-identifier that specifies a value to be used in the evaluation of a function. (2) An operand of the USING phrase of a CALL or INVOKE statement, used for passing values to a called program or an invoked method.

* **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 ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT.

array. An aggregate that consists of data objects, each of which can be uniquely referenced by subscripting. An array is 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 uses a coded character set that is based on 7-bit coded characters (8 bits including parity check). The standard is used for information interchange between data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

IBM has defined an extension to ASCII (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 but no physical representation.

* **AT END condition.** A condition that is caused during the execution of a READ, RETURN, or SEARCH statement under certain conditions:

- A READ statement runs on 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.
- A RETURN statement runs when no next logical record exists for the associated sort or merge file.
- A SEARCH statement runs when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

B

basic document encoding. For an XML document, one of the following encoding categories that the XML parser determines by examining the first few bytes of the document:

- ASCII
- EBCDIC
- Unicode UTF-16, either big-endian or little-endian
- Other unsupported encoding
- No recognizable encoding

big-endian. The default format that the mainframe and the AIX workstation use to store binary data and UTF-16 characters. In this format, the least significant byte of a binary data item is at the highest address and the least significant byte of a UTF-16 character is at the highest address. Compare with *little-endian*.

binary item. A numeric data item that is represented in binary notation (on the base 2 numbering system). The decimal equivalent consists 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 can 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 records that are either contained within the block or that overlap the block. Synonymous with *physical record*.

breakpoint. A place in a computer program, usually specified by an instruction, where external intervention or a monitor program can interrupt the program as it runs.

buffer. A portion of storage that is used to hold input or output data temporarily.

built-in function. See *intrinsic function*.

business method. A method of an enterprise bean that implements the business logic or rules of an application. (Sun)

byte. A string that consists of a certain number of bits, usually eight, treated as a unit, and representing a character or a control function.

byte order mark (BOM). A Unicode character that can be used at the start of UTF-16 or UTF-32 text to indicate the byte order of subsequent text; the byte order can be either big-endian or little-endian.

bytecode. Machine-independent code that is generated by the Java compiler and executed by the Java interpreter. (Sun)

C

callable services. In Language Environment, a set of services that a COBOL program can invoke by using the conventional Language Environment-defined call interface. All programs that share the Language Environment conventions can use these services.

called program. A program that is the object of a `CALL` statement. At run time the called program and calling program are combined to produce a *run unit*.

* **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 choose between a number of resulting actions.

cataloged procedure. A set of job control statements that are placed in a partitioned data set called the procedure library (SYS1.PROCLIB). You can use cataloged procedures to save time and reduce errors in coding JCL.

CCSID. See *coded character set identifier*.

century window. A 100-year interval within which any two-digit year is unique. Several types of century window are available to COBOL programmers:
- For windowed date fields, you use the `YEARWINDOW` compiler option.
- For the windowing intrinsic functions `DATE-TO-YYYYMMDD`, `DAY-TO-YYYYDDD`, and `YEAR-TO-YYYY`, you specify the century window with *argument-2*.
- For Language Environment callable services, you specify the century window in CEESCEN.

* **character.** The basic indivisible unit of the language.

character encoding unit. A unit of data that corresponds to one code point in a coded character set. One or more character encoding units are used to represent a character in a coded character set. Also known as *encoding unit*.

For `USAGE NATIONAL`, a character encoding unit corresponds to one 2-byte code point of UTF-16.

For `USAGE DISPLAY`, a character encoding unit corresponds to a byte.

For `USAGE DISPLAY-1`, a character encoding unit corresponds to a 2-byte code point in the DBCS character set.

character position. The amount of physical storage or presentation space required to hold or present one character. The term applies to any class of character. For specific classes of characters, the following terms apply:
- *Alphanumeric character position*, for characters represented in `USAGE DISPLAY`
- *DBCS character position*, for DBCS characters represented in `USAGE DISPLAY-1`
- *National character position*, for characters represented in `USAGE NATIONAL`; synonymous with *character encoding unit* for UTF-16

character set. A collection of elements that are used to represent textual information, but for which no coded representation is assumed. See also *coded character set*.

character string. A sequence of contiguous characters that form a COBOL word, a literal, a `PICTURE` character string, or a comment-entry. A character string 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 restarted later.

* **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. Classes can be defined hierarchically, allowing one class to inherit from another.

* **class condition.** The proposition (for which a truth value can be determined) that the content of an item is wholly alphabetic, is wholly numeric, is wholly DBCS, is wholly Kanji, or consists exclusively of the characters that are listed in the definition of a class-name.

* **class definition.** The COBOL source unit that defines a class.

class hierarchy. A tree-like structure that shows relationships among object classes. It places one class at the top and one or more layers of classes below it. Synonymous with *inheritance hierarchy*.

* **class identification entry.** An entry in the CLASS-ID paragraph of the IDENTIFICATION DIVISION; this entry contains clauses that specify the class-name and assign selected attributes to the class definition.

class-name (object-oriented). The name of an object-oriented COBOL class definition.

* **class-name (of data).** A user-defined word that is defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION; this word assigns a name to the proposition (for which a truth value can be defined) that the content of a data item consists exclusively of the characters that are listed in the definition of the class-name.

class object. The runtime object that represents a class.

* **clause.** An ordered set of consecutive COBOL character strings whose purpose is to specify an attribute of an entry.

client. In object-oriented programming, a program or method that requests services from one or more methods in a class.

* **COBOL character set.** The set of characters used in writing COBOL syntax. 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
	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, one code page could assign characters and meanings to 256 code points for 8-bit code, and another code page could assign characters and meanings to 128 code points for 7-bit code. For example, one of the IBM code pages for English on the workstation is IBM-1252 and on the host is IBM-1047. A *coded character set*.

code point. A unique bit pattern that is defined in a coded character set (code page). Graphic symbols and control characters are assigned to code points.

coded character set. A set of unambiguous rules that establish a character set and the relationship between the characters of the set and their coded representation. Examples of coded character sets are the character sets as represented by ASCII or EBCDIC code pages or by the UTF-16 encoding scheme for Unicode.

coded character set identifier (CCSID). An IBM-defined number in the range 1 to 65,535 that identifies a specific code page.

* **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 byte position within a print line or within a reference format line. The columns are numbered from 1, by 1, starting at the leftmost position of the line and extending to the rightmost position of the line. A column holds one single-byte character.

* **combined condition.** A condition that is the result of connecting two or more conditions with the AND or the OR logical operator. See also *condition* and *negated combined condition*.

* **comment-entry.** An entry in the IDENTIFICATION DIVISION that can be any combination of characters from the character set of the computer.

* **comment line.** A source program line represented by an asterisk (*) in the indicator area of the line and any characters from the character set of the computer in area A and area B of that line. The comment line serves only for documentation. A special form of comment line represented by a slant (/) in the indicator area of the line and any characters from the character set of the computer in area A and area B of that line causes page ejection before printing the comment.

* **common program.** A program that, despite being directly contained within another program, can be called from any program directly or indirectly contained in that other program.

compatible date field. 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, and the other has YYXX.
- One has DATE FORMAT YYYYXXXX, and the other has YYYYXX.

A windowed date field can be subordinate to a data item that is an expanded date group. The two date fields are compatible if the subordinate date field has USAGE DISPLAY, starts two bytes after the start of the group expanded date field, and the two fields meet at least one of the following conditions:

- The subordinate date field has a DATE FORMAT pattern with the same number of Xs as the DATE FORMAT pattern of the group date field.
- The subordinate date field has DATE FORMAT YY.
- The group date field has DATE FORMAT YYYYXXXX and the subordinate date field has DATE FORMAT YYXX.

- PROCEDURE DIVISION: Two date fields are compatible if they have the same date format except for the year part, which can 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 COBOL source code is translated, by a COBOL compiler, to a COBOL object program.

compiler. A program that translates source code written in a higher-level language into machine-language object code.

compiler-directing statement. A statement that causes the compiler to take a specific action during compilation. The standard compiler-directing statements are COPY, REPLACE, and USE.

*** complex condition.** A condition in which one or more logical operators act upon one or more conditions. See also *condition*, *negated simple condition*, and *negated combined condition*.

complex ODO. Certain forms of the OCCURS DEPENDING ON clause:

- Variably located item or group: A data item described by an OCCURS clause with the DEPENDING ON option is followed by a nonsubordinate data item or group. The group can be an alphanumeric group or a national group.
- Variably located table: A data item described by an OCCURS clause with the DEPENDING ON option is followed by a nonsubordinate data item described by an OCCURS clause.
- Table with variable-length elements: A data item described by an OCCURS clause contains a subordinate data item described by an OCCURS clause with the DEPENDING ON option.
- Index name for a table with variable-length elements.
- Element of a table with variable-length elements.

component. (1) A functional grouping of related files. (2) In object-oriented programming, a reusable object or program that performs a specific function and is designed to work with other components and applications. JavaBeans(TM) is Sun Microsystems, Inc.'s architecture for creating components.

*** computer-name.** A system-name that identifies the computer where 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 or the operating system and result 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. When used in these language specifications in or in reference to 'condition' (*condition-1*, *condition-2*,. . .) of a general format, the term refers to a conditional expression that consists 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. See also *simple condition*, *complex condition*, *negated simple condition*, *combined condition*, and *negated combined condition*.

*** 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 phrase that specifies the action to be taken upon determination of the truth value of a condition that results from the execution of a conditional statement.

*** conditional statement.** A statement that specifies that the truth value of a condition is to be determined and that the subsequent action of the object program depends 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 can assume; or a user-defined word assigned to a status of an implementor-defined switch or device.

* **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.

contained program. A COBOL program that is nested within another COBOL program.

* **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 that contains 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. Synonymous with *copy file*.

* **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 value. A character string that identifies the monetary units stored in a numeric-edited item. Typical examples are $, USD, and EUR. A currency-sign value can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign ($) is used as the default currency-sign value. See also *currency symbol*.

currency symbol. A character used in a PICTURE clause to indicate the position of a currency sign value in a numeric-edited item. A currency symbol can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign ($) is used as the default currency sign value and currency symbol. Multiple currency symbols and currency sign values can be defined. See also *currency sign value*.

* **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. The division of a COBOL program or method that describes the data to be processed by the program or method: the 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.

* **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 of the ACCEPT statement.
- The result of certain arithmetic operations. For details, see Arithmetic with date fields (*Enterprise COBOL Language Reference*).

The term *date field* refers to both *expanded date field* and *windowed date field*. See also *nondate*.

date format. The date pattern of a date field, specified in either of the following ways:
- Explicitly, by the DATE FORMAT clause or DATEVAL intrinsic function argument-2
- Implicitly, by statements and intrinsic functions that return date fields. For details, see Date field (*Enterprise COBOL Language Reference*).

DBCS. See *double-byte character set (DBCS)*.

DBCS character. Any character defined in IBM's double-byte character set.

DBCS character position. See *character position*.

DBCS data item. A data item that is described by a PICTURE character string that contains at least one symbol G, or, when the NSYMBOL(DBCS) compiler option is in effect, at least one symbol N. A DBCS data item has USAGE DISPLAY-1.

* **debugging line.** 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 that consists 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 DECLARATIVE 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 the unedited numeric value of the item.

* **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.

dependent region. In IMS, the MVS virtual storage region that contains message-driven programs, batch programs, or online utilities.

* **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 to refer to any other symbol.

* **digit position.** The amount of physical storage required to store a single digit. This amount can 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.

display floating-point data item. A data item that is described implicitly or explicitly as USAGE DISPLAY and that has a PICTURE character string that describes an external floating-point data item.

* **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 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 (DLL)*.

DLL application. An application that references imported programs, functions, or variables.

DLL linkage. A CALL in a program that has been compiled with the DLL and NODYNAM options; the CALL resolves to an exported name in a separate module, or to an INVOKE of a method that is defined in a separate module.

do construct. In structured programming, a DO statement is used to group a number of statements in a procedure. In COBOL, an inline 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.

document type definition (DTD). The grammar for a class of XML documents. See *XML type definition*.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 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 2 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 CALL. A `CALL` *literal* statement in a program that has been compiled with the `DYNAM` option and the `NODLL` option, or a `CALL` *identifier* statement in a program that has been compiled with the `NODLL` option.

dynamic link library (DLL). A file that contains executable code and data that are bound to a program at load time or run time, rather than during linking. Several applications can share the code and data in a DLL simultaneously. Although a DLL is not part of the executable file for a program, it can be required for an executable file to run properly.

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). A DSA is allocated upon invocation of a program or function and persists for the duration of the invocation instance. DSAs are generally allocated within stack segments managed by Language Environment.

*** EBCDIC (Extended Binary-Coded Decimal Interchange Code).** A coded character set based on 8-bit coded characters.

EBCDIC character. Any one of the symbols included in the EBCDIC (Extended Binary-Coded-Decimal Interchange Code) set.

edited data item. A data item that has been modified by suppressing zeros or inserting editing characters or both.

*** editing character.** A single character or a fixed two-character combination belonging to the following set:

Character	Meaning
	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)

EJB. See *Enterprise JavaBeans*.

EJB container. A container that implements the EJB component contract of the J2EE architecture. This contract specifies a runtime environment for enterprise beans that includes security, concurrency, life cycle management, transaction, deployment, and other services. An EJB container is provided by an EJB or J2EE server. (Sun)

EJB server. Software that provides services to an EJB container. An EJB server can host one or more EJB containers. (Sun)

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.

encapsulation. [In object-oriented programming, the technique that is used to hide the inherent details of an object. The object provides an interface that queries and manipulates the data without exposing its underlying structure. Synonymous with *information hiding*.

enclave. When running under Language Environment, an enclave is analogous to a run unit. An enclave can create other enclaves by a `LINK` and the use of the system() function of C.

encoding unit. See *character encoding unit*.

end class marker. A combination of words, followed by a separator period, that indicates the end of a COBOL class definition. The end class marker is:

```
END CLASS class-name.
```

end method marker. A combination of words, followed by a separator period, that indicates the end of a COBOL method definition. The end method marker 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 marker.** A combination of words, followed by a separator period, that indicates the end of a COBOL source program. The end program marker is:

`END PROGRAM` *`program-name`*`.`

enterprise bean. A component that implements a business task and resides in an EJB container. (Sun)

Enterprise JavaBeans. A component architecture defined by Sun Microsystems, Inc. for the development and deployment of object-oriented, distributed, enterprise-level applications.

* **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 where the source program is compiled and those where the object program is run. It 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, program switches or all of these. When an environment-name is associated with a mnemonic-name in the `ENVIRONMENT DIVISION`, the mnemonic-name can be substituted in any format in which such substitution is valid.

environment variable. Any of a number of variables that define some aspect of the computing environment, and are accessible to programs that operate in that environment. Environment variables can affect the behavior of programs that are sensitive to the environment in which they operate.

execution time. See *run time*.

execution-time environment. See *runtime environment*.

expanded date field. A date field containing an expanded (four-digit) year. See also *date field* and *expanded year*.

expanded year. A date field that consists only of a four-digit year. Its value includes the century: for example, 1998. Compare with *windowed year*.

* **explicit scope terminator.** A reserved word that terminates the scope of a particular `PROCEDURE DIVISION` statement.

exponent. A number that indicates the power to which another number (the base) is to be raised. Positive exponents denote multiplication; negative exponents denote division; and 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.

Extensible Markup Language. See *XML*.

extensions. COBOL syntax and semantics supported by IBM compilers in addition to those described in Standard COBOL 85.

external code page. For XML documents, the value specified by the `CODEPAGE` compiler option.

* **external data.** The data that is described in a program as external data items and external file connectors.

* **external data item.** A data item that is described as part of an external record in one or more programs of a run unit and that can be referenced from any program in which it is described.

* **external data record.** A logical record that is described in one or more programs of a run unit and whose constituent data items can be referenced from any program in which they are described.

external decimal data item. See *zoned decimal data item* and *national decimal data item*.

* **external file connector.** A file connector that is accessible to one or more object programs in the run unit.

external floating-point data item. See *display floating-point data item* and *national floating-point data item*.

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

factory data. Data that is allocated once for a class and shared by all instances of the class. Factory data is declared in the WORKING-STORAGE SECTION of the DATA DIVISION in the FACTORY paragraph of the class definition, and is equivalent to Java private static data.

factory method. A method that is supported by a class independently of an object instance. Factory methods are declared in the FACTORY paragraph of the class definition, and are equivalent to Java public static methods. They are typically used to customize the creation of objects.

* **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 that 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 entry.** A SELECT clause and all its subordinate clauses that declare the relevant physical attributes of a file.

FILE-CONTROL paragraph. A paragraph in the ENVIRONMENT DIVISION in which the data files for a given source unit are declared.

* **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 that conform to a specific set of data-record and file-description protocols, and a set of programs that manage these files.

* **fixed file attributes.** Information about a file that is established when a file is created and that 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 bytes.

fixed-point item. 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 can be either binary, packed decimal, or external decimal.

floating point. 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.

floating-point data item. A numeric data item that contains 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 that the exponent specifies.

* **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 can include a reference-modifier. A function-identifier that references

an alphanumeric function can be specified anywhere in the general formats that an identifier can be specified, subject to certain restrictions. A function-identifier that references an integer or numeric function can be referenced anywhere in the general formats that an arithmetic expression can be specified.

function-name. A word that names the mechanism whose invocation, along with required arguments, determines the value of a function.

function-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 FUNCTION-POINTER` clause contains the address of a function entry point. Typically used to communicate with C and Java programs.

G

garbage collection. The automatic freeing by the Java runtime system of the memory for objects that are no longer referenced.

* **global name.** A name that is declared in only one program but that can be referenced from the program and from any program contained within the program. Condition-names, data-names, file-names, record-names, report-names, and some special registers can be global names.

global reference. A reference to an object that is outside the scope of a method.

group item. (1) A data item that is composed of subordinate data items. See *alphanumeric group item* and *national group item*. (2) When not qualified explicitly or by context as a national group or an alphanumeric group, the term refers to groups in general.

grouping separator. A character used to separate units of digits in numbers for ease of reading. The default is the character comma.

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*.

hide. To redefine a factory or static method (inherited from a parent class) in a subclass.

hierarchical file system. A collection of files and directories that are organized in a hierarchical structure and can be accessed by using UNIX System Services.

* **high-order end.** The leftmost character of a string of characters.

hiperspace. In a z/OS environment, a range of up to 2 GB of contiguous virtual storage addresses that a program can use as a buffer.

I

IBM COBOL extension. COBOL syntax and semantics supported by IBM compilers in addition to those described in Standard COBOL 85.

`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` can 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 that is a function, a function-identifier is used.

IGZCBSO. The Enterprise COBOL bootstrap routine. It must be link-edited with any module that contains a Enterprise COBOL 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 can consist of a sequence of imperative statements.

* **implicit scope terminator.** A separator period that terminates the scope of any preceding unterminated statement, or a phrase of a statement that 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. A mechanism for using the implementation of a class as the basis for another class. By definition, the inheriting class conforms to the inherited classes. Enterprise COBOL does not support *multiple inheritance*; a subclass has exactly one immediate superclass.

inheritance hierarchy. See *class hierarchy*.

* **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 at run time.

* **input-output statement.** A statement that causes files to be processed by performing operations on individual records or on 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 that defines the state of an object. The instance data introduced by a class is defined in the WORKING-STORAGE SECTION of the DATA DIVISION in the OBJECT paragraph of the class definition. The state of an object also includes the state of the instance variables introduced by 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.

Interactive System Productivity Facility (ISPF). An IBM software product that provides a menu-driven interface for the TSO or VM user. ISPF includes library utilities, a powerful editor, and dialog management.

interlanguage communication (ILC). The ability of routines written in different programming languages to communicate. ILC support allows the application developer to readily build applications from component routines written in a variety of languages.

intermediate result. An intermediate field that contains the results of a succession of arithmetic operations.

* **internal data.** The data that is described in a program and excludes 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 that is described in one program in a run unit. An internal data item can have a global name.

internal decimal data item. A data item that is described as USAGE PACKED-DECIMAL or USAGE COMP-3, and that has a PICTURE character string that defines the item as numeric (a valid combination of symbols 9, S, P, or V). Synonymous with *packed-decimal data item*.

* **internal file connector.** A file connector that is accessible to only one object program in the run unit.

internal floating-point data item. A data item that is described as USAGE COMP-1 or USAGE COMP-2. COMP-1 defines a single-precision floating-point data item. COMP-2 defines a double-precision floating-point data item. There is no PICTURE clause associated with an internal floating-point data item.

* **intrarecord data structure.** The entire collection of groups and elementary data items from a logical record that a contiguous subset of the data description entries defines. 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 predefined function, such as a commonly used arithmetic function, called by a built-in function reference.

* **invalid key condition.** A condition, at run time, caused when a specific value of the key associated with an indexed or relative file is determined to be not valid.

* **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; this entry 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 that 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.

is-a. A relationship that characterizes classes and subclasses in an inheritance hierarchy. Subclasses that have an is-a relationship to a class inherit from that class.

ISPF. See *Interactive System Productivity Facility (ISPF).*

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.

J

J2EE. See *Java 2 Platform, Enterprise Edition (J2EE).*

Java 2 Platform, Enterprise Edition (J2EE). An environment for developing and deploying enterprise applications, defined by Sun Microsystems, Inc. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multitiered, Web-based applications. (Sun)

Java batch-processing program (JBP). An IMS batch-processing program that has access to online databases and output message queues. JBPs run online, but like programs in a batch environment, they are started with JCL or in a TSO session.

Java batch-processing region. An IMS dependent region in which only Java batch-processing programs are scheduled.

Java Database Connectivity (JDBC). A specification from Sun Microsystems that defines an API that enables Java programs to access databases.

Java message-processing program (JMP). An IMS Java application program that is driven by transactions and has access to online IMS databases and message queues.

Java message-processing region. An IMS dependent region in which only Java message-processing programs are scheduled.

Java Native Interface (JNI). A programming interface that allows Java code that runs inside a Java virtual machine (JVM) to interoperate with applications and libraries written in other programming languages.

Java virtual machine (JVM). A software implementation of a central processing unit that runs compiled Java programs.

JavaBeans. A portable, platform-independent, reusable component model. (Sun)

JBP. See *Java batch-processing program (JBP).*

JDBC. See *Java Database Connectivity (JDBC).*

JMP. See *Java message-processing program (JMP).*

job control language (JCL). A control language used to identify a job to an operating system and to describe the job's requirements.

JVM. See *Java virtual machine (JVM).*

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 that serve to identify the ordering of data.

* **key of reference.** The key, either prime or alternate, currently being used to access records within an indexed file.

* **keyword.** A reserved word or function-name whose presence is required when the format in which the word appears is used in a source program.

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 (such as Enterprise COBOL, COBOL for OS/390 & VM, COBOL for MVS & VM, C/C++ for MVS & VM, PL/I for MVS & VM) that produce object code conforming to the Language Environment conventions.

last-used state. A state that a program is in if its internal values remain the same as when the program was exited (the values 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) that 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 can 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 the compiler is to use for compiling a given source program.

* **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. The Lilian date format is named in honor of Luigi Lilio, the creator of the Gregorian calendar.

* **linage-counter.** A special register whose value points to the current position within the page body.

link. (1) The combination of the link connection (the transmission medium) and two link stations, one at each end of the link connection. A link can be shared among multiple links in a multipoint or token-ring configuration. (2) To interconnect items of data or portions of one or more computer programs; for example, linking object programs by a linkage editor to produce an executable file.

LINKAGE SECTION. The section in the DATA DIVISION of the called program or invoked method that describes data items available from the calling program or invoking method. Both the calling program or invoking method and the called program or invoked method can refer to these data items

linker. A term that refers to either the z/OS linkage editor or the z/OS binder.

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. The default format that Intel processors use to store binary data and UTF-16 characters. In this format, the most significant byte of a binary data item is at the highest address and the most significant byte of a UTF-16 character is at the highest address. Compare with *big-endian*.

local reference. A reference to an object that is within the scope of your method.

locale. A set of attributes for a program execution environment that indicates culturally sensitive considerations, such as character code page, collating sequence, date and 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 the 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 can be either an elementary item or a group of items. 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 that receives control when the programs are run within a process.

makefile. A text file that contains a list of the files for your application. The make utility uses this file to update the target files with the latest changes.

* **mass storage.** A storage medium in which data can be organized and maintained in both a sequential manner and a nonsequential manner.

* **mass storage device.** A device that has a large storage capacity, such as a magnetic disk.

* **mass storage file.** A collection of records that is stored in a mass storage medium.

* **megabyte (MB).** 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.

message-processing program (MPP). An IMS application program that is driven by transactions and has access to online IMS databases and message queues.

message queue. The data set on which messages are queued before being processed by an application program or sent to a terminal.

method. Procedural code that defines an operation supported by an object and that is executed by an INVOKE statement on that object.

*** method definition.** The COBOL source code that defines a method.

*** method identification entry.** An entry in the METHOD-ID paragraph of the IDENTIFICATION DIVISION; this entry contains a clause that specifies the method-name.

method invocation. A communication from one object to another that requests the receiving object to execute a method.

method-name. The name of an object-oriented operation. When used to invoke the method, the name can be an alphanumeric or national literal or a category alphanumeric or category national data item. When used in the METHOD-ID paragraph to define the method, the name must be an alphanumeric or national literal.

*** mnemonic-name.** A user-defined word that is associated in the ENVIRONMENT DIVISION with a specified implementor-name.

module definition file. A file that describes the code segments within a load module.

MPP. See *message-processing program (MPP).*

multitasking. A mode of operation that provides for the concurrent, or interleaved, execution of two or more tasks.

multithreading. Concurrent operation of more than one path of execution within a computer. Synonymous with *multiprocessing*.

N

name. A word (composed of not more than 30 characters) that defines a COBOL operand.

national character. (1) A UTF-16 character in a USAGE NATIONAL data item or national literal. (2) Any character represented in UTF-16.

national character position. See *character position*.

national data item. A data item of category national, national-edited, or numeric-edited of USAGE NATIONAL.

national decimal data item. An external decimal data item that is described implicitly or explicitly as USAGE NATIONAL and that contains a valid combination of PICTURE symbols 9, S, P, and V.

national-edited data item. A data item that is described by a PICTURE character string that contains at least one instance of the symbol N and at least one of the simple insertion symbols B, 0, or /. A national-edited data item has USAGE NATIONAL.

national floating-point data item. An external floating-point data item that is described implicitly or explicitly as USAGE NATIONAL and that has a PICTURE character string that describes a floating-point data item.

national group item. A group item that is explicitly or implicitly described with a GROUP-USAGE NATIONAL clause. A national group item is processed as though it were defined as an elementary data item of category national for operations such as INSPECT, STRING, and UNSTRING. This processing ensures correct padding and truncation of national characters, as contrasted with defining USAGE NATIONAL data items within an alphanumeric group item. For operations that require processing of the elementary items within a group, such as MOVE CORRESPONDING, ADD CORRESPONDING, and INITIALIZE, a national group is processed using group semantics.

*** 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.

native method. A Java method with an implementation that is written in another programming language, such as COBOL.

*** negated combined condition.** The NOT logical operator immediately followed by a parenthesized combined condition. See also *condition* and *combined condition*.

*** negated simple condition.** The NOT logical operator immediately followed by a simple condition. See also *condition* and *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 SECTION and LINKAGE SECTION that bear no hierarchic relationship to other data items.

nondate. Any of the following:

- A data item whose date description entry does not include the DATE FORMAT clause
- A literal
- A date field that has been converted using the UNDATE function
- A reference-modified date field
- The result of certain arithmetic operations that can include date field operands; for example, the difference between two compatible date fields

null. A figurative constant that is used to assign, to pointer data items, the value of an address that is not valid. 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 data item. (1) A data item whose description restricts its content to a value represented by characters chosen from the digits 0 through 9. If signed, the item can also contain a +, -, or other representation of an operational sign. (2) A data item of category numeric, internal floating-point, or external floating-point. A numeric data item can have USAGE DISPLAY, NATIONAL, PACKED-DECIMAL, BINARY, COMP, COMP-1, COMP-2, COMP-3, COMP-4, or COMP-5.

numeric-edited data item. A data item that contains numeric data in a form suitable for use in printed output. It can consist of external decimal digits from 0 through 9, the decimal separator, commas, the currency sign, sign control characters, and other editing characters. A numeric-edited item can be represented in either USAGE DISPLAY or USAGE NATIONAL.

* **numeric function.** A function whose class and category are numeric but that for some possible evaluation does not satisfy the requirements of integer functions.

* **numeric literal.** A literal composed of one or more numeric characters that can contain 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. Each object in the class is said to be an instance of the class.

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, where the object program is run, is described.

* **object computer entry.** An entry in the OBJECT-COMPUTER paragraph of the ENVIRONMENT DIVISION; this entry 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 instance. See *object*.

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-oriented programming. A programming approach based on the concepts of encapsulation and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on the data objects that comprise the problem and how they are manipulated, not on how something is accomplished.

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 or class definition. Where there is no danger of ambiguity, the word *program* can be used in place of *object program*.

object reference. A value that identifies an instance of a class. If the class is not specified, the object reference is universal and can apply to instances of any class.

* **object time.** The time at which an object program is executed. Synonymous with *run time*.

* **obsolete element.** A COBOL language element in Standard COBOL 85 that was deleted from Standard COBOL 2002.

ODO object. In the example below, X is the object of the OCCURS DEPENDING ON clause (ODO object).

```
WORKING-STORAGE SECTION
01  TABLE-1.
    05  X                PICS9.
    05  Y OCCURS 3 TIMES
          DEPENDING ON X    PIC X.
```

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.** (1) The general definition of operand is "the component that is operated upon." (2) For the purposes of this document, any lowercase word (or words) that appears in a statement or entry format can be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

operation. A service that can be requested of an object.

* **operational sign.** An algebraic sign that is associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

* **optional file.** A file that is declared as being not necessarily present each time the object program is run. 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. Its presence is optional to the user when the format in which the word appears is used in a source unit.

* **output file.** A file that is opened in either 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.

overload. To define a method with the same name as another method that is available in the same class, but with a different signature. See also *signature*.

override. To redefine an instance method (inherited from a parent class) in a subclass.

P

package. A group of related Java classes, which can be imported individually or as a whole.

packed-decimal data item. See *internal decimal data item*.

padding character. An alphanumeric or national character that is used to fill the unused character positions in a physical record.

page. A vertical division of output data that represents a physical separation of the data. The separation is based on internal logical requirements or external characteristics of the output medium or both.

* **page body.** That part of the logical page in which lines can be written or spaced or both.

* **paragraph.** In the PROCEDURE DIVISION, a paragraph-name followed by a n period and by zero, one, or more sentences. In the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION, 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 DIVISION and ENVIRONMENT DIVISION. 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. (1) Data passed between a calling program and a called program. (2) A data element in the USING phrase of a method invocation. Arguments provide additional information that the invoked method can use to perform the requested operation.

Persistent Reusable JVM. A JVM that can be serially reused for transaction processing by resetting the JVM between transactions. The reset phase restores the JVM to a known initialization state.

* **phrase.** 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.

port. (1) To modify a computer program to enable it to run on a different platform. (2) In the Internet suite of protocols, a specific logical connector between the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) and a higher-level protocol or application. A port is identified by a port number.

portability. The ability to transfer an application program from one application platform to another with relatively few changes to the source program.

preinitialization. The initialization of the COBOL runtime environment in preparation for multiple calls from programs, especially non-COBOL programs. The environment is not terminated until an explicit termination.

* **prime record key.** A key whose contents uniquely identify a record within an indexed file.

* **priority-number.** A user-defined word that classifies sections in the PROCEDURE DIVISION for purposes of segmentation. Segment numbers can contain only the characters 0 through 9. A segment number can be expressed as either one or two digits.

private. As applied to factory data or instance data, accessible only by methods of the class that defines the data.

* **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 code. 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), XML PARSE.

PROCEDURE DIVISION. The COBOL division that contains instructions for solving a problem.

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 can 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. Typically used to communicate with COBOL and Language Environment programs.

process. The course of events that occurs during the execution of all or part of a program. Multiple processes can run concurrently, and programs that run within a process can share resources.

program. (1) A sequence of instructions suitable for processing by a computer. Processing may include the use of a compiler to prepare the program for execution, as well as a runtime environment to execute it. (2) A logical assembly of one or more interrelated modules. Multiple copies of the same program can be run in different processes.

* **program identification entry.** In the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION, an entry that 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 marker, a user-defined word or alphanumeric literal that identifies a COBOL source program.

project. The complete set of data and actions that are required to build a target, such as a dynamic link library (DLL) or other executable (EXE).

* **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
	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 that 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. Synonymous with *record description*.

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.

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).

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. 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 category alphanumeric, category DBCS, or category national data item by specifying the leftmost character and length relative to the leftmost character position of a `USAGE DISPLAY`, `DISPLAY-1`, or `NATIONAL` 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*.

*** 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, nonnumeric literal, or index-name has a specific relationship to the value of another arithmetic expression, data item, nonnumeric literal, or index name. See also *relational operator*.

* **relational operator.** A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meanings are:

Character	**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

* **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 logical ordinal position of the record 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 that is an integer.

* **reserved word.** A COBOL word that is specified in the list of words that can be used in a COBOL source program, but that must not appear in the program as a user-defined word or system-name.

* **resource.** A facility or service, controlled by the operating system, that an executing program can use.

* **resultant identifier.** A user-defined data item that is to contain the result of an arithmetic operation.

reusable environment. A reusable environment is created when you establish an assembler program as the main program by using either the old COBOL interfaces for preinitialization (functions ILBOSTP0 and IGZERRE, and the `RTEREUS` runtime option), or the Language Environment interface, CEEPIPI.

ring. In the COBOL editor, a set of files that are available for editing so that you can easily move between them.

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. Synonymous with *object time*.

runtime environment. The environment in which a COBOL program executes.

* **run unit.** A stand-alone object program, or several object programs, that interact by means of COBOL `CALL` or `INVOKE` statements and function at run time as an entity.

S

SBCS. See *single-byte character set (SBCS)*.

scope terminator. A COBOL reserved word that marks the end of certain `PROCEDURE DIVISION` statements. It can 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 any of these divisions: `ENVIRONMENT`, `DATA`, or `PROCEDURE`. In the `ENVIRONMENT DIVISION` and `DATA DIVISION`, 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 that, together with its contained programs, is compiled separately from all other programs.

* **separator.** A character or two or more 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.

session bean. In EJB, an enterprise bean that is created by a client and that usually exists only for the duration of a single client/server session. (Sun)

* **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.

signature. (1) The name of an operation and its parameters. (2) The name of a method and the number and types of its formal parameters.

* **simple condition.** Any single condition chosen from this set:

- Relation condition
- Class condition
- Condition-name condition
- Switch-status condition
- Sign condition

See also *condition* and *negated simple condition*.

single-byte character set (SBCS). A set of characters in which each character is represented by a single byte. See also *ASCII* and *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.

* **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, where the source program is compiled, is described.

* **source computer entry.** An entry in the SOURCE-COMPUTER paragraph of the ENVIRONMENT DIVISION; this entry 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 a source program can be represented by other forms and symbols, in this document the term always refers to a syntactically correct set of COBOL statements. A COBOL source program commences with the IDENTIFICATION DIVISION or a COPY statement and terminates with the end program marker, if specified, or with the absence of additional source program lines.

source unit. A unit of COBOL source code that can be separately compiled: a program or a class definition. Also known as a *compilation unit*.

* **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-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`; this entry 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 COBOL 85. The COBOL language defined by the following standards:

- *ANSI INCITS 23-1985, Programming languages - COBOL*, as amended by *ANSI INCITS 23a-1989, Programming Languages - COBOL - Intrinsic Function Module for COBOL* and *ANSI INCITS 23b-1993, Programming Languages - Correction Amendment for COBOL*
- *ISO 1989:1985, Programming languages - COBOL*, as amended by *ISO/IEC 1989/AMD1:1992, Programming languages - COBOL: Intrinsic function module* and *ISO/IEC 1989/AMD2:1994, Programming languages - Correction and clarification amendment for COBOL*

* **statement.** A syntactically valid combination of words, literals, and separators, beginning with a verb, written in a COBOL source program.

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.

* **subclass.** A class that inherits from another class. When two classes in an inheritance relationship are considered together, the subclass is the inheritor or inheriting class; the superclass 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 that is 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 can 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.

substitution character. A character that is used in a conversion from a source code page to a target code page to represent a character that is not defined in the target code page.

* **superclass.** A class that is inherited by another class. See also *subclass*.

surrogate pair. In the UTF-16 format of Unicode, a pair of encoding units that together represents a single Unicode graphic character. The first unit of the pair is called a *high surrogate* and the second a *low surrogate*. The code value of a high surrogate is in the range X'D800' through X'DBFF'. The code value of a low surrogate is in the range X'DC00' through X'DFFF'. Surrogate pairs provide for more characters than the 65,536 characters that fit in the Unicode 16-bit coded character set.

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.

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 pseudo-text that is any of the following characters:

- 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 nonnumeric literals, the opening quotation mark and the closing quotation mark that bound the literal.
- Any other sequence of contiguous COBOL characters except comment lines and the word COPY bounded by separators that are neither a separator nor a literal.

thread. A stream of computer instructions (initiated by an application within a process) that is in control of a process.

token. In the COBOL editor, a unit of meaning in a program. A token can contain data, a language keyword, an identifier, or other part of the language syntax.

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*.

troubleshoot. To detect, locate, and eliminate problems in using computer software.

* **truth value.** The representation of the result of the evaluation of a condition in terms of one of two values: true or false.

typed object reference. A data-name that can refer only to an object of a specified class or any of its subclasses.

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.

Unicode. A universal character encoding standard that supports the interchange, processing, and display of text that is written in any of the languages of the modern world. There are multiple encoding schemes to represent Unicode, including UTF-8, UTF-16, and UTF-32. Enterprise COBOL supports Unicode using UTF-16 in big-endian format as the representation for the national data type.

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.

unrestricted storage. Storage below the 2-GB bar. It can be above or below the 16-MB line. If it is above the 16-MB line, it is addressable only in 31-bit mode.

* **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 can 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 can 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 that is repeated a variable number of times. Such an item must contain an OCCURS DEPENDING ON clause in its data description entry or be subordinate to such an item.

* **variably located group.** A group item following, and not subordinate to, a variable-length table in the same record. The group item can be an alphanumeric group or a national group.

* **variably located item.** A data item following, and not subordinate to, a variable-length table in the same record.

* **verb.** A word that expresses an action to be taken by a COBOL compiler or object program.

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 that are executed automatically when the end of a unit or reel has been reached before end-of-file has been reached.

VSAM file system. A file system that supports COBOL sequential, relative, and indexed organizations.

W

Web service. A modular application that performs specific tasks and is accessible through open protocols like HTTP and SOAP.

white space. Characters that introduce space into a document. They are:

- Space
- Horizontal tabulation
- Carriage return
- Line feed
- Next line

as named in the Unicode Standard.

windowed date field. A date field containing a windowed (two-digit) year. See also *date field* and *windowed year*.

windowed year. A date field that consists only of a two-digit year. This two-digit year can be interpreted using a century window. For example, 06 could be interpreted as 2006. See also *century window*. Compare with *expanded year*.

* **word.** A character string of not more than 30 characters that forms a user-defined word, a system-name, a reserved word, or a function-name.

* **WORKING-STORAGE SECTION.** The section of the DATA DIVISION that describes working-storage data items, composed either of noncontiguous items or working-storage records or of both.

workstation. A generic term for computers used by end users including personal computers, 3270 terminals, intelligent workstations, and UNIX terminals. Often a workstation is connected to a mainframe or to a network.

wrapper. An object that provides an interface between object-oriented code and procedure-oriented code. Using wrappers allows programs to be reused and accessed by other systems.

X

x. The symbol in a PICTURE clause that can hold any character in the character set of the computer.

XML. Extensible Markup Language. A standard metalanguage for defining markup languages that was derived from and is a subset of SGML. XML omits the more complex and less-used parts of SGML and makes it much easier to write applications to handle document types, author and manage structured information, and transmit and share structured information across diverse computing systems. The use of XML does not require the robust applications and processing that is necessary for SGML. XML is developed under the auspices of the World Wide Web Consortium (W3C).

XML data. Data that is organized into a hierarchical structure with XML elements. The data definitions are defined in XML element type declarations.

XML declaration. XML text that specifies characteristics of the XML document such as the version of XML being used and the encoding of the document.

XML document. A data object that is well formed as defined by the W3C XML specification.

XML type definition. An XML element that contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition, or DTD.

Y

year field expansion. Explicitly expanding date fields that contain two-digit years to contain four-digit years in files and databases and then using these fields in expanded form in programs. This is the only method for assuring reliable date processing for applications that have used two-digit years.

Z

zoned decimal data item. An external decimal data item that is described implicitly or explicitly as `USAGE DISPLAY` and that contains a valid combination of `PICTURE` symbols `9`, `S`, `P`, and `V`. The content of a zoned decimal data item is represented in characters 0 through 9, optionally with a sign. If the `PICTURE` string specifies a sign and the `SIGN IS SEPARATE` clause is specified, the sign is represented as characters + or -. If `SIGN IS SEPARATE` is not specified, the sign is one hexadecimal digit that overlays the first 4 bits of the sign position (leading or trailing).

List of resources

Enterprise COBOL for z/OS

Compiler and Runtime Migration Guide, GC27-1409

Customization Guide, GC27-1410

Debug Tool Reference and Messages, SC18-9304

Debug Tool User's Guide, SC18-9302

Fact Sheet, GC27-1407

Language Reference, SC27-1408

Licensed Program Specifications, GC27-1411

Programming Guide, SC27-1412

Support

Performance Tuning, www.ibm.com/support/docview.wss?uid=swg27001475

If you have a problem using Enterprise COBOL for z/OS, see the following site, which provides up-to-date support information: www.ibm.com/software/awdtools/cobol/zos/support/.

Related publications

CICS Transaction Server for z/OS

Application Programming Guide, SC34-6433

Application Programming Reference, SC34-6434

Customization Guide, SC34-6429

External Interfaces Guide, SC34-6449

z/OS C/C++

Programming Guide, SC09-4765

Run-Time Library Reference, SA22-7821

DB2 UDB for z/OS

Application Programming and SQL Guide, SC18-7415

Command Reference, SC18-7416

SQL Reference, SC18-7426

z/OS DFSMS

Access Method Services for Catalogs, SC26-7394

Checkpoint/Restart, SC26-7401

Macro Instructions for Data Sets, SC26-7408

Using Data Sets, SC26-7410

Utilities, SC26-7414

DFSORT

Application Programming Guide, SC26-7523

Installation and Customization, SC26-7524

IMS

Application Programming: Database Manager, SC18-7809

Application Programming: Design Guide, SC18-7810

Application Programming: EXEC DLI Commands for CICS and IMS, SC18-7811

Application Programming: Transaction Manager, SC18-7812

Connect Guide and Reference, SC18-9287

Java Guide and Reference, SC18-7821

z/OS ISPF

Dialog Developer's Guide and Reference, SC34-4821

User's Guide Vol. 1, SC34-4822

User's Guide Vol. 2, SC34-4823

z/OS Language Environment

Concepts Guide, SA22-7567

Customization, SA22-7564

Debugging Guide, GA22-7560

Programming Guide, SA22-7561

Programming Reference, SA22-7562

Run-Time Messages, SA22-7566

Run-Time Migration Guide, GA22-7565

Writing Interlanguage Communication Applications, SA22-7563

z/OS MVS

JCL Reference, SA22-7597

JCL User's Guide, SA22-7598

Program Management: User's Guide and Reference, SA22-7643

System Commands, SA22-7627

z/OS TSO/E

Command Reference, SA22-7782

Primer, SA22-7787

User's Guide, SA22-7794

z/OS UNIX System Services

Command Reference, SA22-7802

Programming: Assembler Callable Services Reference, SA22-7803

User's Guide, SA22-7801

z/Architecture(R)

Principles of Operation, SA22-7832

Softcopy publications for z/OS

The following collection kit contains z/OS and related product publications:

z/OS CD Collection Kit, SK3T-4269

Unicode and character representation

Unicode, www.unicode.org/

Character Data Representation Architecture: Reference and Registry, SC09-2190

z/OS Support for Unicode: Using Conversion Services, SA22-7649

Java

The Java Language Specification, Second Edition, by Gosling et al., java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html

The Java Native Interface, java.sun.com/j2se/1.3/docs/guide/jni/index.html

The Java 2 Enterprise Edition Developer's Guide, java.sun.com/j2ee/sdk_1.2.1/techdocs/guides/ejb/html/DevGuideTOC.html

Java 2 on z/OS, www.ibm.com/servers/eserver/zseries/software/java/

Persistent Reusable Java Virtual Machine User's Guide, SC34-6201

WebSphere Application Server for z/OS

Applications, SA22-7959

XML

Extensible Markup Language (XML), www.w3.org/XML/

XML specification, www.w3.org/TR/REC-xml/

Index

Readers' Comments — We'd Like to Hear from You

Enterprise COBOL for z/OS
Programming Guide
Version 3 Release 4

Publication No. SC27-1412-05

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	E-mail address

Readers' Comments — We'd Like to Hear from You
SC27-1412-05

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

IBM Corporation
Reader Comments
DTX/E269
555 Bailey Avenue
San Jose, CA
U.S.A. 95141-9989

Fold and Tape **Please do not staple** Fold and Tape

SC27-1412-05

Cut or Fold
Along Line

Program Number: 5655-G53

Printed in USA

SC27-1412-05