IBM COBOL for VSE/ESA

Programming Guide

Release 1

SC26-8072-02

IBM COBOL for VSE/ESA

Programming Guide

Release 1

SC26-8072-02

Note!

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

Third Edition (June 1998)

This edition applies to Version 1 Release 1 Modification 1 of IBM COBOL for VSE/ESA, Program Number 5686-068, and to all subsequent releases and modifications until otherwise indicated in new editions.

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

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

IBM Corporation, Department BWE/H3
P.O. Box 49023
San Jose, California, 95161-9023
U.S.A.

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

Contents

Notices

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

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

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

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

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

Programming Interfaces

This *COBOL/VSE Programming Guide* is intended to help you create and compile IBM COBOL for VSE/ESA application programs. This book documents General-Use Programming Interface and Associated Guidance Information provided by IBM COBOL for VSE/ESA.

General-Use programming interfaces allow the customer to write programs that obtain the services of IBM COBOL for VSE/ESA.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AD/Cycle
CICS
CICS/ESA
CICS/VSE
COBOL/370
DFSORT
Enterprise Systems Architecture/370
IBM
IMS/ESA
MVS/ESA
OS/2
OS/390
SQL/DS
System/370
System/390
VisualAge
VM/ESA

Other company, product or service names may be the trademarks or service marks of others.

About This Book

The purpose of this book is to help you write and compile IBM COBOL for VSE/ESA programs. A companion volume, *LE/VSE Programming Guide*, provides instructions on link-editing and running your programs.

This book assumes experience in developing application programs and some knowledge of COBOL. It focuses on using IBM COBOL for VSE/ESA (hereafter referred to as COBOL/VSE) to accomplish your programming objectives and not on the definition of the COBOL/VSE language. For complete information on COBOL/VSE syntax, refer to *COBOL/VSE Language Reference*.

Previous IBM COBOL products provided their own environment and services for running programs, and the application programming guides for these products commonly included information on how to link-edit and run your programs. However, IBM Language Environment for VSE/ESA (LE/VSE) provides the run-time environment and run-time services required to run your COBOL/VSE programs. Therefore, you will find information on link-editing and executing programs in the *LE/VSE Programming Guide*.

As mentioned, the focus of this book is to provide guidance on creating and compiling COBOL/VSE programs. For information on migrating DOS/VS COBOL and VS COBOL II programs to COBOL/VSE, see *COBOL/VSE Migration Guide*.

For an overview of all the COBOL/VSE and LE/VSE publications, see "Bibliography" on page 448. For a comparison of commonly used COBOL/VSE and LE/VSE terms, see "Comparison of Commonly Used LE/VSE and COBOL/VSE Terms" on page xxi.

Abbreviated Terms

Certain terms are used in a shortened form in this book. Abbreviations for the product names used most frequently in this book are listed alphabetically in Figure 1. Abbreviations for other terms, if not commonly understood, are listed in the glossary in the back of this book.

Figure 1. Common Abbreviations in this Book

Term Used	Long Form
CICS®	CICS/VSE®
COBOL/VSE	IBM COBOL for VSE/ESA
DL/I	DL/I DOS/VS
LE/VSE	IBM Language Environment for VSE/ESA
VSE	VSE/ESA

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

- ISO 1989:1985, Programming languages - COBOL
- ISO 1989/Amendment 1, Programming Languages - COBOL - Amendment 1: Intrinsic function module

- X3.23-1985, American National Standard for Information Systems - Programming Language - COBOL
- X3.23a-1989, American National Standard for Information Systems - Programming Language - Intrinsic Function Module for COBOL

Note that the two ISO standards are identical to the American National standards.

Syntax Notation

In this book, syntax is described using the structure defined below.

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

  The ►►— indicates the beginning of a syntax diagram.

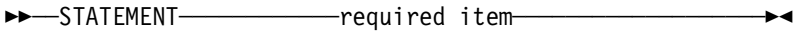
  The —► symbol indicates that the statement syntax is continued on the next line.

  The ►— symbol indicates that a statement is continued from the previous line.

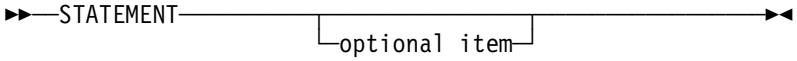
  The —►◄ indicates the end of a syntax diagram.

  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).

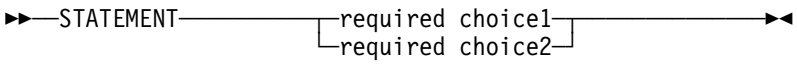


- Optional items appear below the main path.

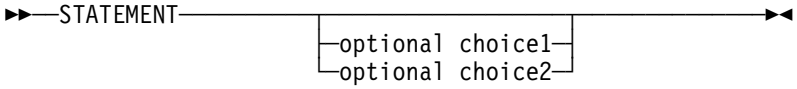


- When you can choose from two or more items, they appear vertically in a stack.

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



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



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

How Examples Are Shown

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

To improve clarity, examples are separated from the explanatory text. They may be indented, printed in a different font style, or are shown in boxes.

Publications Provided with COBOL/VSE

Publications provided with the COBOL/VSE product include the following:

Figure 2. IBM COBOL for VSE/ESA Publications

Task	Publication	Order number
Evaluation and Planning	*General Information*	GC26-8068
	Migration Guide	GC26-8070
	Installation and Customization Guide	SC26-8071
Programming	*Programming Guide*	SC26-8072
	Language Reference	SC26-8073
	Reference Summary	SX26-3834
Diagnosis	*Diagnosis Guide*	SC26-8528
Warranty	*Licensed Program Specifications*	GC26-8069

General Information
: Contains high-level information designed to help you evaluate the COBOL/VSE product. This book describes new compiler and language features, application development with LE/VSE, and product support for industry standards.

Migration Guide
: Contains detailed migration and compatibility information for current users of DOS/VS COBOL and VS COBOL II who wish to migrate to, or reuse existing applications on, COBOL/VSE. This book also describes several migration aids or tools to help you plan a migration path for your installation.

Installation and Customization Guide
: Provides information you will need in order to install and customize the COBOL/VSE product. Detailed planning information includes:

  - System and storage requirements for COBOL/VSE
  - Information about changing compiler option defaults during installation
  - Information for installing the product in shared storage

Programming Guide
: Contains guidance information for writing and compiling application programs using COBOL/VSE, including information on the following topics:

  - Programming using new product features, such as intrinsic functions

- Processing techniques for VSAM and SAM files
- Debugging techniques using compiler options and listings
- Nested programming techniques
- Year 2000 processing, and using the millennium language extensions
- Subsystem considerations

Language Reference
Provides syntax and semantic information about the COBOL language as implemented by IBM, including rules for writing source programs, and descriptions of IBM language extensions. This book is meant to be used in conjunction with the *COBOL/VSE Programming Guide*, which provides programming task-oriented information.

Reference Summary
Contains a convenient summary of the COBOL/VSE language syntax—including new intrinsic functions—as well as syntax for compiler options, compiler-directing statements, and the COBOL/VSE reserved word list.

Diagnosis Guide
Provides instructions for diagnosing failures in the COBOL/VSE compiler product that are not caused by user error. This book will help you construct a keyword string that allows you or IBM Service to search the product failure database for previously documented problems and appropriate corrections.

Licensed Program Specifications
Contains a product description and product warranty information for the COBOL/VSE compiler.

Language Environment for VSE/ESA Publications

Other publications useful for developing applications with COBOL/VSE include the following publications provided with the LE/VSE product:

Figure 3. IBM Language Environment for VSE/ESA Publications

Task	Publication	Order number
Evaluation and Planning	*Fact Sheet*	GC33-6679
	Concepts Guide	GC33-6680
	Installation and Customization Guide	SC33-6682
Programming	*Programming Guide*	SC33-6684
	Programming Reference	SC33-6685
	C Run-Time Programming Guide	SC33-6688
	Writing Interlanguage Communication Applications	SC33-6686
	Debugging Guide and Run-Time Messages	SC33-6681
Migrating	*Run-Time Migration Guide*	SC33-6687
Warranty	*Licensed Program Specifications*	GC33-6683

Fact Sheet
Provides a brief overview and description of LE/VSE.

Concepts Guide
Provides a detailed overview of program models and intended architecture for LE/VSE, the common run-time environment.

Installation and Customization Guide
Contains information needed to plan for installing and customizing the LE/VSE product.

Programming Guide
Provides detailed information on the following topics:

- Directions for linking and running programs that use LE/VSE services
- Information on storage management, run-time message handling, and condition handling models

This book also contains language-specific run-time information.

Programming Reference
Provides detailed information on callable services and run-time options, and how to use them.

Writing Interlanguage Communication Applications
Provides instructions for writing programs that use interlanguage communication (ILC).

C Run-Time Programming Guide
Provides information on customizing and using locales, to allow your programs to present information in country-specific formats.

Debugging Guide and Run-Time Messages
Provides detailed information on debugging techniques and services. Provides a listing of run-time messages and their explanations, as well as abend codes.

Licensed Program Specifications
Contains a product description and warranty information.

Comparison of Commonly Used LE/VSE and COBOL/VSE Terms

For a better understanding of the various terms used throughout the *IBM Language Environment for VSE/ESA* and *IBM COBOL for VSE/ESA* publications, and what terms are meant to be equivalent, see the following table.

Figure 4. Comparison of Commonly-used LE/VSE and COBOL/VSE Terms

LE/VSE Term	COBOL/VSE 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	Working-Storage data (defined without EXTERNAL clause)
Routine	Program
Scalar	Elementary Item
Stack frame	Register save area associated with the program in the TGT

Summary of Changes

This section lists the major changes that have been made to the COBOL/VSE product and this manual since Release 1. Technical changes are marked in the text by a change bar in the left margin.

Changes in Modification Level 1

- The INTDATE compiler option, allowing COBOL integer dates to be treated as either ANSI-standard or LE/VSE-compatible.
- New intrinsic functions to convert 2-digit year dates to 4-digit year dates.
- New specifications on the format-2 ACCEPT statement to receive the system date with a 4-digit year.
- The millennium language extensions, enabling compiler-assisted date processing for dates containing 2-digit and 4-digit years (see Chapter 22, “Using the Millennium Language Extensions” on page 366).

  Requires VisualAge COBOL Millennium Language Extensions for VSE/ESA (program number 5686-MLE) to be installed with your compiler.

Changes in the Third Edition

- Extensions to currency support for displaying financial data, including:
  - Support for currency signs of more than one character
  - Support for more than one type of currency sign in the same program
  - Support for the euro currency sign, as defined by the Economic and Monetary Union (EMU)

  (see “Using Currency Signs” on page 91).

Part 1. Overview of COBOL/VSE Programming

This part of the book provides an overview of product features and how they support you in the program development process.

Chapter 1. Product Features

COBOL (COmmon Business Oriented Language) is a programming language that efficiently processes business information. COBOL emphasizes the description and handling of data items and of input/output records. Instead of extensive algebraic or logical processing, COBOL applications manipulate large files of data in a relatively simple way.

The COBOL/VSE Compiler is an IBM licensed program that supports System/370™ Enterprise Systems Architecture (ESA). COBOL/VSE runs under VSE/ESA and requires LE/VSE.

The compiler accepts and compiles COBOL programs written in COBOL 85 Standard (except for Report Writer, Communications, and Level 2 of the Debug module). It also accepts and compiles a number of IBM extensions to COBOL 85 Standard. (See page xvi for the definition of COBOL 85 Standard as used in this book.)

The COBOL language as accepted by the compiler is described in *COBOL/VSE Language Reference*.

COBOL/VSE Features

The following features are available with COBOL/VSE:

24- or 31-Bit Addressing
The compiler and the object programs it produces can be run in either 24- or 31-bit addressing mode.

COBOL 85 Standard support
COBOL/VSE incorporates most of the major and minor language enhancements to the required modules for the high level of the COBOL 85 Standard. (See page xvi for the definition of COBOL 85 Standard as used in this book.)

COBOL/VSE supports all required modules at the intermediate level. It also supports all the required modules at the high level with the exception of the following language features:

- EXTEND phrase of the OPEN statement
- REVERSED phrase of the OPEN statement
- OF/IN phrase of the COPY statement

LE/VSE support
COBOL/VSE is a member of the set of languages that uses LE/VSE, a common run-time library and set of services. As such, development of multilanguage applications is greatly simplified and enhanced, since the application programmer uses a common interface to the run-time support—common run-time options, a set of powerful callable services, a common condition-handling mechanism, and a set of interlanguage call conventions.

In addition, LE/VSE support provides:

- Preinitialization of the run-time environment for multiple invocations of the same application (in a non-CICS environment)

- Customization of the run-time environment via a user exit facility
- Storage Management and Tuning
- Dynamic storage allocation for data areas
- One common dump for all LE-conforming languages in an easily understandable format
- National language support for message handling

These features are described in the *LE/VSE Programming Guide*.

VSAM (Virtual Storage Access Method) support

VSAM provides:

- Fast storage and retrieval of records
- Password protection
- Centralized and simplified data and space management
- Advanced error recovery facilities
- System and user catalogs

Using VSAM, COBOL supports:

- Sequential files (through VSAM ESDS capabilities)
- Indexed files with alternate indexes (through VSAM KSDS capabilities)
- Relative files (through VSAM RRDS capabilities)

Expanded VSAM file status codes enhance your ability to identify and handle exception conditions using the FILE STATUS clause.

Program-related information and listings

You can request and control a variety of listing and program-related information:

- Print or suppress the listing of your source program by using the SOURCE compiler option.
- Produce a listing of the assembler-language expansion of your source code by using the LIST compiler option.
- Control the appearance and content of your source listing by using the TITLE statement, LANGUAGE, LINECOUNT, SEQUENCE, SPACE, OFFSET, and NUMBER compiler options.
- Obtain cross-reference information for statements (verbs), data-names, procedure-names, and program-names, by using the VBREF and XREF compiler options.
- Obtain Data Division map listing, nested program structure map, global tables, and literal pools by using the MAP compiler option.
- Control the error level for which diagnostic messages are to appear in your listing by using the FLAG compiler option.
- Flag specific elements in your source listing. Use FLAGSTD to identify the level or subset of Standard COBOL that is to be regarded as the conformance level. Use FLAGMIG as an aid in migrating from VS COBOL II Release 2 level language to COBOL/VSE.
- Obtain a system dump for diagnostic use by using the DUMP compiler option.

- Use the ADATA compiler option to produce the SYSADAT file containing program data. This information can be used by utilities that previously parsed the compiler output listing.

Optimized object code
Produces object programs that generally use less processing time.

Library management facility
This default feature allows installations running with multiple LE/VSE partitions to save considerable main storage by sharing some or all of the LE/VSE library routine modules.

Syntax-checking compilation
Saves machine time while debugging source syntax errors. When the NOCOMPILE compiler option without any subparameter is used, the source program is scanned for syntax errors and diagnostic messages are produced, but no object code is produced. When you use the NOCOMPILE option with W, E, or S, a full compilation is produced with object code when no diagnostic message is found with higher severity than that requested. When you use the COMPILE option, a full compilation is produced even in the presence of serious errors. All diagnostics and object code will be produced.

Separately located installation defaults
Allows an installation to define the default compiler options, rather than have the definition in the compiler code.

Installation control of defaults
Prevents specific installation defaults from being replaced. (This allows enforcement of certain customer standards.)

Reentrant compiler
Allows the COBOL/VSE compiler to reside in the Shared Virtual Area so that it can be shared by multiple users.

Reentrant object code option
Allows the object code for a COBOL/VSE program to reside in the Shared Virtual Area so that one copy of it may be shared among all callers.

Intrinsic functions
Allows a variety of arithmetic, string-handling, and date management functions to be handled by reference to a function name from within a statement.

SSRANGE
Checks subscript and index values to see that the composite of the subscripts does not address an area beyond the region of the table. Checks to see that a variable-length item is within its defined maximum length. It also checks reference modification values to see that they do not address an area outside the region of the subject data item.

RTEREUS
In a non-CICS environment, RTEREUS is a run-time option that implicitly initializes the run-time environment for reuse when the first COBOL program is invoked.

User exits
The EXIT compiler option provides you with a way to specify that a user-supplied program, or programs, be given control at particular points in the compilation process. You may supply programs that will supply source input,

provide copy-library functions, receive the listing output, or be given access to records written to the Associated Data file.

Reserved word control
Allows you to specify (with the WORD compiler option) an alternative reserved word table to be used by the compiler. A user-defined reserved word table can be used instead of the default reserved word table supplied by IBM. A CICS-specific reserved word table is also provided as an alternative table, for flagging syntax not supported under CICS.

Hexadecimal notation for nonnumeric literals
Allows you to use hexadecimal notation in nonnumeric literals.

Batch compiling
A batch compiling technique and the use of the NAME compiler option allows you to create one or more phases with a single invocation of the compiler.

VS COBOL II Release 2 to COBOL/VSE migration tools
The CMPR2 and FLAGMIG compiler options help you to migrate your VS COBOL II Release 2 applications to COBOL/VSE.

Structured programming support
Constructs are provided that enable you to develop and maintain a structured application program. Such things as nested programs, WITH TEST BEFORE and AFTER phrases, and explicit scope terminators, aid in the development of structured programs.

Nonnumeric literal with double-byte characters
Allows you to use both EBCDIC and double-byte characters in nonnumeric literals.

National Language Support
COBOL/VSE supports a compiler option, LANGUAGE, which allows you to select a national language for compiler listing headings and compiler messages.

Currency sign compiler option
The CURRENCY compiler option allows you to select a symbol other than the dollar sign ($) to be the currency symbol you use in the PICTURE clause when you do not use the CURRENCY SIGN clause in your program.

Mixed-case headings and messages
The LANGUAGE compiler option allows you to select between mixed-case and uppercase compiler listing headings and compiler messages.

Numeric sign processing compiler option
With the NUMPROC compiler option you can select from three different kinds of sign processing, including one that provides sign processing similar to that used by DOS/VS COBOL.

Numeric truncation compiler option
The TRUNC compiler option allows you to select from three different types of numeric truncation for binary receiving fields.

Relaxed compiler limits
Many limits imposed by DOS/VS COBOL are relaxed in COBOL/VSE. Some of the Data Division limits are shown in Figure 5 on page 6. For a comparison of

other limits, see *COBOL/VSE Migration Guide*, or for a list of compiler limits, see Appendix A, "COBOL/VSE Compiler Limits" on page 398.

Figure 5. Compiler Limits Relaxed by COBOL/VSE

Language Element	DOS/VS COBOL Limit	COBOL/VSE Limit
Working-Storage Section	1 megabyte	128 megabytes
Level-77 data-names	1 million	16 million
Level-01 through -49 data-names	1 million	16 million
Elementary item size	32 kilobytes	16 megabytes
Linkage Section	1 megabyte	128 megabytes

Major IBM Extensions

ADDRESS special register
Allows the address of a record area to be passed and received, using a CALL statement.

LENGTH special register
Provides the number of single byte character positions used by an identifier.

TITLE page heading specification
Displays the literal in the title portion of the page heading of succeeding pages. It replaces the title portion provided by the compiler or a prior TITLE statement.

Double-Byte Character Set (DBCS)
Supports USAGE option (DISPLAY-1) and PICTURE characters (G and N) that define a data item that has 2-byte characters. Support includes DBCS user names and literals, special registers for shift-out and shift-in characters, and comparison between DBCS items, based on the binary collating sequence. In addition, DBCS items may also be reference modified. In addition, nonnumeric literals can contain a mix of DBCS and EBCDIC characters, and can also be used as titles in the TITLE statement. COBOL/VSE also provides two service routines (IGZCA2D and IGZCD2A) for converting alphanumeric and DBCS data items. DBCS is used primarily in conjunction with applications that support large character sets; for example, the Kanji character set used in Japan.

Hexadecimal notation for nonnumeric literals
Allows you to use hexadecimal notation in nonnumeric literals.

POINTER data item
Supports a USAGE option (POINTER), which defines a data type that can be used to hold addresses.

PROCEDURE-POINTER data item
Supports a USAGE option (PROCEDURE-POINTER), which defines a data type that holds the address of a program entry point.

Complex OCCURS DEPENDING ON
Adds support to OCCURS DEPENDING ON (ODO) for the following IBM extensions:

- In all formats with the OCCURS DEPENDING ON clause, any subordinate entry may be variable in length—that is, may contain an OCCURS DEPENDING ON clause.

- An entry containing an OCCURS DEPENDING ON clause need not be the last in a record or group. It may be followed by nonsubordinate entries, any of which may contain an OCCURS DEPENDING ON clause.

Floating-point data
Includes support for single-precision internal floating-point (COMP-1), double-precision internal floating-point (COMP-2), and external floating-point (DISPLAY) data types and fractional exponentiation.

Parentheses in abbreviated conditions
Provides support for the use of parentheses in abbreviated expressions to specify an intended order of evaluation and to clarify the expression for readability.

APPLY WRITE-ONLY
Allows you to store data more efficiently on external media.

TALLY special register
An internally defined data-name that can be referenced or modified by the user. It is commonly used in conjunction with such verbs as INSPECT, STRING, and UNSTRING.

SORT special registers
Compiler-defined data-names that may be referenced or modified by the user and will be used in the SORT interface. The six SORT special registers are SORT-CONTROL, SORT-CORE-SIZE, SORT-FILE-SIZE, SORT-MESSAGE, SORT-MODE-SIZE, and SORT-RETURN.

Date processing
Allows date fields with 2-digit years to be used with automatic recognition of the century, based on a century window. This requires VisualAge COBOL Millennium Language Extensions for VSE/ESA (5686-MLE) to be installed with your compiler.

Chapter 2. Program Development Process

Figure 6 shows the basic process for developing a program. The highlighted step is supported by the COBOL/VSE Compiler product and is the primary focus of this book. The other steps are supported by LE/VSE.

The sections following the figure describe each of these steps at a general level and refer you to other parts of this book for detailed information or to other publications as appropriate. Most of these publications and their order numbers are listed in “Bibliography” on page 448.

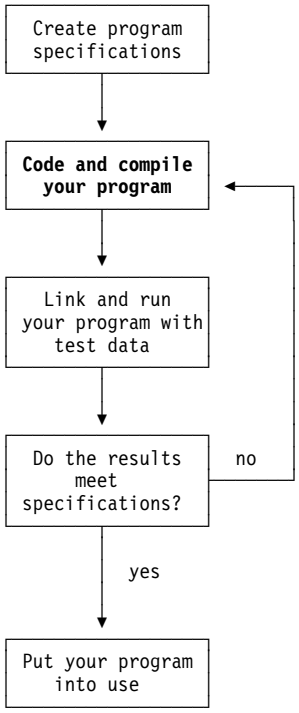


Figure 6. Program Development Process

Create Program Specifications

Before you can start coding your program, you need to understand clearly what it should do. In other words, you must know the **requirements** for your program.

Once you have determined the requirements, you can formulate a programming solution to the problem. You can express this solution in terms of a detailed program **design**. It is often necessary, especially for larger programming projects, to publish the requirements and design in some type of specifications document. This document must be detailed enough to enable you to verify that the program, when completed, meets the specifications.

As you create the specifications for your program, you might want to consider the factors that are outlined in the following section.

Determining Requirements

As mentioned, you must clearly understand what your program must do before you can design it. In other words, once you have determined:

- In what computer environment your program will run
- What inputs will be provided to your program
- What outputs or responses your program must produce

then you can decide:

- What processing your program needs to perform

Computer Environment

For your program's computer environment, you might need to determine:

- Which subsystem: CICS, DL/I, SQL/DS™?
- Which storage media: Tape, disk, or unit-record devices?
- Which access method: SAM or VSAM?
- Which input device: Terminal, unit-record devices or DBCS or special graphics input device?
- Which output device: Terminal, printer, DBCS or special graphics printer, tape, disk, or unit-record devices?

Inputs

For your program's inputs (information provided by the end user or by other programs), you might need to determine:

- Does the input data already exist?
- Where will the input data come from?
  - Will it be passed from another program? How?
  - Will it come from an online or batch system?
- What will the input data look like?
- What possible errors can the input data contain?
- What initial editing will you need to do?
- What is the size and type of each input value?
- What kinds of errors might the user make?
- Will more than one person use the program at the same time?

Outputs or Responses

For your program's outputs or responses, you might need to determine:

- What information does the end user require?
- How many reports will your program create?
- What comments or explanatory text need to appear on the reports?
- What format is required for the printed results?
- Where should results be printed or stored?
- Will results be used as input to another program?
- What error messages are required?
- Will your program continue running after less-than-severe errors?

Designing a Solution

Once you understand the problem in terms of specific requirements, you can design the processing that will solve the problem. “Design” implies a detailed description of various parts and how they work together.

Your design might include descriptions of:

- The flow of your program
- The computational algorithms, if any
- The functions, subroutines, and subprograms used, if any
- The dependencies on input and output
- The overall logic of the program

Top-Down Design

Approaching your programming problem using the technique of top-down structure with stepwise refinement helps create an application design that can be more efficiently coded and easily tested. Identify the major tasks and components necessary and then repeatedly break these tasks and components down into smaller and smaller pieces.

Modularity

Sometimes your solution will be simple enough to code as a single, self-sufficient program. More often, however, your solution will require more than one program. The coordinated programs that make up the application are bound together as one **run unit** (or **enclave** in LE/VSE terminology).

When deciding on how to break your program down into its pieces or into other programs, you should be concerned primarily with function and secondarily with size and independence.

A simple and easy-to-follow design simplifies the coding, testing, and maintenance of your program. You can simplify your program by grouping major functions and identifying common subfunctions.

Code and Compile Your Program

After creating specifications for your program, the next step is to code and compile a program that meets those specifications. This book guides you through the process of coding and compiling COBOL/VSE programs.

If your COBOL/VSE program needs to call programs written in other LE/VSE-conforming languages, or is to be called by such programs, refer to *LE/VSE Writing Interlanguage Communication Applications* for instructions on using interlanguage communication (ILC).

Code Your Program

Coding a program involves choosing the appropriate COBOL/VSE language statements necessary to accomplish your objectives and entering these source statements into a file.

Programming Style

While there are many ways to code a program that implements your design, you should always keep in mind that future maintenance of your program requires that it be easy to understand and modify. If your programming style follows the principles of structured programming outlined below, your programs will be easier to maintain.

For information on how to implement the following principles in the COBOL/VSE language, see "The Procedure Division" on page 30.

Modularity: The hierarchy of your program should be obvious and oriented to the functions performed by your program. At the highest level is the main program or driver, followed by secondary program units, which could be subprograms, sections, or paragraphs, depending on the size of your application. At the lower levels, some program units might be shared or reused by higher units. In any case, the units of your program generally should not exceed one page of code (about 50 lines).

For instructions on using subprograms, refer to Chapter 16, "Subprograms and Data Sharing" on page 260.

Three Control Structures: Your program should be built using only the three logic structures sequence, selection, and iteration, as follows:

Sequence

Grouped COBOL statements are executed one after another in the order that they appear in the program. There is no change of the control flow, for example, with a GO TO statement, to another part of the program.

Selection

One or more COBOL statements are executed conditionally, depending on the results of a test. For simple conditions, the IF ... THEN ... ELSE sequence provides a two-path alternative (the IF construct). For multiple conditions, the EVALUATE statement provides a number of path options, one of which is taken for each condition (the CASE construct).

Iteration

One or more COBOL statements are executed repeatedly, either:

WHILE a condition remains true (the DO-WHILE construct), or
UNTIL a condition becomes true (the DO-UNTIL construct).

These are implemented using variations of the COBOL PERFORM statement.

One Entry: Each of the program units that make up your program should have only one entry point and return to the place from which it was invoked. Avoid the use of GO TO, ALTER, and ENTRY statements.

Descriptive Comments and Names: A well-structured program clearly describes the functions that it performs, both in the comments you enter as documentation and in the names you give to program units, paragraphs, and data items. These comments and names should reflect function and not implementation details.

Visual Clarity: To enhance visual clarity, blank lines should separate logical constructions, and indentation should be used to show nesting or subordination. For example, sequences of statements embedded in selection or iteration structures should always be indented.

Compile Your Program

After writing a program, you compile the source file (or files, if you have subprograms) to produce an object module(s). When you compile, you can choose what types of listings of your programs the compiler will produce. Figure 7 shows the basic process of compilation.

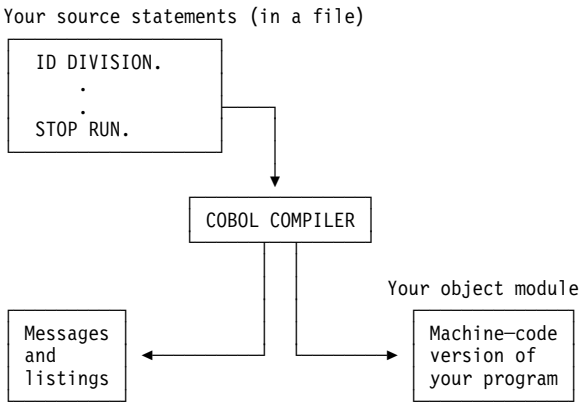


Figure 7. Compilation Process

You can compile several COBOL programs with a single invocation of the compiler using a batch technique.

Part 3, "Compiling Your Program" on page 205 contains detailed instructions for compiling your program.

Link and Run Your Program with Test Data

After compilation, your program will consist of one or more object modules which contain unresolved references to each other, as well as a reference to the LE/VSE run-time library. These references are resolved during link-editing or, dynamically, during execution.

So, after compiling the COBOL/VSE program, the next step is to link and run it with test data to verify it produces the results you expect.

LE/VSE provides the run-time environment and services needed to run your program. Therefore, for instructions on linking and running COBOL/VSE and all other LE-conforming language programs, refer to *LE/VSE Programming Guide*.

Do the Results Meet Specifications?

When you run your program with test data, it might be obvious that its behavior does not meet your specifications or expectations. In this case, you must modify your program as necessary, recompile, relink, and test it again. Assuming your design represents a valid solution to user requirements, your program will be ready for general use when it has successfully handled all the test cases necessary to verify compliance with your specifications. If general use of the program shows that it does not meet user requirements, you may have to change your design.

It is also possible that your program will behave unexpectedly for no apparent reason. In this case, you will need to determine the cause of the problem using debugging techniques and facilities, make the necessary changes, and test your program again. Once you have fixed your program so that it exhibits correct behavior for all the necessary test cases, it is ready for general use.

To discover problems in your program, you can use certain COBOL/VSE language elements and facilities, and analyze storage dumps. See Chapter 18, “Debugging” on page 292 for more information on these various problem determination techniques.

CICS Programs: For CICS programs you can also use the CICS Execution Diagnostic Facility (EDF) to debug your program.

Put Your Program into Use

Once you have verified the results of your program, it is ready for general use. Because LE/VSE provides the run-time environment and services necessary to run your COBOL/VSE program, make sure that the target computer environment in which your program will run has access to LE/VSE.

Part 2. Coding Your Program

This part of the book explains how to accomplish various programming objectives using the COBOL/VSE language. It discusses the most common topics, starting with basic ones and then building on those in succeeding chapters. More complex programming topics are treated in Part 4, “Advanced Topics” on page 259.

Chapter 3. Program Structure

Program specifications are the basis for coding your application program. The data descriptions in your program will match the data described in the specifications; likewise, the logic of your program will match the logic in the specifications. Furthermore, the order in which you code the parts of your program will follow the "top-down" hierarchy established at the design stage.

A COBOL program consists of four divisions, and each division has a specific logical function to perform when solving your data processing problems. The Identification Division is required; the other three (the Environment Division, Data Division, and Procedure Division) are optional.

The following example illustrates the simplest COBOL program that will compile without errors. It contains the only required division header and one other statement:

```
Identification Division.
Program-ID. Miniprog.
```

Although this program does compile, it does not actually do anything. To write meaningful programs, you need to understand all of the COBOL divisions, which are explained in this chapter.

The Identification Division

As shown in the following sample entry, you assign a name to the program and provide other identifying information about it in the Identification Division:

```
Identification Division.
Program-ID. Miniprog.
Author. Peter Programmer.
Installation.  Computing Laboratories.
Date-Written.  08/24/94.
Date-Compiled. 09/26/94.
```

The division header and the PROGRAM-ID paragraph are the only **required** elements.

The PROGRAM-ID paragraph names your program. Other programs use the name assigned by the PROGRAM-ID paragraph to call this program. The name you use in the PROGRAM-ID appears in the header on each page, after the first, of the program listing generated when the program is compiled. If you specify the NAME compiler option, the name you use in the PROGRAM-ID is placed on the Linkage Editor PHASE statement or librarian CATALOG command to identify the object module resulting from the compilation.

You can specify the program attributes COMMON and INITIAL with the PROGRAM-ID clause.

The COMMON attribute specifies that the program may be called by the containing program or by any program within the containing program. However, the COMMON program may not be called by any program contained within itself. Only contained programs may have the COMMON attribute. For more information see "Nested Programs" on page 263.

The INITIAL attribute specifies that whenever a program is called, it is placed in its initial state, and any of its contained programs are also placed in their initial state. Essentially, a program is in its initial state when: data items with VALUE clauses are set to the specified value, changed GOTO and PERFORM statements are set to their initial states, and internal files are closed.

The other Identification Division paragraphs are optional and treated as documentation. You can use them for descriptive information about your program.

The DATE-COMPILED paragraph, for example, inserts the compilation date in place of any comments you enter.

Both the date and time of the compilation are stored in the object code in the format MM/DD/YY HH.MM.SS. This will be the value used in the WHEN-COMPILED Special Register.

Listing Header in the Identification Division

The header on the first page of your source statement listing contains the identification of the compiler and the current release level, the announcement date, the current date, plus the date and time of compilation and the page number. You can change the header on succeeding pages of the listing with the compiler-directing TITLE statement. If you do not use the TITLE statement, the header remains the same throughout the listing. For example:

```
PP 5686-068 IBM COBOL for VSE/ESA 1.1.1                Date 06/16/1998  Time 13:41:27   Page     1
```

For the succeeding pages in the listing, you can change the first 65 characters of this header with the TITLE statement. For example:

```
TITLE 'DATA ENTRY PROGRAM'.
```

The TITLE statement forces a new page and produces the specified header on the new page and succeeding pages of the listing. Besides the title you specify, each new header line contains:

- The name of the program being compiled—as specified on the PROGRAM-ID statement (the name space is blank for pages in the listing that precede the processing of the PROGRAM-ID statement)
- The date and time of the compilation
- The current page number

For example:

```
DATA ENTRY PROGRAM                      IGYCARPB  Date 06/16/1998  Time 13:41:27   Page     2
```

You can specify one header for your entire listing or you can change the header several times throughout the listing.

The title is left-justified at the top of the page. It must be a nonnumeric literal.

The TITLE statement:

- Can be specified in any division (however, it is recommended that you code it as the last statement in the Identification Division)
- Cannot be continued on the next line
- Is **not** printed on the source listing
- Has no effect on the compilation or execution of a program

Errors to Watch for in the Identification Division

To avoid the mistakes most commonly made in the Identification Division, verify the following:

- Is the division header spelled correctly and without hyphens?
- Are all the necessary periods included?
- Are the hyphens properly placed in PROGRAM-ID, DATE-WRITTEN, and DATE-COMPILED?
- Is everything coded in the correct margin or area?

The Environment Division

In the two optional sections of the Environment Division, you can specify the particular computer environment in which you are working. The **Configuration Section** specifies the characteristics of your computer and the **Input-Output Section** relates your program files to the external file names known by the operating system.

Configuration Section

Figure 8 on page 20 shows a sample of some of the entries you might include in the Configuration Section. It is an optional section in which you can:

- Describe the computer
  - On which the source program is compiled
  - On which the object program is executed
- Set status indicators
- Specify the collating sequence
- Specify the character to be used for the currency sign
- Specify the interchange of functions for the comma and the period
- Specify symbolic characters
- Specify a user-defined class

The Configuration Section can only appear in the outermost program of a nested program structure.

Describe the Computer

With the SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs, you define the computer(s) where your program will be compiled and executed. With the SPECIAL-NAMES paragraph, you relate environment-names defined by IBM to mnemonic-names that you define. That is, you associate system devices with mnemonic-names used in ACCEPT and DISPLAY statements.

SOURCE-COMPUTER Paragraph

This paragraph documents the computer on which your program will be compiled.

The WITH DEBUGGING MODE clause specifies that the debugging lines in your program (those statements which are coded with a "D" in column 7) are compiled.

OBJECT-COMPUTER Paragraph

The MEMORY SIZE clause documents the amount of main storage your program uses.

The PROGRAM COLLATING SEQUENCE clause specifies the collating sequence, associating it with a mnemonic-name defined in the SPECIAL-NAMES paragraph.

SPECIAL-NAMES Paragraph

The SPECIAL-NAMES paragraph allows you to specify special system features, such as the name of the alphabet your program uses for program collating sequence.

This paragraph associates system functions with names used in your program, such as ACCEPT/DISPLAY devices, UPSI switches, and WRITE ADVANCING actions. The environment-names you can use for ACCEPT/DISPLAY devices and for WRITE ADVANCING are documented in *COBOL/VSE Language Reference*.

Set Status Indicators

User Programmable Status Indicator (UPSI) switches were used on older computers for processing special conditions at run time. Although this method is still supported, the recommended ways to set status indicators at run time are:

- Read data and code conditional processing based on the value of the data
- Pass run-time parameters in the PARM parameter of the EXEC statement in the job control procedure

These preferred techniques require:

- The USING option of the Procedure Division header

  A description in the Linkage Section of the records that will contain the parameters passed
- The passing of the parameters in run-time JCL

Specify the Collating Sequence

Using the PROGRAM COLLATING SEQUENCE clause and the ALPHABET clause, you can establish the collating sequence to be used in the following operations:

- Nonnumeric comparisons explicitly specified in relation conditions and condition-name conditions
- HIGH-VALUE and LOW-VALUE settings
- SEARCH ALL

- SORT and MERGE unless replaced by a COLLATING SEQUENCE phrase on the SORT or MERGE statement

The sequences used may be based on one of these alphabets:

- EBCDIC (specify NATIVE if the native character set is EBCDIC). This is the default if the ALPHABET clause is omitted
- ASCII (specify STANDARD-1)
- ISO 7-bit code, International Reference Version (specify STANDARD-2)
- A change of the EBCDIC sequence that you define in the SPECIAL-NAMES paragraph

Each separate SORT or MERGE operation can replace the alphabet specified in the PROGRAM COLLATING SEQUENCE clause.

Figure 8 shows the Environment Division coding used to specify a collating sequence where uppercase and lowercase letters are similarly treated for comparisons and for sorting or merging. Please note that 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 included in the SPECIAL-NAMES paragraph.

```
Identification Division.

    ⋮

Environment Division.
  Configuration Section.
   Source-Computer. IBM-370.
   Object-Computer. IBM-370
      Program Collating Sequence Special-Sequence.
   Special-Names.
      Alphabet Special-Sequence Is
          "A" Also "a"
          "B" Also "b"
          "C" Also "c"
          "D" Also "d"
          "E" Also "e"
          "F" Also "f"
          "G" Also "g"
          "H" Also "h"
          "I" Also "i"
          "J" Also "j"
          "K" Also "k"
          "L" Also "l"
          "M" Also "m"
          "N" Also "n"
          "O" Also "o"
          "P" Also "p"
          "Q" Also "q"
          "R" Also "r"
          "S" Also "s"
          "T" Also "t"
          "U" Also "u"
          "V" Also "v"
          "W" Also "w"
          "X" Also "x"
          "Y" Also "y"
          "Z" Also "z".
```

Figure 8. Example of an Alternate Collating Sequence

Specify Currency Sign

The literal specified with the CURRENCY SIGN clause takes precedence over the default currency sign established by the CURRENCY compiler option.

Comma / Period Interchange

The functions of the period and the comma in PICTURE character strings and in numeric literals may be exchanged through use of the DECIMAL-POINT IS COMMA clause.

Specify Symbolic Characters

By using the SYMBOLIC CHARACTER clause, you can give symbolic names to any character of the specified alphabet. For example, to give a name to the back-space character (X'16' in the EBCDIC alphabet) you would code:

```
SYMBOLIC CHARACTERS BACKSPACE IS 23
```

You use ordinal position to identify the character, position 1 corresponds to character X'00'.

Specify a User-Defined Class

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

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

The class name can only be referenced in a class condition.

Input-Output Section

Your COBOL/VSE programs can process either SAM sequential files or VSAM files with sequential, indexed or relative organization. Use the FILE-CONTROL and I-O CONTROL paragraphs to identify and describe the characteristics of your program files, associate them with the external files where they physically reside, and specify information to control efficient transmission of the data records between your program and the external medium.

Note: For CICS programs, code only the Environment Division header and, optionally, the Configuration Section. CICS does not allow COBOL definition of files.

Figure 9 shows examples of FILE-CONTROL entries for a SAM disk file, a SAM tape file, and a VSAM indexed file.

FILE-CONTROL Entry for a SAM Disk File:

```
SELECT PRINTFILE
    ASSIGN TO UPDPRINT
    ORGANIZATION IS SEQUENTIAL
    ACCESS IS SEQUENTIAL.
```

FILE-CONTROL Entry for a SAM Tape File:

```
SELECT TAPE-FILE
    ASSIGN TO SYS006-TAPE1.
```

FILE-CONTROL Entry for a VSAM File:

```
SELECT COMMUTER-FILE
    ASSIGN TO COMMUTER
    ORGANIZATION IS INDEXED
    ACCESS IS RANDOM
    RECORD KEY IS COMMUTER-KEY
    FILE STATUS IS
       COMMUTER-FILE-STATUS
       COMMUTER-VSAM-STATUS.
```

Figure 9. Examples of FILE-CONTROL entries

For both SAM and VSAM files, the SELECT and ASSIGN clauses are required. The SELECT clause chooses a file in the COBOL program to be associated with an external file. The ASSIGN clause then associates the program's name for the file with the external name for the actual data file.

Use the ORGANIZATION clause to describe the file's organization and the ACCESS MODE clause to define the manner in which the records in the file will be made available for processing—sequential, random, or dynamic. For SAM files, both the ORGANIZATION and ACCESS MODE clause are optional. SAM files always have sequential organization. For VSAM files, you may have additional statements in the FILE-CONTROL paragraph depending on the type of VSAM file you are using.

Chapter 9, “File Input/Output Overview” on page 123 provides a general overview on files and file processing. For more specifics on Input-Output Section entries and other details on SAM and VSAM file processing, see Chapter 10, “Processing SAM Files” on page 133, or Chapter 11, “Processing VSAM Files” on page 152.

You need to define to the operating system all files that you process in your COBOL/VSE program. Figure 10 shows the relationship of JCL statements to the FILE-CONTROL and FD entries in your program. For information about JCL statements for file access, see “Job Control Language for SAM Files” on page 142, and “Job Control Language for VSAM files” on page 172.

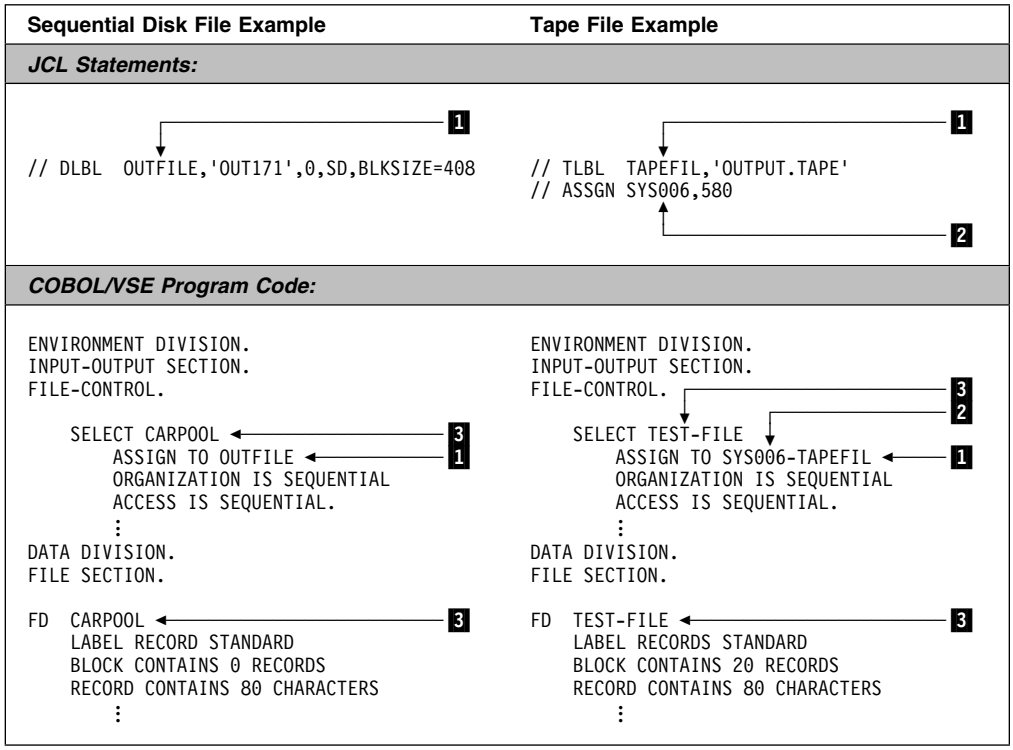


Figure 10. Examples of JCL, FILE-CONTROL Entries, and FD Entries

The numbers in Figure 10 correspond to the numbers below:

1 The *filename* in the JCL corresponds to the *name* in the ASSIGN clause:

```
// DLBL OUTFILE     corresponds to    ASSIGN TO OUTFILE
// TLBL TAPEFIL     corresponds to    ASSIGN TO SYS006-TAPEFIL
```

2 For the tape file, the programmer logical unit in the ASSGN JCL statement corresponds to SYS*nnn* in the ASSIGN clause:

```
// ASSGN SYS006      corresponds to    ASSIGN TO SYS006-TAPEFIL
```

3 When you specify a file in the FILE-CONTROL entry, the file must be described in an FD entry for *file-name* in the Data Division.

```
SELECT CARPOOL      corresponds to    FD CARPOOL
SELECT TEST-FILE    corresponds to    FD TEST-FILE
```

Set Status Keys for Error Handling

Using the FILE STATUS clause, you can set up error-handling procedures for use when a nonzero file status code is returned.

Coding the FILE STATUS clause for each defined file and coding a test for the file status key value after each I/O statement is strongly recommended. For VSAM files, you can use a second *data-name* in the FILE STATUS clause to get VSAM return code, component code and reason code information.

```
FILE STATUS IS data-name-1 [data-name-2]
```

Both *data-names* must be defined in the Data Division.

Note: Neither *data-name* in the FILE STATUS clause can be variably located. (See “Complex OCCURS DEPENDING ON” on page 106 for more information on variably located data items.)

For more information on using the FILE STATUS clause, see “File Status Key” on page 198 and “VSAM Return Code (VSAM Files Only)” on page 201.

Another way of handling input/output errors is to set up error/exception declaratives. For more information on using error declaratives, see “Input/Output Error Handling Techniques” on page 194.

Vary the Input/Output File at Run Time

The *file-name* you specify in your SELECT sentence is used as a constant throughout your COBOL program, while the name of the file on the DLBL or TLBL statement can be associated with a different file at run time.

Changing a *file-name* in your COBOL program requires changing input/output statements and recompiling the program. In contrast, you can change the *file identifier* in the DLBL or TLBL statement at run time.

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

```
SELECT MASTER
    ASSIGN TO SYS011-S-MASTRA
```

The three possible input files are MASTER1, MASTER2, and MASTER3. Therefore, one of the following JCL statements or sets of JCL statements is coded in the job step that calls for program execution:

```
// DLBL MASTRA,'MASTER1',0,SD

// DLBL MASTRA,'MASTER2',0,SD

// TLBL MASTRA,'MASTER3'
// ASSGN SYS011,TAPE
```

Any reference within the program to MASTRA is a reference to the file identifier in the DLBL or TLBL *file identifier*.

The *system-name* portion of the *assignment-name* that appears in the ASSIGN clause and the *filename* of the DLBL or TLBL statement must always be the same. You can vary the file itself in the *file identifier* in the DLBL or TLBL statement.

The *file-name* that follows the SELECT statement (MASTER in the previous example) must be the same as the FD *file-name* entry.

MULTIPLE FILE TAPE Clause

The MULTIPLE FILE TAPE clause is treated as documentation. It is syntax-checked, but has no effect on the execution of the program. For files with standard labels, the function is performed by SAM, the Sequential Access Method. When each file is processed in the sequence stored on the tape, and a rewind is not needed for any file except the last, the function is not needed.

Note: An arbitrary access to a file without labels or with nonstandard labels on a multiple file tape is not supported.

APPLY WRITE-ONLY Clause

The APPLY WRITE-ONLY clause makes optimum use of buffer and device space when creating a sequential file with blocked V-mode records. With APPLY WRITE-ONLY, a buffer is shortened only when the next record does not fit in the unused remainder of the buffer.

Also note that the AWO compiler option puts the APPLY WRITE-ONLY clause on all eligible files.

Without APPLY WRITE-ONLY, a buffer is shortened when there is not enough space remaining in it to accommodate the maximum size record.

This clause has meaning only when the file is opened as OUTPUT or EXTEND and has standard sequential organization.

The Data Division

In the Data Division, you define the characteristics of your data and group your data definitions into one of the three sections within this division:

Data used in input/output operations (File Section)
Data developed for internal processing (Working-Storage Section)
Data from another program (Linkage Section)

To save processing time:

- Group data definitions of constants together.
- Group data definitions of variables together, separately from the constants.

- Place the most frequently referenced data items close together in the Working-Storage Section.
- Define data in a way that avoids unnecessary conversions.

To save yourself time, use the COPY statement in the Data Division.

File Section (Data Used in Input/Output Operations)

The data you use in input and output operations is defined in the File Section. Entries in the File Section are summarized in Figure 11 on page 25. Naming and describing the input and output files is the File Section's primary function. Data items defined in the File Section are never available to Procedure Division statements until the file has been successfully opened.

The FD *file-name* is the name used in the Procedure Division when you OPEN and CLOSE files and READ records from them. (For VSAM files, the *file-name* in the FD entry is also used for a START or DELETE request.)

The record description following the FD entry describes the fields of the records in the file. You can code this as a level-01 description of the entire record. In the Working-Storage Section, you can code a "working copy" that describes the fields of the record.

The *record-name* established is the object of WRITE and REWRITE statements.

For **SAM files only**, you can specify one of the following:

- The record format in the RECORDING MODE clause. Without this specification, the compiler determines the record format based on the RECORD clause and on the level-01 record descriptions. For more information on record format and length, see "Data Division Entries for SAM Files" on page 134.
- A blocking factor for the file in the BLOCK CONTAINS clause. If the BLOCK CONTAINS clause is omitted, the file will be unblocked.

For **VSAM files**, the BLOCK CONTAINS clause is treated as a comment.

Figure 11 (Page 1 of 2). File Section Entries

Clause	To Specify	Notes
FD	The *file-name* to be referred to in Procedure Division input/output statements	Must match *file-name* in the SELECT clause
BLOCK CONTAINS	Size of physical record	SAM: If equal to 0, BLKSIZE must be specified on JCL, or, for files defined using the VSE/VSAM Space Management for SAM Feature, in the VSAM catalog. BLOCK CONTAINS 0 should only be specified for files assigned to direct-access storage devices (DASD). The BLOCK CONTAINS clause is ignored for files assigned to unit record devices. VSAM: Treated as comment

Figure 11 (Page 2 of 2). File Section Entries

Clause	To Specify	Notes
RECORD CONTAINS	Size of logical records (fixed or variable length)	If specified for an input file assigned to a DASD device and defined using the VSE/VSAM Space Management for SAM Feature, must match the record size in the VSAM catalog. Alternatively, RECORD CONTAINS 0 may be used for such files. The RECORD CONTAINS 0 should not be used for files other than those described above.
RECORD IS VARYING	Size of logical records (variable length)	If specified must match information on JCL or file label; compiler checks match with record descriptions
LABEL RECORDS	Discussed under "Processing Labels for SAM Files" on page 148	VSAM: Treated as comment
STANDARD	Labels exist	SAM: Treated as comment
OMITTED	Labels do not exist	SAM: Treated as comment
data-name	Labels defined by the user	SAM: 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	SAM only
CODE-SET	ASCII or EBCDIC files	SAM only
RECORDING MODE	Physical record description	SAM only

The FD File Name

The FD *file-name* must match the SELECT *file-name*. Through the *assignment-name*, the *file-name* is associated with the *system-name* of the file.

Record Descriptions in the File Section

The record description following the FD *file-name* describes the logical records in the file, following the COBOL rules for data description (level numbers, picture clause, and so forth).

A common programming practice is to keep the FD input and output record descriptions at the 01 level and code more detailed data descriptions (with the fields used in processing) in Working-Storage. READ INTO is used to bring the records into Working-Storage. Processing occurs on the copy of data in Working-Storage. A WRITE FROM statement then writes processed data into the record area defined in the File Section.

For variable-length records, use READ instead of READ INTO. By specifying a READ INTO statement for a file in format U, V, or S, the record size just read for that file is used in the compiler-generated MOVE statement. Consequently, you may 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.

The record length must correspond to the record length of the file.

The record length must allow for the control character if you are going to specify the NOADV compiler option; otherwise the record length should be the same as that of the record to be printed.

The CODE-SET Clause

For SAM files, the CODE-SET clause of the FD statement can be used to identify the file as being either EBCDIC or ASCII.

Sharing Files Using the EXTERNAL and GLOBAL Clauses

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).

EXTERNAL: Is used for separately compiled programs. A file that is defined as EXTERNAL can be referenced by any program in the run unit which describes that file. See “Input-Output Using EXTERNAL Files” on page 127 for an example.

GLOBAL: Is used for programs in a nested, or contained, structure. If a program contains another program (directly or indirectly), both programs can access a common file by referencing a GLOBAL file name. For more information on contained programs and the GLOBAL clause, see “Nested Programs” on page 263.

Working-Storage Section (Data Developed for Internal Processing)

You can write a program that processes data without performing any input/output operations. In such a program, all the data is defined in the Working-Storage Section. Most programs, however, have a combination of input and output file processing and internal data manipulation; the data files are defined in the File Section and the data developed by the program is defined in the Working-Storage Section.

Initialize all the data in the Working-Storage Section before you use it. For methods of initializing data, see “Initializing a Variable (INITIALIZE Statement)” on page 45.

Data read from input files is quite often copied into the Working-Storage Section before processing begins (using the READ INTO statement).

A CALL statement with the USING option is more efficient when a single item is passed than when many level-01 items are passed.

Several other techniques help to make working with Working-Storage data in your program easier and to make passing this data to subprograms more efficient:

- Keep indentation consistent
- Group data entries
- Use standard data item names
- Use meaningful prefixes and suffixes
- Use the EXTERNAL clause for data items passed to separately compiled subprograms
- Use the GLOBAL clause for data items passed to contained subprograms

Keep Indentation Consistent

The key purpose of indentation is to help the reader understand program relationships and functions. Some ways to do this in the Data Division are:

- Begin all PICTURE clauses in the same column The choice of column depends on the length of the longest *data-name* and on the depth of the level structure. Starting all PICTURE clauses somewhere between columns 32 through 45 is reasonable.
- Indent continuation lines. This makes it clear that they are part of the same entry. Use a consistent number of spaces, such as 4.
- Highlight record structure by indenting each successive level by 2 to 4 spaces.
- Group individual data items (items that are not part of records) under a higher level. For example, group individual items that are not part of other records under level-01 entries with descriptive names, such as FLAGS or ERROR-MESSAGES.

In addition to indentation, use blank lines to make record groupings clearer.

Conventions vary from one programming organization to another. You should follow the standards in effect for your organization.

Group Data Entries

Group entries under one or only a few level-01 *data-names*. Then use level-05 and greater to describe logical record areas.

Use widely incremented level numbers like 01, 05, 10, 15, instead of 01, 02, 03, 04, to allow room for future insertions of group levels.

Use level-88 *condition-names*. Then, if the conditional values must be changed, the Procedure Division coding for conditional tests does not need to be changed.

Use Standard Data Item Names

Some general rules to follow in naming your data items are:

- Make sure they conform to internal standards
- Do not use reserved words
- Select meaningful names

Use Meaningful Prefixes or Suffixes

Careful use of prefixes and suffixes helps to:

- Make groups and subordination clear
- Distinguish between similar record-fields
- Make record and field-names meaningful

Make groups and subordination clear.

Assign a prefix or suffix for each group item. Use this prefix or suffix on every subordinate item (except FILLER) to associate a file with its records and work areas.

This technique makes it easier for a person unfamiliar with the program to find fields in the program listing and to determine which fields are logically part of the same record or area. For example:

```
01 STATUS-AREA.
   05 COMMUTER-FILE-STATUS        PIC X(2).
   05 COMMUTER-VSAM-STATUS.
      10 VSAM-R15-RETURN-CODE     PIC 9(2) COMP.
      10 VSAM-FUNCTION-CODE       PIC 9(1) COMP.
      10 VSAM-FEEDBACK-CODE       PIC 9(3) COMP.
```

Distinguish between similar record-fields.
If, for example, three files all have a date field, instead of DATE1, DATE2, and DATE3, use MASTER-DATE, DETAIL-DATE, and REPORT-DATE.

Make record and field-names meaningful.
One convention for making *data-names* meaningful is prefixing. For example:

Prefix	For Names of Items in
T	Transaction files
M	Old master files
NM	New master files

Another convention is to make the suffix or prefix of all file names different from all record names. For example:

Make the prefix of all *file-names* **FILE**
Make the suffix of all *record names* **RECORD** or **REC**.

Using such conventions consistently throughout your program helps readers understand the structure and function of the program.

Use the EXTERNAL clause

By using the EXTERNAL clause, any program in the run unit that includes declarations for data items may access the data items. Only level-01 items may be specified as EXTERNAL. For more details, see “Sharing Data Using the EXTERNAL Clause” on page 281.

Use the GLOBAL clause

The GLOBAL clause allows the data items to be accessed by any subprogram contained within the program that includes the declarations. Only level-01 items may be specified as GLOBAL. For more details, see “Nested Programs” on page 263.

Linkage Section (Data from Another Program)

Separately Compiled Programs

Many times an application's solution consists of many, separately compiled programs that call and pass data to one another. The Linkage Section in the called program describes the data that is passed from another program. The calling program must use a CALL ... BY REFERENCE or BY CONTENT statement to pass the data. For details on using data from other programs, see “Passing Data BY REFERENCE or BY CONTENT” on page 273.

Nested Programs

An application's solution may also consist of nested programs—programs that contain other programs. Level-01 Linkage Section data items may be specified with the GLOBAL clause. This allows Linkage Section data items to be accessed by any nested program that includes the declarations. For more details, see "Nested Programs" on page 263.

Limits in the Data Division

The COBOL/VSE compiler limits the maximum size of data division elements. For a complete list of these compiler limits, see Appendix A, "COBOL/VSE Compiler Limits" on page 398.

The Procedure Division

In the Procedure Division, you code the executable statements that process the data you have defined in the other divisions. The Procedure Division contains the logic of your program.

The Procedure Division begins with the division header and a *procedure-name* header. It is divided into sections, paragraphs, sentences, and statements:

A **section** is a logical subdivision of your processing logic. A section may contain several paragraphs.

A **paragraph** subdivides a section, procedure, or program. It contains a set of related statements that provide a function and is one of the basic building blocks of a structured program. A paragraph can be the subject of the PERFORM statement.

A **sentence** is a series of one or more COBOL statements ending with a period. Many structured programs do not have separate sentences. Each paragraph may contain one sentence. Using scope terminators instead of periods to show the logical end of the scope of a statement is preferred. Scope terminators, both explicit and implicit, are discussed beginning on page 33.

A **statement** performs a defined step of COBOL processing, such as adding two numbers. A statement is a syntactically correct combination of words, beginning with a COBOL verb.

Procedure Division Structure

In structured programming, the language implementation techniques in COBOL apply to the Procedure Division.

In the COBOL/VSE language, there are four categories of statements:

Imperative
Conditional
Compiler-directing
Delimited scope

Imperative Statements

An imperative statement specifies that an unconditional action is to be taken. The statements in the following table are imperative when they are used without any conditional phrases (such as ON EXCEPTION, NOT ON EXCEPTION, AT END, NOT AT END, and so on). Additionally, conditional statements that are terminated by their explicit scope terminators are classified as imperative.

Figure 12. COBOL Statements Causing Unconditional Action to be Taken

Type of Imperative Statement	COBOL Statement
An arithmetic statement	ADD COMPUTE DIVIDE INSPECT (TALLYING) MULTIPLY SUBTRACT
A data-manipulation statement	ACCEPT (DATE, DAY, DAY-OF-WEEK, TIME) INITIALIZE INSPECT (REPLACING or CONVERTING) MOVE SET STRING UNSTRING
A procedure-branching statement	ALTER CALL EXIT EXIT PROGRAM GO TO PERFORM STOP STOP RUN
An I/O statement	ACCEPT (SYSIN, CONSOLE) CLOSE DELETE DISPLAY OPEN READ REWRITE START WRITE
One of the following miscellaneous statements	CANCEL CONTINUE ENTRY MERGE RELEASE RETURN SORT

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.

For example, an arithmetic statement without ON SIZE ERROR is an imperative statement. But an arithmetic statement with the conditional option ON SIZE ERROR and without a scope terminator is a conditional statement. Data-manipulation statements or CALL statements with ON OVERFLOW and I/O statements with INVALID KEY, AT END, AT END-OF-PAGE, or RETURN with AT END and without a scope terminator are all conditional statements.

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

Compiler-Directing Statements

A compiler-directing statement is not part of the program logic. Compiler-directing statements inform the compiler about the program structure, copy processing, listing control, and control flow. The following statements are considered compiler directing:

BASIS	REPLACE
*CBL	SERVICE LABEL
*CONTROL	SKIP1
COPY	SKIP2
DELETE	SKIP3
EJECT	TITLE
ENTER	USE
INSERT	

Delimited Scope Statements

In general, a delimited scope statement uses an explicit scope terminator to turn a conditional statement into an imperative statement; the resulting imperative statement can then be nested. Explicit scope terminators may also be used, however, to terminate the scope of an imperative statement and clearly end the statement. Explicit scope terminators are provided for certain COBOL verbs and are listed under “Explicit Scope Terminators” on page 33.

Because a period implicitly terminates the scope of all previous statements, do not use a period within a delimited scope statement.

Unless specified otherwise, a delimited scope statement may be specified wherever an imperative statement is allowed by language rules.

- Use a delimited scope statement 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 instead of a period to terminate the scope of an IF statement within a nested IF.

- Use a delimited scope statement if you want 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 in-line 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 in-line PERFORM statement, but is **invalid** for the out-of-line PERFORM statement.

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

Be careful when coding an explicit scope terminator for an imperative statement that is nested within a conditional statement. You must 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, instead of the first, as intended by the programmer.

```
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 could be recoded in one of the following ways:

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

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

Explicit Scope Terminators

An explicit scope terminator marks the end of certain Procedure Division statements. Explicit scope terminators may be used with both the conditional and imperative forms of these statements.

The following are explicit scope terminators:

END-ADD	END-IF	END-SEARCH
END-CALL	END-MULTIPLY	END-START
END-COMPUTE	END-PERFORM	END-STRING
END-DELETE	END-READ	END-SUBTRACT
END-DIVIDE	END-RETURN	END-UNSTRING
END-EVALUATE	END-REWRITE	END-WRITE

Example of Using Explicit Scope Terminators:

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

Implicit Scope Terminators

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

An unterminated conditional statement may not be contained by another statement. Except for nesting statements with IF statements, nested statements must be imperative statements and must follow the rules for imperative statements. You should not nest conditional statements.

Example of Using Implicitly Terminated Statements:

```
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.
```

Each of the two periods in the above program fragment terminate the IF statements, making the code equivalent to the following example which has 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
END-IF
IF ITEM = "B"
  ADD 2 TO TOTAL
END-IF
```

Explicit scope terminators make a program easier to understand and prevent the unintentional termination of statements that an implicit terminator may cause. For instance, 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.
```

In this case, the two statements:

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

will be executed 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 termination of statements, you should use explicit scope terminators instead of implicit scope terminators, especially within paragraphs. You should only use implicit scope terminators at the end of a paragraph or the end of the program.

Scope of Statements

Statements that include explicit scope terminators are termed delimited scope statements. (See “Delimited Scope Statements” on page 32, “Explicit Scope Terminators” on page 33, and “Implicit Scope Terminators” on page 34.)

When statements are nested within other statements, a separator period that terminates the sentence **also** terminates all nested statements.

When a delimited scope statement is nested within another delimited scope statement with the same verb, each explicit scope terminator terminates the statement initiated by the most recent, and as yet unterminated, occurrence of that verb.

For statements nested within statements allowing optional conditional phrases, any optional conditional phrase encountered is considered the nearest preceding unterminated statement. If no phrase has been associated with it, then it can be associated with the scope terminator.

Structured Programming Practices

The following specific structured programming practices are suggested for use in the Procedure Division:

- To improve readability:
  - Limit paragraphs to one listing page
  - Write paragraph and section names on separate lines
  - Indent to show program logic
  - Align the PICTURE clauses
  - Put the VALUE clause on a separate line, if it is long
  - Use in-line PERFORM statements if the paragraphs are short
- To end the scope of your statements, use explicit scope terminators instead of periods.
- To be sure you have included all necessary functions, compare your code to the design documents.
- To achieve top-down programming:

- Avoid the PERFORM . . . THRU structure unless you use the THRU option to an EXIT for a paragraph. (You should enter a program at the beginning of a paragraph and exit at its end.)
- Avoid skipping around in the code with GO TO statements

  The GO TO statement transfers control to **one** other paragraph, and the GO TO . . . DEPENDING ON statement transfers control to **one of several** different paragraphs, on the basis of the value of the control item you define. Errors caused by a GO TO used with DEPENDING ON are difficult to debug, because, during execution, it may be difficult to determine the value the control item had at any particular time.

  The difference between a GO TO branch and a PERFORM is that PERFORM returns control to the statement that follows the PERFORM imperative statement. In contrast, a GO TO branches to another part of the code and stays there.
- Avoid the ALTER statement

  ALTER can be used to change the name of a paragraph to which a GO TO statement transfers control. If you must use this statement (it is not recommended that you do), keep a tally to indicate whether or not the ALTER statement has been executed. Otherwise, errors caused by the statement are difficult to debug.
- Avoid STOP with *literal*

  STOP with *literal* suspends execution temporarily and sends a message (the *literal*) to the system operator. This use is not recommended in an application program.

COBOL Tools for Structured Programming

COBOL/VSE offers several language elements that help support a structured approach in your Procedure Division coding:

- EVALUATE statement—permits “case” constructions
- In-line PERFORM statement—permits “do” constructions
- TEST BEFORE and TEST AFTER in the PERFORM statements—function as “do-while” and “do-until” constructions
- Scope terminators—permit nesting of structured programming constructions (see “Delimited Scope Statements” on page 32)

For language syntax, see *COBOL/VSE Language Reference*.

EVALUATE Statement

The case structure is implemented in COBOL/VSE by the EVALUATE statement. For example:

```
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
      IF CARPOOL-SIZE >= 7 THEN
        MOVE "BIG GROUP" TO PRINT-CARPOOL-STATUS
      END-IF
    END-IF
  END-IF
END-IF
```

In the EVALUATE statement, expressions to be tested are called selection-subjects. In the example above, `CARPOOL-SIZE` is the selection-subject. The answer selected is called a selection-object. When evaluated, each pair of selection-subjects and selection-objects must belong to the same class (numeric, character, CONDITION TRUE or FALSE).

Use the EVALUATE statement to select from a set of processing actions. Using EVALUATE, you specify a condition to be evaluated and select a processing action based on that evaluation. You can specify up to 255 evaluate subjects and objects in an EVALUATE statement. There is no limit to the number of WHEN clauses that can be specified in an EVALUATE statement, but one page is a practical limit to observe.

The compiler looks at the first WHEN condition. If this condition is satisfied, the processing actions associated with this phrase are selected.

The preceding example of the EVALUATE statement shows that when several conditions evaluate to a range of values and each condition leads to the same processing action, you can use the THRU phrase to easily implement this logic. Alternatively, you can also use multiple WHEN statements when several conditions lead to the same processing action. Multiple WHEN statements give you more flexibility for specifying the same processing action for conditions that do not evaluate to values that fall within a range or evaluate to alphanumeric values.

For the following EVALUATE statement:

```
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
```

The execution of the EVALUATE statement ends when:

- The statements associated with the selected WHEN phrase are executed
- The statements associated with the WHEN OTHER phrase are executed
- No WHEN conditions are satisfied

WHEN phrases are tested in the order they were coded. Therefore you should order these phrases with optimum performance in mind. The WHEN phrase containing selection-objects most likely to be satisfied should be coded first. Code the other WHEN phrases in descending order of probability of satisfaction occurrence—except, of course, the WHEN OTHER phrase, which must come last.

With EVALUATE statements, you can test several conditions and specify a different action for each. For example, in Figure 13 on page 39, both selection- subjects in a WHEN phrase must satisfy the TRUE condition before the phrase is executed. If both subjects do not evaluate to TRUE, the next WHEN phrase is processed.

```
Identification Division.
  Program-ID. MiniEval.
Environment Division.
  Configuration Section.
  Source-Computer. IBM-370.
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.
```

Figure 13. EVALUATE Statement Example

In-Line PERFORM Statement

The traditional out-of-line PERFORM statement requires an implicit branch to a separate paragraph and an implicit return (see “Iterative Loops (PERFORM Statement)” on page 120). If the performed paragraph is in the subsequent sequential flow, it will be executed one more time. To avoid this additional execution, the paragraph is placed outside the normal sequential flow, as shown in Figure 14 on page 40. (The performed paragraph can be thought of as an internal subroutine.)

In structured programming, using an **in-line** PERFORM statement, the paragraph performs one logical function. For readability, this paragraph should be contained on one listing page (about 50 lines or less).

The subject of an in-line PERFORM must be an imperative statement. Therefore, statements other than imperative statements within an in-line PERFORM must be coded with their scope terminators, shown also in Figure 14.

```
Perform 100-Initialize-Paragraph
Read Update-Transaction-File Into WS-Transaction-Record
  At End
    Set Transaction-EOF To True
End-Read

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
  Perform 410-Re-Initialize-Fields
  Read Update-Transaction-File Into WS-Transaction-Record
    At End
      Set Transaction-EOF To True
  End-Read
End-Perform
```

Figure 14. In-Line and Out-of-Line PERFORM Statements

The choice of whether to put a PERFORM statement in-line or out-of-line depends on several factors:

- Is it performed from several places?

  Use out-of-line PERFORM when you perform the same piece of code from several places in your program.

- Which will be easier to read?

  If the PERFORM is put in-line, will the logical flow of the program be less clear because the logical portions of the program flow over several pages? One use of the in-line PERFORM that would violate structured programming practices would be to let a paragraph flow over several pages.

  If, however, the PERFORM paragraph is short, an in-line PERFORM may save the trouble of skipping around in the listing.

- Which makes sense, given the efficiency trade-offs?

  An in-line PERFORM paragraph does not require branching.

  But remember, PERFORM coding can improve code optimization, so efficiency concerns should not be overemphasized.

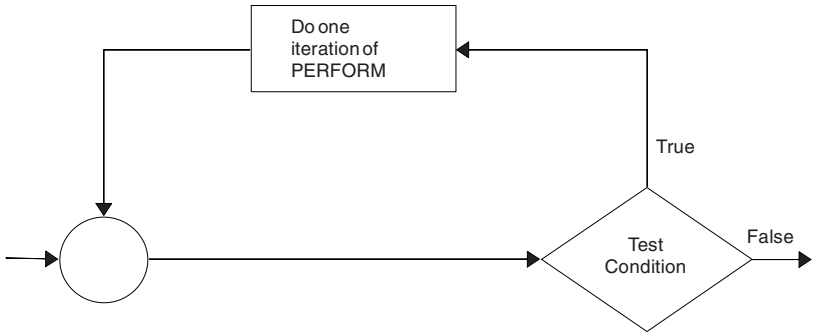
TEST BEFORE or TEST AFTER Loop

The traditional COBOL PERFORM statement allowed just one type of test. The condition to terminate the “do loop” was always tested before the loop was entered. If the test condition was false, the loop was not executed even once. (In structured programming terminology, this was a “do-while” loop.)

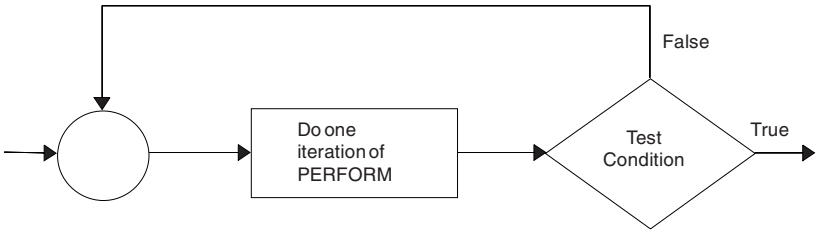
Because you can now use the TEST BEFORE or TEST AFTER phrase with the PERFORM statement, you can choose to have your test either before the loop entry or after it.

With TEST AFTER, the loop is executed the first time—regardless of the condition.

TEST BEFORE corresponds to “do-while.”



TEST AFTER corresponds to “do-until.”



COBOL Tools for Top-Down Coding

The following tools are available to encourage top-down coding:

- PERFORM statements for program modules

  For small program modules that are not shared among programs, you can write the lower-level paragraphs as stubs. During early program development, PERFORM statements can point to these stubs.

- Nested COPY statements

  The COPY statement in COBOL/VSE allows you to nest COPY statements to any depth, and you can code them in any program division. You can write and debug the high-level modules of your program first. These high-level modules contain COPY statements that point to unwritten stubs that “stand in” for the lower-level code.

  Because COPY statements can be nested, you can code COPY statements at every code sequence level in your program. At a later stage in coding, these same COPY statements can point to the completed code sequence.

- CALL statements

  For large and complex programs, use the CALL statement to separate logically distinct portions of your program into called programs. These called subprograms are high-level modules of your application program. The CALL statement executes the subprograms.

  Each subprogram is a complete program within itself. You develop and code the highest-level control modules first. In these control modules you can code statements that invoke the next lower-level subprograms. These subprograms can be stubs that will later be developed into complete subprograms.

Chapter 4. Data Representation and Assignment

A simple COBOL program might process data entered only from the terminal, whereas most COBOL programs process data from files. Whether your data comes from the terminal or from a file, your program must have a way of referring to each item of data as it processes.

In addition, items of information that are constant for each run of the program, as well as items that are derived from manipulation of other items or received as feedback from other software systems, must also be represented in a way suitable for processing.

This chapter introduces COBOL/VSE fundamentals for representing items of data and assigning values to those items. Subsequent chapters discuss the details of specific data types and the processing techniques associated with each.

For complete definitions of each of the language elements and rules described below, see *COBOL/VSE Language Reference*.

Variables, Structures, Literals, and Constants

The concept of data being represented as variables, structures, literals, and constants is fairly universal in high-level programming languages. Although the terminology might be different in COBOL, such data representations are used.

Variables (Data Items)

Because the idea behind writing programs is to create a process that can be used repeatedly for different sets of data, you will want to represent many of the items your program deals with as variables (or in COBOL terminology, data items). For example, your variables might be customer names, employee addresses, or inventory parts. You refer to a variable by a *data-name*, which you define in the Data Division of your program:

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

The data used in a COBOL program can be divided into three classes—alphabetic, alphanumeric, and numeric. For complete details on the classes and categories of data and the PICTURE clause rules for defining data, see *COBOL/VSE Language Reference*. The discussion of “The Data Division” on page 24 also provides additional information.

Structures (Group Items and Records)

Related data items are often parts of a larger data structure, and these hierarchic structures are defined in the Data Division. A data item that includes subordinated data items is called a group item. An elementary data item is a data item that does not have any subordinate items and has a PICTURE clause. A record can be either an elementary data item or a group of data items.

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

```
Data Division.
File Section.
FD  Customer-File
    Recording Mode is F
    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
⋮
```

For more information on defining records and group items, refer to the discussion of “The Data Division” on page 24.

Literals

There are some cases when you know the value you want to use for a data item. In these cases, you do not need to define or refer to a data-name; you simply use a literal representation of the data value in the Procedure Division.

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

```
Move "Invalid Data" To Customer-Name
```

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

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

In these examples, "Invalid Data" is a nonnumeric literal, and 03519 is a numeric literal.

Constants (Data Items with a VALUE)

There is no special COBOL construct specifically for constants, but most programmers simply define a data item with an initial VALUE (as opposed to initializing a variable using the INITIALIZE statement):

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

Figurative Constants

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

For example:

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

This statement fills the entire length, as defined in the Data Division, with spaces.

Assignment and Terminal Interactions

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

- To establish a constant, use the VALUE clause in the definition of the data item. See the previous section, “Constants (Data Items with a VALUE),” for some examples.
- To assign values to a variable or large data area, use one of the following methods:
  - INITIALIZE statement as explained later in this chapter
  - MOVE statement as explained later in this chapter
  - STRING or UNSTRING statement as discussed in Chapter 5, “String Handling” on page 54
  - VALUE clause as discussed in “Assigning Values When You Define the Table (VALUE Clause)” on page 100
- To replace characters or groups of characters in a data item, use the INSPECT statement which is discussed in “Tallying and Replacing Data Items (INSPECT Statement)” on page 62.
- To receive input values from the terminal, use the ACCEPT statement. See “Assigning Terminal Input to Variables (ACCEPT Statement)” on page 48.
- To receive input values from a file, use the READ (or READ INTO) statement. This subject is discussed in Chapter 9, “File Input/Output Overview” on page 123.

- To assign the results of arithmetic, use the COMPUTE statement or one of the other arithmetic statements listed in the table on page 31.

Initializing a Variable (INITIALIZE Statement)

The following examples illustrate some uses of the INITIALIZE statement. (In these examples, the symbol b indicates a space.)

Example 1:

```
INITIALIZE identifier-1
```

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

Example 2:

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

ALPHABETIC-1	ANJUST Before	ANJUST After
ABCD	bbbbbbbb	bbbbABCD

Example 3:

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

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

Example 4:

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

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

Example 5:

```
01 ALPHANUM-EDIT-1         PIC XXBX/XXX.
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	ABbC/DEF	M/bb/bbb

Initializing a Structure (INITIALIZE Statement)

Resetting the values of subordinate items by initializing the group item is an advantage of the INITIALIZE statement. The following example shows how you can reset fields in a transaction record produced by a program to spaces and zeros.

Note: The fields are not identical in each record produced.

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

	TRANSACTION-OUT Before	TRANSACTION-OUT After
Record 1	R00138300024000000000000000	b00000000000000000000000000
Record 2	R00139000048000000000000000	b00000000000000000000000000
Record 3	S00141000012000000000000000	b00000000000000000000000000
Record 4	C00138300000000425000000000	b00000000000000000000000000
Record 5	C00201000000000000100000000	b00000000000000000000000000
Note: The symbol b represents a blank space.		

Assigning Values to Variables or Structures (MOVE Statement)

Assignment in the most common sense of the term is accomplished with the MOVE statement. For example, the following statement:

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

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

In the case of variables containing numbers, moves can be more complicated since there are several ways numbers are represented. These are discussed in Chapter 6, “Numbers and Arithmetic” on page 73. In general, the *algebraic values* of numbers are moved if possible (as opposed to the digit-by-digit type of move performed with character data):

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

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

The compiler assumes you know what the description of the item is that you are moving. Therefore, it will perform all MOVE statements regardless of whether items “fit,” even if that means a destructive overlap could occur at run time. In such cases, you will get a warning message when you compile your program. Therefore, when you move a group item to another group item, be sure the subordinate data descriptions are compatible.

For example:

```
Data Division.
File Section.
FD  Customer-File
    Recording Mode is F
    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).
 ⋮
Procedure Division.
 ⋮
Move Customer-Name To Orig-Customer-Name
 ⋮
```

Assigning Terminal Input to Variables (ACCEPT Statement)

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

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

Then the statement:

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

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

Displaying Data Values (DISPLAY Statement)

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

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

would display this message:

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

Where the DISPLAY Output Goes

The UPON phrase of the DISPLAY statement allows you to direct the output to an output device such as the terminal or the system logical output device. You can use UPON CONSOLE to direct the output to the terminal. For example:

```
Display Record-Count " records read from file." Upon Console.
```

If you omit the UPON phrase, the output is directed to the system logical output device, which is the filename specified in the OUTDD compiler option. When this is the same as the filename specified in the LE/VSE MSGFILE run-time option, the output from the COBOL DISPLAY statement is directed to the LE/VSE message file. For more information, see the *LE/VSE Programming Guide*.

Assigning Arithmetic Results

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

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

However, when significant left-order digits would be lost in execution, the COMPUTE statement can detect the condition and allow you to deal with it, whereas the MOVE statement carries out the assignment with destructive truncation. When you use the ON SIZE ERROR phrase of the COMPUTE statement, the compiler generates code to detect a size-overflow condition. If the condition occurs, the code you specified in the ON SIZE ERROR phrase is executed, and the content of z remains unchanged. If the ON SIZE ERROR phrase is not specified, the assignment is carried out with truncation. There is no ON SIZE ERROR support for the MOVE statement.

Assigning Results of COBOL/VSE and LE/VSE Calculations (COMPUTE Statement)

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

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

For information on intrinsic functions, see section “Built-in (Intrinsic) Functions.”

Results of date, time, mathematical calculations and other operations can be assigned to data items using LE/VSE callable services. These LE/VSE services are available via a standard COBOL CALL statement, and the values they return are passed in the parameters in the CALL statement. For example, you can invoke the LE/VSE service CEESIABS to find the absolute value of a variable with the statement:

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

As a result of this call, the variable `Result` is assigned to be the absolute value of the value that is in the variable `Arg`; the variable `Feedback-code` contains the return code indicating whether the service completed successfully. You have to define all the variables in the Data Division using the correct descriptions, according to the requirements of the particular callable service you are using. For the example above, the variables could be defined like this:

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

For an overview of LE/VSE callable services available from COBOL/VSE, see “LE/VSE Callable Services” on page 347.

For detailed information on each LE/VSE callable service such as syntax, parameter descriptions, usage notes, and examples, refer to the discussion on callable services in the *LE/VSE Programming Reference*.

Built-in (Intrinsic) Functions

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

For example:

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

Compute A = 1 + Function Log10(x)

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

The groups of highlighted words in the examples are referred to as **function identifiers**. A function identifier is the combination of the COBOL reserved word "function" followed by a **function-name** (such as MAX), followed by any arguments to be used in the evaluation of the function (such as x, y, z). As such, a function identifier represents both the function's invocation and the data value returned by

the function. Since it actually represents a data item, a function identifier can be used in most places in the Procedure Division where a data item having the attributes of the returned value can be used. For exact information on where function identifiers can be used, see *COBOL/VSE Language Reference*.

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. You define only the nonliteral data items that you use as arguments. Figurative constants are not allowed as arguments.

Function identifiers are loosely spoken of in this book as function references. Whereas the COBOL word "function" is a reserved word, the names of the various functions (function-names) are not reserved—you can use them in other contexts, such as for the name of a variable and without references to a function. For example, you could use "SQRT" to invoke an intrinsic function and/or to name a variable in your program:

```
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)
 ⋮
```

Note: Functions are not allowed when the CMPR2 compiler option is in effect.

A function identifier represents a value that is either a character string (alphanumeric data class) or a number (numeric data class) depending on the type of function. The functions MAX, MIN, DATEVAL, and UNDATE can return either type of value depending on the type of arguments you supply. Figure 15 on page 52 lists the functions available with COBOL/VSE according to their function types—alphanumeric, numeric, or either. The function type defines the data type of the function result and where the function can be referenced. See the *COBOL/VSE Language Reference* for individual descriptions of each function, specific requirements for their arguments, and allowable places to reference them.

Three functions, DATEVAL, UNDATE, and YEARWINDOW are provided with the millennium language extensions to assist with manipulating and converting windowed date fields. For details on the millennium language extensions, see Chapter 22, "Using the Millennium Language Extensions" on page 366. The three functions are described individually in the *COBOL/VSE Language Reference*.

Figure 15. Function Types

Alphanumeric	Numeric	Alphanumeric or Numeric
CHAR CURRENT-DATE LOWER-CASE REVERSE UPPER-CASE WHEN-COMPILED	ACOS ANNUITY ASIN ATAN COS DATE-TO-YYYYMMDD DATE-OF-INTEGER DAY-OF-INTEGER DAY-TO-YYYYDDD FACTORIAL INTEGER INTEGER-OF-DATE INTEGER-OF-DAY INTEGER-PART LENGTH LOG LOG10 MEAN MEDIAN MIDRANGE MOD NUMVAL NUMVAL-C ORD ORD-MAX ORD-MIN PRESENT-VALUE RANDOM RANGE REM SIN SQRT STANDARD-DEVIATION SUM TAN VARIANCE YEAR-TO-YYYY YEARWINDOW	DATEVAL MAX MIN UNDATE

Nesting Functions

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

For example:

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

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

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

Substrings of Function Identifiers

You can include a substring specification (reference modifier) in your function identifier for alphanumeric functions. Chapter 5, “String Handling” on page 54 discusses these details and some uses for alphanumeric functions.

Additional Information on Intrinsic Functions

In addition to the three general categories of intrinsic functions shown in Figure 15 on page 50, numeric intrinsic functions are further classified according to the type of numbers they return. For these details and information on some uses for numeric functions, see “Numeric Intrinsic Functions” on page 83.

For information on using intrinsic functions in table handling, see “Processing Table Items (Intrinsic Functions)” on page 113. This discussion includes a description of the efficient ALL subscript feature that enables you to easily reference all of the elements of an array as function arguments.

Arrays (Tables) and Pointers

Representing data as an array of elements and referring to data by the address of the data (pointers) are fairly universal concepts in high-level programming languages. These concepts are outlined below.

Arrays (Tables)

In COBOL, arrays are called tables, and there are language constructs available for representing and handling tables. This subject is treated comprehensively in Chapter 7, “Handling Tables (Arrays)” on page 94, including the assignment of values to arrays and considerations for moving variable-length items.

Pointers

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

Pointer data items can be:

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

You can use pointer data items when you want to accomplish limited base addressing, particularly when you want to pass and receive addresses of a variably located record area. You can also use pointers to handle a chained list. For information on these topics, read the appropriate sections in Chapter 16, “Subprograms and Data Sharing” on page 260.

Procedure Pointers

A pointer to an entry point can be defined as a procedure pointer. A data item defined with the USAGE IS PROCEDURE-POINTER clause in the Data Division may contain the entry address for a procedure entry point. A procedure-pointer data item has the same format as the LE/VSE entry variable data type.

See “Passing Entry Point Addresses with Procedure Pointers” on page 280 for more information.

Chapter 5. String Handling

COBOL/VSE provides language constructs for performing various operations associated with string data items. For example:

- Joining data items
- Splitting data items
- Referencing substrings of data items
- Tallying and replacing data items
- Using double-byte character items (DBCS)
- Converting data items

This chapter discusses the many techniques you can use to manipulate string data.

Joining Data Items (STRING Statement)

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

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

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

For more information about ON OVERFLOW with STRING and UNSTRING, see Chapter 13, “Error Handling” on page 192.

STRING Statement Example

The following example illustrates some of the considerations that apply to the STRING statement.

In the File Section, the following input record is defined:

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

In the Working-Storage Section, the programmer has defined 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 programmer wants to construct an output line consisting of portions of the information from RCD-01. The line is to consist of a line number, customer name and address, invoice number, next billing date, and balance due, shortened to the dollar figure shown. (The symbol ␣ indicates a blank space.)

The record, as read, contains the following information:

```
J.B.␣SMITH␣␣␣␣␣
444␣SPRING␣ST.,␣CHICAGO,␣ILL.␣␣␣␣␣␣
A14275
$4,736.85
$2,400.00
09/22/94
$2,336.85
10/22/94
```

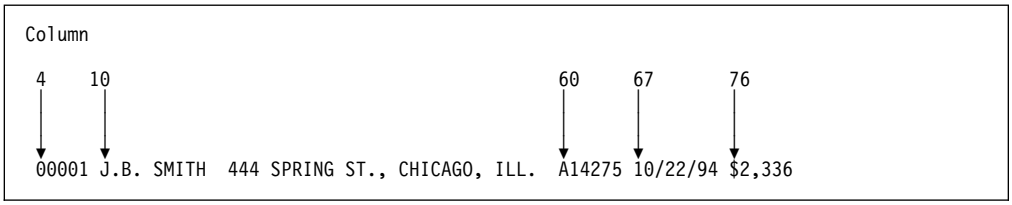
In the Procedure Division, the programmer initializes RPT-LINE to SPACES, and sets LINE-POS to 4 (that is to be used as the POINTER field). When the pointer option is specified, you can use the explicit pointer field to control placement of data in the receiving field. This STRING statement is then issued:

```
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.
```

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

1. The field LINE-NO is moved into positions 4 through 8 of RPT-LINE
2. A space is moved into position 9
3. The group item CUST-INFO is moved into positions 10 through 59
4. INV-NO is moved into positions 60 through 65
5. A space is moved into position 66
6. DATE-DUE is moved into positions 67 through 74
7. A space is moved into position 75
8. The portion of BAL-DUE that precedes the decimal point is moved into positions 76 through 81.
9. The value of LINE-POS is 82 after the STRING statement is executed

At the end of execution of the STRING statement, RPT-LINE appears as shown in the following:



Splitting Data Items (UNSTRING Statement)

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

As with the STRING statement, you can specify delimiters that, when encountered in the sending field, cause the current receiving field to be switched to the next one specified. You can get back the number of characters placed in each receiving field, and you can keep a count of the total number of characters transferred. If all the receiving fields are filled before the end of the sending item is reached, you can specify special actions for the program to take.

UNSTRING Statement Example

The following example illustrates some of the considerations that apply to the UNSTRING statement.

In the Data Division, the programmer has defined the following input record to be acted upon 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.
```

The next two records are defined as receiving fields for the UNSTRING statement. DISPLAY-REC is to be used for printed output. WORK-REC is to be used for further internal processing.

```
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).

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).
```

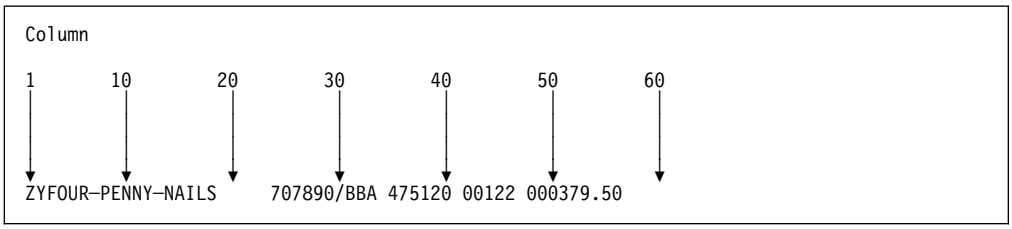
The programmer has also defined the following fields for use as control fields in the UNSTRING statement:

```
77 DBY-1                    PIC X.
77 CTR-1                    PIC S9(3).
77 CTR-2                    PIC S9(3).
77 CTR-3                    PIC S9(3).
77 CTR-4                    PIC S9(3).
77 DLTR-1                   PIC X.
77 DLTR-2                   PIC X.
77 CHAR-CT                  PIC S9(3).
77 FLDS-FILLED              PIC S9(3).
```

In the Procedure Division, the programmer writes the following UNSTRING statement to move subfields of INV-RCD to the subfields of DISPLAY-REC and WORK-REC:

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

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



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

1. Positions 3 through 18 (FOUR-PENNY-NAILS) of INV-RCD are placed in ITEM-NAME, left-justified within the area, and the unused character positions are padded with spaces. The value 16 is placed in CTR-1.
2. Because ALL SPACES is specified as a delimiter, the 5 contiguous SPACE characters are considered to be one occurrence of the delimiter.
3. Positions 24 through 29 (707890) are placed in INV-NO. The delimiter character, /, is placed in DLTR-1, and the value 6 is placed in CTR-2.
4. Positions 31 through 33 are placed in INV-CLASS. See Note.
5. Positions 35 through 40 (475120) are examined and placed in M-UNITS. The value 6 is placed in CTR-3. See Note.

6. Positions 42 through 46 (00122) are placed in FIELD-A and right-justified within the area. The high-order digit position is filled with a 0 (zero). See Note.

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

   At the end of execution of the UNSTRING statement:

   - DISPLAY-REC contains the following data:

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

   - WORK-REC contains the following data:

     ```
     475120000122BBA
     ```

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

7. Positions 48 through 53 (000379) are placed in DISPLAY-DOLS. The period (.) delimiter character in DBY-1 is placed in DLTR-2, and the value 6 is placed in CTR-4.
8. Because all receiving fields have been acted upon and 2 characters of data in INV-RCD have not been examined, the ON OVERFLOW exit is taken, and execution of the UNSTRING statement is completed.

Note: The UNSTRING statement has a slightly different behavior when the CMPR2 compiler option is in effect. For VS COBOL II Release 2 compatibility and migration details, see *COBOL/VSE Migration Guide*.

Referencing Substrings of Data Items (Reference Modifiers)

You can use reference modifiers with character string data items to reference a substring of the data item. Remember that intrinsic functions which return character string values are also considered alphanumeric data items, and thus can include a reference modifier. Figure 15 on page 50 shows which intrinsic functions return alphanumeric data.

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

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

As this shows, you specify the wanted substring in parentheses immediately following the data item. Within the parentheses, you specify the ordinal position (from the left) of the character you want the substring to start with, followed by a colon and the length of the substring. The length is optional and if omitted, the substring will automatically extend to the end of the data item.

If the leftmost character position or the length value is a noninteger, truncation will occur to create an integer.

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

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

The SSRANGE compiler option and the CHECK run-time option detect out-of-range reference modifiers and flag violations with a run-time message.

A Sample Problem

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

```
HHMMSSss
```

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

```
HH:MM:SS
```

Without Reference Modification

Without reference modification, the following data items would have to be defined:

```
01  TIME-GROUP.
    05  INTERESTING-FIELDS.
        10  HOURS                   PIC XX.
        10  MINUTES                 PIC XX.
        10  SECONDS                 PIC XX.
    05  UNINTERESTING-FIELDS.
        10  HUNDREDTHS-OF-SECONDS   PIC XX.
01  EXPANDED-TIME-GROUP.
    05  INTERESTING-FIELDS.
        10  HOURS                   PIC XX.
        10                          PIC X   VALUE ":".
        10  MINUTES                 PIC XX.
        10                          PIC X   VALUE ":".
        10  SECONDS                 PIC XX.
```

The following code would retrieve the TIME value, convert it to its expanded format, and display the new value:

```
 ACCEPT TIME-GROUP FROM TIME.
 MOVE CORRESPONDING
    INTERESTING-FIELDS OF TIME-GROUP TO
    INTERESTING-FIELDS OF EXPANDED-TIME-GROUP.
 DISPLAY "CURRENT TIME IS: " EXPANDED-TIME-GROUP.
```

With Reference Modification

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

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

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

```
ACCEPT REFMOD-TIME-ITEM FROM TIME.
DISPLAY "CURRENT TIME IS: "
   REFMOD-TIME-ITEM (1:2)
   ":"
   REFMOD-TIME-ITEM (3:2)
   ":"
   REFMOD-TIME-ITEM (5:2).
```

The reference:

```
REFMOD-TIME-ITEM (1:2)
```

causes a reference beginning at character position 1, for a length of 2, thus retrieving the portion of the time value that corresponds to the number of hours.

The reference:

```
REFMOD-TIME-ITEM (3:2)
```

causes a reference beginning at character position 3, for a length of 2, thus retrieving the portion of the time value that corresponds to the number of minutes.

The reference:

```
REFMOD-TIME-ITEM (5:2)
```

causes a reference beginning at character position 5, for a length of 2, thus retrieving the portion of the time value that corresponds to the number of seconds.

With Reference Modification of an Intrinsic Function

The simplest solution to our problem would be to reference a substring of the CURRENT-DATE function:

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

This code requires no Data Division entries and fewer lines of code.

Using Variables as Reference Modifiers

So far, all of the substringing examples have used numeric literals as the integers in the reference modifier. However, these values can also be variables that are defined as integers.

For example:

```
05 Left-posn          Pic 99  Value 4.
05 Name-length        Pic 99  Value 5.
⋮
Move Customer-Record(Left-posn:Name-length) To Customer-Name
```

In this example, the substring of `Customer-Record` that would be moved depends on the values of `Left-posn` and `Name-length` at run time.

Using Arithmetic Expressions as Reference Modifiers

You can also use an arithmetic expression as either of the integers in a reference modifier. An arithmetic expression that creates a fixed-point noninteger is shortened to create an integer. An arithmetic expression that creates a floating-point noninteger is rounded to create an integer.

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

The program would have the following data:

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

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

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

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

In the previous example, both the leftmost character position and the length were specified in the reference modifier for `RIGHTY` to illustrate the use of arithmetic expressions as reference modifiers.

However, remember that specifying the length is optional. If it is omitted, the substring created will automatically extend to the end of the item. Thus, coding:

```
MOVE RIGHTY ( I + 1 : ) TO LEFTY
```

is equivalent to:

```
MOVE RIGHTY ( I + 1 : LENGTH OF RIGHTY - I ) TO LEFTY
```

Omitting the length, when possible, is recommended as a simpler, less error-prone coding technique.

Using Intrinsic Functions as Reference Modifiers

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

For example:

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

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

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

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

For a list that shows which numeric functions return integer and noninteger results, see the *COBOL/VSE Language Reference*.

Referencing Substrings of Table Items

You can also reference substrings of table entries, including variable-length entries. This is discussed in Chapter 7, “Handling Tables (Arrays)” on page 94.

Tallying and Replacing Data Items (INSPECT Statement)

The INSPECT statement is useful for filling selective portions of a data item with a value, or for replacing portions with a corresponding portion of another data item. You can also use it for counting the number of times a specific character (zero, space, asterisk, for example) occurs in a data item.

INSPECT Statement Examples

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

Example 1:

```
77  COUNTR                PIC   9   VALUE ZERO.
01  DATA-1                PIC X(6).
  ⋮
    INSPECT DATA-1
            TALLYING COUNTR FOR CHARACTERS AFTER INITIAL "S"
            REPLACING ALL "A" BY "O"
```

DATA-1 Before	COUNTR After	DATA-1 After
ANSELM	3	ONSELM
SACKET	5	SOCKET
PASSED	3	POSSED

Example 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
0000ALABAMA	4	0000ALABAMA
CHATHAM0000	0	CH2THAM0000

Example 3:

```
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. The table shows examples of the contents of DATA-4 before and after the conversion statement is performed.

Example 4:

```
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

Using Double-Byte Character (DBCS) Data

Direct use of byte-oriented nonnumeric operations (for example, STRING and UNSTRING) on nonnumeric data items containing double-byte characters leads to unpredictable results. When statements that operate on a byte-by-byte basis are used with nonnumeric data items containing double-byte characters, you should first convert these items to pure DBCS data. This allows you to use supported functions on the DBCS data item. Once you have achieved the results you want, you may convert the DBCS data item back to a nonnumeric data item containing double-byte characters.

You can use reference modifiers with DBCS data items to reference a substring of a DBCS data item the same way you can for non-DBCS items:

```
Move dbcs-item-1(2:5) to dbcs-item-2
```

The values within the parentheses represent the leftmost "character" position you want the substring to begin with and the number of "character" positions to move. If the second value is omitted, the substring will automatically extend to the end of the DBCS item.

COBOL/VSE provides two service routines (IGZCA2D and IGZCD2A) which allow you to convert between nonnumeric data items and DBCS data items using the COBOL/VSE CALL interface. The DBCS compiler option does not affect the operation of these service routines.

Note: The examples in this section use the following notation to describe DBCS items:

< and > denote Shift-Out (SO) and Shift-In (SI), respectively

D0, D1, D2,... Dn, denote any DBCS character except for double-byte EBCDIC characters

.A, .B, .C, ..., denote any double-byte EBCDIC character; the period, ".", represents the value X'42'

A single letter, such as A, B, or s, denotes any single-byte EBCDIC character.

Nonnumeric to DBCS Data Conversion

The IGZCA2D service routine may be used to convert nonnumeric data that contains double-byte characters to pure DBCS data. You may use the IGZCA2D service routine by passing four parameters to the routine using the CALL statement. These parameters are:

parameter-1
specifies the sending field for the conversion. It is treated as a nonnumeric data item.

parameter-2
specifies the receiving field for the conversion. It is treated as a DBCS data item. Reference modification cannot be used with parameter-2.

parameter-3
specifies the number of bytes in parameter-1 that will be converted. It may be the LENGTH special register of parameter-1 or a four-byte USAGE IS BINARY data item containing the number of bytes of parameter-1 to be converted. Shift codes are counted as one byte each.

parameter-4
specifies the number of bytes in parameter-2 that will receive the converted data. It may be the LENGTH special register of parameter-2 or a four-byte USAGE IS BINARY data item containing the number of bytes of parameter-2 to receive the converted data.

Parameter-2 must be passed to the routine BY REFERENCE, while parameter-1, parameter-3, and parameter-4 may be passed BY REFERENCE or BY CONTENT.

Note: The compiler will **not** perform syntax checking on these parameters. You are responsible for ensuring that the parameters are correctly set and passed to the conversion routine using the CALL statement. If the parameters are not correctly set and passed to the conversion routine, results may be unpredictable.

If parameter-1 contains double-byte character data, the conversion routine removes the shift codes, leaving the DBCS data unchanged. The single-byte EBCDIC data in parameter-1 is converted to double-byte EBCDIC characters. An EBCDIC space (X'40') will be converted to a DBCS space (X'4040'), instead of X'4240'.

The contents of parameter-1, parameter-3, and parameter-4 remain unchanged by the service routine.

The valid range for the contents of parameter-3 and parameter-4 is 1 to 16 million.

The service routine sets the RETURN-CODE special register to reflect the status of the conversion. Figure 16 describes the meanings of these return codes.

Figure 16. 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 cut off on the right.
6	Parameter-1 was converted and the results were placed in parameter-2. An EBCDIC character in the range X'00' to X'3F' or X'FF' was encountered. The valid EBCDIC character has been converted into an out-of-range DBCS character.
8	Parameter-1 was converted and the results were placed in parameter-2. An EBCDIC character in the range X'00' to X'3F' or X'FF' was encountered. The valid EBCDIC character has been 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. An EBCDIC character in the range X'00' to X'3F' or X'FF' was encountered. The valid EBCDIC character has been converted into an out-of-range DBCS character. The DBCS data in parameter-2 was cut off 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 specified in parameter-4. No conversion occurred.

In the Procedure Division, you can write the following CALL statement to convert the nonnumeric data in `alpha-item` to DBCS data. The results of the conversion will be placed in `dbcs-item`.

```
CALL "IGZCA2D" USING BY REFERENCE alpha-item dbcs-item
                     BY CONTENT LENGTH OF alpha-item
                                LENGTH OF dbcs-item
```

If the contents of `alpha-item` and `dbcs-item` before the conversion were:

```
alpha-item = AB<D1D2D3>CD
dbcs-item  = D4D5D6D7D8D9D0
```

and the lengths were:

```
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
```

with a RETURN-CODE of 0.

DBCS to Nonnumeric Data Conversion

The IGZCD2A routine converts pure DBCS data to nonnumeric data which may contain double-byte characters. The DBCS compiler option has no effect on the operation of the service routine. The IGZCD2A service routine requires four parameters. These parameters are:

parameter-1
specifies the sending field for the conversion. It is treated as a DBCS data item.

parameter-2
specifies the receiving field for the conversion. It is treated as a nonnumeric data item.

parameter-3
specifies the number of bytes in parameter-1 that will be converted. It may be the LENGTH special register of parameter-1 or a four-byte USAGE IS BINARY data item containing the number of bytes of parameter-1 to be converted.

parameter-4
specifies the number of bytes in parameter-2 that will receive the converted data. It may be the LENGTH special register of parameter-2 or a four-byte USAGE IS BINARY data item containing the number of bytes of parameter-2 to receive the converted data. Shift codes are counted as one byte each.

Parameter-2 must be passed to the routine BY REFERENCE, while parameter-1, parameter-3, and parameter-4 may be passed BY REFERENCE or BY CONTENT.

Note: The compiler will **not** perform syntax checking on these parameters. You are responsible for ensuring that the parameters are correctly set and passed to the conversion routine. If the parameters are not correctly set and passed to the conversion routine, results may be unpredictable.

If parameter-1 contains DBCS characters which are not double-byte EBCDIC characters, shift codes will be inserted around these DBCS characters. All double-byte EBCDIC characters will be converted to single-byte EBCDIC characters. The DBCS space (X'4040') will be converted to an EBCDIC space (X'40').

The contents of parameter-1, parameter-3, and parameter-4 remain unchanged by the service routine.

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 parameter-4 is 1 to 16 million.

The service routine sets the RETURN-CODE special register to reflect the status of the conversion. Figure 17 describes the meanings of these return codes.

Figure 17. 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 EBCDIC spaces.
4	Parameter-1 was converted and the results were placed in parameter-2. Parameter-2 was cut off 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 specified in parameter-3. No conversion occurred.

[1] If a truncation occurs within the DBCS characters, the truncation will occur on an even-byte boundary and a shift-in (SI) will be inserted. If necessary, the nonnumeric data will be padded with an EBCDIC space after the shift-in.

In the Procedure Division, you can write the following CALL statement to convert the DBCS data in `dbcs-item` to nonnumeric data with double-byte characters. The results of the conversion will be placed in `alpha-item`.

```
CALL "IGZCD2A" USING BY REFERENCE dbcs-item alpha-item
                     BY CONTENT LENGTH OF dbcs-item
                                LENGTH OF alpha-item
```

If the contents of `dbcs-item` and `alpha-item` before the conversion were

```
dbcs-item  = .A.BD1D2D3.C.D
alpha-item = sssssssssss
```

and the lengths were

```
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
```

with a RETURN-CODE of 0.

Converting Data Items (Intrinsic Functions)

Intrinsic functions are available to convert character string data items to:

- Uppercase or lowercase
- Reverse order
- Numbers

Besides using intrinsic functions to convert characters, you can also use the INSPECT statement. See the examples under “Tallying and Replacing Data Items (INSPECT Statement)” on page 62.

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

The following code:

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

would display the following messages on the terminal:

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

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

Converting to Reverse Order (REVERSE)

The following code:

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

would reverse the order of the characters in `Orig-cust-name`. For example, if the starting value were JOHNSONbbb, the value after execution of the statement would be bbbNOSNHOJ.

Converting to Numbers (NUMVAL, NUMVAL-C)

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

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

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

Using NUMVAL and NUMVAL-C reduces the need for you to statically declare numeric data in a fixed format and input data in a precise manner. For example, for this code:

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

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

```
+001.23
-300.00
```

However, using the NUMVAL function, you could code:

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

and the input could be:

```
 1.23
 -300
```

Note: Both NUMVAL and NUMVAL-C return a long (double-precision) floating-point value. A reference to either of these functions, therefore, represents a reference to a numeric data item. For more information about the characteristics of numeric data, see Chapter 6, “Numbers and Arithmetic” on page 73.

Evaluating Data Items (Intrinsic Functions)

Several intrinsic functions can be used in evaluating data items:

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

Evaluating Single Characters for Collating Sequence (CHAR, ORD)

If you want to know the ordinal position of a certain character in the collating sequence, you can reference the ORD function specifying the character in question as the argument, and ORD will return an integer representing that ordinal position.

One convenient way to do this is to use the substring of a data item as the argument to ORD:

```
IF Function Ord(Customer-record(1:1)) Is > 13 THEN ...
```

On the other hand, if you know what position in the collating sequence you want but do not know what character it corresponds to, then reference the CHAR function specifying the integer ordinal position as the argument, and CHAR will return the required character:

```
INITIALIZE Customer-Name REPLACING ":" BY Function Char(29)
```

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

If you have two or more alphanumeric data items and want to know which data item contains the largest value (evaluated according to the collating sequence), you can use the MAX function, supplying the data items in question as arguments. If you want to know which item contains the smallest value, you would use the MIN function. The MAX and MIN functions simply return the contents of one of the variables you supply.

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

For example, with these data definitions:

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

then the following statement:

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

would assign VALLEJObbb to the first ten character positions of `Customer-record`.

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

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

This would assign the integer 3 to x, if the same arguments were used as in the previous example. If MIN and ORD-MIN were used respectively in the two examples above, then THOMASbbbb and the integer 2 would be the values returned.

Note: This group of functions can also be used for numbers, in which case the algebraic values of the arguments are compared. For more information, see the appropriate section of Chapter 6, "Numbers and Arithmetic" on page 73.

The above examples would probably be more realistic if `Arg1`, `Arg2` and `Arg3` were instead successive elements of an array (table). For information on using table elements as function arguments, see the section on "Processing Table Items (Intrinsic Functions)" on page 113 in Chapter 7, "Handling Tables (Arrays)."

Returning Variable-Length Results with Alphanumeric Functions

The results of alphanumeric functions may 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))
```

In this case, R2 is evaluated to be larger than R1. Therefore, assuming that the symbol b represents a blank space, the string "fbbbb" would be moved to R3 (the unfilled character positions in R3 are padded with spaces), and L evaluates to the value 5. If R1 were the value "g" then R1 would be larger than R2, and the string "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.

Therefore, be aware that you may be dealing with variable-length output from alphanumeric functions and plan your program code accordingly. For example, you may need to think about using variable-length record files when it is possible that the records you will be writing may be of different lengths:

```
File Section.
FD  Output-File.
01  Customer-Record    Pic X(80)

Working-Storage Section.
01  R1            Pic x(50).
01  R2            Pic x(70).
 ⋮
    Write Customer-Record from Function Max(R1 R2)
```

Finding the Length of Data Items (LENGTH)

The LENGTH function is useful in many programming contexts for determining the length of string items. The following COBOL statement shows moving a data item, such as a customer name, into the particular field in a record that is for customer names:

```
Move Customer-name To Customer-record(1:Function Length(Customer-name))
```

Note: The LENGTH function can also be used on a numeric data item or a table entry. Numeric data and tables are discussed in Chapter 6, "Numbers and Arithmetic" on page 73 and in Chapter 7, "Handling Tables (Arrays)" on page 94.

In addition to the LENGTH function, another technique to find the length of a data item is to use the LENGTH OF special register. Thus, coding either `Function Length(Customer-Name)` or `LENGTH OF Customer-Name` would return the same result– the length of `Customer-Name` in bytes.

Whereas the LENGTH function may only be used where arithmetic expressions are allowed, the LENGTH OF special register can be used in a greater variety of contexts. For example, the LENGTH OF special register may be used as an argument to an intrinsic function that allows integer arguments. (An intrinsic function may not be used as an operand to the LENGTH OF special register.) The LENGTH OF special register can also be used as a parameter in a CALL statement.

Finding the Date of Compilation (WHEN-COMPILED)

If you want to know the date and time the program was compiled as provided by the system on which the program was compiled, you can use the WHEN-COMPILED function. The result returned has 21 character positions with the first 16 positions in the format:

```
YYYYMMDDhhmmsshh
```

to show the four-digit year, month, day, and time (in hours, minutes, seconds, and hundredths of seconds) of compilation.

The WHEN-COMPILED special register is another technique you can use to find the date and time of compilation. It has the format:

```
MM/DD/YYhh.mm.ss
```

The WHEN-COMPILED special register supports only a two-digit year and carries the time out only to seconds. This special register can only be used as the sending field in a MOVE statement.

Chapter 6. Numbers and Arithmetic

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

General COBOL View of Numbers (PICTURE clause)

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

Defining Numeric Items

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

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

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

```
05  Price             Pic s99v99.
```

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

For information on how you can control the way the compiler handles non-separate signs, see the NUMPROC compiler option description under "NUMPROC" on page 243.

Separate Sign Position (for Portability)

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

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

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

Extra Positions for Displayable Symbols (Numeric Editing)

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

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

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

How to Use Numeric-Edited Items as Numbers

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

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

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

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

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

Computational Data Representation (USAGE Clause)

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

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

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

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

External Decimal (USAGE DISPLAY) Items

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

What USAGE DISPLAY Items Are For

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

Should You Use Them for Arithmetic

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

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

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

External Floating-Point (USAGE DISPLAY) Items

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

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

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

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

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

Binary Items

BINARY, COMP, and COMP-4 are synonyms on all platforms.

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

How Much Storage BINARY Occupies

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

Binary items with 9 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.

Why Use Binary

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

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

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

Truncation of Binary Data (TRUNC Compiler Option)

Use the TRUNC(STD|OPT|BIN) compiler option (described in "TRUNC" on page 252) to indicate how binary data (BINARY, COMP, and COMP-4) is truncated.

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

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

Why Use Packed Decimal

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

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

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

COMP-1 and COMP-2 data items are stored in System/390® hexadecimal format.

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

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

The characteristics of conversions between floating-point format and other number formats are discussed in the next section, “Data Format Conversions” on page 78.

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

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

Internal Representation of Numeric Items

Figure 18 shows how the different numeric data types are represented internally in program storage.

Figure 18. 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 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
Binary	PIC S9999 BINARY COMP COMP-4	+ 1234	04 D2
		- 1234	FB 2E
	PIC 9999 BINARY COMP COMP-4	+ 1234	04 D2
Internal Decimal	PIC S9999 PACKED-DECIMAL COMP-3	+ 1234	01 23 4C
		- 1234	01 23 4D
	PIC 9999 PACKED-DECIMAL COMP-3	+ 1234	01 23 4F
		- 1234	01 23 4F
Internal Floating Point	COMP-1	+ 1234	43 4D 20 00
Internal Floating Point	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	+ 1234	4E F1 F2 4B F3 F4 C5 4E F0 F2
		- 1234	60 F1 F2 4B F3 F4 C5 4E F0 F2

Data Format Conversions

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

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

What Conversion Means

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 and comparisons with the same rules that are used for MOVE and COMPUTE statements. The rules for moves are defined in *COBOL/VSE Language Reference*.

When possible, the compiler performs the move to preserve the numeric "value" as opposed to a direct digit-for-digit move. (For more information on truncation and predicting the loss of significant digits, refer to Appendix B, "Intermediate Results and Arithmetic Precision" on page 401.)

Conversion Takes Time

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

Conversions and Precision

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

Conversions Where Loss of Precision Is Possible

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

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

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

When a USAGE COMP-1 data item is moved to a fixed-point data item with more than 9 digits, the fixed-point data item will receive only 9 significant digits, and the remaining digits will be zero.

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

If a USAGE COMP-1 data item is moved to a fixed-point data item of 9 or more digits and then returned to the USAGE COMP-1 data item, the original value is recovered.

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

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

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

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

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

Sign Representation and Processing

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

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

Positive: C, A, E, and F.

Negative: D and B.

NUMPROC Compiler Option

The COBOL NUMPROC compiler option affects sign processing for external decimal and internal decimal data. NUMPROC has no effect on binary data or floating-point data. For additional details on the NUMPROC compiler option, see its description under “NUMPROC” on page 243.

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 be sure you 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. When processing is done with DISPLAY or PACKED-DECIMAL data, the sign in the sending item is converted to the preferred sign before the operation is performed. The preferred sign is also generated in the receiver. NUMPROC(NOPFD) is less efficient than NUMPROC(PFD), but it should be used whenever data that does not use preferred signs might exist.

If an unsigned, external decimal sender is moved to an alphanumeric receiver, the sign is unchanged (even with NUMPROC(NOPFD)).

NUMPROC(MIG)

When NUMPROC(MIG) is in effect, the compiler generates code that is similar to that produced by DOS/VS COBOL. This option can be especially useful as a tool when migrating DOS/VS COBOL programs to COBOL/VSE.

Checking for Incompatible Data (Numeric Class Test)

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

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

How to Do a Numeric Class Test

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

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

The numeric class test checks the contents of a data item against a set of values that are valid for the particular PICTURE and USAGE of the data item. For example, a packed decimal item would be 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 non-separate).

Interaction of NUMPROC and NUMCLS Options

The numeric class test is affected by the NUMPROC compiler option and the NUMCLS option (which is set at installation time). For information on the NUMPROC compiler option, refer to its description under “NUMPROC” on page 243. To determine the NUMCLS setting used at your installation, consult your system programmer.

Figure 19 and Figure 20 show the values that the compiler considers valid for the sign. Only external decimal, external floating-point, and packed decimal items are checked for sign, because binary and internal floating-point items use an operational sign that is always valid.

Figure 19. Sign Representation with NUMCLS(PRIM)

	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, see Figure 20.

Figure 20. Sign Representation with NUMCLS(ALT)

	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)	+, -

Performing Arithmetic

COBOL provides various language features to perform arithmetic:

- ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE statements (discussed in “COMPUTE and Other Arithmetic Statements” on page 82).
- Arithmetic expressions (discussed in “Arithmetic Expressions” on page 82).
- Intrinsic functions (discussed in “Numeric Intrinsic Functions” on page 83).
- LE/VSE callable services.

  Callable services are introduced in “LE/VSE Callable Services” on page 86 and discussed more fully in *LE/VSE Programming Reference*.

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

COMPUTE and Other Arithmetic Statements

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

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

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

or to many data items:

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

When to Use Other Arithmetic Statements

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

```
Add 1 To Increment
```

instead of:

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

Or,

```
Subtract Overdraft From Balance
```

instead of:

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

Or,

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

instead of:

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

You might also prefer to use the DIVIDE statement (with its REMAINDER phrase) for division in which you want to process a remainder. The REM intrinsic function also provides the ability to process a remainder. For an example of the REM function, see “Mathematics” on page 86.

Arithmetic Expressions

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

Figure 21. Operator Evaluation

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

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

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

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

Numeric Intrinsic Functions

Intrinsic functions can return an alphanumeric or numeric value.

Numeric intrinsic functions:

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

  For more information on the practical application of intrinsic functions, including examples of their usage, refer to “Intrinsic Function Examples” on page 84.

Many of the capabilities of intrinsic functions are also provided by LE/VSE callable services. For a comparison of the two, see “LE/VSE Callable Services” on page 86.

Types of Numeric Functions

Numeric functions are classified into these categories:

Integer Those that return an integer

Floating-Point Those that return a long floating-point value

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

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

Nesting Functions and Arithmetic Expressions

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

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

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

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

ALL Subscripting and Special Registers

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

- You can reference all the elements of an array as function arguments by using the ALL subscript. This feature is used with tables, and examples of its use are shown under “Processing Table Items (Intrinsic Functions)” on page 113.
- The integer-type special registers are allowed as arguments wherever integer arguments are allowed.

Intrinsic Function Examples

You can use intrinsic functions to perform several different kinds of arithmetic, as outlined in Figure 22.

Figure 22. Types of Arithmetic that Numeric Intrinsic Functions Handle

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

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

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

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

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

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

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

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

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

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

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

The ANNUITY function can be used in business problems that require you to determine the amount of an installment payment (annuity) necessary to repay the principal and interest of a loan. The series of payments is characterized by an equal

amount each period, periods of equal length, and an equal interest rate each period. The following example shows how you could calculate the monthly payment required to repay a $15,000 loan at 12% annual interest in three years (36 monthly payments, interest per month = .12/12):

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

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

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

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

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

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

LE/VSE Callable Services

Many of the capabilities of intrinsic functions are also provided by LE/VSE callable services. LE/VSE callable services are a means of assigning results of arithmetic to data items.

LE/VSE provides a set of callable services that include mathematical types of functions (“Math-Oriented Callable Services and Intrinsic Functions” on page 87) and date and time operations (“Date Callable Services and Intrinsic Functions” on page 88). Some of these return the same results as corresponding COBOL intrinsic functions, some produce slightly different results than the corresponding intrinsic functions, and some provide services for which there is no corresponding COBOL intrinsic function (see *LE/VSE Programming Reference*).

Math-Oriented Callable Services and Intrinsic Functions

Figure 23. Same Numeric Results

COBOL Intrinsic Function	LE/VSE Callable Service
ACOS	CEESDACS
ASIN	CEESDASN
ATAN	CEESDATN
COS	CEESDCOS
LOG	CEESDLOG
LOG10	CEESDLG1
REM	CEESSMOD
SIN	CEESDSIN
SQRT	CEESDSQT
TAN	CEESDTAN

Although these functions produce the same results, how you use intrinsic functions and LE/VSE 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 LE/VSE callable service with a CALL statement, you must ensure that the parameters are defined with the numeric data types required by that particular service, generally COMP-1 and COMP-2.

The error handling of intrinsic functions and LE/VSE callable services sometimes differs. If you pass an explicit feedback token when calling the LE/VSE math services, you must check the feedback code after the call and take explicit action to deal with any errors. (However, if the feedback token is explicitly OMITTED, you do not need to check the token after each call. Any errors will be automatically signaled by LE/VSE.)

Figure 24 shows the numeric intrinsic function and the LE/VSE callable service that return nonequivalent values.

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 will produce different random numbers from the same seed.

Figure 24. Nonequivalent Numeric Results

COBOL Intrinsic Function	LE/VSE Callable Service
RANDOM	CEERAN0

Date Callable Services and Intrinsic Functions

Both the COBOL intrinsic functions and the LE/VSE date callable services are based on the Gregorian calendar. However the starting dates can be different, depending on the setting of the INTDATE compiler installation option. When the IBM default setting of INTDATE(ANSI) is in effect, COBOL uses January 1, 1601 as day 1. When INTDATE(LILIAN) is in effect, COBOL uses October 15, 1582 as day 1. LE/VSE always uses October 15, 1582 as day 1.

This means that if you set your installation default to INTDATE(LILIAN) you will get equivalent results from COBOL intrinsic functions and LE/VSE callable date services.

INTDATE(ANSI) in Effect

Figure 25. Compatible Numeric Results with INTDATE(ANSI) in Effect

COBOL Intrinsic Function	LE/VSE Callable Service
INTEGER-OF-DATE	CEECBLDY

Figure 26. Incompatible Numeric Results with INTDATE(ANSI) in Effect

COBOL Intrinsic Function	LE/VSE Callable Service
DATE-OF-INTEGER	CEEDATE with `picture_string YYYYMMDD`
DAY-OF-INTEGER	CEEDATE with `picture_string YYYYDDD`
INTEGER-OF-DATE	CEEDAYS

INTDATE(LILIAN) in Effect

Figure 27. Compatible Numeric Results with INTDATE(LILIAN) in Effect

COBOL Intrinsic Function	LE/VSE Callable Service
DATE-OF-INTEGER	CEEDATE with `picture_string YYYYMMDD`
DAY-OF-INTEGER	CEEDATE with `picture_string YYYYDDD`
INTEGER-OF-DATE	CEEDAYS

Figure 28. Incompatible Numeric Results with INTDATE(LILIAN) in Effect

COBOL Intrinsic Function	LE/VSE Callable Service
INTEGER-OF-DATE	CEECBLDY

Fixed-Point versus Floating-Point Arithmetic

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

- General arithmetic.

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

- Expressions and functions.

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

- Arithmetic comparisons.

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

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

The following discussion explains when arithmetic and arithmetic comparisons are evaluated in fixed-point and floating-point. For details on the precision of arithmetic evaluations, see Appendix B, “Intermediate Results and Arithmetic Precision” on page 401.

Floating-Point Evaluations

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

- An operand or result field is floating-point.

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

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

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

- An exponent contains decimal places.

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

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

Fixed-Point Evaluations

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

Arithmetic Comparisons (Relation Conditions)

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

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

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

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

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

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

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

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

Examples of Fixed-Point and Floating-Point Evaluations

For the examples shown on page 88, if you define the data items in the following manner:

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

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

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

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

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

Using Currency Signs

Many programs need to process financial information and present that information to the user with the relevant currency signs in the output. With COBOL currency support, in addition to using symbols such as the dollar sign ($) to display financial output, you can:

- Use currency signs of more than one character, for example, USD, DEM, EUR
- Use more than one currency sign in a program
- Use the euro sign established by the Economic and Monetary Union (EMU) as a currency sign, if your code page supports it for your printer or display unit

This section describes these features, and gives examples of how you can use them.

Specifying Currency Signs

You use the CURRENCY SIGN clause to specify the symbols to be used for displaying financial information, and the picture characters that relate to those signs. 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
```

Using Hex Literals for Currency Signs

You can use a hexadecimal literal to indicate the currency sign value. This may be useful when the data entry method for the source program does not allow the entry of the intended character(s) easily. The following example shows the hex value used as the the currency sign:

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

Multiple Currency Signs

By using more than one CURRENCY SIGN clause in your program, you can allow for multiple currency signs to be displayed. The following example shows how values can be displayed in both Euro currency (as EUR) and French francs (as FRF):

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EuroExample.
Environment Division.
Configuration Section.
Special-Names.
    Currency Sign is "FRF "  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-FRF         Pic S99999V99.
01  Deposit-Report.
    02  Report-in-Franc    Pic -FFFFF9.99.
    02  Report-in-Euro     Pic -UUUUU9.99.

01  EUR-to-FRF-Conv-Rate   Pic 9V99999  Value 6.78901.

PROCEDURE DIVISION.
Report-Deposit-in-FRF-and-EUR.
    Move Deposit-in-Euro to Report-in-Euro

    Compute Deposit-in-FRF Rounded
          = Deposit-in-Euro * EUR-to-FRF-Conv-Rate
      on Size Error
        Perform Conversion-Error
      not on Size Error
        Move Deposit-in-FRF 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 FRF"
           Display "Euro value: " Report-in-Euro.
```

The above example will produce the following display output:

```
Deposit in Euro  =  EUR 8000.00
Deposit in Franc = FRF 54312.08
```

Note that the exchange rate used in this example to perform the currency conversion is for illustrative purposes only; it is not the official exchange rate.

Euro Currency Sign

You can use the euro sign established by the Economic and Monetary Union (EMU) as a currency sign if your code page supports it for your printer or display unit.

However, if there is no corresponding character on your keyboard, you will need to specify it as a hexadecimal value in the CURRENCY SIGN clause (for details, see “Using Hex Literals for Currency Signs” on page 91). The hexadecimal value for the euro sign is either X'9F' or X'5A', depending on the code page in use.

Figure 29 shows the code pages that support the euro sign as code point X'9F'.

Figure 29. Code Pages with Euro Sign as Code Point X'9F'

Code Page	Applicable Countries	Modified From
IBM-1140	USA, Canada, Netherlands, Portugal, Australia, New Zealand	IBM-037
IBM-1141	Austria, Germany	IBM-273
IBM-1144	Italy	IBM-280
IBM-1145	Spain, Latin America - Spanish	IBM-284
IBM-1146	UK	IBM-285
IBM-1147	France	IBM-297
IBM-1148	Belgium, Canada, Switzerland	IBM-500
IBM-1149	Iceland	IBM-871

Figure 30 shows the code pages that support the euro sign as code point X'5A'.

Figure 30. Code Pages with Euro Sign as Code Point X'5A'

Code Page	Applicable Countries	Modified From
IBM-1142	Denmark, Norway	IBM-277
IBM-1143	Finland, Sweden	IBM-278

Chapter 7. Handling Tables (Arrays)

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

Defining a Table (OCCURS Clause)

Use the COBOL OCCURS clause in the Data Division entry to define a table. The OCCURS clause eliminates the need for separate entries for repeated data items; it also supplies the information necessary for the use of subscripts or indexes. For more information on the format of the OCCURS clause, refer to *COBOL/VSE Language Reference*.

Give the table a group name, then define a subordinate item (the table *element*) that is to be repeated *n* times:

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

While the table element has a collective name, the individual occurrences do not have unique data-names. To refer to them, specify the data-name of the table element, together with the occurrence number of the wanted item within the element. The occurrence number is called a **subscript**, and the technique of supplying the occurrence number of individual table elements is called **subscripting**. A related technique, called **subscripting using index-names (indexing)** is also available for table references.

A **subscript** indicates the position of an entry. If you had a one-dimensional table called YEAR-TABLE that contained the 12 months (January through December), the subscript for January would be 1. The notation for March would be YEAR-TABLE (3). See “Subscripting” on page 96.

An **index** is a symbol used to locate an item in a table. An index differs from a subscript in that an index is a value to be added to the address of a table to locate an item (the displacement from the beginning of the table). See “Subscripting Using Index-Names (Indexing)” on page 97.

The following figures show how to code tables:

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

Tables of up to seven dimensions may be defined using this same method.

One Dimension

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

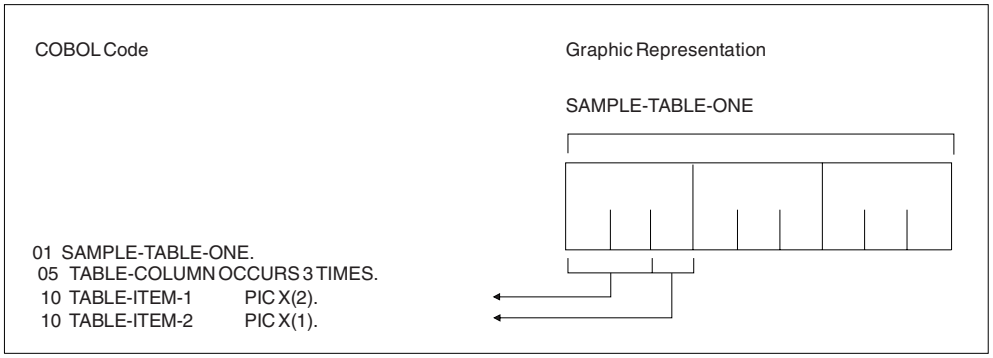


Figure 31. Coding a One-Dimensional Table

`Sample-Table-One` is the group item that contains the table. `Table-Column` names the table element of a one-dimensional table that occurs 3 times.

Two Dimensions

To create tables of more than one dimension, use nested OCCURS clauses. That is, create a table of tables.

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

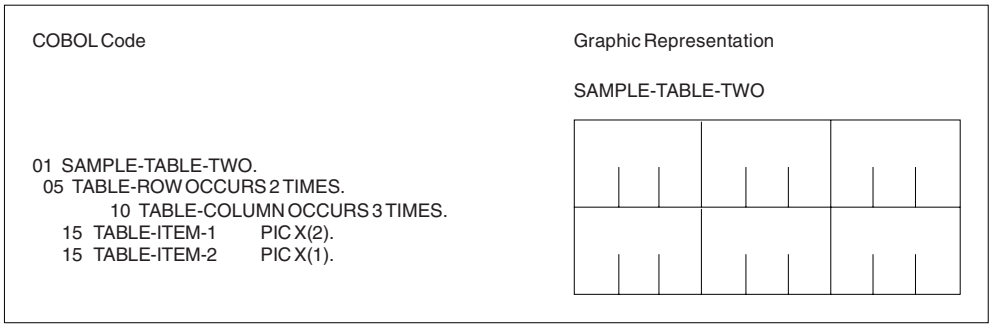


Figure 32. Coding a Two-Dimensional Table

`Sample-Table-Two` is the name of a two-dimensional table. `Table-Row` is an element of a one-dimensional table that occurs 2 times. `Table-Column` is an element of a two-dimensional table that occurs 3 times within each occurrence of `Table-Row`.

Three Dimensions

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

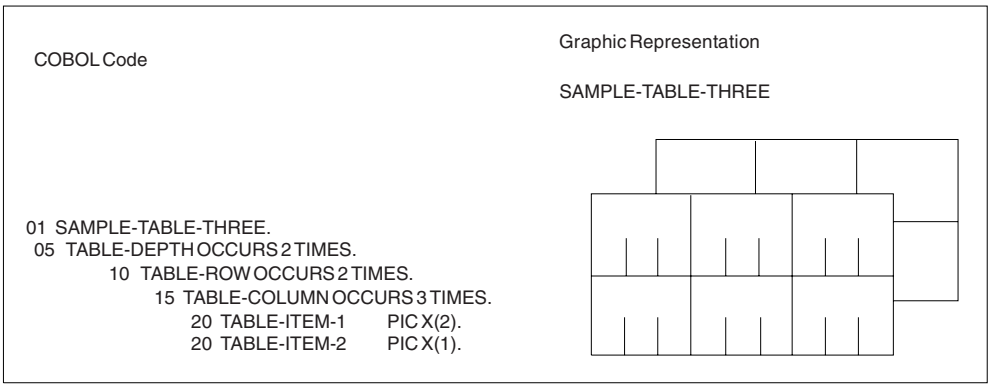


Figure 33. Coding a Three-Dimensional Table

In `Sample-Table-Three`, `Table-Depth` is an element of a one-dimensional table that occurs 2 times. `Table-Row` is an element of a two-dimensional table that occurs 2 times within each occurrence of `Table-Depth`. `Table-Column` is an element of a three-dimensional table that occurs 3 times within each occurrence of `Table-Row`.

Referring to an Item in a Table

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

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

An *index* is a symbol used to locate an item in a table. An index differs from a subscript in that an index is a value to be added to the address of a table to locate an item (the displacement from the beginning of the table). See page 97.

Subscripting

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

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

A **literal subscript** must be an integer and must have a value of 1 or greater. For example, valid literal subscript references to `Sample-Table-Three` are:

```
Table-Column (2, 2, 1)
Table-Column (2 2 1)       (The spaces are required for subscripting.)
```

In the table reference `Table-Column (2, 2, 1)`, the first value (2) refers to the second occurrence within `Table-Depth`, the second value (2) refers to the second occurrence within `Table-Row`, and the third value (1) refers to the first occurrence within `Table-Column`.

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

The data-name used as a **variable subscript** must be described as an elementary numeric integer data item. A valid, variable subscript reference to `Sample-Table-Two`, (assuming that `Sub1` and `Sub2` are data-names containing positive integer values within the range of the table), is :

```
Table-Column (Sub1 Sub2)
```

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

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

```
Table-Column (Sub1 - 1, Sub2 + 3)
```

indicates that the value in *data-name* Sub1 is to be decremented by one, and the value in Sub2 is to be incremented by three.

Subscripting Using Index-Names (Indexing)

You can also refer to table elements by using an index. An index is a displacement from the start of the table, based on the length of the table element. Use the index in SET, SEARCH, SEARCH ALL, PERFORM VARYING, or relational condition statements. An *index-name* must be initialized through a SET, PERFORM VARYING, or SEARCH ALL statement before it is used in a table reference.

The compiler determines the index of an entry based on the following formula:

$$I = L * (S-1)$$

where:

I is the index value

L is the length of a table entry

S is the subscript (occurrence number) of an entry

You define the *index-name* for a table in the INDEXED BY clause of the OCCURS clause in the table definition.

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

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

```
05  Table-Item occurs 10 indexed by Inx-A PIC X(8).
```

For the fifth occurrence of `Table-Item`, the binary value contained in `Inx-A` is (5 - 1) * 8 = 32.

In **relative indexing**, the *index-name* is followed by a space, followed by a + or a -, followed by another space, followed by an unsigned numeric literal. The literal is considered to be an occurrence number, and is converted to an index value before being added to or subtracted from the *index-name*. For example, if you specify indexing for `Sample-Table-Three` as follows:

```
01 Sample-Table-Three
   05 Table-Depth occurs 3 times indexed by Inx-A.
      10 Table-Column occurs 4 times indexed by Inx-B.
         15 Table-Row occurs 8 times indexed by Inx-C    PIC X(8).
```

then a relative indexing reference to

```
Table-Row (Inx-A + 1, Inx-B + 2, Inx-C - 1)
```

causes the following calculation of the displacement:

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

This means:

Each occurrence of `Table-Depth` is 256 characters in length
Each occurrence of `Table-Column` is 64 characters in length
Each occurrence of `Table-Row` is 8 characters in length

One or more index references (direct or relative) may be specified together with literal subscripts.

To compare two different occurrences of a table element, use a direct indexing reference together with a relative indexing reference, or use subscripting, which is easier to read in your code.

An index can be shared among different tables. That is, you can use the index defined with one table to index another table if both table descriptions are identical. To be identical, the tables must have the same number of occurrences, as well as occurrences of the same length.

You can use **index data items** to store index values. You can use the SET statement to assign to an index the value that you stored in the index data item. The index data item holds the physical displacement value. You define an index data item with the USAGE IS INDEX clause.

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

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

Subscripting and Indexing Restrictions

1. A *data-name* must not be subscripted or indexed when it is being used as a subscript or qualifier.
2. An index can be modified only by a PERFORM, SEARCH, or SET statement.
3. When a literal is used in a subscript, it must be a positive or unsigned integer.
4. When a literal is used in relative indexing or relative subscripting, it must be an unsigned integer.

Referring to a Substring of a Table Item

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

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

you can change the third and fourth bytes in the first element of TABLE-ELEMENT so that both bytes contain the value "?". This could be performed with the following MOVE statement:

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

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

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

ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ

`ANY-TABLE` would look like this after the change:

AB??EFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ

Putting Values in a Table

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

- Load the table dynamically
- Initialize the table
- Assign values when you define the table

Loading the Table Dynamically

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

To load a table, use:

The PERFORM . . . VARYING statement
Either subscripting or indexing

See Figure 35 and Figure 36 on page 103.

When reading data to load your table:

1. Make sure that the data does not exceed the space allocated for the table.
2. If the data must be in sequence, check the sequence
3. If the data contains the subscript that determines its position in the table, check the subscript for a valid range.

When testing for the end of a table, use a named value giving the item count rather than using a literal. Then, if you make the table bigger, you need to change only one value, instead of all references to a literal.

Initializing the Table (INITIALIZE Statement)

You can load your table with a value during execution with the INITIALIZE statement. You can move:

- Spaces into alphabetic, alphanumeric, and alphanumeric-edited items
- Zeros into numeric and numeric-edited items
- A particular value

You can use INITIALIZE to move a value into each table element. For example, to fill a table with 3s:

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

You cannot use INITIALIZE for a variable-length table (one that was defined using OCCURS DEPENDING ON).

Assigning Values When You Define the Table (VALUE Clause)

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

Define static values in Working-Storage in one of three ways:

1. First describe the table storage area by simply arranging subordinate data description entries, specifying the initial value of each subordinate entry in a VALUE clause. Then code a REDEFINES entry to describe the table as a record that contains a repeating subordinate entry, defined with an OCCURS clause. For an example of this method, see Figure 34 on page 102.

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

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

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

   A VALUE clause can also be present on a group item which contains an OCCURS clause with the DEPENDING ON option. In this case, each subordinate structure that contains the DEPENDING ON option is initialized using the maximum number of occurrences. If the table is defined with the DEPENDING ON option, all the elements are initialized using the maximum defined value of the DEPENDING ON object. In both cases, if the ODO object has a VALUE clause, it is logically initialized after the ODO subject has been initialized. For example:

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

   causes `Y(1)` to be initialized to `'A'`, `Y(2)` to `'B'`,... `Y(5)` to `'E'`, and finally the object of the ODO (`X`) is initialized to `'3'`. Any subsequent reference to `TABLE-THREE` (such as DISPLAY) would refer to the first 3 elements, `Y(1)` through `Y(3)`. This behavior is different if the CMPR2 compiler option is in effect. See *COBOL/VSE Migration Guide* for more details on VS COBOL II Release 2 compatibility and migration.

3. You can use the VALUE clause on a table element to initialize the element to the indicated value. As an example, this code:

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

   causes all the `X` elements (1 through 5) to be initialized to `'AA'`, all the `Y` elements (1 through 5) to be initialized to `'19'`, and all the `Z` elements (1 through 5) to be initialized to `'BB'`. `T-OBJ` is then set to `'3'`.

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

Figure 34. Sample Table

The procedure shown in Figure 35 on page 103 processes the entire table shown in Figure 34, using subscripting and the PERFORM...VARYING statement.

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

Figure 35. Processing the Sample Table, Using Subscripting

The procedure shown in Figure 36 processes the entire table, using indexing.

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

Figure 36. Processing the Sample Table, Using Indexing

Creating Variable-Length Tables (DEPENDING ON Clause)

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

The cases to consider when using the ODO clause are:

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

ODO Object outside the Group

You must ensure that the object of the OCCURS DEPENDING ON clause (which specifies the number of occurrences of the table elements) contains a value that correctly specifies the current number of occurrences of the table elements. Figure 37 on page 104 shows how to define a variable-length 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 "FALSE".
     01 MISC-VALUES.
       05 LOCATION-TABLE-LENGTH        PIC 9(3) VALUE ZERO.
       05 LOCATION-TABLE-MAX           PIC 9(3) VALUE 100.
  ****************************************************************
  ***               L O C A T I O N   T A B L E                ***
  ***               FILE CONTAINS LOCATION CODES.              ***
  ****************************************************************
     01 LOCATION-TABLE.
       05 LOCATION-CODE OCCURS 1 TO 100 TIMES
              DEPENDING ON LOCATION-TABLE-LENGTH   PIC X(80).
```

Figure 37. Defining a Variable-Length Table

Figure 38 shows a “do-until” structure used to control loading of a variable-length table. When initialization is complete, LOCATION-TABLE-LENGTH will contain the subscript of the last item in the table. (This variable-length table is defined in Figure 37.)

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

Figure 38. Loading a Variable-Length Table

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

If the content of the ODO object does not match its PICTURE clause, the program may abend. See Chapter 6, “Numbers and Arithmetic” on page 73 for more information on data and sign representation.

ODO Object and Subject Contained in Sending Group Item

The following example illustrates a group item (REC-1) whose subordinate items contain an OCCURS clause with the DEPENDING ON option and the object of that DEPENDING ON option.

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

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

If you wanted to move REC-1 to REC-2, the length of REC-1 is determined immediately prior to the MOVE, using the current value in FIELD-1. If the contents of FIELD-1 do not conform to its PICTURE, that is, if FIELD-1 does not contain an external decimal item, the program will abend. (See Chapter 6, “Numbers and Arithmetic” on page 73 for more information on data and sign representation).

As you can see, you must be sure that in this situation you have the correct value placed in the ODO object (FIELD-1) before the MOVE is initiated.

ODO Object and Subject Contained in Receiving Group Item

The following example illustrates a group item (MAIN-AREA) whose subordinate items contain an OCCURS clause with the DEPENDING ON option and the object of that DEPENDING ON option.

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

If you wanted to do a MOVE to REC-2, the length of REC-2 is determined using the maximum number of occurrences. In this example, that would be 5 occurrences of FIELD-4 plus FIELD-3 for a length of 26.

In this case, the ODO object (FIELD-3) need not be set before referencing REC-2 as a receiving item.

This behavior is different if the CMPR2 compiler option is in effect. See *COBOL/VSE Migration Guide* for details on VS COBOL II Release 2 compatibility and migration.

However, if REC-2 were followed by a data item which is in the same record but is not subordinate to REC-2, then the actual length of REC-2 is used and the ODO object must be set before the reference. In the following example, REC-2 is followed by REC-3.

```
01  MAIN-AREA
    03  REC-2.
        05  FIELD-3                      PIC S9.
        05  FIELD-5                      PIC S9.
        05  FIELD-4 OCCURS 1 TO 5 TIMES
             DEPENDING ON FIELD-3         PIC X(05).
    03  REC-3.
        05  FIELD-6 OCCURS 1 TO 5 TIMES
             DEPENDING ON FIELD-5         PIC X(05).
```

If you did a MOVE to REC-2 in this case, the actual length of REC-2 would be calculated immediately prior to the move using the current value of the ODO object (FIELD-3), and a compiler message would be issued letting you know that the actual length, instead of the maximum length, was used. This case requires that you set the value of the ODO object (FIELD-3) prior to using the item as a receiving field.

Complex OCCURS DEPENDING ON

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

- A data item described by an OCCURS clause with the DEPENDING ON option is followed by a nonsubordinate item.
- 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 with the DEPENDING ON option.
- A data item described by an OCCURS clause with the DEPENDING ON option is nested within another data item described by an OCCURS clause with the DEPENDING ON option.

If a group item contains both the subject and object of the ODO, and it is a receiving item, then the maximum length of the item is used. In this situation it is not necessary to set the value of the ODO object before a reference is made. If the receiving item is followed by a data item which is in the same record but is not subordinate to the receiver, then the actual length is used and a compiler message is issued to inform you that the actual length, not the maximum, will be used. In this situation it is necessary to set the value of the ODO object before any reference to the item. This behavior is different if the CMPR2 compiler option is in effect. See *COBOL/VSE Migration Guide* for more details on VS COBOL II Release 2 compatibility and migration.

All other cases of ODO objects in an 01-level must be set before any reference is made to a complex ODO item in the 01-level.

Note: An ODO object cannot be variably located. For instance, in the following example, before EMPLOYEE-NUMBER can be referred to, COUNTER-1 and COUNTER-2 **must** be set, even though EMPLOYEE-NUMBER does not directly depend on either of the ODO objects for its value.

The length of the variable portions of each record is the product of the ODO object and the length of the subject of the OCCURS clause. The length is calculated at the time of a reference to one of the following:

1. a group item of variable length
2. a data item following, and not subordinate to, a variable-length table in the same level-01 record (variably located item)
3. a group item following, and not subordinate to, a variable-length table in the same level-01 record (variably located group)
4. an index name for a table that has variable-length elements
5. an element of a table that has variable-length elements

Any item that meets one of these five criteria is considered to be a “complex ODO item.” The following example illustrates each of the possible occurrences of a complex ODO item.

```
01 FIELD-A.                                              1

   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).    2
   02 TABLE-2 OCCURS 5 TIMES                            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.
```

Whenever a reference is made to one of these five data items, the actual length, if used, is computed as follows:

- The contents of COUNTER-1 are multiplied by 3 to calculate the length of TABLE-1.
- The contents of COUNTER-2 are multiplied by 2 and added to the length of TABLE-ITEM to calculate the length of TABLE-2.
- The length of FIELD-A is calculated by adding the length of COUNTER-1, COUNTER-2, TABLE-1, EMPLOYEE-NUMBER, and TABLE-2 times 5.

If a data item described by an OCCURS clause with the DEPENDING ON option is followed in the same level-01 record by nonsubordinate data items, a change in the value of the ODO object, and a subsequent reference to a complex ODO item during the course of program execution, will have the following effects:

- The size of any group containing the related OCCURS clause will reflect the new value of the ODO object.
- Whenever a MOVE to a group containing an ODO object is executed, the MOVE is made based on the current contents of the object of the DEPENDING ON option.

  Note: The value of the ODO object may change because a MOVE is made to it or to the group in which it is contained. The value of the ODO object may also change because the group in which it is contained is a record area that has been changed by execution of a READ statement.

- The location of any nonsubordinate items following the item described with the OCCURS clause will be affected by the new value of the ODO object. If you wish to preserve the contents of these items, the following procedure can be used: Prior to the change in the ODO object, move all nonsubordinate items following the variable item to a work area; after the change in the ODO object, move all the items back.

  You must be careful when using complex-ODO index names. If you set an index name (like 'INDX' in the previous example) for a table with variable-length entries ('TABLE-2'), and then change the value of the ODO object ('COUNTER-2'), be aware that the offset in your index is no longer valid for the table, since the table has changed. If, at this point, you were to code statements that used your index name, thinking the index name had a valid value for the table, the statements would produce unexpected results. This would apply to coding:

  – A reference (using your index name) to an element of the table
  – A format-1 SET statement of the type SET INTEGER-DATA-ITEM TO INDEX-NAME
  – A format-2 SET statement of the type SET INDEX-NAME UP/DOWN BY INTEGER

  To avoid making this type of error, you can do the following:

  1. Save the value of your index name (in the form of its integer occurrence number) in an integer data item before changing the ODO object.
  2. Immediately after changing the ODO object, restore the value of your index name from the integer data item.

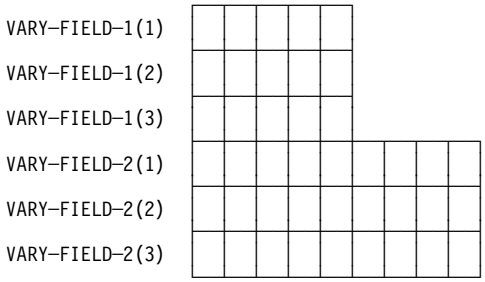
  For example:

```
77   INTEGER-DATA-ITEM-1      PIC 99.

    SET INDX TO 5
   *          INDX IS VALID AT THIS POINT.
    SET INTEGER-DATA-ITEM-1 TO INDX
    MOVE NEW-VALUE TO COUNTER-2.
   *          INDX IS NOT VALID AT THIS POINT.
    SET INDX TO INTEGER-DATA-ITEM-1.
   *          INDX IS NOW VALID AND CAN BE
   *          USED WITH EXPECTED RESULTS.
```

  The following example applies to updating a record containing an OCCURS clause with the DEPENDING ON option and at least one other subsequent entry. In this case, the subsequent entry is another OCCURS clause with the DEPENDING ON option.

```
WORKING-STORAGE SECTION.
01  VARIABLE-REC.
    05  FIELD-1                              PIC X(10).
    05  CONTROL-1                            PIC S99.
    05  CONTROL-2                            PIC S99.
    05  VARY-FIELD-1 OCCURS 1 TO 10 TIMES
         DEPENDING ON CONTROL-1               PIC X(5).
    05  GROUP-ITEM-1.
        10  VARY-FIELD-2
             OCCURS 1 TO 10 TIMES
             DEPENDING ON CONTROL-2           PIC X(9).
01  STORE-VARY-FIELD-2.
    05  GROUP-ITEM-2.
        10  VARY-FLD-2
             OCCURS 1 TO 10 TIMES
             DEPENDING ON CONTROL-2           PIC X(9).
```

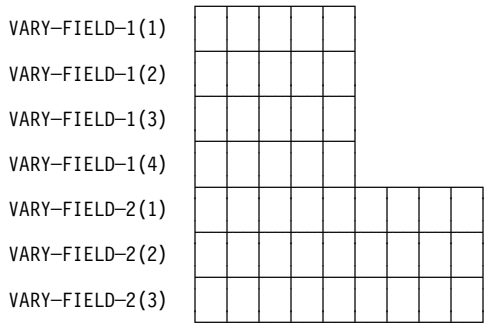
Assume that both CONTROL-1 and CONTROL-2 contain the value 3. In this situation, storage for VARY-FIELD-1 and VARY-FIELD-2 would look like this:



In order to add a fourth field to VARY-FIELD-1, the following steps are required to prevent VARY-FIELD-1 from overlaying the first 5 bytes of VARY-FIELD-2:

```
MOVE GROUP-ITEM-1 TO GROUP-ITEM-2
ADD 1 TO CONTROL-1
MOVE "additional field" TO
  VARY-FIELD-1 (CONTROL-1)
MOVE GROUP-ITEM-2 TO GROUP-ITEM-1
```

The updated storage for VARY-FIELD-1 and VARY-FIELD-2 would now look like this:



The intent of this last example is to emphasize that if you want to preserve the values contained in data items that follow a variable-length item within the same record, you must move them to another field prior to changing the length

of the variable-length item, and then move them back after the length indicator has been changed.

Searching a Table (SEARCH Statement)

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

To perform **serial searches**:

- Use PERFORM . . . VARYING with subscripting or indexing (discussed in "Creating Variable-Length Tables (DEPENDING ON Clause)" on page 103).
- Use SEARCH and indexing

To perform **binary searches**:

Use indexing and the SEARCH ALL statement.

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

Serial Search

Use the SEARCH statement to perform a serial search beginning at the current index setting. (You can use the SET statement to modify the index setting before using the SEARCH statement.) The conditions in the WHEN option are evaluated in the order in which they are written.

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

Only one level of a table (a table element) can be referenced with one SEARCH statement. SEARCH statements can be nested if you delimit the statement with END-SEARCH, because an imperative statement must follow the WHEN condition and because the SEARCH statement is itself conditional.

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

Arranging the table so that the data used most often is at the beginning also enables more efficient serial searching. If the table is large and is presorted, a binary search is more efficient.

Binary Search (SEARCH ALL Statement)

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

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

For example, a table defined like this:

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

could be searched using the following instructions:

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

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

SEARCH Statement Examples

Example 1:

```
01 ARRAY-VALUES.
   05 FILLER                  PIC 999V99  VALUE  11.11.
   05 FILLER                  PIC 999V99  VALUE  22.22.
   05 FILLER                  PIC 999V99  VALUE  33.33.
   05 FILLER                  PIC 999V99  VALUE  44.44.
   05 FILLER                  PIC 999V99  VALUE  55.55.
   05 FILLER                  PIC 999V99  VALUE  66.66.
   05 FILLER                  PIC 999V99  VALUE  77.77.
   05 FILLER                  PIC 999V99  VALUE  88.88.
   05 FILLER                  PIC 999V99  VALUE  99.99.
   05 FILLER                  PIC 999V99  VALUE 111.11.

01 ARRAY-TBL REDEFINES ARRAY-VALUES.
   05 ARRAY-ELEMENT OCCURS 10 TIMES
        ASCENDING KEY IS AE-KEY
        INDEXED BY AE-INDEX.
      10 AE-KEY               PIC 999V99.

SET AE-INDEX TO 1
MOVE 2 TO RETURN-CODE
SEARCH ARRAY-ELEMENT
  AT END
    MOVE 4 TO RETURN-CODE
  WHEN AE-KEY(AE-INDEX) = 77.77
    CONTINUE
  WHEN AE-KEY(AE-INDEX) = 88.88
    MOVE 0 TO RETURN-CODE
END-SEARCH
```

Values after execution:

```
RETURN-CODE = 2
AE-INDEX points to the TABLE-ELEMENT that equals 77.77
```

Example 2:

```
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).

SET TE1-INDEX TO 1
SET TE2-INDEX TO 4
SET TE3-INDEX TO 1
MOVE "A1234" TO KEY1 (TE1-INDEX, TE2-INDEX, TE3-INDEX + 2)
MOVE "AAAAAAAA00" TO KEY2 (TE1-INDEX, TE2-INDEX, TE3-INDEX + 2)
SEARCH TABLE-ENTRY3
  AT END
    MOVE 4 TO RETURN-CODE
  WHEN TABLE-ENTRY3(TE1-INDEX, TE2-INDEX, TE3-INDEX)
      = "A1234AAAAAAAA00"
    MOVE 0 TO RETURN-CODE
END-SEARCH
```

Values after execution:

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

Processing Table Items (Intrinsic Functions)

As pointed out in Chapter 4, “Data Representation and Assignment” on page 42, which introduces intrinsic functions, some intrinsic functions process numeric arguments, some process alphanumeric arguments, and some process either type of argument. Therefore, you can use any of the intrinsic functions to process alphanumeric or numeric table items as long as the table item's data description is compatible with the function's argument requirements. The *COBOL/VSE Language Reference* describes the required data formats for the arguments of the various intrinsic functions.

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

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

You may often need to process the data in tables iteratively. Traditionally, this processing has been performed with constructs such as the PERFORM and SEARCH statements. However, for intrinsic functions that accept multiple arguments, you can use the ALL subscript (valid only in intrinsic functions) to reference all the items in the table or single dimension of the table. The iteration is handled

automatically, making your code shorter and simpler. The following example determines the maximum value in Table-One:

```
Compute Table-Max = Function Max(Table-One(ALL))
```

The next example sums a cross-section of Table-Two:

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

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

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

The following example also shows the power of intrinsic functions used with the ALL subscript. Notice the simplicity of the code and that you do not have to explicitly write constructs to loop through all the elements of the table.

```
01  Employee-Table.
    02  Emp-Count          Pic s9(4)  usage binary.
    02  Emp-Record         occurs 1 to 500 times
                           depending on Emp-Count.
        03  Emp-Name       Pic x(20).
        03  Emp-Idme       Pic 9(9).
        03  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))
```

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

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

Efficient Coding for Tables

For efficient table-handling, follow these suggestions:

- If the table is searched sequentially, put the data values most likely to satisfy the search criteria at the beginning of a table.
- Using *index-names* instead of subscripts is more efficient, but subscripting may be easier to understand and maintain. Relative index references are executed as fast as direct index references. For additional details, see “Subscripting” on page 96 and “Subscripting Using Index-Names (Indexing)” on page 97.
- Use binary (COMP) data items with 9 or fewer digits for subscripts and OCCURS DEPENDING ON objects. Use fewer than 5 digits, if possible.
- Avoid referencing errors by coding subscript and index checks into your program.

Chapter 8. Selection and Iteration

This chapter explains the control structures you can use to choose alternative program actions based on the outcome of a decision and how to control looping within your program.

Selection (IF and EVALUATE Statements)

Selection is the means of providing for alternative program actions depending on the tested value of data items.

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

The IF and EVALUATE statements, as well as conditional expressions are discussed below.

IF Statement

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

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

IF Statement with a Null Branch

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

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

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

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

Nested IF Statements

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

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

Consider the following pseudocode for a nested IF statement:

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

In this case, an IF is nested, along with a sequential structure, in one branch of another IF. The structure for this logic is shown in Figure 39 on page 116.

When you code a structure like the one in Figure 39, the END-IF closing the inner nested IF becomes very important. Use END-IF instead of a period, because a period would terminate the outer IF structure as well.

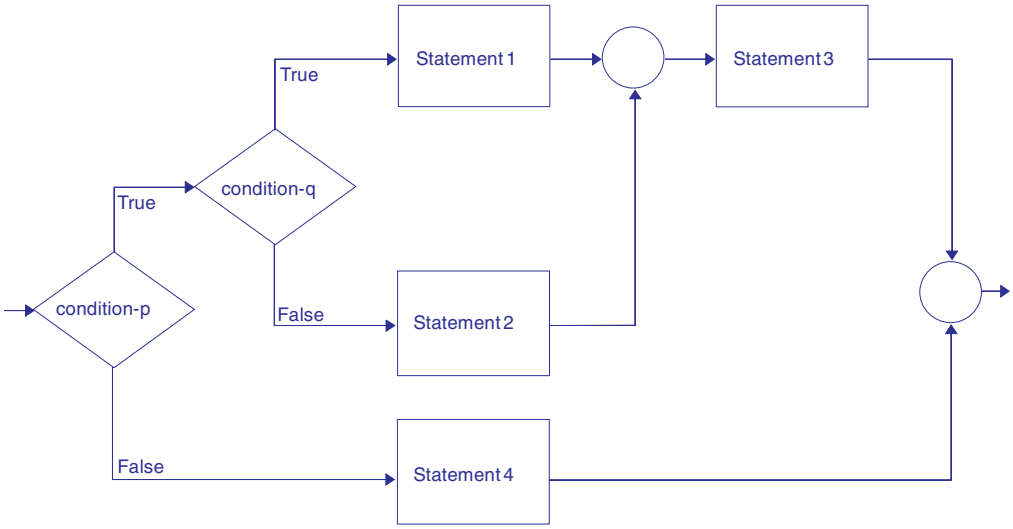


Figure 39. A Control Logic Structure for Nested IF Statements

When IF statements are nested, readability and debugging will be easier if each IF statement has its own END-IF scope-terminator and if proper indentation is used. For example:

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

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

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

If this programmer intended that ELSE PERFORM D depends on IF A = 1, the code would look like this:

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

EVALUATE statement

The EVALUATE statement is an expanded form of the IF statement.

An IF statement allows your program to act on one of two conditions: true or false. When the condition is true, one action is performed. When the condition is not true, a different action is performed (the ELSE clause). If you had three or more possible conditions instead of just two, and you were limited to using IF statements, you would need to nest or cascade the IF statements. Such nested IF statements are a common source of logic errors and debugging problems.

With the EVALUATE statement, you can test any number of conditions in a single statement and have separate actions for each. In structured programming terms, this is a “case” construct. It can also be thought of as a form of decision table. For an example and discussion of the EVALUATE statement used to implement a case structure, see “EVALUATE Statement” on page 36.

Conditional Expressions

The IF and EVALUATE statements let you specify alternative program actions, depending on the true or false value of a condition expression. COBOL lets you specify any of these simple conditions:

- Class condition—for testing whether data is alphabetic, numeric, DBCS, or Kanji. This conditional expression is discussed in “Checking for Incompatible Data (Numeric Class Test)” on page 80. NUMPROC(PFD) which bypasses invalid sign processing may affect the outcome of a test for numeric data.
- User-defined condition—for testing a level-88 condition name to discover whether or not a data item contains a particular value or range of values.
- Relation condition—for comparing two items
- Sign condition—for testing whether a numeric operand is less than, greater than, or equal to zero.
- Switch-status condition—for testing whether an UPSI switch is on or off.

You can create combined conditions by use of logical connectives (AND, OR, or NOT) and you can combine combined conditions.

Specific rules for using conditions are given in *COBOL/VSE Language Reference*.

Use level-88 items to define *condition-names* that you can test to control the processing of **switches** and **flags**.

Condition-Names (Switches and Flags)

Some program decisions are based on whether the value of a data item is true or false, on or off, yes or no. Define level-88 items with meaningful names to act as switches to control these two-way decisions in your program. For example, to test for an end-of-file condition for an input file named Transaction-File, you could use the following data definitions:

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

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

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

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

Some program decisions are based not on an on or off condition of a data item, but instead, depend on the particular value (or range of values) of a data item. To test for more than two specific values, you can assign more than one condition-name to a field by using multiple level-88 items. When condition-names are used to give more than just on or off values to a field, the field is generally referred to as a flag, not a switch.

Consider a program that updates a master file. The updates are read from a transaction file. The records of the transaction file contain a field for the function 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".
    05  ...
```

The code in the Procedure Division for testing these *condition-names* might look like this:

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

Flags and switches make your code easier to modify. If you need to change the values for a condition, you have to change only the level-88 *condition-name* value. The name of the condition as you use it in the Procedure Division need not be changed. For example, a program that uses a *condition-name* to test a field for a given numeric range—a salary range—need not be changed. If the program must be modified to check for a different salary range, you would need to change only the *condition-name* value in the Data Division. You do not need to make changes in the Procedure Division.

Resetting Condition-Names (Switches and Flags)

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

- SET *condition-name* TO TRUE

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

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

  The **SET** statement in the following example is exactly equivalent to
  `Move "y" to Transaction-EOF-Flag.`

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

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

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

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

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

- SWITCH-OFF

  Establish a data item with this description:

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

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

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

    Move SWITCH-OFF to Transaction-EOF-Switch
  ```

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

Iterative Loops (PERFORM Statement)

For looping (repeating the same code), use one of the forms of the PERFORM statement. You can use the PERFORM statement to control the looping with a definite number or with a decision.

PERFORM statements can be in-line or out-of-line. For suggestions on deciding which format to use, see “In-Line PERFORM Statement” on page 39.

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

Coding a Loop to Be Executed a Definite Number of Times

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

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

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

Conditional Looping

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

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

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

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

When control reaches the PERFORM statement, the condition (MONTH EQUAL DECEMBER) is tested. If it is satisfied, control is transferred to the INSPECT statement. If it 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 may want to code the WITH TEST BEFORE clause.)

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

Looping through a Table

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

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

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

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

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

 INSPECT ...
```

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

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

Executing a Group of Paragraphs or Sections

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

When you use PERFORM . . . THRU, use a paragraph-EXIT statement to clearly indicate the end point for the series of paragraphs.

Intrinsic functions can make the task of the iterative processing of tables simpler and easier for you to code. For information on using the ALL subscript with intrinsic functions to reference all the items in a table, see “Processing Table Items (Intrinsic Functions)” on page 113.

Chapter 9. File Input/Output Overview

This section contains general information on COBOL input/output coding and on using input/output files.

Reading and writing data is an essential part of every program. Your program retrieves information, processes it as you specify, and then produces the results.

The source of the information and the target for the results can be one or more of the following:

A direct-access device
A magnetic tape
A printer
A terminal
A card reader or punch (unit-record device)
Another program to which you pass data

The information as it exists on the external device is a **physical record** or **block**. It is an actual physical collection of information that is handled as a unit by the system during input/output operations.

Your COBOL program does not handle these physical records. The information as it is used by your COBOL program is a **logical record** (which may or may not be the same as the physical record). A logical record can be a complete physical record, it can be part of a physical record, or it can include parts or all of one or more physical records. Your COBOL program handles only these logical records, and it handles them exactly as you have defined them.

In COBOL, a collection of physical and logical records is a **file**; that is, a sequence of pieces of information that your program can process.

File Organization and Input/Output Devices

Depending on the input/output device, your file organization will be either sequential, indexed, or relative. You should decide on the devices and file types to be used when you design your program. SAM and VSAM are the two access methods available with COBOL/VSE that will handle the input/output requests to the operating system for the storage and retrieval of records from the input/output devices.

Sequential File Organization

The arrangement of records is established by the physical order in which they are entered when the file is created. 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 record transmission (access) mode allowed for sequential files is sequential only.

Indexed File Organization

Each record in the file contains a field whose contents form the record key. The position of this key field is the same in each record.

The index component of the file provides the logical arrangement of the main file, ordered by record key. The actual physical arrangement of the records in the main file is not significant to your COBOL program.

An indexed file can also make use of alternate indexes—keys that let you access the file using a different logical arrangement of the records.

The record transmission (access) modes allowed for indexed files are sequential, random, or dynamic. When indexed files are read or written sequentially, the sequence is that of the key values. For a description of random and dynamic record transmission, see "File Access Modes" on page 156.

Relative File Organization

Records in the file are identified by their location relative to where the file begins. The first record in the file has a relative record number of 1, the tenth record has a relative record number of 10, and so forth.

The record transmission (access) modes allowed for relative files are sequential, random, or dynamic. When relative files are read or written sequentially, the sequence is that of the relative record number. For a description of random and dynamic record transmission, see "File Access Modes" on page 156.

File Organization on Sequential-Only Devices

Terminals, printers, card readers, and punches are called unit-record devices. They work a "line" at a time. After that line is processed, it exits the device. Therefore, you must work sequentially with each record as it is presented to your program or as your program sends it out.

On a tape, the records are always arranged sequentially, and your program must process them sequentially. Use SAM physical sequential files. The records on tapes may be of fixed or variable length, and the rate of data transfer is faster than it is for cards.

File Organization on Direct-Access Storage Devices

Direct-access storage devices hold many records at a time. Record arrangement in the file is significant because it determines the ways your program can process the data. Because of this, your program's use of direct-access storage can be more varied than its use of the sequential-only devices.

Several types of direct-access file organization are possible:

Sequential (VSAM or SAM)
Indexed (VSAM)
Relative (VSAM)

If you need to process records randomly, use VSAM indexed or relative files.

Sometimes, the file processing method has been determined for you by the specifications for your application program or by the standards of your installation. But, if the decision is yours, you need to consider several things:

- If a large percentage of the file is referenced or updated in your application program, sequential processing is faster than indexed or relative. If a small percentage of records is processed during each run of your application program, use indexed or relative access.
- A SAM or VSAM sequential file is the simplest file type. Either works for an application that uses only sequential access of fixed-length or variable-length records and no insertion of records between existing ones.
- An indexed file is the most flexible file. It may be used for applications requiring both sequential and random access in the same program. A VSAM indexed file can make use of fixed-length or variable-length records.
- A relative file works well for an application that performs random insert and delete operations.

Figure 40 shows the possible file organizations, access modes, and record length attributes for COBOL files.

Figure 40. Summary of File Organizations, Access Modes, and Record Lengths

File Organization	Sequential Access	Random Access	Dynamic Access	Fixed Length	Variable Length
SAM (Physical Sequential)	YES	NO	NO	YES	YES
VSAM Sequential (ESDS)	YES	NO	NO	YES	YES
VSAM Indexed (KSDS)	YES	YES	YES	YES	YES
VSAM Relative (RRDS)	YES	YES	YES	YES	YES

COBOL Input/Output Coding

You code your COBOL program according to the types of files and record lengths you decide to use. The general format of input/output coding is shown in Figure 41 on page 126. Explanations of user-supplied information (lowercase) follow the figure.

```
IDENTIFICATION DIVISION.
    ⋮
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT filename ASSIGN TO assignment-name
    ORGANIZATION IS org ACCESS MODE IS access
    RECORD/RELATIVE KEY IS keyname
    FILE STATUS IS status
    ⋮
DATA DIVISION.
  FILE SECTION.
   FD filename
   01 recordname.
      nn . . . fieldlength & type
      nn . . . fieldlength & type
    ⋮
WORKING-STORAGE SECTION
 01 status  PICTURE 99.
    ⋮
PROCEDURE DIVISION.
    ⋮
      OPEN iomode filename
    ⋮
      READ filename
    ⋮
      WRITE recordname
    ⋮
      CLOSE filename
    ⋮
  STOP RUN.
```

Figure 41. Overview of COBOL Input/Output Coding

The user-supplied information in Figure 41 can be explained as follows:

filename
Any legal COBOL name. You must use the same file name on the SELECT and the FD statements, and on the READ, OPEN, and CLOSE statements. Additionally, the file name is required if you use the START or DELETE statements. This name is not necessarily the actual name of the file as it is known to the system. Each file requires its own SELECT, FD, and input/output statements.

assignment-name
Any 1- to 30-character name you choose, provided that it follows COBOL and system naming rules. This *assignment-name* becomes important later, when you get ready to run your program, at which time you specify it on a DLBL, TLBL, or ASSGN statement.

org
Specifies the organization: SEQUENTIAL, INDEXED, or RELATIVE. This clause is optional for SAM files because SAM is implicitly sequential.

access
Specifies the access mode, SEQUENTIAL, RANDOM, or DYNAMIC. For sequential file-processing, you can omit this clause.

keyname
Defines a data item you will later fill with the key value of a record you want before you do a READ, START, or WRITE. A *keyname* is used only if you are using a VSAM indexed or relative file.

status
Contains the 2-character COBOL FILE STATUS key.

recordname
Contains the name of the record used in the WRITE and REWRITE statements.

fieldlength
Contains the logical length of the field.

type
Must match the record format of the file. If you break the record description entry beyond the level-01 description, each element should map accurately against the fields of the record.

iomode
Specifies INPUT or OUTPUT mode. If you are only reading from a file, specify INPUT. If you are only writing to it, specify OUTPUT or EXTEND. If you are doing both, specify I-O.

Note: EXTEND is only supported for sequentially accessed VSAM files. For SAM files it is not supported and will, if specified, result in the OPEN statement terminating with a file status code of '37'.

See Chapter 10, “Processing SAM Files” on page 133 and Chapter 11, “Processing VSAM Files” on page 152 for more complete details on processing SAM and VSAM files.

File Availability

The concepts of file availability and creation affect OPEN processing, OPTIONAL files, and file status codes 05 and 35. The successful execution of an OPEN statement determines the availability of the file.

Note: Optional files are not necessarily present each time the program is run. You can define files opened in INPUT, I-O, or EXTEND mode as optional by defining them with the SELECT OPTIONAL phrase in the FILE-CONTROL paragraph of your program.

For example, an OPEN I-O of a nonoptional file that is not available results in file status 35. If the file is OPTIONAL, the OPEN I-O will create the file and return file status 05.

File availability and creation are defined differently for SAM and VSAM files.

For details about SAM files see “Availability of SAM Files” on page 140, and for details about VSAM files see “Availability of VSAM Files” on page 168.

Input-Output Using EXTERNAL Files

Using the EXTERNAL clause for files allows separately compiled programs within the run unit to have access to common files. The example on page 129 shows some of the advantages of using EXTERNAL files:

- The main program can reference the record area of the file, even though the main program does not contain any I/O statements.

- Each subprogram can control a single I/O function, such as OPEN or READ.
- Each program has access to the file

The example on page 129 also illustrates that the following items are required to successfully process an EXTERNAL file:

- The *file-name* in the SELECT clause of all the programs accessing the file must match.
- The *assignment-name* in the ASSIGN clause of all the programs accessing the file must match.
- The *data-name* in the FILE STATUS clause of all the programs that will check the file status code must match.
- EXTERNAL must be coded in the file's FD entry in all the programs accessing the file.
- For all programs that want to check the same file status field, the EXTERNAL clause must be coded on the level-01 data definition for the file status field in each program.

The following table gives the names of the main program and subprograms used in the example shown in Figure 43 on page 129 and describes their functions.

Figure 42. Program Names for Input/Output Using EXTERNAL Files

Program Name	Function
ef1	This is the main program. It calls all the subprograms, and then verifies the contents of a record area.
ef1openo	This program opens the external file for output and checks the File Status Code.
ef1write	This program writes a record to the external file and checks the File Status Code.
ef1openi	This program opens the external file for input and checks the File Status Code.
ef1read	This program reads a record from the external file and checks the File Status Code.
ef1close	This program closes the external file and checks the File Status Code.

Additionally, COPY statements ensure that each subprogram contains an identical description of the file.

Each program in the example declares a data item with the EXTERNAL clause in its Working-Storage Section. This item is used to check file status codes, and is also placed using the COPY statement.

Each program uses three Copy Library members:

- The first is named `efselect` and is placed in the FILE-CONTROL paragraph.

```
Select ef1
   Assign To ef1
   File Status Is efs1
   Organization Is Sequential.
```

- The second is named `effile` and is placed in the File Section.

```
Fd ef1 Is External
          Record Contains 80 Characters
          Recording Mode F.
01  ef-record-1.
    02  ef-item-1   Pic X(80).
```

- The third is named `efwrkstg` and is placed in the Working-Storage Section.

```
01  efs1            Pic 99 External.
```

```
 Identification Division.
 Program-ID.
     ef1.
*
* This is the main program that controls the external file
* processing.
*
 Environment Division.
 Input-Output Section.
 File-Control.
     Copy efselect.
 Data Division.
 File Section.
     Copy effile.
 Working-Storage Section.
     Copy efwrkstg.
 Procedure Division.
     Call "ef1openo"
     Call "ef1write"
     Call "ef1close"
     Call "ef1openi"
     Call "ef1read"
     If ef-record-1 = "First record" Then
       Display "First record correct"
     Else
       Display "First record incorrect"
       Display "Expected: " "First record"
       Display "Found   : " ef-record-1
     End-If
     Call "ef1close"
     Goback.
 End Program ef1.
```

Figure 43 (Part 1 of 4). Input/Output Using EXTERNAL Files

```
       Identification Division.
       Program-ID.
           ef1openo.
      *
      * This program opens the external file for output.
      *
       Environment Division.
       Input-Output Section.
       File-Control.
           Copy efselect.
       Data Division.
       File Section.
           Copy effile.
       Working-Storage Section.
           Copy efwrkstg.
       Procedure Division.
           Open Output ef1
           If efs1 Not = 0
             Display "file status " efs1 " on open output"
             Stop Run
           End-If
           Goback.
       End Program ef1openo.
       Identification Division.
       Program-ID.
           ef1write.
      *
      * This program writes a record to the external file.
      *
       Environment Division.
       Input-Output Section.
       File-Control.
           Copy efselect.
       Data Division.
       File Section.
           Copy effile.
       Working-Storage Section.
           Copy efwrkstg.
       Procedure Division.
           Move "First record" to ef-record-1
           Write ef-record-1
           If efs1 Not = 0
             Display "file status " efs1 " on write"
             Stop Run
           End-If
           Goback.
       End Program ef1write.
```

Figure 43 (Part 2 of 4). Input/Output Using EXTERNAL Files

```
       Identification Division.
       Program-ID.
           ef1openi.
      *
      * This program opens the external file for input.
      *
       Environment Division.
       Input-Output Section.
       File-Control.
           Copy efselect.
       Data Division.
       File Section.
           Copy effile.
       Working-Storage Section.
           Copy efwrkstg.
       Procedure Division.
           Open Input ef1
           If efs1 Not = 0
             Display "file status " efs1 " on open input"
             Stop Run
           End-If
           Goback.
       End Program ef1openi.
       Identification Division.
       Program-ID.
           ef1read.
      *
      * This program reads a record from the external file.
      *
       Environment Division.
       Input-Output Section.
       File-Control.
           Copy efselect.
       Data Division.
       File Section.
           Copy effile.
       Working-Storage Section.
           Copy efwrkstg.
       Procedure Division.
           Read ef1
           If efs1 Not = 0
             Display "file status " efs1 " on read"
             Stop Run
           End-If
           Goback.
       End Program ef1read.
```

Figure 43 (Part 3 of 4). Input/Output Using EXTERNAL Files

```
       Identification Division.
       Program-ID.
           ef1close.
      *
      * This program closes the external file.
      *
       Environment Division.
       Input-Output Section.
       File-Control.
           Copy efselect.
       Data Division.
       File Section.
           Copy effile.
       Working-Storage Section.
           Copy efwrkstg.
       Procedure Division.
           Close ef1
           If efs1 Not = 0
             Display "file status " efs1 " on close"
             Stop Run
           End-If
           Goback.
       End Program ef1close.
```

Figure 43 (Part 4 of 4). Input/Output Using EXTERNAL Files

Checking for Input/Output Errors

After each input/output statement is executed for a file, the FILE STATUS key is updated with a value that indicates the success or the failure of the operation. Using a FILE STATUS clause, you can test the FILE STATUS key after each input/output statement and set up error-handling procedures to use when a nonzero file status code is returned. Checking the file status key is highly recommended. For VSAM files, you can use a second *data-name* in the FILE STATUS clause to get additional VSAM return code information.

For details on using the FILE-STATUS clause, see “File Status Key” on page 198.

Another way of handling input/output errors is to set up error/exception declaratives as explained under “Input/Output Error Handling Techniques” on page 194.

Chapter 10. Processing SAM Files

There are certain COBOL language statements needed to process Sequential Access Method (SAM) files. After identifying and describing the SAM files in the Environment and Data Divisions, you can process the records in these files in the Procedure Division of your program.

SAM files are unkeyed files where the records are placed one after another, according to entry order. Your program can process these files only sequentially; that is, you can retrieve (READ) records in the same order as they exist in the file. Each record is placed after the preceding record. After you have created a record, you cannot change its length or its position in the file, and you cannot erase it. You can, however, update SAM files on disk by using the REWRITE statement.

SAM files can be on unit-record, tape, or direct-access storage devices.

SAM processing is best for tables and intermediate storage.

For information on how SAM files are organized and how the system processes them, see the *VSE/ESA System Macros User's Guide*.

COBOL Coding for SAM Files

Environment Division Entries for SAM Files

Use the FILE-CONTROL entry to define the files in your COBOL program to be SAM files and to associate them with the *filenames* for the external files (an external file name is the name by which a file is known to the operating system). For example:

```
FILE-CONTROL.
    SELECT COMMUTER-FILE-MST
    ASSIGN TO COMMUTR
    ORGANIZATION IS SEQUENTIAL
    ACCESS MODE IS SEQUENTIAL.
```

Your ASSIGN clause may include an “S-” before the *name* to document that the file is a SAM file. For example:

```
    ASSIGN TO S-COMMUTR
```

For files on disk or tape, the *name* in the ASSIGN clause (“COMMUTR” in this example) is the same as that on the DLBL or TLBL JCL statement. This relates the file in the program to the external file where the data resides. For more information, see “Input-Output Section” on page 21.

For files on devices other than disk or tape, specify the SYS*nnn* format. For example:

```
    ASSIGN TO SYS011-S-CARDIN
```

This example refers to an input device such as a card reader, which will be assigned to SYS011 (using the JCL ASSGN statement) at run time.

For a full description of the syntax and parameters of the SELECT statement, see the *COBOL/VSE Language Reference*. For a description of the JCL statements used to relate program files with external data files, see *VSE/ESA System Control Statements*.

Data Division Entries for SAM Files

In the FD entry, specify the record format and whether or not the records are blocked. In the associated record description entry or entries, define the *record-name* and record length.

You can explicitly specify a record format of F, V, S, or U in the RECORDING MODE clause, or let COBOL determine the record format from the RECORD clause or from the record descriptions associated with your FD entry for the file. If you want the records to be blocked, you must specify the BLOCK CONTAINS clause in your FD entry.

The following example shows how the FD entry may look for a file with 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).
```

The term **logical record** is used in a slightly different way in the COBOL language and in SAM. For format-V and format-S files, the SAM logical record includes a 4-byte prefix before 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, the definitions of SAM logical record and COBOL logical record are identical.

Fixed-Length Records (Format F)

Fixed-length records are in format F. You may specify RECORDING MODE F.

If you omit the RECORDING MODE clause, the compiler determines the recording mode to be F, if:

> The largest level-01 record associated with the file is not greater than the block size specified in the BLOCK CONTAINS clause.

In addition, you must do one of the following:

- Use the RECORD CONTAINS *integer* clause (RECORD Clause Format 1).

  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 specify all level-01 record description entries associated with the file to be of the same fixed size, and none contains an OCCURS DEPENDING ON clause. 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 within a block (the blocking factor) is constant for every block in the file, except the last block, which may be shorter.

The layout of format-F records is shown in Figure 44 on page 135. See also *VSE/ESA System Macros User's Guide*.

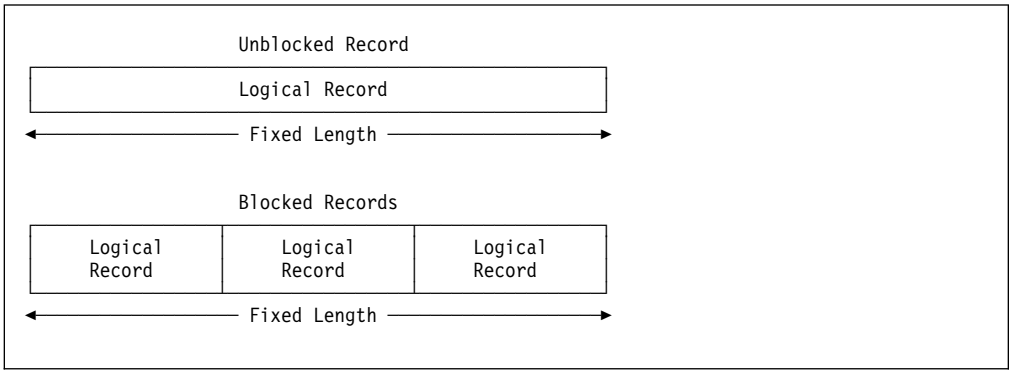


Figure 44. Format-F Records and Blocks

Variable-Length Records (Format V or D)

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.

If you 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 specified in the BLOCK CONTAINS clause.

In addition, you must also do one of the following:

- Use the RECORD IS VARYING clause (RECORD Clause Format 3).
- Use the RECORD CONTAINS *integer-1* TO *integer-2* clause (RECORD Clause Format 2) with integer-1 the minimum length and integer-2 the maximum length of the level-01 records associated with the file.
- Omit the RECORD clause, but specify multiple level-01 records (associated with the file) that are of different sizes, or some of which contain an OCCURS DEPENDING ON clause.

The RECORD clause is sensitive to the CMPR2 compiler option. See *COBOL/VSE Migration Guide* for more information on VS COBOL II Release 2 compatibility and migration.

Format-V records have control fields preceding your data. The control fields are shown in Figure 45 on page 136. Please note how the SAM logical record length is determined by adding 4 bytes (for the control fields) to the record length that is defined in your program. Your program must not include these 4 bytes in its description of the record and record length.

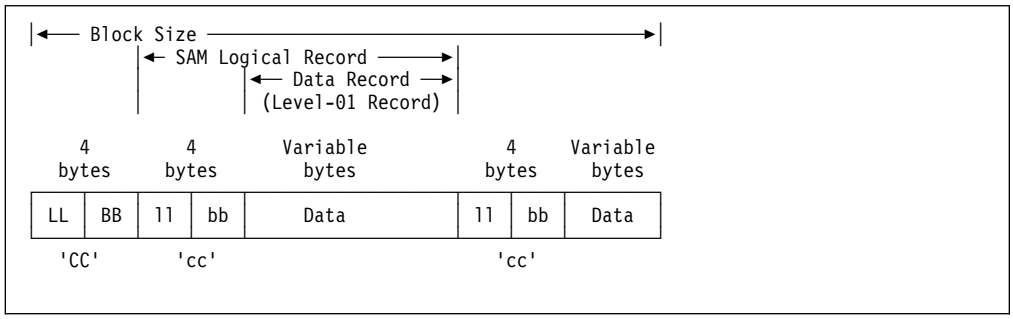


Figure 45. Format-V Records and Blocks

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 SAM logical record contain control information.

ll Represents 2 bytes designating the SAM logical record length (including the 'cc' field).

bb Represents 2 bytes reserved for system use.

For **unblocked** format-V records, the length of the block is:

CC + cc + the data portion.

For **blocked** format-V records, the length of the block is:

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 will be moved without the control bytes.

By specifying 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 may 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.

Spanned Records (Format S)

Spanned records are in format S. A spanned record is a logical record that can be contained in one or more physical blocks. You can specify RECORDING MODE S for spanned records in SAM files assigned to magnetic tape or to direct-access devices.

If you use VSE/VSAM you should note that the VSE/VSAM Space Management for SAM Feature does **not** support the spanned record format.

If you omit the RECORDING MODE clause in the FD entry, the compiler determines the recording mode to be S, if:

> The maximum record length plus 4 is greater than the block size specified in the BLOCK CONTAINS clause.

When you are creating files containing format-S records, if a record is larger than the remaining space in a block, a segment of the record is written to fill the block, and the rest of the record is stored in the next block or blocks, depending on its length.

When you are retrieving files with format-S records, your program can only retrieve complete records.

Spanned records are preceded by control fields, as shown in Figure 46.

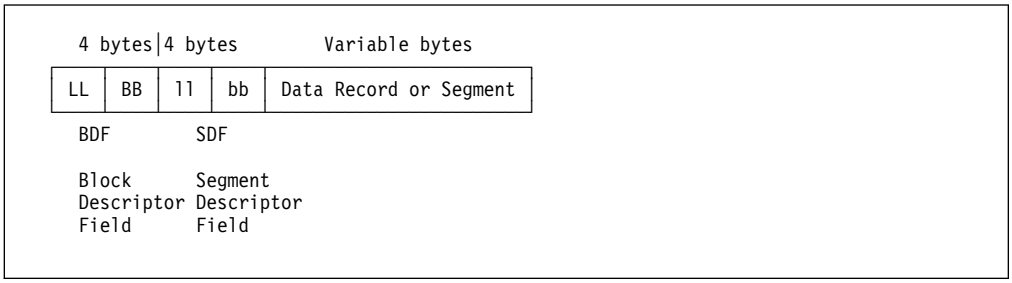


Figure 46. Format-S Records

Each block is preceded by a block descriptor field (BDF). 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 segment descriptor field (SDF). There is one segment descriptor field for each record segment within 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.

A spanned blocked file is a file including fixed-length physical blocks that you define. The logical records can be either fixed or variable in length and their size may be smaller than, equal to, or larger than the physical block size. There are no required relationships between logical records and physical block sizes.

A spanned unblocked file is a file made up of physical blocks, each containing one logical record or one segment of a logical record. The logical records may 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.

You can efficiently use external storage and still organize your files with logical record lengths by defining files with format-S records.

- You can specify 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.

By specifying a READ INTO statement for a format-S file, the record size just read for that file is used in the MOVE statement generated by the compiler. Consequently, you may 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.

Undefined Records (Format U)

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.

The compiler determines the recording mode to be U only if you specify RECORDING MODE U.

The record length is determined in your program based on how you use the RECORD clause. If you:

- Use the RECORD CONTAINS *integer* clause (RECORD Clause Format 1).

  The record length is the value specified for *integer*, regardless of the lengths of the level-01 record description entries associated with the file.

- Use the RECORD IS VARYING clause (RECORD Clause Format 3).

  If you specify values for *integer-1* and *integer-2* (RECORD IS VARYING FROM *integer-1* TO *integer-2*), the maximum record length is the value specified for *integer-2*, regardless of the lengths of the level-01 record description entries associated with the file. 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 (RECORD Clause Format 2) with *integer-1* and *integer-2* matching the minimum length and the maximum length of the level-01 record description entries associated with the file.

  The maximum record length is the value specified for *integer-2*.

- 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.

Each block on external storage is treated as a logical record; there are no record-length or block-length fields. Format-U records are shown in Figure 47 on page 139.

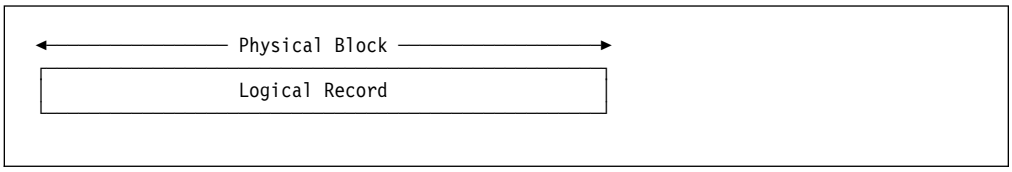


Figure 47. Format-U Records

When you specify a READ INTO statement for a format-U file, the size of the record just read for that file is used in the MOVE statement generated by the compiler. Consequently, you may 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.

Block Sizes

In a COBOL program, you establish the size of a physical record with the BLOCK CONTAINS clause. If you do not use this clause, the compiler assumes that the records are not blocked. Blocking SAM files on disk or tape can enhance processing speed and minimize storage requirements.

If you specify the block size explicitly in the BLOCK CONTAINS clause, it must not be greater than the maximum block size for the device. The block size specified for a format-F file must be an integral multiple of the record length.

If your program uses SAM files on tape, use a physical block size of at least 12 to 18 bytes. Otherwise, the block will be treated as “noise” and skipped over when a parity check occurs while:

Reading a block of records of fewer than 12 bytes

Writing a block of records of fewer than 18 bytes

If you use the VSE/VSAM Space Management for SAM Feature to process a SAM ESDS file, the BLOCK CONTAINS clause is sensitive to the CMPR2 compiler option. When a program compiled with the CMPR2 compiler option accesses a previously implicitly or explicitly defined SAM ESDS file, the block size is determined from the BLOCK CONTAINS clause, unless BLOCK CONTAINS 0 is specified. When a program compiled with the NOCMPR2 compiler option opens a previously implicitly or explicitly defined SAM ESDS file, the block size of the file is determined from the VSE/VSAM catalog. For more information, see *COBOL/VSE Migration Guide*.

BLOCK CONTAINS 0: If your program uses files assigned to direct-access storage devices, it may be more flexible if you code BLOCK CONTAINS 0 in your source program and set the block size at run time in the DLBL statement by using the BLKSIZE parameter. For information about specifying the block size in the DLBL statement of your JCL, see “Job Control Language for SAM Files” on page 142.

If you use VSE/VSAM and the VSE/VSAM Space Management for SAM Feature, you may also benefit from coding BLOCK CONTAINS 0 in your source program.

COBOL/VSE will attempt to determine the block size from the VSAM catalog at OPEN time.

BLOCK CONTAINS 0 should not be specified for:

- Files assigned to tape devices
- Files with fixed-length records, opened as I-O, unless the file is a previously defined SAM ESDS file

The BLOCK CONTAINS clause is ignored for files assigned to unit-record devices.

Block size for ASCII files: If you specify the BLOCK CONTAINS clause for an ASCII sequential file that has a block prefix, be sure to indicate the length of the block prefix in the block size you specify.

Availability of SAM Files

A SAM file is available if one of the following conditions is true:

- If the file resides on a direct-access storage device, a DLBL statement is present for the file, and there is a corresponding VTOC entry for the file, or, for a SAM ESDS file created using the VSE/VSAM Space Management for SAM Feature, there is a corresponding entry in the VSAM catalog.
- If the file does not reside on a direct-access storage device, the system logical unit number specified in the ASSIGN clause is assigned to a valid input device.

If a SAM file is unavailable and the COBOL language defines that the file be created (such as an OPTIONAL file being opened as I-O), COBOL/VSE will create the file using the file information provided by your JCL.

If you attempt to create a SAM file which resides on a direct-access storage device and you have not coded a DLBL statement for the file, the following will occur:

- If you have coded an ERROR declarative procedure in your program for the SAM file, that procedure will be given control.
- If you have defined a file status key for the file, it will be set to 96.
- If you have not defined a file status key, and you did not code an ERROR declarative, then your program will abend.

If you attempt to create a SAM file, and the system logical unit number for the file has been assigned to UA, the following will occur:

- If you have coded an ERROR declarative procedure in your program for the SAM file, that procedure will be given control.
- If you have defined a file status key for the file, it will be set to 90.
- If you have not defined a file status key, and you did not code an ERROR declarative, then your program will abend.

A SAM file opened in INPUT mode is unavailable if one of the following conditions is true:

- If the file resides on a direct-address storage device, a DLBL statement is present for the file, and there is no corresponding VTOC entry.

- If the system logical unit number specified in the ASSGN clause, or on the EXTENT statement for a SAM file on a direct-access storage device, is assigned to UA.

The behavior for handling unavailable files is different if the CMPR2 compiler option is in effect. See *COBOL/VSE Migration Guide*.

Creating SAM Files

When you create a SAM file, use your run-time JCL to:

- Select the type of input/output device to be allocated for the file.

  For a direct-access storage device or a diskette device, use the DLBL JCL statement.

  For a magnetic tape device, use the TLBL JCL statement.

  For a unit-record device or unlabeled tape, use the ASSGN JCL statement.
- Name the file using the file-ID (file-identifier) parameter of the DLBL or TLBL JCL statement.
- Give instructions for the volume on which the file will reside and for volume mounting.

  For a direct-access storage device or a diskette device, use the EXTENT and ASSGN JCL statements.

  For a magnetic tape device, use the volume serial number parameter of the TLBL JCL statement, or let the system choose an output volume.
- For direct-access storage devices or a diskette device, allocate the type and amount of space the file needs. To do this, use the EXTENT JCL statement.

  For a SAM ESDS file, created using the VSE/VSAM Space Management for SAM Feature, you may alternatively use the RECORDS and RECSIZE parameters of the DLBL JCL statement.
- For a SAM ESDS file, indicate whether you want to keep or delete the file after it is closed. To do this, use the DISP parameter of the DLBL JCL statement.
- For a direct-access storage device, specify a different block size than that coded in your program. To do this, use the BLKSIZE parameter of the DLBL JCL statement.

  This parameter does not apply to SAM ESDS files.

Retrieving SAM Files

You retrieve SAM files by using job control statements.

SAM ESDS files
Some file information, such as volume, space, and block size, is stored in the VSAM catalog. In this case, all you have to specify in the DLBL statement are the *file-ID*, the catalog name (if not your installation default catalog), and the disposition (DISP parameter).

Other SAM files
For a file which resides on a direct-access storage device or a diskette device, some information, such as space, is stored in the volume table of contents (VTOC), but you must specify the volume information as well as the *file-ID*.

For a file which resides on a magnetic tape device, you must specify the unit and volume information, as well as the *file-ID*.

For a file which resides on a unit-record device, you must specify the unit information.

Job Control Language for SAM Files

Some of the information about the SAM file must always be specified in the FILE-CONTROL entry, the FD entry, and other COBOL clauses. Other information must be specified in the JCL statement for output files. For input files, the system can obtain information from the VTOC (for direct-access storage devices or diskette devices), or the VSE/VSAM catalog (for SAM ESDS files).

Certain characteristics of SAM files cannot be expressed in the COBOL language, but they may be specified in the JCL statements.

For files which reside on direct-access storage devices, code the parameters of the DLBL JCL statement to specify information about the file, including:

- Block size (BLKSIZE=), if BLOCK CONTAINS 0 RECORDS was specified at compile time. This parameter can be used to specify a block size for SAM files. For SAM ESDS files the block size will be determined from the VSAM catalog, if the file has been previously defined.

  Note that, for output files with fixed-length records, the BLKSIZE parameter must be 8 + a multiple of the record length.
- Block size (BLKSIZE=) for a file opened as OUTPUT if you want to replace the block size specified in the BLOCK CONTAINS clause in your program. If you specify the BLKSIZE parameter in your DLBL JCL statement, VSE, and consequently COBOL/VSE, will use the block size from the BLKSIZE parameter and not the block size specified in your program. For SAM ESDS files the block size will be determined from the VSAM catalog and not from your program, if the file has been previously defined and your program was compiled with the NOCMPR2 compiler option.
- Disposition (DISP=OLD), if you wish to add records to a SAM ESDS file, or magnetic tape file, which you previously created.

Ensuring File Attributes Match Your Program

When the *fixed file attributes* specified in the JCL statements, the VTOC, or the VSE/VSAM catalog (SAM ESDS) for a file and the attributes specified for that file in the SELECT and FD statements of your COBOL program are not consistent, an OPEN statement in your program may not execute successfully. Mismatches in the attributes for file organization, record format (fixed or variable), or record length result in a file status code 39, and the OPEN statement fails.

To prevent common file status 39 problems, follow these guidelines:

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 file. For format-V or format-S files, the maximum record length specified in your program must be exactly 4 bytes smaller than the length attribute of the file. For format-F files, the record length specified in your program must exactly match the length attribute of the file. For format-U files, the maximum

record length specified in your program must exactly match the length attribute of the file.

For details on how logical record lengths are determined from the FD entry and record descriptions in your program, see “Variable-Length Records (Format V or D)” on page 135, “Fixed-Length Records (Format F)” on page 134, and “Undefined Records (Format U)” on page 138.

The easiest way to define variable-length records in your program is to use RECORD IS VARYING FROM *integer-1* TO *integer-2* in the FD entry and specify an appropriate value for *integer-2*. For example, assume that you have determined the length attribute of the file to be 104. Keeping in mind that the maximum record length is determined from the RECORD IS VARYING clause (in which values are specified) 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, use either the RECORD CONTAINS *integer* clause, or omit this clause and specify 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 file. An alternative way to avoid record length conflicts for SAM ESDS files with fixed-length records is to simply code RECORD CONTAINS 0.

Processing New Files: When your COBOL program will write records to a new file, ensure that the file attributes you specify in the JCL statements do not conflict with the attributes you have specified in your program. For example, if you use the BLKSIZE parameter of the DLBL JCL statement to specify the block size of a file with fixed-length records, being opened as OUTPUT, the block size must be 8 + a multiple of the record length.

Processing Printer Files: When your COBOL program will write records to a printer file, ensure that the record length is not larger than the maximum record length allowed by the device to which the file is assigned. For example, the largest record length that may be specified for printer file assigned to a 1403 printer is 132 bytes. One byte is added to the record length to account for the printer control character. If you specify the NOADV compiler option when you compile your program, the maximum record length you may specify in your program is 133 bytes. For information about the maximum record lengths allowed for printer

devices, see the description of the DTFPR system macro in *VSE/ESA System Macros Reference*.

Coding Input/Output Statements for SAM Files

Code the following input/output statements to process a SAM file:

OPEN
Makes the file available to your program.

For all SAM files, you can open the file as INPUT or OUTPUT (depending on device capabilities).

For direct-access storage SAM files, you can also open the file as I-O.

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 at the end of the file.

REWRITE
Updates a record.

CLOSE
Releases the connection between the file and your program.

For the complete syntax of COBOL/VSE statements, see *COBOL/VSE Language Reference*.

Error Processing for SAM

When an input/output statement operation fails, COBOL/VSE will not perform corrective action for you. You choose whether or not your program will continue executing after a less-than-severe input/output error occurs. COBOL/VSE provides these techniques for intercepting and handling certain SAM input/output errors:

The end-of-file phrase (AT END)
The EXCEPTION/ERROR declarative
The FILE STATUS clause
The INVALID KEY phrase

If you do not specify a file status key or you do not code a declarative, then a serious input/output processing error will cause your program to abend.

For details on detecting input/output processing problems, see “Input/Output Error Handling Techniques” on page 194.

Opening a SAM File

Before it can use any READ, WRITE or REWRITE statements to process records in a file, your program must first open the file with an OPEN statement. The concepts of file availability and **optional** files affect OPEN processing, file creation, and file status codes 05, 35, and 96.

Note: Optional files are files that are not necessarily present each time the program is executed. You can define files opened in INPUT and I-O mode as optional by defining them with the SELECT OPTIONAL phrase in the FILE-CONTROL paragraph of your program.

An OPEN statement can execute successfully if the file is available. A file is considered to be available when it has been identified to the operating system with a file name definition. Without a file name definition, a file is not dynamically created.

For additional information on file availability and creation, see the availability topic under “Availability of SAM Files” on page 140

An OPEN operation executes successfully only when the *fixed file attributes* specified in the JCL statements, the VTOC, or the VSE/VSAM catalog (SAM ESDS) for a file and the attributes specified for that file in the SELECT and FD statements of your COBOL program are consistent. Mismatches in the attributes for file organization, the code set, the maximum record size, or the record type (fixed or variable) result in a file status code 39 and the OPEN statement fails.

Preventing the Reopening of a File during Program Execution: Specify CLOSE WITH LOCK so that the file cannot be opened again during program execution.

Processing Tape Files in Reverse Order: Use the REVERSED option of the OPEN statement. The OPEN statement does not reposition the file. Prior to the OPEN statement, the file should be positioned correctly. Subsequent READ statements read the data records in reverse order, starting with the last record.

SAM OPEN NO REWIND: OPEN NO REWIND is supported only for tape files. When the NO REWIND phrase is specified, the OPEN statement does not cause the file to be repositioned. The use of the NO REWIND phrase causes the file to be OPENed at the current file pointer.

OPEN NO REWIND for disk files is not supported. If coded, the phrase is ignored when the file is OPENed.

Processing Multiple Tape Files

The MULTIPLE FILE TAPE clause of the I-O-CONTROL paragraph is syntax-checked, but has no effect on the execution of the program. Use the file-sequence-number of the TLBL JCL statement to specify the position of a file on a multiple file tape. If the tape is unlabeled, use the NO REWIND option of the OPEN and CLOSE statements to ensure the tape is correctly positioned.

Adding Records to a SAM File

You cannot add to a SAM file by opening the file as EXTEND. If you attempt to open a SAM file as EXTEND you will cause the following to occur:

1. If you have coded an ERROR declarative procedure in your program for the SAM file, that procedure will be given control.
2. If you have defined a file status key for the file, it will be set to 37.
3. If none of the above has been done, your program will abend.

You can, however, use VSE system services to add records to a SAM ESDS file, or a magnetic tape file. To do this, specify a disposition of OLD on the DLBL, or a disposition of OLD or MOD on the TLBL JCL statement for the file, open the file as OUTPUT, and use the WRITE statement to add records immediately after the last record in the file.

If you want to add records to a file opened as I-O, you must close the file and open it as OUTPUT.

Updating a SAM File

You can only update SAM files that reside on direct-access storage devices.

You can replace an existing record with another record of the same length by:

- Opening the file as I-O
- Using REWRITE to update an existing record in the file. (The last file processing statement before REWRITE must have been a successful READ statement.)

Writing Your File to a Printer or VSE/POWER Spool File

Controlling the Size of Your Printed Page: You can use the LINAGE clause of the FD entry to control the size of your printed page. In the LINAGE clause, you set 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 treats the file as if you had also specified 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 clause, COBOL first calculates the sum of LINAGE-COUNTER plus *nn*. Subsequent actions depend on the size of *nn*. The END-OF-PAGE imperative statement is executed **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 statement is executed.

Note that any AT END-OF-PAGE or NOT AT END-OF-PAGE imperative statements are executed only if the write operation completes successfully. If the WRITE operation is unsuccessful, control is passed to the end of the WRITE statement, omitting all conditional phrases.

If you use the END-OF-PAGE phrase without the LINAGE clause, the END-OF-PAGE condition exists when channel 12 is sensed (defined either in the FCB or on the carriage control tape). The printer file must be defined as an unblocked, single-buffered file (RESERVE 1 AREA and no BLOCK CONTAINS clause). You should ensure that every WRITE statement in the program using the ADVANCING option advances the printer only one line at a time; otherwise, channel 12 may not be sensed and results may be unpredictable.

Controlling the Vertical Positioning of Records You Write: 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 option from your WRITE statement, you get the equivalent of:

```
AFTER ADVANCING 1 LINE
```

Segmenting Your Printer Files: If you use the VSE/POWER SEGMENT macro to segment your printer file, you should close the file before issuing the SEGMENT macro, and then reopen the file. This will ensure that all records written to the file are received by VSE/POWER before the file is segmented. If you do not close the file, not all records written may appear in the correct segment of the VSE/POWER output.

Alternatively, if the file is defined as single-buffered (RESERVE 1 AREA) and unblocked, all records will be written to the file before the file is segmented and will appear in the correct segment of the VSE/POWER output.

If you use the DISPLAY verb to direct output to SYSLST and concurrently use WRITE ... ADVANCING to a file also assigned to SYSLST, or you use WRITE ... ADVANCING to write to multiple files which are assigned to SYSLST, you should also define the files as single-buffered and unblocked to ensure the records are written in the correct order.

Closing a SAM File

Use the CLOSE statement to disconnect your program from the SAM file. If you try to close a file that is already closed, you will get a logic error.

Certain options which may be specified on the COBOL/VSE CLOSE statement are invalid for certain device types in VSE. These are indicated in the following chart.

CLOSE Statement Format	Unit Record	Sequential Disk Files	Sequential Tape Files
CLOSE			
CLOSE REEL/UNIT	X		
CLOSE REEL/UNIT WITH REWIND	X	X	
CLOSE REEL/UNIT FOR REMOVAL	X	X	
CLOSE WITH NO REWIND	X		
CLOSE WITH LOCK			

" X " means option is invalid for device type.

In instances where such invalid options are specified, the CLOSE statement is processed and a file-status set to indicate the invalid option.

Automatic Closing of Files: If you neglect to close a SAM file in your application, the file is automatically closed for you under the following conditions:

- At the termination of the run unit (STOP RUN, or GOBACK from the main program) **all** open files defined in any COBOL/VSE program within the run unit are closed, both SAM and VSAM.
- At abnormal termination of the run unit (when the LE/VSE run-time option TRAP(ON) is in effect), **all** open files defined in any COBOL/VSE program within the run unit are closed, both SAM and VSAM.

- When CANCEL is used for a COBOL/VSE subprogram, any open nonexternal files defined in that subprogram are closed.
- When a COBOL/VSE subprogram with the INITIAL attribute returns control, any open nonexternal files defined in that subprogram are closed.

File status codes are set when these implicit CLOSE operations are performed, but EXCEPTION/ERROR declaratives are not invoked. Also, LABEL declaratives are not invoked when implicit CLOSE operations are performed.

Processing Labels for SAM Files

Labels can be used to identify magnetic-tape and direct-access volumes. The labels are used to locate the files and are identified and verified by label processing routines of the operating system.

There are two different kinds of labels: standard and nonstandard. Nonstandard user labels are not supported by COBOL/VSE.

Standard labels consist of volume labels and groups of file labels. The volume label group precedes or follows data on the volume; it identifies and describes the volume. The file label groups precede or follow each file on the volume, and identify and describe the file.

- The file labels that **precede** the file are called **header** labels.
- The file labels that **follow** the file are called **trailer** labels. They are similar to the header labels, except that they also contain a count of blocks in the file.
- The file label groups can optionally include standard user labels.
- The volume label groups can optionally include standard user labels.

Standard Label Format

Standard labels are 80-character records that are recorded in EBCDIC or ASCII. The first 4 characters are always used to identify the labels. Figure 48 shows what these *identifiers* are for tape.

Figure 48. Identifiers for Tape Labels

Identifier	Description
VOL1	Volume label
HDR1 or HDR2	File header labels
EOV1 or EOV2	File trailer labels (end-of-volume)
EOF1 or EOF2	File trailer labels (end-of-file)
UHL1 to UHL8	User header labels
UTL1 to UTL8	User trailer labels

The format of the direct-access volume label is the same as the format of the tape volume label group, except that on each volume a file occupies, the initial file label contains file control information. This information appears in the volume table of contents (VTOC) and contains the equivalent of the tape file header and trailer information, in addition to space allocation and other control information.

Standard User Labels

Standard user labels contain user-specified information about the associated file. User labels are optional within the standard label groups. The format used for user header labels (UHL1-8) and user trailer labels (UTL1-8) consists of a label 80 characters in length recorded in:

EBCDIC on DASD or on IBM standard labelled tapes
ASCII on ANSI/ISO/FIPS labelled tapes

The first 3 bytes consist of the characters that identify the label:

UHL for a user header label (at the beginning of a file)
UTL for a user trailer label (at the end-of-volume or end-of-file)

The next byte contains the relative position of this label within a set of labels of the same type. From 1 through 8 labels are permitted.

The remaining 76 bytes consist of user-specified information.

Trailer and Header Labels: User labels are generally created, examined, or updated when the beginning or end of a file or volume (reel) is reached. End- or beginning-of-volume exits are allowed (that is, intermediate trailers and headers may be created or examined).

Trailer labels for files opened as INPUT or I-O are processed when the file has reached AT END condition. You can create, examine, or update up to eight header labels and eight trailer labels on each volume of the file. These labels reside on the initial volume of a multivolume file on a direct-access storage device.

User-Label Track: For direct-access storage volumes, LIOCS writes standard user labels on the first track of your file's first (or only) extent on each volume. The user-label track will contain both user header and user trailer labels.

LABEL Declarative

The USE AFTER LABEL declarative provides label handling procedures at the COBOL source level for handling user labels. The AFTER option indicates processing of standard user labels. List the labels as *data-names* in the LABEL RECORDS clause in the File Description entry for the file.

When the file is opened as:

INPUT
The label is read and control is passed to the LABEL declarative if a USE . . . LABEL declarative is specified for the OPEN option or for the file.

OUTPUT
A buffer area for the label is provided and control is passed to the LABEL declarative, if a USE . . . LABEL declarative is specified for the OPEN option or for the file.

INPUT or I-O
Control is passed to the LABEL declarative for processing trailer labels when a CLOSE statement is executed for the file that has reached the AT END condition.

A special exit may be specified by the statement GO TO MORE-LABELS. When an exit is made from a label declarative section by means of this statement, the system will do one of the following:

- Write the current beginning or ending label and then reenter the USE section at its beginning for further creating of labels. After creating the last label, you must exit by executing the last statement of the section.
- Read an additional beginning or ending label, and then reenter the USE section at its beginning for further checking of labels. When processing user labels, the section will be reentered 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 executed for a user label, the declarative section is not reentered to check or create any immediately succeeding user labels.

Processing SAM ASCII Tape Files

If your program processes an ASCII SAM tape file, you must:

- Specify the ASCII alphabet
- Specify the record formats
- Define the block length

In addition, if your program processes numeric data items from ASCII files, you should use the separately signed numeric data type (SIGN IS LEADING SEPARATE).

Specify the ASCII Alphabet

In the SPECIAL-NAMES paragraph, specify:

```
ALPHABET-NAME IS STANDARD-1
```

In the FD statement for the file, specify:

```
CODE-SET IS ALPHABET-NAME
```

Note: STANDARD-1 means ASCII.

Specify the Record Formats

You can process SAM ASCII tape files with these record formats:

Fixed-Length (format F)
Undefined (format U)
Variable-length (format V)

If you are using variable-length records, you cannot explicitly code format D. Instead, specify RECORDING MODE V. The format information is internally converted to D mode. D-mode records have a 4-byte record descriptor for each record.

Process ASCII File Labels

Standard label processing for ASCII files is no different from standard label processing for EBCDIC files. The system translates ASCII code into EBCDIC before processing.

All ANS user labels are optional. ASCII files may 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. You may not use USE BEFORE STANDARD LABEL procedures.

To create or verify user labels (user label exit), code a USE AFTER STANDARD LABEL procedure.

ASCII files on tape may have:

ANS labels
ANS and user labels
No labels

Any labels on an ASCII tape must be in ASCII code. Tapes containing a combination of ASCII and EBCDIC cannot be read.

Processing SAM 3540-Diskette Unit Files

COBOL/VSE supports 3540 Diskette unit file management. The 3540 diskette unit is quite different from standard direct-access devices. The physical characteristics on the device include:

1. The 3540 diskette is divided into character sectors, with each sector containing 128 characters.
2. Each logical record may occupy no more than one sector, and may be from 1 to 128 characters long.
3. All records in the file must be the same size. Only fixed-length records can reside on a 3540 diskette.
4. Blocking factors can only be 1, 2, 13, or 26 records.

If your program processes a 3540 Diskette unit file, you should be aware of the physical characteristics of the device when you code your:

Data Division entries

Input/Output statements

Chapter 11. Processing VSAM Files

Virtual Storage Access Method (VSAM) is an access method for files on direct-access storage devices. The basic ways to use VSAM are:

To load a file
To retrieve records from a file
To update a file

VSAM processing has some advantages over SAM:

- Data can be protected against unauthorized access
- Cross-system compatibility
- Device independence (no need to be concerned with block size and other control information) is provided
- JCL for COBOL programs using VSAM files is simpler (information needed by the system is provided in the VSAM catalog)

VSAM processing is the **only** way for your COBOL/VSE program to use indexed or relative file organizations.

This chapter provides a brief introduction to VSAM file organization and access modes, describes the coding your COBOL programs need to identify and process VSAM files, and explains how VSAM files must be defined and identified to the operating system before your program can process them.

If you have complex requirements, or are going to be a frequent user of VSAM, you should review the VSE/VSAM publications. A list of these is given in “Bibliography” on page 448.

VSAM Terminology

VSAM and COBOL use slightly different terminology when referring to files. For example, file organization refers to sequential, indexed, and relative files. The corresponding VSAM names are: entry-sequenced, key-sequenced, and relative-record files. The term *file* in the following discussion may refer either to a COBOL file or to a VSAM file.

Figure 49 shows some examples of how VSAM terminology is different from the terminology used for SAM files.

Figure 49 (Page 1 of 2). VSAM Terminology

VSAM Term	Similar Non-VSAM Term
ESDS	SAM file
KSDS	ISAM file
RRDS	DAM file
Control interval size (CISZ)	Block size
Buffers (BUFNI/BUFND)	BUFNO
Access Method Control Block (ACB)	Define The File (DTF)

Figure 49 (Page 2 of 2). VSAM Terminology

VSAM Term	Similar Non-VSAM Term
Cluster (CL)	File
Cluster Definition	File allocation
Record size	Record length

VSAM File Organization

The physical organization of VSAM files differs considerably from those used by other access methods. VSAM files are held in control intervals and control areas; the size of these is normally determined by the access method, and the way in which they are used is not visible to you.

There are three types of file organization you can use with VSAM (see Figure 50 on page 155 for a comparison):

VSAM sequential file organization
Also referred to as VSAM ESDS (Entry-Sequenced Data Set) organization.

VSAM indexed file organization
Also referred to as VSAM KSDS (Key-Sequenced Data Set) organization.

VSAM relative file organization
Also referred to as VSAM fixed-length or variable-length RRDS (Relative-Record Data Set) organization.

Note: Throughout this book, the term **VSAM relative record file (or RRDS)** is used to mean relative-record files with fixed-length and variable-length records, unless they need to be differentiated.

VSAM files can be processed in COBOL/VSE programs only after they are defined with access method services, explained under "Defining VSAM Files (Access Method Services)" on page 168.

VSAM Sequential File Organization

In VSAM **sequential file** organization **(ESDS)**, the records are stored in the order in which they were entered. VSAM entry-sequenced files are equivalent to SAM sequential files, except that tape storage or unit record devices cannot be used with VSAM. The order of the records is fixed.

Records in sequential files can only be accessed (read or written) sequentially.

After you have placed a record into the file, you cannot shorten, lengthen, or delete it. However, you can update (REWRITE) a record if the length does not change. New records are added at the end of the file.

VSAM Indexed File 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 within 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. In your COBOL program, you specify this key through the clause:

```
RECORD KEY IS data-name
```

where *data-name* is the name of the key field as you defined it in the record description entry in the Data Division.

You can also specify one or more alternate keys to use for retrieving records. Using alternate keys, you can access the file to read records in some sequence other than the prime key sequence. For example, you could access the file through employee department rather than through employee number. Alternate keys need not be unique. More than one record will be accessed, given a department number as a key. This is permitted if alternate keys are specified as allowing duplicates.

You define the alternate key in your COBOL program with the ALTERNATE RECORD KEY IS clause:

```
ALTERNATE RECORD KEY IS data-name
```

where *data-name* is the name of the key field as you defined it in the record description entry in the Data Division.

To use an alternate index, you need to define a file (using access method services) called the Alternate Index (AIX). For information on defining the Alternate Index, see “Creating Alternate Indexes” on page 169. This file 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.

Indexed files are identified as such in your COBOL program by the ORGANIZATION IS INDEXED clause.

VSAM Relative-Record File Organization

A VSAM **relative record file (RRDS)** contains records ordered by their relative key—the relative key being the relative record number representing the record's location relative to where the file begins. The relative record number identifies the record which can be either fixed or variable in length.

Your COBOL program may use some type of randomizing routine that will associate a key value in each record with the relative record number for that record.

Although there are many techniques used to convert a record key to a relative record number, the most commonly used randomizing algorithm is the division/remainder technique. With this technique, you divide the key by a value equal to the number of slots in the file to produce a quotient and remainder. When you add one to the remainder, the result will be a valid relative record number.

Relative files are identified as such in your COBOL program by the ORGANIZATION IS RELATIVE clause. You may define a relative key to associate each logical record with its relative record number with the RELATIVE KEY IS clause.

Alternate indexes are not supported for VSAM RRDS.

Relative-Record File Organization with Fixed-Length Records

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, while the 10th slot has a relative record number of 10.

Each record in the file occupies one slot, and you store and retrieve records according to the relative record number of that slot.

When you load the file, you have the option of skipping over slots and leaving them empty.

Relative-Record File Organization with Variable-Length Records

In a VSAM variable-length RRDS, the records are ordered according their relative record number. Records are stored and retrieved according to the relative record number you specify.

When you load the file, you have the option of skipping over relative record numbers. Unlike fixed-length RRDS, there are no slots in variable-length RRDS. Instead, there is user-defined free space to allow for more efficient record insertions.

To use a VSAM variable-length RRDS, you:

- Define the file in your COBOL program with the ORGANIZATION IS RELATIVE clause.
- Use the appropriate FD statements in your COBOL program to describe the records with variable-length sizes.
- Define the VSAM file through access method services as a RRDS.

Figure 50 (Page 1 of 2). Comparison of VSAM file organizations

Entry-Sequenced file	**Key-Sequenced file**	**Relative-Record file**
Records are in order in which they are written.	Records are in collating sequence by key field.	Records are in relative record number order.
Access is sequential.	Access is by key through an index.	Access is by relative record number, which is treated like a key.
May have one or more alternate indexes, though not supported in COBOL.	May have one or more alternate indexes.	May not have alternate indexes.
A record's RBA (relative byte address) cannot change.	A record's RBA can change.	A record's relative record number cannot change.
Space at the end of the file is used for adding records.	Distributed free space is used for inserting records and changing their lengths in place.	For fixed-length RRDS, empty slots in the file are used for adding records. For variable-length RRDS, distributed free space is used for adding records and changing their lengths in place.

Figure 50 (Page 2 of 2). Comparison of VSAM file organizations

Entry-Sequenced file	Key-Sequenced file	Relative-Record file
A record cannot be deleted, but you can reuse its space for a record of the same length.	Space given up by a deleted or shortened record is automatically reclaimed within a control interval.	Space given up by a deleted record can be reused.
Can have spanned records.	Can have spanned records.	Cannot have spanned records.
Can be reused as a work file unless it has an alternate index, is associated with key ranges, or exceeds 123 extents per volume.	Can be reused as a work file unless it has an alternate index, is associated with key ranges, or exceeds 123 extents per volume.	Can be reused as a work file.

File Access Modes

You can only access records in VSAM sequential files sequentially. You can access records in VSAM indexed and relative files in three ways, sequentially, randomly, or dynamically.

1. **Sequential access**—Specify ACCESS IS SEQUENTIAL in the FILE CONTROL entry.

   For indexed files, records are accessed in the order of the key field selected (either primary or alternate).

   For relative files, records are accessed in the order of their relative record numbers.

2. **Random access**—Specify ACCESS IS RANDOM in the FILE-CONTROL entry.

   For indexed files, records are accessed according to the value you place in a key field.

   For relative files, records are accessed according to the value you place in the relative key.

3. **Dynamic access**—Specify ACCESS IS DYNAMIC in the FILE-CONTROL entry.

   Dynamic access is a mixed sequential-random access within 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.

   For example, suppose you had an indexed file of employee records, and the employee's hourly wage formed the record key. Also, suppose your program was interested in those employees earning between $7.00 and $9.00 per hour and those earning $15.00 per hour and above. To do this, retrieve the first record randomly (with a random-retrieval READ) based on the key of 0700. Next, begin reading sequentially (i.e. using READ NEXT) until the salary field exceeds 0900. You would then switch back to a random read, this time based on a key of 1500. After this random read, switch back to reading sequentially until you reach the end of the file.

Figure 51 summarizes VSAM file organization, access modes, and record formats (fixed or variable length).

Figure 51. VSAM File Organizations, Access Modes, and Record Lengths

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

COBOL Coding for VSAM Files

There are certain COBOL language statements needed to process VSAM files. After identifying and describing the VSAM files in the Environment and Data Division, you can process the records in the files in the Procedure Division of your program.

Remember that VSAM files must be defined with access method services before your COBOL/VSE program can do any file processing.

Environment Division Entries for VSAM Files

Use the FILE-CONTROL entry to define the VSAM file organizations and access methods for the files in your COBOL program. Figure 52 shows typical FILE-CONTROL entries for VSAM files. The first example is for a VSAM sequential file (ESDS). The second example shows the statements for a VSAM indexed file (KSDS) that will be accessed dynamically. In addition to the primary key, `COMMUTER-NO`, there is an alternate key for this file, `LOCATION-NO`. Example 3 is for a relative-record file (RRDS) to be accessed randomly by the value placed in the relative key, `ITEM-NO`.

Example 1: VSAM Sequential File

```
SELECT S-FILE
  ASSIGN TO SEQUENTIAL-AS-FILE
  ORGANIZATION IS SEQUENTIAL
  ACCESS IS SEQUENTIAL
  FILE STATUS KEY IS FSTAT-CODE VSAM-CODE.
```

Example 2: VSAM Indexed File

```
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 KEY IS FSTAT-CODE VSAM-CODE.
```

Example 3: VSAM Relative File

```
SELECT R-FILE
  ASSIGN TO RELATIVE-FILE
  ORGANIZATION IS RELATIVE
  ACCESS IS RANDOM
  RELATIVE KEY IS RFILE-RELATIVE-KEY
  FILE STATUS KEY IS FSTAT-CODE VSAM-CODE.
```

Figure 52. Example File-Control Entries for VSAM Files

The ORGANIZATION clause will be:

- ORGANIZATION IS SEQUENTIAL—for VSAM sequential files (ESDS)
- ORGANIZATION IS INDEXED—for VSAM indexed files (KSDS)
- ORGANIZATION IS RELATIVE—for VSAM relative files (RRDS)

The FILE STATUS clause specifies fields that are updated by VSAM after each input/output statement to indicate the success or failure of the operation. See "File Status Key" on page 198 and "VSAM Return Code (VSAM Files Only)" on page 201 for information on how to set up the fields to check the returned values.

The index (key) for VSAM indexed files: For a VSAM indexed file, your RECORD KEY definition must agree with the definition in the catalog entry.

If you are using VSAM indexed files with **alternate indexes**, your ALTERNATE RECORD KEY definitions must agree with the definitions in the catalog entry. Any password entries that are cataloged should be coded directly after the ALTERNATE RECORD KEY phrase. Specify WITH DUPLICATES only if your alternate index was cataloged as having duplicate keys.

For further details on using alternate indexes and an example of the relationship between the COBOL FILE-CONTROL entry and the DLBL statements for a VSAM indexed file with alternate indexes, see "Creating Alternate Indexes" on page 169.

Data Division Entries for VSAM Files

VSAM records can be fixed or variable in length. COBOL determines the record format from the RECORD clause and the record descriptions associated with your FD entry for the file.

Since the concept of blocking has no meaning for VSAM files, you may omit the BLOCK CONTAINS clause. The clause is syntax-checked, but it has no effect on the execution of the program.

Fixed-Length Records

The compiler determines the records to be fixed length, if you do one of the following:

- Use the RECORD CONTAINS *integer* clause (RECORD Clause Format 1).
- Omit the RECORD clause and define all the level-01 records (associated with the file) to be the same fixed size.

Variable-Length Records

The compiler determines the records to be variable length, if you do one of the following:

- Use the RECORD IS VARYING clause (RECORD Clause Format 3).

  If you specify values for *integer-1* and *integer-2* (RECORD IS VARYING FROM *integer-1* to *integer-2*), the maximum record length is the value specified for *integer-2*, regardless of the lengths specified in the level-01 record description entries associated with the file. 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 (RECORD Clause Format 2) with integer-1 the minimum length and integer-2 being the maximum length of the level-01 records associated with the file.

  The maximum record length is the value specified for *integer-2*.

- Omit the RECORD clause, but specify multiple level-01 records (associated with the file) that are of different sizes, or some of which 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.

The RECORD clause is sensitive to the CMPR2 compiler option. See *COBOL/VSE Migration Guide* for more information on VS COBOL II Release 2 compatibility and migration.

By specifying 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 may 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 by the READ statement, the size of the record corresponds to its level-01 record description.

Coding Input/Output Statements for VSAM Files

VSAM file processing involves seven COBOL statements:

OPEN
Connect the VSAM file to your COBOL program for processing.

WRITE
Add records to a file or load a file.

START
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, described under “File Position Indicator” on page 161.

READ and READ NEXT
Retrieve records from a file.

REWRITE
Update records.

DELETE
Logically remove records from indexed and relative files only.

CLOSE
Disconnect the VSAM file from your program.

All of the following determine which input/output statements are valid for a given VSAM file:

- Access mode (sequential, random, or dynamic)
- File organization (ESDS, KSDS, or RRDS)
- Mode of OPEN statement (INPUT, OUTPUT, I-O, or EXTEND)

Figure 53 on page 160 shows the possible combinations with **sequential** files (ESDS). The 'X' indicates that the specified statement may be used with the open mode given at the top of the column.

Figure 53. Valid COBOL Statements with Sequential Files (ESDS)

Access Mode	COBOL/VSE 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

Figure 54 shows the possible combinations with **indexed** (KSDS) and **relative** (RRDS) files. The 'X' indicates that the specified statement may be used with the open mode given at the top of the column.

Figure 54 (Page 1 of 2). Valid COBOL Statements with Indexed Files (KSDS) and Relative Files (RRDS)

Access Mode	COBOL/VSE 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	

Figure 54 (Page 2 of 2). Valid COBOL Statements with Indexed Files (KSDS) and Relative Files (RRDS)

Access Mode	COBOL/VSE Statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	

File Position Indicator

The file position indicator indicates the next record to be accessed for sequential COBOL requests. You do not specify the file position indicator anywhere in your program. The file position indicator is set by successful OPEN, START, READ, and READ NEXT statements. Subsequent READ or READ NEXT requests then use the established file position indicator position 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.

Error Processing for VSAM

All errors in processing a VSAM file, whether logic errors in your program or input/output errors on the external storage media, return control to your COBOL program.

When an input/output statement operation fails, COBOL/VSE will not perform corrective action for you. Such errors return control to your program. You choose whether or not your program will continue executing after a less-than-severe input/output error occurs. COBOL/VSE provides the following techniques for intercepting and handling certain VSAM input/output errors:

The end-of-file phrase (AT END)
The EXCEPTION/ERROR declarative
The FILE STATUS clause (file status key and VSAM return code)
The INVALID KEY phrase

If you do not specify a file status key and you do not code a declarative, serious VSAM processing errors can go undetected by your program. VSAM file processing problems do not usually cause an abend, and it is possible your program will be processing wrong data.

If you continue processing after such errors occur, you may impair the integrity of your data. Good coding practice demands that you check the status key value after every input/output request (including OPEN and CLOSE). Each VSAM file should have its own status key defined in your program.

For details on detecting input/output processing problems, see “Input/Output Error Handling Techniques” on page 194.

Opening a File (ESDS, KSDS, or RRDS)

Before your program can use any WRITE, START, READ, REWRITE, or DELETE statements to process records in a file, it must first open the file with an OPEN statement. The concepts of file availability and creation affect OPEN processing, OPTIONAL files, and file status codes 05 and 35.

For example, an OPEN EXTEND of a nonoptional file that is not available results in file status 35, and the OPEN statement fails. If the file is OPTIONAL, the OPEN EXTEND will create the file and return file status 05. The successful execution of an OPEN statement determines the availability of the file.

An OPEN operation executes successfully only when the *fixed file attributes* specified in the VSAM catalog for the file and the attributes specified for that file in the SELECT and FD statements of your COBOL program are consistent. Mismatches in the attributes for file organization (sequential, relative, or indexed), the prime record key, the alternate record keys, the maximum record size, or the record type (fixed or variable) result in a file status code 39 and the OPEN statement fails.

How you code the OPEN statement in your COBOL program for a VSAM file depends on whether the file is empty (has never contained records) or a loaded file. For either type of file, **you should check the file status key after each OPEN statement**.

Opening an Empty File

To open a file that has never contained records (an empty file):

Use OPEN OUTPUT for ESDS files.

Use OPEN OUTPUT or OPEN EXTEND for KSDS and RRDS files (either coding has the same effect). If you have coded the file for random dynamic access, you can also use OPEN I-O if the file is optional.

Optional files are files that are not necessarily present each time the program is run. You can define files opened in INPUT, I-O, or OUTPUT mode as optional by defining them with the SELECT OPTIONAL phrase in the FILE-CONTROL section of your program.

Initially Loading Records Sequentially into a File: Initially loading a file means writing records into the file for the first time. This is not the same as writing records into a file that has contained records that have all been deleted.

To initially load a VSAM file:

Use OPEN I-O (for optional files) or OPEN OUTPUT or OPEN EXTEND
Use sequential processing because it is faster (ACCESS IS SEQUENTIAL)
Use WRITE to add a record to the file

Using OPEN OUTPUT to load a VSAM file will significantly improve the performance of your program. Using OPEN I-O or OPEN EXTEND will have a negative impact on your program's performance.

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.

Figure 55 shows the COBOL statements used for loading a VSAM file.

Initially Loading a File Randomly or Dynamically: Although sequential processing is more efficient, you can use random or dynamic processing to load a file. Because VSAM does not support such processing, COBOL/VSE 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 COBOL/VSE processing steps prepare the file for use and give it the status of a loaded file, having been used at least once. In addition to this extra overhead for preparing files for use, remember that random processing does not consider any user-defined free space and, as a result, any future insertions may be inefficient. Conversely, sequential processing maintains user-defined free space.

Loading a VSAM file with Access Method Services: You can load or update a VSAM file with the IDCAMS REPRO command. See *VSE/VSAM Commands and Macros* for information about REPRO. **REPRO should be used whenever possible**.

Figure 55 shows the COBOL statements used for loading a VSAM file.

Figure 55. Statements Used to Load Records into a VSAM File

	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 LABEL RECORDS	FD entry LABEL RECORDS	FD entry LABEL RECORDS
Procedure Division	OPEN OUTPUT OPEN EXTEND WRITE CLOSE	OPEN OUTPUT OPEN EXTEND WRITE CLOSE	OPEN OUTPUT OPEN EXTEND WRITE CLOSE

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

- For a VSAM entry-sequenced or relative-record file opened as EXTEND, the added records are placed after the last existing records in the file.

- For a VSAM key-sequenced file opened as EXTEND, each record you add must have a record key higher than the highest record in the file.

Reading Records from a VSAM File

Use the READ statement to retrieve records from a file. To read a record, you must have opened the file INPUT or I-O. **You should check the file status key after each READ statement**.

Records in VSAM sequential files can be retrieved only in the sequence in which they were written.

Records in VSAM indexed and relative record files can be retrieved:

Sequentially
According to the ascending order of the key you are using, the RECORD KEY or the ALTERNATE RECORD KEY, beginning at the current position of the file position indicator for indexed files, or according to ascending relative record locations for relative files.

Randomly
In any order, depending on how you set the RECORD KEY or ALTERNATE RECORD KEY or the RELATIVE KEY prior to 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). See “File Access Modes” on page 156 for a complete description of the features of each access mode.

When you want to read **sequentially**, beginning at a specific record, use START before the READ NEXT to set the file position indicator to point to a particular record (see “File Position Indicator” on page 161). When you specify START followed by READ NEXT, the next record is read and the file position indicator is reset to the next record. The file position indicator can be moved around randomly through the use of START, but all reading is done sequentially from that point. You can continue to read records sequentially, or you can use the START statement to move the file position indicator:

```
START file-name KEY IS EQUAL TO ALTERNATE-RECORD-KEY
```

When a direct READ is executed for a VSAM indexed file, based on an alternate index for which duplicates exist, only the first record in the file (base cluster) with that alternate key value is retrieved. You need a series of READ NEXT statements to retrieve each of the file records with the same alternate key. A FILE STATUS value of '02' is returned if there are more records with the same alternate key value to be read; a value of '00' is returned when the last record with that key value has been read.

Updating Records in a VSAM File

The COBOL language statements that can be used to update a VSAM file in the Environment and Data Divisions are the same as those shown in Figure 55 on page 163.

Figure 56 shows the statements that you can use in the Procedure Division for sequential (ESDS), indexed (KSDS), and relative-record (RRDS) files.

Figure 56. Procedure Division Statements Used to Update VSAM Files

ESDS	KSDS	RRDS
ACCESS IS SEQUENTIAL: OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE CLOSE	*ACCESS IS SEQUENTIAL*: OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE DELETE CLOSE	*ACCESS IS SEQUENTIAL*: OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE DELETE CLOSE
ACCESS IS RANDOM: not applicable	*ACCESS IS RANDOM*: OPEN I-O READ WRITE REWRITE DELETE CLOSE	*ACCESS IS RANDOM*: OPEN I-O READ WRITE REWRITE DELETE CLOSE
ACCESS IS DYNAMIC *Sequential Processing*: not applicable	*ACCESS IS DYNAMIC* *Sequential Processing*: OPEN I-O READ NEXT WRITE REWRITE START DELETE CLOSE	*ACCESS IS DYNAMIC* *Sequential Processing*: OPEN I-O READ NEXT WRITE REWRITE START DELETE CLOSE
ACCESS IS DYNAMIC *Random Processing*: not applicable	*ACCESS IS DYNAMIC* *Random Processing*: OPEN I-O READ WRITE REWRITE DELETE CLOSE	*ACCESS IS DYNAMIC* *Random Processing*: OPEN I-O READ WRITE REWRITE DELETE CLOSE

Adding Records to a VSAM file

The COBOL WRITE statement adds a record to a file, without replacing any existing records. The record to be added must not be larger than the maximum record size specified when the file was defined. **You 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, new records must be written in ascending key sequence. If the file is opened EXTEND, the record keys of the records to be added must be higher than the highest primary record key on the file when the file was opened.

For relative files, the records must be in sequence. If you include a RELATIVE KEY data item in the SELECT clause the relative record number of the record just written is placed in that data item.

Adding Records Randomly or Dynamically

When you write records to an indexed file and ACCESS IS RANDOM or ACCESS IS DYNAMIC, the records can be written in any order.

Replacing Records in a VSAM File

To replace records in a VSAM file, use REWRITE on a file that you have opened for I/O. If you attempt to use REWRITE on a file that is not opened I-O, the record is not rewritten and the status key is set to 49. **You should check the file status key after each REWRITE statement**.

- For files accessed sequentially, you must read the record before you issue a REWRITE statement.
- For sequential files, the length of the record you rewrite must be the same as the length of the original record.
- For indexed files, you can change the length of the record you rewrite.

To replace records randomly or dynamically, the record to be rewritten need not be read by the COBOL program. Instead, to position the record you want to update:

- For indexed files, move the record key to the RECORD KEY data item, and then issue the REWRITE.
- For relative files, move the relative record number to the RELATIVE KEY data item, and then issue the REWRITE.

Deleting Records from a VSAM File

Open the file for I/O and use the DELETE statement to remove an existing record from an indexed or relative file. You cannot use DELETE on a sequential file.

When ACCESS IS SEQUENTIAL, or if the file contains spanned records, the record to be deleted must first be read by the COBOL program. 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 ACCESS IS RANDOM or ACCESS IS DYNAMIC, and if the records are not spanned, the record to be deleted need not be read by the COBOL program. To delete a record, the key of the record to be deleted is moved to the RECORD KEY data item and the DELETE is issued. **You should check the file status key after each DELETE statement**.

Closing VSAM Files

Use the CLOSE statement to disconnect your program from the VSAM file. If you try to close a file that is already closed, you will get a logic error. **You should check the file status key after each CLOSE statement**.

Automatic Closing of Files: If you neglect to close a file (SAM or VSAM) in your application, the file is automatically closed for you under the following conditions:

- At the termination of the run unit (STOP RUN, or GOBACK from the main program) **all** open files defined in any COBOL/VSE program within the run unit are closed, both SAM and VSAM.
- At abnormal termination of the run unit (when the LE/VSE run-time option TRAP(ON) is specified), **all** open files defined in any COBOL/VSE program within the run unit are closed, both SAM and VSAM.
- When CANCEL is used for a COBOL/VSE subprogram, any open nonexternal files defined in that program are closed.
- When a COBOL/VSE subprogram with the INITIAL attribute returns control, any open nonexternal files defined in that program are closed.

File status codes are set when these implicit CLOSE operations are performed, but EXCEPTION/ERROR declaratives are not invoked. Also, LABEL declaratives are not invoked when implicit CLOSE operations are performed.

Protecting VSAM Files with a Password

COBOL/VSE supports explicit passwords on VSAM files to prevent unauthorized access and update. To use explicit passwords, specify the optional PASSWORD clause in your program's SELECT statement. Use this clause only if the catalog entry for the file includes a read or an update password.

- If the catalog entry includes a **read** password, the file cannot be opened and accessed in a COBOL program unless the password clause is specified in the FILE-CONTROL paragraph and described 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, the file can be opened and accessed, but not updated unless the password clause is specified in the FILE-CONTROL paragraph and described in the Data Division.
- If the catalog entry includes **both** a read password and an update password, specify the update password in order to both read and update the file in your program.

If your program only retrieves records and does not update them, you need only specify the read password. If your program loads files or updates them, you need to specify the update password.

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 every alternate index in order for the file to be fully password-protected. Where you place the PASSWORD clause becomes important because each alternate index has its own password. The PASSWORD clause must directly follow the key clause to which it applies.

An example of the COBOL code used for a VSAM indexed file with password protection is as follows:

```
    ⋮
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".
```

Availability of VSAM Files

A VSAM file is defined as available if it has a DLBL statement (see Figure 10 on page 22 for an example), has been defined by access method services, and has previously contained a record. A VSAM file is unavailable if it has a DLBL statement, has been defined by access method services, but has never contained a record.

If an indexed or relative file is unavailable and the COBOL language defines that the file be created, COBOL/VSE will write a dummy record in the file and then delete the record. This prepares the file for use.

Notes:

1. A return code of zero will always be returned upon completion of the OPEN statement for a VSAM sequential file.
2. The IDCAMS REPRO command can be used to empty a file. Deleting records in this manner will reset the files high-use Relative Byte Address (RBA) to zero. The file is effectively empty and appears to COBOL as if it never contained a record.

Defining VSAM Files (Access Method Services)

VSAM entry-sequenced, key-sequenced, and relative-record files can be processed in COBOL/VSE only after defining them through access method services (IDCAMS).

A VSAM **cluster** is a logical definition for a VSAM file 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.

You use the access method services DEFINE CLUSTER command to define your VSAM files (clusters). This process includes creating an entry in the VSAM catalog without any data transfer. Specify the following information about the cluster:

- Name of the entry

- Name of the catalog to contain this definition and its password (may use default name)
- Organization—sequential, indexed, or relative
- Device and volumes the file will occupy
- Space required for the file
- Record size and control interval sizes (CISIZE)
- Passwords (if any) required for future access
- For VSAM Indexed files (KSDS) only, specify length and position of the prime key within the records
- For VSAM Fixed-Length Relative-Record files (RRDS):

  DEFINE CLUSTER NUMBERED
  RECORDSIZE(*n*,*n*)
  where:

  n is greater than or equal to the maximum size COBOL record.

  When a file is defined in this manner, all records will be padded to the fixed slot size *n*. If the RECORD IS VARYING ON *data-name* form of the RECORD clause is used, a WRITE or REWRITE will use the length specified in the 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 will always return the fixed slot size in the DEPENDING ON *data-name*.

- For VSAM Variable-Length Relative-Record files (RRDS):

  DEFINE CLUSTER NUMBERED
  RECORDSIZE(*avg*,*m*)
  where:

  avg is the average size COBOL record expected and is strictly less than *m*.

  m is the maximum size COBOL record expected.

For further information, see *VSE/VSAM Commands and Macros*.

Creating Alternate Indexes

An alternate index provides access to the records in a file using more than one key. It accesses records in the same way as the prime index key of an indexed file (KSDS).

When planning to use an alternate index, you must know:

- The type of file (base cluster) with which the index will be associated
- Whether the keys will be unique or nonunique
- Whether the index is to be password-protected
- Some of the performance aspects of using alternate indexes

To use an alternate index, you need to follow these steps:

- Define the alternate index, using the DEFINE ALTERNATEINDEX command.

- Relate the alternate index to the base cluster (the file to which the alternate index gives you access), using the DEFINE PATH command. The base cluster and alternate index are described by entries in the same catalog.
- Build the alternate index, using typically the BLDINDEX command.

Defining the Alternate Index

Since an alternate index is, in practice, a VSAM file that contains pointers to the keys of a VSAM file, 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).

To define and build a catalog entry for the alternate index, use the access method services command:

DEFINE ALTERNATEINDEX

In it, you specify:

- The name of the alternate index
- The name of its related VSAM indexed file
- The location in the record of any alternate indexes and whether they are unique or not
- Whether or not alternate indexes are to be updated when the file is modified
- The name of the catalog to contain this definition and its password (may use default name)

Note: If you intend to use an alternate index on a file that is to be opened for I-O, the base cluster must be defined with share options of 2 or greater. Otherwise, the open I-O will fail.

In your COBOL program the alternate index is identified solely by the ALTERNATE RECORD KEY clause of the FILE CONTROL paragraph. The ALTERNATE RECORD KEY definitions must match the definitions you have made in the catalog entry.

Any password entries you have cataloged should be coded directly after the ALTERNATE RECORD KEY phrase.

Figure 57 maps the relationships between the COBOL FILE-CONTROL entry and the DLBL statements for a VSAM indexed file with alternate indexes.

```
// DLBL MASTRA,'CLUSTER.NAME',0,VSAM,DISP=OLD  1
// DLBL MASTRA1,'PATH1',0,VSAM,DISP=OLD   2
// DLBL MASTRA2,'PATH2',0,VSAM,DISP=OLD   3

FILE-CONTROL.
  SELECT  MASTER-FILE ASSIGN TO MASTRA   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.
```

Figure 57. COBOL FILE-CONTROL Entry / DLBL Statements for VSAM Indexed File with Alternate Indexes

The example shows the connection between a program using two alternate indexes and the required DLBL statements. In this example:

1 The base cluster name is CLUSTER.NAME.

2 The name of the first alternate index path is PATH1.

3 The name of the second alternate index path is PATH2.

4 The *systemname* for the base cluster is specified with the ASSIGN clause.

The *file-names* for the alternate indexes are formed from the *file-name* for the base cluster suffixed by 1 for the first alternate path, 2 for the second alternate path, and so on. Therefore, the *file-name* for the base cluster must not be longer than six characters so that the *file-names* for the alternate paths will not exceed the VSE maximum of seven characters. Coding a seven-character *file-name* for the base cluster will result in I/O for the alternate record keys being processed on the primary key instead of the alternate keys.

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.

Defining Alternate Index Paths

After you define an alternate index, you need to 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 create a path, issue the access method services command:

DEFINE PATH

In it, you can specify:

- The name of the path
- The alternate index to which the path is related
- The name of the catalog that contains the alternate index

Building the Alternate Index

After you have defined the alternate index and its path and have loaded the VSAM indexed file, you can specify the access method services command:

BLDINDEX

This command causes the alternate index to be loaded with index records. BLDINDEX reads all the records in your VSAM indexed file (or base cluster) and extracts the data needed to build the alternate index.

In the BLDINDEX command, you need to identify an input and an output file. The input file is the indexed file (base cluster), and the output file is the alternate index or its path.

Alternatively, as described in the next section, you can use the run-time option AIXBLD to build the alternate index at run time. However, this may adversely affect run-time performance.

Dynamically Invoking Access Method Services

You can dynamically invoke access method services at run time for VSAM indexed files (KSDS) to complete the file and index definition procedures. To do this, you must use the run-time option AIXBLD, which is described in the *LE/VSE Programming Reference*, and make sure that:

1. Your program opens the VSAM file as OUTPUT.
2. The access method services utility program is available.
3. You provide a job catalog (IJSYSUC) JCL DLBL statement for the VSAM catalog containing the definition of the alternate index; no job catalog DLBL is required if the definition of the alternate index is contained in the VSAM master catalog.

Use this procedure only when necessary, since it uses a large amount of processing time. When possible, preload the base cluster before defining alternate indexes.

Job Control Language for VSAM files

All VSAM files have to be predefined and cataloged through the access method services command, DEFINE. Most of the information about a VSAM file is in the catalog, as opposed to in the run-time JCL statements. Consequently, you need to specify only minimal JCL for a VSAM file. Usually, the input and data buffers are the only variables you are concerned about. The basic JCL statement you need for your VSAM files is:

```
// DLBL file-name,'file-ID',,VSAM,CAT=catname
```

file-ID
Must be the same as the name specified in the access method services command DEFINE CLUSTER or DEFINE PATH.

VSAM
Indicates a VSE/VSAM file label.

CAT=*catname*
This parameter specifies the file-name of the DLBL statement for the catalog owning this VSAM file. Specify this parameter only if the VSAM file is owned by a VSAM catalog other than the job catalog, or, if there is no job catalog, the master catalog.

You may also specify the following parameters in the DLBL statement for your VSAM file:

BUFSP
Specifies the number of bytes of virtual storage to be allocated as buffer space for the file. This parameter takes precedence over the information provided in the catalog or in the ACB, if its value is higher.

BUFND
Specifies the number of I/O buffers to hold control intervals containing data records.

BUFNI
Specifies the number of I/O buffers to hold control intervals containing index records.

Considerations for VSAM Performance

Most likely, your system programmer is responsible for COBOL/VSAM performance tuning. There are some things that you, as an application programmer, might want to control:

- Invoking access methods services

  Build your alternate indexes in advance, using IDCAMS.

- Buffering

  The default is one index (BUFNI) and two data buffers (BUFND). Specify additional data buffers for sequential access and specify additional index buffers for random access. Specify both BUFND and BUFNI when ACCESS IS DYNAMIC. (See the *VSE/VSAM Programmer's Reference*.)

  Avoid coding additional buffers unless your application will run interactively; and then code buffers only when response-time problems arise that may be caused by I/O delay.

- Loading records, using access methods services

  The access methods services REPRO command can update an indexed file as fast or faster than any COBOL program when:

  – The target indexed file already contains records.

  – The input sequential file contains records to be updated or inserted into the indexed file.

  If you do use a COBOL program to load the file, use OPEN OUTPUT and ACCESS SEQUENTIAL. See “File Access Modes” on page 156 for a complete description of each of the access modes.

- File access modes

  For optimum performance, access records sequentially. The least efficient method of accessing records is dynamic access.

- Key design

  For optimum key compression, design the key within the records so that the high-order portion is relatively constant while the low-order portion changes often.

- Multiple alternate indexes

  Because updates have to be applied through the primary paths and reflected through multiple alternate paths, the use of multiple alternate indexes may cause performance degradation.

- Relative File Organization

  Although they are not as space efficient, VSAM fixed-length relative files are more run-time efficient than VSAM variable-length relative files.

- Control Interval Sizes (CISZ)

  VSAM will calculate CISZ to best fit the direct-access storage device (DASD) usage algorithm, which may not, however, be efficient for your application.

  Provide your system programmer with information about the data access and future growth of your VSAM files. From this information, your system programmer can determine the best CISZ.

An average CISZ of 4096 is suitable for most applications. A smaller CISZ means faster retrieval for random processing at the expense of inserts (that is, more CI splits and consequently more space in the file). 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.

Chapter 12. File Sorting and Merging

Arranging records in a particular sequence is a common requirement in data processing. Such record sequencing can be accomplished using sort or merge operations.

- The **sort** operation accepts unsequenced input and produces output in a specified sequence.
- The **merge** operation compares records from two or more sequenced files and combines them in order.

COBOL has special language features that assist in sort and merge operations. For information on the COBOL sort and merge language, see the *COBOL/VSE Language Reference*. For additional information on sorting and merging records, see the *DFSORT/VSE Application Programming Guide*.

With COBOL/VSE, your IBM sort/merge licensed program must be DFSORT/VSE or an equivalent product. Whenever DFSORT/VSE is mentioned, any other equivalent SORT product can be used.

COBOL programs containing SORT or MERGE statements can reside above or below the 16-megabyte line.

To sort or merge files, you need to do the following:

Figure 58. Preparing to Sort or Merge Files

Action	Code
Describe the input and output files for sorting or merging.	FILE-CONTROL and FD entries (if needed)
Describe sort files and merge files.	FILE-CONTROL and SD entries (always needed)
Specify the sort or merge operation.	SORT or MERGE statements in the Procedure Division
Establish an environment in which the sort product is available.	See “Coding Run-Time JCL for SORT” on page 183

Describing the Files

Sort files and merge files must be described with SELECT statements in the Environment Division and SD (Sort File Description) entries in the Data Division. (For an example, see Figure 59 on page 177.) The sort file or merge file described in an SD entry is the working file used during the sort or merge operation. You cannot execute any input/output statements for this file, and you do not create JCL statements in the run-time JCL for the file.

Code FD (File Description) entries, if needed, to describe files used as input to or output from a sort or merge operation. You can also sort or merge records that are defined only in Working-Storage.

If you are only sorting or merging data items from Working-Storage and are not using files as input to or output from a sort or merge operation, you still need SD and FILE-CONTROL entries for the sort file or merge file.

Every SD entry must contain a record description, for example:

```
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).
```

Do not specify RECORDING MODE, BLOCK CONTAINS, or LABEL RECORDS in a sort file description.

The sort files and merge files are processed with SORT or MERGE statements in the Procedure Division. The statement specifies the key field(s) within the record upon which the sort or merge is to be sequenced. You can specify a key or keys as ascending or descending, or when you specify more than one key, as a mixture of the two.

You can mix SORT and MERGE statements in the same program. Within the limits of virtual storage, a COBOL/VSE program can contain any number of sort or merge operations, each with its own independent input or output procedure.

The SORT Statement

You can specify **input** procedures to be performed on the sort records **before** they are sorted (SORT ... INPUT PROCEDURE).

You can specify **output** procedures to be performed on the sort records **after** they are sorted (SORT ... OUTPUT PROCEDURE).

You can use input or output procedures to add, delete, change, edit, or otherwise modify the records.

You can use the SORT statement to:

- Sort data items (including tables) in Working-Storage
- Read records directly into the new file without any preliminary processing (SORT ... USING)
- Transfer sorted records directly to a file without any further processing (SORT ... GIVING)

A COBOL program containing a sort operation is usually organized so that one or more input files are read and operated on by an input procedure. Within the input procedure, a RELEASE statement (analogous to the WRITE statement) places a record into the file to be sorted. That is, when input procedure execution is completed, all the records that are to be sorted have been given to DFSORT/VSE. If you do not want to modify or process the records before the sorting operation begins, the SORT statement USING option releases the unmodified records to the new file.

```
   ID Division.
     Program-ID. SmplSort.
   Environment Division.
     Input-Output Section.
     File-Control.
 * Assign Name For A Sort File Is
 * Treated As Documentation.
 *
       Select Sort-Work-1 Assign To SortFile.
       Select Sort-Work-2 Assign To SortFile.
       Select Input-File  Assign To InFile.
   Data Division.
     File Section.
      SD Sort-Work-1
        Record Contains 100 Characters.
      01 Sort-Work-1-Area.
        05 Sort-Key-1              Pic  X(10).
        05 Sort-Key-2              Pic  X(10).
        05 Filler                  Pic  X(80).

      SD Sort-Work-2
        Record Contains 30 Characters.
      01 Sort-Work-2-Area.
        05  Sort-Key               Pic  X(5).
        05  Filler                 Pic  X(25).

      FD Input-File
        Label Records Are Standard
        Block Contains 0 Characters
        Record Contains 100 Characters
        Recording Mode Is F.
      01 Input-Record              Pic X(100).
 *    .
 *    .
 *    .
     Working-Storage Section.
      01 EOS-Sw                    Pic  X.
      01 Filler.
        05 Table-Entry Occurs 100 Times
          Indexed By X1            Pic X(30).
 *    .
 *    .
 *    .
```

Figure 59. Environment and Data Division Entries for a Sort Program

After all the input records have been passed to DFSORT/VSE, the sorting operation is executed. This operation arranges the entire set of records in the sequence specified by the key(s).

After completion of the sorting operation, sorted records can be made available, one at a time, through a RETURN statement, for modification in an output procedure. If you do not want to modify or process the sorted records, the SORT statement GIVING option names the output file and writes the sorted records to an output file.

The MERGE Statement

You have access to output procedures (used after merging) that can modify the output records.

Unlike the SORT statement, you cannot specify an input procedure in the MERGE statement; you must use MERGE ... USING.

The files to be merged must already be in the same sequence. The merge program then combines them into one sequenced file.

The MERGE statement execution begins the merge processing. This operation compares keys within the records of the input files, and passes the sequenced records one-by-one to the RETURN statement of an output procedure or to the file named in the GIVING phrase.

If you want to process the merged records, they can be made available to your COBOL program, one at a time, through a RETURN statement in an output procedure. If you do not want to modify or process the merged records, the MERGE statement GIVING phrase names the merged output file into which the merged records will be written.

Note: When using DFSORT/VSE, the maximum number of input files that may be merged is 9.

Specifying the Sort Criteria

In the SORT statement, you specify the key on which the file will be sorted. The key must be defined in the record description of the file to be sorted. In the following example, notice that SORT-GRID-LOCATION and SORT-SHIFT are defined in the Data Division before they are used 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).

PROCEDURE DIVISION.
   .
   .
   .
  SORT SORT-FILE
      ON ASCENDING KEY SORT-GRID-LOCATION SORT-SHIFT
      INPUT PROCEDURE 600-SORT3-INPUT
      OUTPUT PROCEDURE 700-SORT3-OUTPUT.
   .
   .
   .
```

To sort on more than one key, as shown in the example above, list the keys in descending order of importance. The example also shows the use of an input and an output procedure. Use an input procedure if you want to process the records before you sort them, and use an output procedure if you want to further process the records after you sort them.

Note: The key used in the SORT statement cannot be variably located. (See "Complex OCCURS DEPENDING ON" on page 106 for more information on variably located data items.)

Restrictions on Sort-Key Length

The maximum number of keys is 64, as long as the total length of the keys does not exceed 3072 bytes.

Alternate Collating Sequences

You can sort records on EBCDIC, ASCII, or another collating sequence. The default collating sequence is EBCDIC or the PROGRAM COLLATING SEQUENCE you specified in the Configuration Section (if any). You can replace the sequence named in the PROGRAM COLLATING SEQUENCE by using the COLLATING SEQUENCE option of the SORT statement. Consequently, you can use different collating sequences for multiple sorts in your program.

When you sort an ASCII file, you have to request the ASCII collating sequence. To do this, use the COLLATING SEQUENCE *alphabet-name* option of the SORT statement, where the *alphabet-name* has been defined in the SPECIAL-NAMES paragraph as STANDARD-1.

Windowed Date Fields

You can specify windowed date fields as sort keys if you are using DFSORT/VSE as your sort program, and your version of DFSORT/VSE supports the Y2PAST option. DFSORT/VSE will use a windowed date sequence to sort the records, rather than a simple binary collating sequence. You can achieve this automatically if you use the DATE FORMAT clause to define a windowed date field, and use this field as a sort key. In this case, the century window used by DFSORT/VSE will be the same as that used by the compilation unit (specified by the YEARWINDOW compiler option).

See Chapter 22, "Using the Millennium Language Extensions" on page 366 for a description of windowed date fields and how you can use them to assist with the Year 2000 problem. See the *DFSORT/VSE Application Programming Guide* for information on DFSORT/VSE and the Y2PAST option.

Coding the Input Procedure

Use SORT ... USING if you do not need to process the records in an input file (or files) before they are released to the sort program. With SORT ... USING *file-name*, the compiler generates an input procedure to open the file, read the records, release the records to the sort program, and close the file.

The input file must not be open when the SORT statement begins execution. If you want to process the records in the input file before they are released to the sort program, use the INPUT PROCEDURE option of the SORT statement.

Each input procedure must be contained in either paragraphs or sections. For example, to release records from Working-Storage (a table) to the new file:

```
       SORT SORT-WORK-2
        ON ASCENDING KEY SORT-KEY
        INPUT PROCEDURE 600-SORT3-INPUT-PROC
        .
        .
        .

  600-SORT3-INPUT-PROC SECTION.
       PERFORM WITH TEST AFTER
          VARYING X1 FROM 1 BY 1 UNTIL X1 = 100
        RELEASE SORT-WORK-2-AREA  FROM  TABLE-ENTRY (X1)
       END-PERFORM.
```

An input procedure contains code for processing records and releasing them to the sort operation. You might want to use an input procedure to:

- Release data items to the new file from Working-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 new file

To transfer records to the new file, all input procedures must contain at least one RELEASE or RELEASE FROM statement. To release A from X, for example, you can enter:

```
MOVE X TO A.
RELEASE A.
```

Figure 60 compares the use of the RELEASE and RELEASE FROM statements.

Figure 60. Comparison of RELEASE and RELEASE FROM

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

Coding the Output Procedure

Use SORT ... GIVING if you want DFSORT/VSE to transfer the sorted records directly from the sort work file into another file without any further processing. With SORT ... GIVING *file-name*, the compiler generates an output procedure to open the file, return the records, write the records, and close the file. At the time the SORT statement is executed, the file named with the GIVING option must not be open.

If, however, you want to select, edit, or otherwise modify the sorted records before writing them from the sort work file into another file, use the OUTPUT PROCEDURE option of the SORT statement.

In the output procedure, you must use the RETURN statement to make each sorted record available to your program (the RETURN statement for a sort file is similar to

a READ statement for an input file). Your output procedure may then contain any statements necessary to process the records that are made available, one at a time, by the RETURN statement.

You can use RETURN INTO, instead of RETURN, to return and process the records into Working-Storage or to an output area. You may also use the AT END and END-RETURN phrases with the RETURN statement. The imperative statements on the AT END phrase will execute after all the records have been returned from the sort file. The END-RETURN explicit scope terminator serves to delimit the scope of the RETURN statement.

When you code output procedures, remember that each output procedure must include at least one RETURN or RETURN INTO statement. Also, each output procedure must be contained in either a section or a paragraph.

Restrictions on Input/Output Procedures

The following restrictions apply to the procedural statements within input and output procedures:

1. The input/output procedure must not contain any SORT, MERGE, STOP RUN, EXIT PROGRAM, or GOBACK statements.
2. The execution of a CALL statement to another program that follows standard linkage conventions is permitted. The called program cannot execute a SORT or MERGE statement. For information on linkage convention considerations with LE/VSE callable services, see the *LE/VSE Programming Guide*.
3. You can use ALTER, GO TO, and PERFORM statements in the input/output procedure to refer to procedure-names outside the input/output procedure. However, you must return to the input/output procedure after a GO TO or PERFORM statement.
4. The remainder of the Procedure Division must not contain any transfers of control to points inside the input/output procedure (with the exception of the return of control from a Declarative Section).
5. During a SORT or MERGE operation, the SD data item is used. You should not use it in the OUTPUT PROCEDURE before the first RETURN statement executes. If data is moved into this record area before the first RETURN statement, the first record to be returned will be overwritten.
6. LE/VSE condition handling does not allow user handlers to be established in an input or output procedure. For details on condition handling considerations and restrictions, see the *LE/VSE Programming Guide*.

Determining Whether the Sort or Merge Was Successful

The DFSORT/VSE program returns a completion code after execution of a sort. The codes are:

0 Successful completion of sort/merge

16 Unsuccessful completion of sort/merge

The *DFSORT/VSE Application Programming Guide* contains a detailed description of conditions under which a sort will be terminated.

The return code or completion code is stored in the SORT-RETURN special register. The contents of SORT-RETURN change with the execution of each SORT or MERGE statement.

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.
       .
       .
       .
```

Premature Termination of a Sort or Merge Operation

The SORT-RETURN special register can also be used to terminate the DFSORT/VSE product operation. Move the integer 16 into the register in an input or output procedure **or** in a Declarative Section entered during sort or merge processing. In an input or output procedure, sort or merge processing will be terminated immediately after the execution of the next RELEASE or RETURN statement. In a declarative section entered during processing of a USING or GIVING file, sort or merge processing will be terminated immediately after the execution of the next implicit RELEASE or RETURN, 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.

If you do not reference SORT-RETURN anywhere in your program, COBOL will test the return code and, if the code is 16, issue a run-time diagnostic message. If you test SORT-RETURN for one or more (but not necessarily all) SORT or MERGE statements, COBOL will not check the return code. (DFSORT/VSE messages are listed in the *DFSORT/VSE Messages, Codes and Diagnosis Guide*.)

By default, DFSORT/VSE diagnostic messages are sent to SYSLST. If you want to change this default, you can use the ROUTE= parameter of the DFSORT/VSE OPTION control statement (see Figure 61 on page 189) or the SORT-MESSAGE special register.

If both the COBOL program and DFSORT/VSE write to SYSLST, uncertain printing results will occur. You can do one of the following:

- Change the DFSORT/VSE message destination to an alternate logical or physical printer via ROUTE= or the SORT-MESSAGE special register (as above).
- Redirect the COBOL program output to an alternate logical or physical printer.

Performing More than One Operation in a Program

You can perform more than one sort or merge in your COBOL program, including:

- Multiple executions of the same sort or merge
- Multiple sorts and/or merges

However, one operation must be completed before another can begin.

Preserving the Original Sequence of Records with Equal Keys

The order of identical collating records can be preserved from input to output in one of these ways:

- Install DFSORT/VSE with the EQUALS option as the default.
- Use the WITH DUPLICATES IN ORDER phrase in the SORT statement; this adds the EQUALS keyword to the SORT-CONTROL statement.

For restrictions when EQUALS is in effect, see the *DFSORT/VSE Application Programming Guide*.

Coding Run-Time JCL for SORT

You need run-time JCL to describe the following files:

Sort Work Files
DLBL, EXTENT and ASSGN statements for SORTWK1, SORTWK2, SORTWK3, ..., SORTWK*n* (where *n* is 9 or less).

You control the number of sort work files that DFSORT/VSE will use during the sort or merge operation by the number of sort work files you specify in your run-time JCL. You must specify the required DLBL information, either in your JCL or in standard labels, for each sort work file in the sequence SORTWK1 to SORTWK*n*, where *n* is the number of sort work files to be used. DFSORT/VSE requires sort work files to be numbered consecutively. For example, if you provide DLBLs for SORTWK1, SORTWK2, and SORTWK4, only the SORTWK1 and SORTWK2 will be used. If, at run time, you use the WORKNM sort option to change the first 4 characters of the sort work file names, you must still specify run-time JCL for the required sort work files in sequence.

SYSLST
An ASSGN statement for sort diagnostic messages, unless the destination is changed via the ROUTE= keyword of the OPTION control statement in the SORT-CONTROL file, or in the SORT-MESSAGE special register.

SORTCKP
Needed if sort will take checkpoints.

Input and Output Files
Define these, if any.

DFSORT/VSE Sublibrary
A LIBDEF statement to define the sublibrary containing the modules.

The size of the program storage in the partition must be large enough to allow sort

to be loaded, and also allow for sort work areas and buffers. The size of program storage can be specified as follows:

```
// EXEC pgmname,SIZE=(pgmname,100K)
```

This will set the size of program storage to the size of the phase containing the COBOL program, plus 100K for SORT.

Improving Sort Performance with FASTSRT

Using the FASTSRT compiler option improves the performance of most sort operations. With FASTSRT, the DFSORT/VSE product performs the I/O on input and/or output files named in **either** or **both** of the following statements:

```
SORT ... USING
SORT ... GIVING
```

You may **not** use the DFSORT/VSE FILNM= option if you specify FASTSRT.

FASTSRT allows DFSORT/VSE (instead of COBOL/VSE) to perform the I/O. Performance of the sort operation may be significantly improved if you block your input and output records.

BLOCK CONTAINS 0 should not be specified for either of the input or output files.

If file status is specified, it will be ignored during the sort.

FASTSRT Requirements for JCL

- In the run-time JCL, the sort work files (SORTWK*n* files) must be assigned to a direct-access device.
- The parameters of the DLBL statement in run-time JCL must match the FD description for the input/output file.

FASTSRT Requirements for Input and Output Files

- SAM files must have a record format of fixed, variable, or spanned.
- VSAM files cannot be password protected.
- BLOCK CONTAINS 0 clause must not be specified for either input or output files.
- Any RELATIVE KEY specified for an output file will not be set by the sort.
- No INPUT declarative (for input files), OUTPUT declarative (for output files), or any file-specific declaratives (for either input or output files) can be specified that apply to FDs used in the sort.
- The same VSAM file cannot be named in both the USING and GIVING phrases.
- A VSAM file will not qualify for FASTSRT (either for the USING or the GIVING clause) if more than one file was specified on the opposing (USING or GIVING) phrase.
- The same SAM file may be used for both FASTSRT input and output, but must be described by two different DLBL or TLBL statements at run time. For example, if FASTSRT is in effect, in the FILE-CONTROL Section you might have:

```
SELECT FILE-IN ASSIGN INPUTF.
SELECT FILE-OUT ASSIGN OUTPUTF.
```

In the Data Division, then, you would have an FD for both FILE-IN and FILE-OUT, where FILE-IN and FILE-OUT are identical, except for their names (they describe the same file).

In the Procedure Division, your SORT statement would look like this:

```
SORT file-name
  ASCENDING KEY data-name-1
  USING FILE-IN GIVING FILE-OUT
```

Then in your run-time JCL, you would use:

```
// DLBL INPUTF,'INOUT',0,SD
// DLBL OUTPUTF,'INOUT',0,SD
```

- If the input and output SAM files are the same, either because the same file name was specified for the USING and GIVING phrases, or because the input and output files are assigned the same file-name, 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. However, if the file was found to be ineligible for FASTSRT on input, it may be eligible for FASTSRT on output.
- A VSAM file that qualifies for FASTSRT (either in the USING or the GIVING phrase) cannot be accessed by the COBOL program until the SORT statement processing has completed. That is, if a VSAM file qualifies for FASTSRT on input (USING phrase), it cannot be accessed (OPEN will fail) in the output procedure and vice versa.
- A SAM file that qualifies for FASTSRT (either in the USING or the GIVING phrase) may be accessed by the COBOL program during the execution of the SORT statement. That is, if it is used for FASTSRT on input, it can be accessed by the COBOL program in the output procedure; if it is used for FASTSRT on output, it can be accessed in the input procedure.
- A variable relative file is ineligible to be either the input or output file for FASTSRT.
- The record descriptions of the SD and FD (for either the input or the output file) must both specify the same format, either fixed or variable, and the largest records of the SD and FD (for either the input file or the output file) must specify the same record length.

In addition, input files and output files must meet specific requirements.

FASTSRT Requirements for Input Files

- Only one file can be mentioned in the USING phrase.
- A VSAM file used as input must not be empty. However, a SAM input file may be empty.
- No input procedure can be used.

FASTSRT Requirements for Output Files

- Only one file can be mentioned in the GIVING phrase.
- The LINAGE clause must not be specified for the output FD entry.

- No output procedure can be used.

Information Messages for FASTSRT

The compiler issues information messages to point out statements in which FASTSRT can improve performance. The compiler also issues messages if you specified FASTSRT, but do not qualify for improved sort performance in your program.

If the requirements for FASTSRT listed above are not met, COBOL will perform all the I/O, and there will be no performance improvement, even though you specified FASTSRT.

Sorting Variable-Length Records

Although you cannot specify RECORDING MODE V in the SD entry (which does not allow the RECORDING MODE clause), the compiler determines that the records in the new file are of variable length if:

- The input file to the new file has variable-length records.
- The SD includes more than one record description and the records are of different lengths.
- The SD includes a RECORD IS VARYING IN SIZE clause.

If the input file to the new file contains variable-length records, specify the record length that occurs most frequently in the input file (the modal length) on the SMS= statement or in the SORT-MODE-SIZE special register. (For the format of the SMS statement, see "Passing Control Statements to DFSORT/VSE.") Use of the SMS statement or SORT-MODE-SIZE special register is optional, but it can improve sort performance.

Passing Control Statements to DFSORT/VSE

Optionally, you can pass information to DFSORT/VSE through control statements read from SYSIPT or contained in a VSE Librarian member. Use this technique only if you want to change the system defaults in order to improve the performance of your sort operations.

The control statements you can include at run time (in the order listed) are:

1. SORT or MERGE (used to replace the SORT or MERGE statement generated by the compiler)
2. SMS=*nnnnn* where *nnnnn* is the length, in bytes, of the most frequent record size (ignored if the SD is not variable)
3. OPTION (except FILNM=)
4. Other DFSORT/VSE control statements (ALTSEQ, ANALYZE, INCLUDE, INREC, OMIT, OUTREC, or SUM)

The purpose of each of these control statements is summarized in Figure 61 on page 189. For keyword values that you can use with OPTION and other DFSORT/VSE control statements, see the *DFSORT/VSE Application Programming Guide*.

Format of the Control Statements

Control statements must be coded in the order listed above, and must be between columns 2 and 71. You can continue a record by ending the line with a comma, and continuing the next line with a new keyword. No labels or comments are allowed on the records, and the record itself cannot be a DFSORT/VSE comment statement. No other syntax checking of the statement is performed.

Specifying Control Statements Source

The use of run-time DFSORT/VSE control statements is optional. You can specify that run-time control statements are to be read by using the SORT-CONTROL special register. You can assign to SORT-CONTROL the value “SYSIPT” (if control statements are to be read from SYSIPT) or the name of a VSE Librarian member. If you provide the name of a VSE Librarian member, the member must be of type “C,” it must be cataloged in a source sublibrary available at run time, and its name must not be IGZSRTCD.

If you provide the name of a VSE Librarian member and it is not found you will receive the message IGZ0027W.

Specifying SORT or MERGE Control Statements

If the compiler generates a SORT statement and you try to override this with a MERGE statement from the SORT-CONTROL member, the overriding MERGE statement is ignored and no message is issued.

If the compiler generates a MERGE statement and you try to override this with a SORT statement from the SORT-CONTROL member, the overriding SORT statement is ignored and no message is issued.

The LE/VSE run-time library appends parameters to the SORT statement depending on the COBOL/VSE SORT options requested. These appends are:

FILES= (determined by the number of sort work DLBL statements found)

EQUALS (if DUPLICATES is coded in the COBOL/VSE SORT statement)

SIZE= (when SORT-FILE-SIZE is set (compatibility only))

These appends will be made to the SORT statement supplied via the SORT-CONTROL member. However, if any of these parameters are already coded on the supplied statement, the supplied option takes precedence.

If you have multiple sorts in a single COBOL program the generated SORT statements can be overriden by using multiple SORT-CONTROL members and changing the value of SORT-CONTROL before each sort. Alternatively, you can set SORT-CONTROL to SYSIPT and use a JCL setup as follows:

```
// EXEC SORTPGM
  SORT FIELDS= ... (sort one override statements)
/*
  SORT FIELDS= ... (sort two override statements)
/*
```

Using Control Statements

The FASTSRT option will not take effect for input if an input procedure is used or for output if an output procedure is used in the SORT statement.

Many functions usually performed in an input or output procedure are the same as those done by the DFSORT/VSE functions:

INREC
OUTREC
INCLUDE
OMIT
SUM

You may be able to eliminate your input and output procedures. To do so, code the appropriate DFSORT/VSE program control statements and place them in COBOL/VSE's SORT-CONTROL file, thereby allowing your SORT statement to qualify for FASTSRT.

For more information, see the *DFSORT/VSE Application Programming Guide*.

SORT Special Registers

The COBOL programmer has control over a number of aspects of sort behavior. For some of these aspects, a special register is available to insert a value before the sort to control the process, or to test the contents after the sort to verify its success. In other cases, COBOL compiler options can affect the sort process. In all cases, sort control statement keywords can be used as an alternative method.

SORT-RETURN
SORT-RETURN is a COBOL special register containing the SORT return code. Test this return code to verify that the sort was successful. For an example of SORT-RETURN verification, see “Determining Whether the Sort or Merge Was Successful” on page 181. You can also use it to terminate sort/merge before its processing is complete by moving the integer 16 to it in an input or output procedure, or in an ERROR declarative entered during sort or merge processing. A RETURN or RELEASE statement must then be executed.

SORT-CONTROL
SORT-CONTROL is an 8-character COBOL special register. If you want to provide run-time DFSORT/VSE control statements, you can assign to SORT-CONTROL the value “SYSIPT” or the name of a VSE Librarian member. Once you have assigned a value to SORT-CONTROL you can cancel it by assigning spaces to SORT-CONTROL.

Figure 61 on page 189 lists those aspects of sort behavior that can be affected by special registers or COBOL compiler options, and the equivalent sort control statement keywords. For a full list of sort keywords, see the *DFSORT/VSE Application Programming Guide*.

Figure 61. Sorting in COBOL/VSE

To Set or Test	Use
Sort completion code	SORT-RETURN special register
Name of file with sort control statements (by default IGZSRTCD)	SORT-CONTROL special register
Modal length of records in a file with variable-length records	SORT-MODE-SIZE special register, or RECORD control statement keyword: LENGTH
Number of sort records	SORT-FILE-SIZE special register, or SORT control statement keyword: SIZE (ignored by DFSORT/VSE)
Amount of main storage to be used	SORT-CORE-SIZE special register, or OPTION control statement keyword: STORAGE
Name of sort message file (default SYSLST)	SORT-MESSAGE special register, or OPTION control statement keyword: ROUTE
Century window for sorting or merging on date fields	YEARWINDOW compiler option, or OPTION control statement keyword: Y2PAST
Format of windowed date fields used as sort or merge keys	Derived from PICTURE, USAGE, and DATE FORMAT clauses, or SORT control statement keyword: FORMAT=Y2*x*

The SORT-CORE-SIZE, SORT-FILE-SIZE, SORT-MESSAGE, and SORT-MODE-SIZE special registers will be used in the SORT interface when they have nondefault values. However, at run time, individual SORT special registers will be replaced by the corresponding parameters on control statements that are included in the SORT-CONTROL file, and a message will be issued. In addition, a compiler warning message (W-level) will be issued for each SORT special register that was set in the program.

Storage Use During a Sort or Merge Operation

In general, the more storage DFSORT/VSE has available, the faster the sorting operation is performed. Certain parameters specified during the installation of DFSORT/VSE determine the amount of storage used during its operation.

Enough program storage must be reserved for:

- The COBOL program to be executed
- DFSORT/VSE modules
- DFSORT/VSE input and output buffers
- DFSORT/VSE working storage

Note: Program storage may be reserved by using the JCL EXEC statement SIZE parameter. Your run-time JCL should look like:

```
// EXEC pgrmid,SIZE=(pgrmid,nnnK)
```

where *nnnK* is the amount of program storage required for DFSORT/VSE.

GETVIS storage must be reserved for:

- COBOL programs that are dynamically loaded from an input or output procedure
- COBOL run-time library routines
- Any storage obtained by these routines
- LE/VSE run-time library routines
- Storage for the DFSORT/VSE GVSIZE option

For a specific execution of a sort or merge, you may replace the values specified at installation. The STORAGE keyword on the DFSORT/VSE OPTION control statement, or the SORT-CORE-SIZE special register, can be used for this purpose. (For the meaning of this key word see the *DFSORT/VSE Application Programming Guide*).

Note: Be careful not to replace the storage allocation to the extent that more than the reserved program storage is used for the sort operation.

Checkpoint/Restart During DFSORT/VSE

It is possible to take a checkpoint during a sort operation. DFSORT/VSE will take only one checkpoint during a sort operation. No checkpoint will be taken during a merge operation.

A checkpoint taken during a DFSORT/VSE operation, unless taken by DFSORT/VSE, cannot be used to restart. Restarts using checkpoints are invalid if the checkpoint was taken by your COBOL/VSE program while SORT or MERGE statements were executing. The restarts are detected and canceled.

If a checkpoint is to be taken during a sorting operation, a checkpoint file must be provided in the run-time JCL. The checkpoint file must:

- be assigned to SYS000 (a tape or a direct-access device, CKD or FBA)
- have standard labels
- have the filename SORTCKP

To cause DFSORT/VSE to take a checkpoint while a SORT statement is executing, code in the following I-O control paragraph:

```
RERUN ON assignment-name
```

SORTING under CICS

Under CICS, you can use the SORT statement (along with a sort program that runs under CICS) to sort small amounts of data. The SORT statement must have both an INPUT PROCEDURE and an OUTPUT PROCEDURE. 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.

Note: There is no IBM sort product that is supported under CICS.

CICS SORT Application Restrictions

The following restrictions apply to COBOL/VSE applications that are written using the SORT statement and will run under CICS.

- SORT statements that include the USING or GIVING phrase are not supported.
- Sort control files are not supported. Data in the SORT-CONTROL special register is ignored.
- Using the following CICS commands in the input/output procedures may cause unpredictable results:

  CICS LINK
  CICS XCTL
  CICS RETURN
  CICS HANDLE
  CICS IGNORE
  CICS PUSH
  CICS POP

- CICS commands, other than those in the preceding list, are allowed provided they are used with the NOHANDLE or RESP option. Unpredictable results may occur if the NOHANDLE or RESP option is not used.
- Any CICS HANDLE or CICS HANDLE ABEND commands specified in the COBOL program prior to executing the SORT statement will not be in effect during the SORT.
- The COBOL sort input/output procedure can contain CALLs to nested programs. Calls to separately compiled programs are not allowed and unpredictable results may occur.

Chapter 13. Error Handling

As you plan and code, you naturally attempt to create a perfect program—one that will be error-free and run without problems. But it is unrealistic to believe that problems will never occur during the execution of your program. Even if your own code is flawless, errors may occur in the system facilities that your program uses.

Anticipate these possibilities by putting code into your program to handle them. Such code can be thought of as built-in distress flares or lifeboats. If such error-handling code is not present in your program, not only could output data and files obviously be ruined, but you might not even be aware of the problem.

The action taken by your error-handling code can vary from attempting to cope with the situation and continue, to issuing a message, to halting execution. In any event, coding a warning message is a good idea.

You might be able to create your own error-detection routines for data-entry errors or for errors as your installation defines them.

COBOL/VSE contains special elements to help you anticipate and correct error conditions. These fall into the following main areas:

- User-initiated dumps
- String and unstring operations
- Arithmetic operations
- Input/output error-handling techniques
- CALL statements
- User-written error-handling routines

User-Initiated Dumps (CALLs to LE/VSE)

Creating a Formatted Dump: You can cause a dump of the LE/VSE run-time environment and the member language libraries at any prespecified point in your program by coding a call to the LE/VSE subroutine CEE5DMP. For example:

```
77  Title-1          Pic x(80)   Display.
77  Options          Pic x(255)  Display.
01  Feedback-code    Pic x(12)   Display.
⋮
Call "CEE5DMP" Using Title-1, Options, Feedback-code
```

In order to have symbolic variables included in the formatted dump produced by LE/VSE, you must compile with the SYM suboption of the TEST compiler option and use the VARIABLES subparameter of CEE5DMP. For further details on using CEE5DMP, see the *LE/VSE Programming Reference*.

You can also specify, through run-time options, that a dump be produced for error conditions of your choosing. For information on these run-time options and their syntax, see the *LE/VSE Programming Reference*. For more information about using dumps, refer to *LE/VSE Debugging Guide and Run-Time Messages*.

Creating a System Dump: You can cause a system dump at any prespecified point in your program by coding a call to the LE/VSE subroutine CEE5ABD.

This callable service terminates the run unit immediately, and a system dump is requested when the ABEND is issued.

For details on CEE5ABD, see the *LE/VSE Programming Reference*.

STRING and UNSTRING Operations

When stringing or unstringing data, the pointer may fall out of the range of the receiving field. In this case, a potential overflow condition exists, but COBOL does not allow the overflow to actually occur; the STRING/UNSTRING operation will not be completed and the receiving field remains unchanged.

If you do not have an ON OVERFLOW clause on the STRING or UNSTRING statement, control passes to the next sequential statement, and you are not notified of the incomplete operation.

Consider the following statement:

```
String Item-1 space Item-2 delimited by Item-3
    into Item-4
    with pointer String-ptr
    on overflow
      Display "A string overflow occurred"
End-String
```

Figure 62. Data Values before and after Statement Executes

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	bbbbbbbb
String-ptr	9(2)	0	0

Note: The symbol b represents a blank space.

Since String-ptr has a value of zero which falls short of the receiving field, an overflow condition occurs and the STRING operation is not completed (a String-ptr greater than 9 would cause the same result). Had ON OVERFLOW not been specified, you would not have been notified that the contents of Item-4 remain unchanged.

Arithmetic Operations

When your program performs arithmetic operations, the results may be larger than the fixed-point field that is to hold them, or you may have attempted a division by 0. In either case, the ON SIZE ERROR clause after the ADD, SUBTRACT, MULTIPLY, DIVIDE, or COMPUTE statement can handle the situation.

For ON SIZE ERROR to work correctly for fixed-point overflow and decimal overflow, you must specify the TRAP(ON) run-time option.

If you code the ON SIZE ERROR clause, the imperative statement of your clause will be executed and your result field will not be changed in the following five cases:

- Fixed-point overflow
- Division by 0
- Zero raised to the zero power
- Zero raised to a negative number
- A negative number raised to a fractional power

Note: You should be aware that floating-point exponent overflow, which occurs when the value of a floating-point arithmetic calculation cannot be represented in the System/370 floating-point operand format, does not cause SIZE ERROR; an abend occurs instead.

Example of Checking for Division by Zero

Code your ON SIZE ERROR imperative statement so that it issues an informative message. For example:

```
DIVIDE-TOTAL-COST.
    DIVIDE TOTAL-COST BY NUMBER-PURCHASED
        GIVING ANSWER
        ON SIZE ERROR
          DISPLAY "ERROR IN DIVIDE-TOTAL-COST PARAGRAPH"
          DISPLAY "SPENT " TOTAL-COST, " FOR " NUMBER-PURCHASED
          PERFORM FINISH
    END-DIVIDE
   .
   .
   .

FINISH.
    STOP RUN.
```

In this example, if division by 0 occurs, the program will do two things: write out a message identifying the trouble and halt program execution.

Input/Output Error Handling Techniques

COBOL/VSE offers five techniques for intercepting and handling certain input/output errors. With the exception of VSAM Return Code they can be used for both SAM and VSAM file processing.

- The end-of-file phrase (AT END)
- The EXCEPTION/ERROR declarative
- The file status key
- The VSAM Return Code
- The INVALID KEY phrase

The most important thing to remember about input/output errors is that you choose whether or not your program will continue executing after a less-than-severe input/output error occurs. COBOL/VSE does not perform corrective action. If you choose to have your program continue (by incorporating error-handling code into your design), you must also code the appropriate error-recovery procedure.

The following figures show the flow of logic after the indicated errors:

Figure 63. List of Logic Flow Figures

Error	Figure
A VSAM input/output error	Figure 64
An out-of-space (INVALID KEY) condition in SAM	Figure 65 on page 196
A SAM input/output error detected by COBOL	Figure 66 on page 196
A SAM input/output error detected by SAM	Figure 67 on page 197

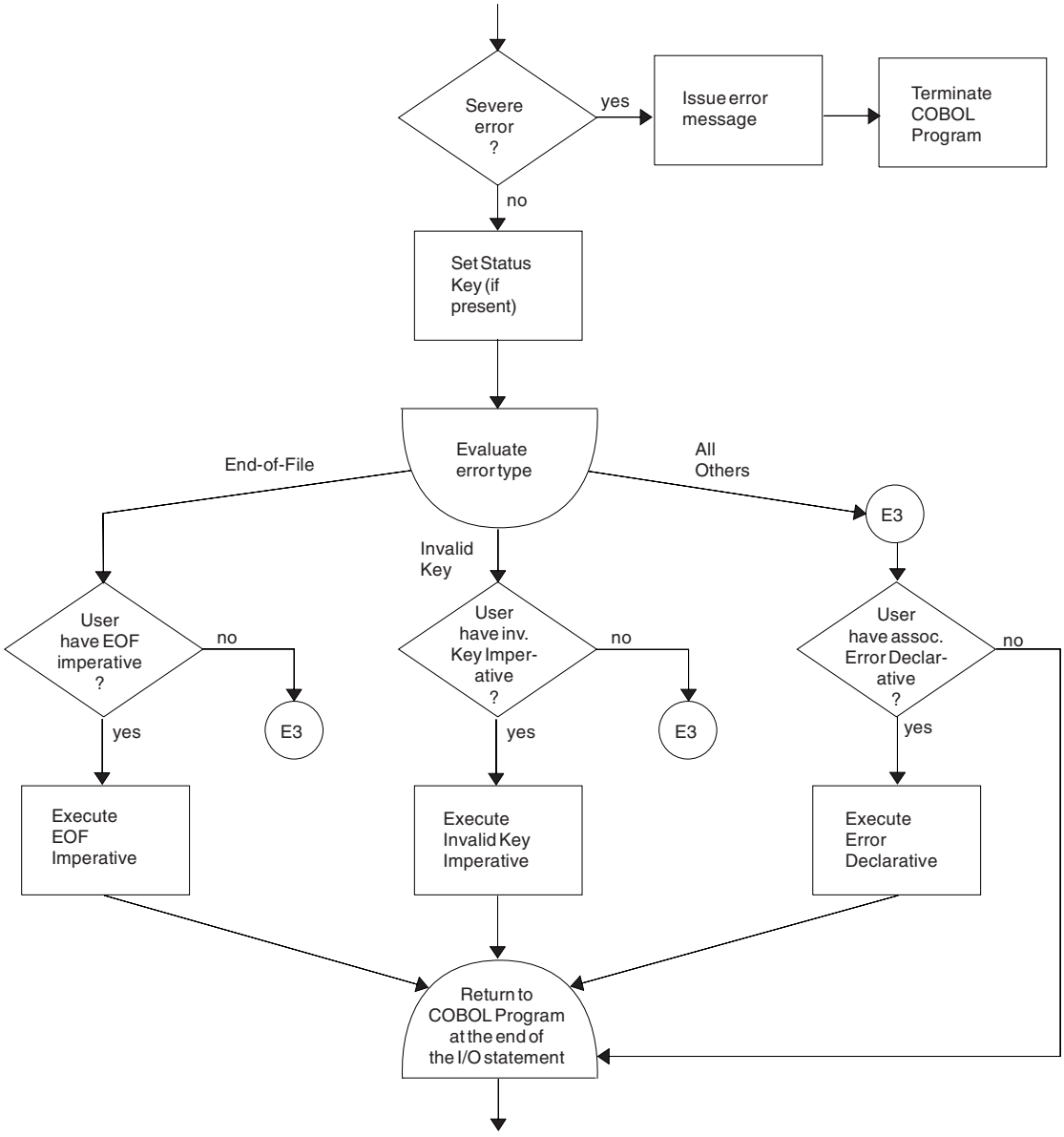
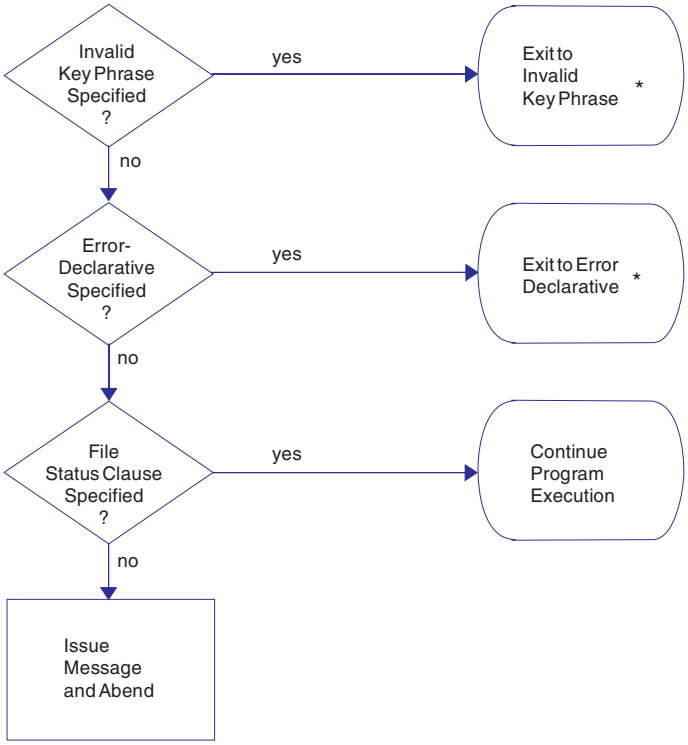


Figure 64. Flow of Logic after a VSAM I/O Error

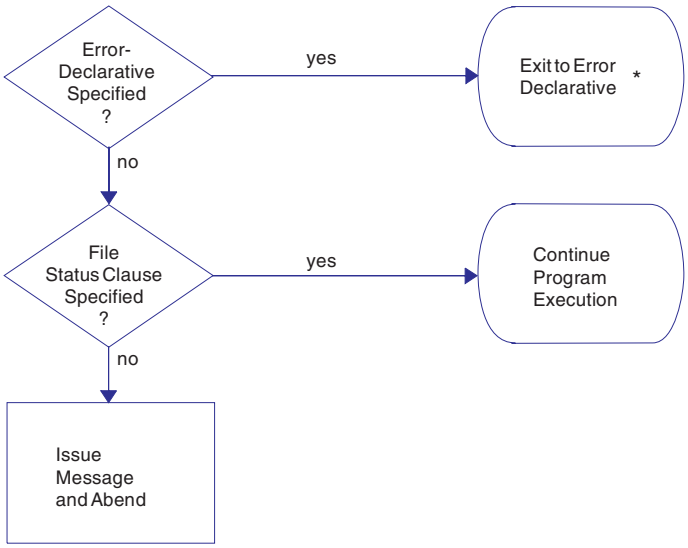
Type of Error: Errors not found by SAM for WRITE or CLOSE REEL/UNIT (INVALID KEY condition).



* Execution of COBOL program then continues after the I/O statement that caused the error.

Figure 65. Flow of Logic after an Out-of-Space (INVALID KEY) Condition in SAM

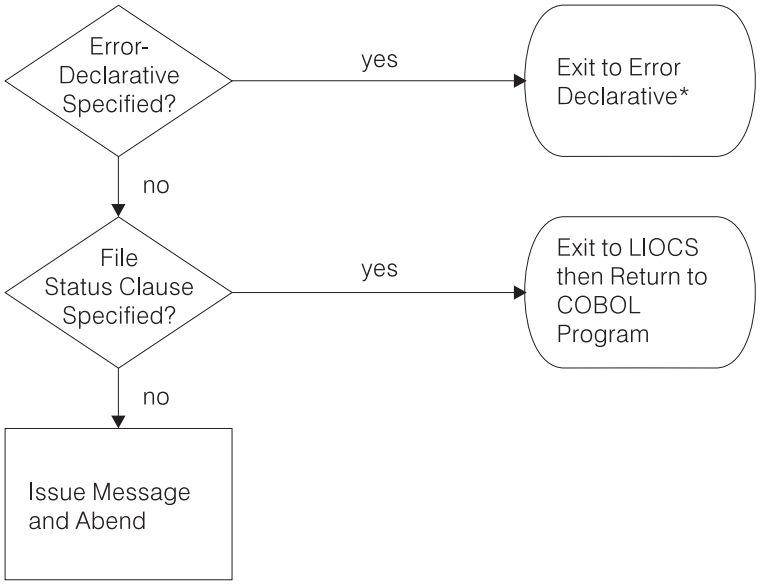
Type of Error: Errors detected by COBOL.



* Execution of COBOL program continues after the I/O statement that caused the error.

Figure 66. Flow of Logic after a SAM I/O Error Detected by COBOL

Type of Error: Errors detected by SAM.



* On exit from declarative, return to SAM.

Figure 67. Flow of Logic after a SAM I/O Error Detected by SAM

End-of-File Phrase (AT END)

An end-of-file condition may or may not represent an error. In many designs, reading sequentially to the end of a file is done intentionally, and the AT END condition is expected.

For example, suppose you are processing a file containing transactions in order to update a master file:

```
PERFORM UNTIL TRANSACTION-EOF = "TRUE"
  READ UPDATE-TRANSACTION-FILE INTO WS-TRANSACTION-RECORD
    AT END
      DISPLAY "END OF TRANSACTION UPDATE FILE REACHED"
      MOVE "TRUE" TO TRANSACTION-EOF
  END READ
        ⋮
END-PERFORM
```

In some cases, however, the condition will reflect an error. You code the AT END phrase of the READ statement to handle either case, according to your program design.

If you code an AT END phrase, then at end-of-file the phrase is executed. If you do not code an AT END phrase, the associated ERROR declarative is executed.

Any NOT AT END phrase that you code is executed only if the READ statement completes successfully. That is, if the READ operation fails because of any condition other than end-of-file, neither the AT END nor the NOT AT END phrase is executed. Instead, control passes to the end of the READ statement after executing any associated declarative procedure.

If you have coded neither an AT END phrase nor an EXCEPTION declarative procedure but have coded a status key clause for the file, control passes to the next sequential instruction after the input/output statement that detected the end-of-file (where presumably you have some code to take appropriate action).

EXCEPTION/ERROR Declarative

You can code one or more ERROR declarative procedures in your COBOL/VSE program that will be given control if an input/output error occurs. You can have:

- A single, common procedure for the entire program
- Group procedures for each file open mode (whether INPUT, OUTPUT, I-O, or EXTEND)
- Individual procedures for each particular file

Place each such procedure in the declaratives section of your Procedure Division. (For syntax details, see *COBOL/VSE Language Reference*.)

In your procedure, you can choose to attempt corrective action, retry the operation, continue, or end execution. You can use the ERROR declaratives procedure in combination with the status key if you want a further analysis of the error.

If you continue processing a blocked file, you may lose the remaining records in a block after the record that caused the error.

Write an ERROR declarative procedure if you want the system to return control to your program after an error occurs. If you do not write such a procedure, your job may be canceled or abnormally terminated after an error occurs.

When writing an ERROR declarative procedure for a SAM file, you should observe the following restrictions.

1. COBOL/VSE cannot handle nested input/output errors on SAM files. If input/output errors occur within an ERROR declarative procedure, the results are unpredictable.
2. Not all input/output errors cause control to be transferred to your ERROR declarative procedure, only input/output errors which the system determines to be “standard” will transfer control.

For a discussion of input/output error handling, see *VSE/ESA System Macros User's Guide*.

File Status Key

The system updates the file status key after each input/output statement executed for a file, placing values in the two digits of the file status key. In general, a zero in the first digit indicates a successful operation, and a zero in both digits means "nothing abnormal to report". Establish a file status key using the file status clause in the FILE-CONTROL and data definitions in the Data Division.

```
FILE STATUS IS data-name-1
```

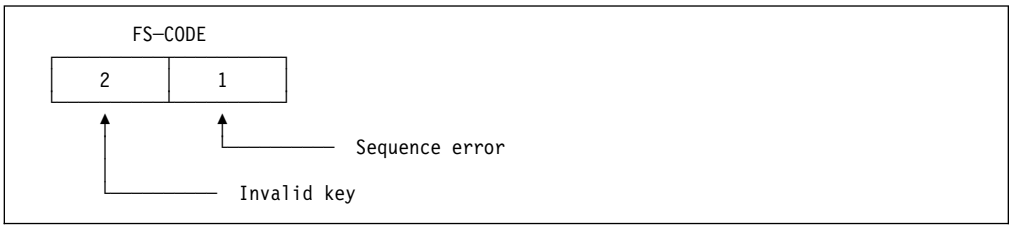
data-name-1
Specifies the 2-character COBOL file status key that should be defined in the Working-Storage Section.

Note: The *data-name* in the FILE STATUS clause cannot be variably located. (See “Complex OCCURS DEPENDING ON” on page 106 for more information on variably located data items.)

Your program can check the COBOL file status key to discover whether an error has been made and, if so, what general type of error it is. For example, if a FILE STATUS clause is specified like this:

```
FILE STATUS IS FS-CODE
```

then *FS-CODE* is used by COBOL/VSE to hold status information like this:



Follow these rules for each SAM or VSAM file:

1. Define a different file status key for each file.

   This is especially important for VSAM files, because, under normal circumstances, errors on a VSAM file return control to your COBOL program—whether they are errors due to the logic of your program or whether they are input/output errors occurring on the storage media.

2. Check the file status key after **every** input/output request.

   After an input or output statement is executed, check the contents of the status key; if it contains a value other than 0, your program can issue an error message, or can take an action based on the value of the code placed in the status key.

   You do not have to reset the status key code, because it is set to 0 before each input/output attempt.

For VSAM files, in addition to the file status key, you can specify a second identifier in the FILE STATUS clause to get more detailed VSAM information on input/output requests. For further details, see “VSAM Return Code (VSAM Files Only)” on page 201.

Note: **For VSAM:** If you do not specify a file status key and you do not code an ERROR declarative, serious processing errors can go undetected by your program, which will go right on processing the wrong data. If you continue processing after such errors occur, you may impair the integrity of your data.

For SAM: If you do not specify a file status key and you do not code an ERROR declarative, then a serious I-O processing error will cause your program to abend.

You can use the status key alone, or in conjunction with the INVALID KEY option, or to supplement the EXCEPTION/ERROR declarative. Using the status key in this way gives you precise information about the results of each input/output operation.

Figure 68 shows that the meanings for the possible file status key values fall into six general categories. The *COBOL/VSE Language Reference* explains each individual status key in detail.

Figure 68. File Status Key Condition Types and Descriptions

Type of Condition	File Status Code	General Description
Successful Completion	00 02 04 05 07	A '0' in the first digit indicates a successful operation condition, and a '0' in both digits means "nothing abnormal to report."
AT END	10 14	A '1' in the first digit indicates an end-of-file condition during a sequential READ operation; there are no more records to READ.
Invalid Key	21 22 23 24	A '2' in the first digit indicates an invalid key condition. There is a problem with a VSAM indexed prime record key value, a VSAM indexed alternate record key value, or a VSAM relative record key value on READ, START, REWRITE, and DELETE requests such as a duplicate key, a record not found, or a boundary violation.
Permanent Error	30 34 35 37 38 39	A '3' in the first digit indicates a permanent error condition with unsuccessful completion of the I/O operation. You may have a boundary violation or have attempted an invalid OPEN statement such as trying to OPEN a file that would not support the open mode specified in the OPEN statement.
Logic Error	41 42 43 44 46 48 49	A '4' in the first digit indicates a logic error condition with unsuccessful completion of the I/O operation. Some possible causes are attempting an OPEN statement for an already open file; a CLOSE statement for an already closed file; a READ, WRITE, DELETE or REWRITE statement on a file not open in the correct mode; or trying to REWRITE a record to a file and the record is a wrong size.
Implementer-Defined	90 91 92 93 94 95 96 97	A '9' in the first digit indicates an implementer-defined condition. COBOL/VSE uses files status codes 90 and 96 for SAM files. For VSAM files, COBOL/VSE uses files status codes 90-97. Some possible causes of VSAM I/O errors include password failures, missing DLBL statement for the file, or invalid or incomplete file information.

Figure 69 on page 201 shows an example of the COBOL coding for performing a simple check on opening a file.

```
      .
      .
      .

Environment Division.
      .
      .
      .
File-Control.
    Select Master-File
      Assign To AS-MASTRA
      File Status Is Master-File-Check
      .
      .
      .

Data Division.
      .
      .
      .

Working-Storage Section.
  01 Master-File-Check       Pic X(2).
      .
      .
      .

Procedure Division.
      .
      .
      .

    Open Input Master-File
    If Master-File-Check Not = Zeros
      Display "Cannot Open File"
      .
      .
      .
```

Figure 69. Using the Status Key to Check an OPEN Statement

VSAM Return Code (VSAM Files Only)

Often the 2-character file status code is too general to pinpoint the disposition of a request. You can get more detailed information about VSAM input/output requests by specifying a second status area:

```
FILE STATUS IS data-name-1 data-name-2
```

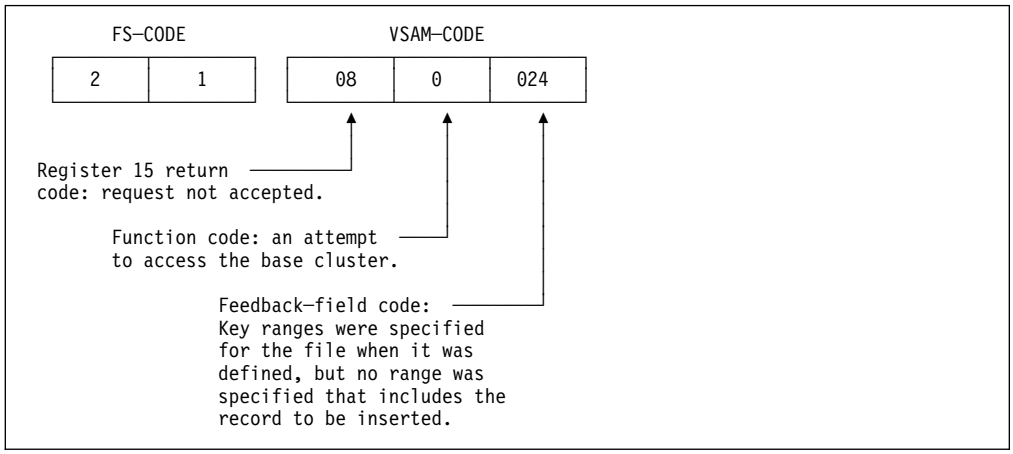
data-name-1
Specifies the 2-character COBOL file status key.

data-name-2
Specifies a 6-byte data item that contains the VSAM return code when the COBOL file status key is not '00'.

The complete status area might be defined in Working-Storage Sections as:

```
01 RETURN-STATUS.
   05 FS-CODE                 PIC X(2).
   05 VSAM-CODE.
      10 VSAM-R15-RETURN      PIC 9(2)  COMP.
      10 VSAM-FUNCTION        PIC 9(1)  COMP.
      10 VSAM-FEEDBACK        PIC 9(3)  COMP.
```

The second area is used by COBOL/VSE to pass along information supplied by VSAM, for example:



For information on interpreting the VSAM codes, see your *VSAM Administration: Macro Instruction Reference* and *VSE/ESA Messages and Codes*.

INVALID KEY Phrase

This phrase will be given control in the event that an input/output error occurs because of a faulty index key. You can include INVALID KEY phrases on READ, START, WRITE, REWRITE, and DELETE requests for VSAM indexed and relative files.

You can also include INVALID KEY on WRITE requests for SAM files. In the case of SAM files, however, the INVALID KEY phrase has limited meaning. It is used only when you attempt to write to a disk that is full.

INVALID KEY phrases differ from ERROR declaratives in these ways:

- INVALID KEY phrases operate for only limited types of errors, whereas the ERROR declarative encompasses all types.
- INVALID KEY phrases are coded directly onto the input/output verb, whereas ERROR declaratives are coded separately.
- INVALID KEY phrases are specific for one single input/output operation, whereas ERROR declaratives are more general.

If you specify INVALID KEY in a statement that causes an INVALID KEY condition, control is transferred to the INVALID KEY imperative statement. In this case, any ERROR declaratives you have coded are not executed.

Any NOT INVALID KEY phrase that you specify is executed only if the statement completes successfully. If the operation fails because of any condition other than INVALID KEY, neither the INVALID KEY nor the NOT INVALID KEY phrase is executed. Instead control passes to the end of the statement after executing any associated ERROR declaratives.

Use the FILE STATUS clause in conjunction with INVALID KEY to evaluate the status key and determine the specific INVALID KEY condition.

For example, assume you have a file containing master customer records and need to update some of these records with information in a transaction update file. You will read each transaction record, find the corresponding record in the master file, and make the necessary updates. The records in both files each contain a field for a customer number, and each record in the master file has a unique customer number.

The File-Control entry for the master file of commuter records includes statements defining indexed organization, random access, MASTER-COMMUTER-NUMBER as the prime record key, and COMMUTER-FILE-STATUS as the file status key. The following example illustrates how you can use FILE STATUS in conjunction with the INVALID KEY to determine more specifically the cause of an I/O statement failure.

```
.
.  (read the update transaction record)
.
MOVE "TRUE" TO TRANSACTION-MATCH
MOVE UPDATE-COMMUTER-NUMBER TO MASTER-COMMUTER-NUMBER
READ MASTER-COMMUTER-FILE INTO WS-CUSTOMER-RECORD
  INVALID KEY
    DISPLAY "MASTER CUSTOMER RECORD NOT FOUND"
    DISPLAY "FILE STATUS CODE IS: " COMMUTER-FILE-STATUS
    MOVE "FALSE" TO TRANSACTION-MATCH
END-READ
```

CALL Statements

When dynamically calling a separately compiled program, the program that you call may be unavailable to the system. For example, the system may be out of storage or it may be unable to locate the phase. If you do not have an ON EXCEPTION or ON OVERFLOW clause on the CALL statement, your application may end abnormally (abend). You can use the ON EXCEPTION clause to execute a series of statements and to perform your own error handling. For example:

```
CALL "REPORTA"
   ON EXCEPTION
     DISPLAY "Program REPORTA not available. Loading REPORTB."
     CALL "REPORTB"
     END-CALL
END-CALL
```

If program REPORTA is unavailable, control will continue with the ON EXCEPTION clause.

The behavior of the ON EXCEPTION/OVERFLOW clause is sensitive to the CMPR2 compiler option. See *COBOL/VSE Migration Guide* for details on VS COBOL II Release 2 compatibility and migration.

Note: The ON EXCEPTION clause applies only to the availability of the called program. If an error occurs while the called program is running, the ON EXCEPTION clause will not be executed.

User-Written Error-Handling Routines

You can handle most error conditions that might occur during program execution by using the ON EXCEPTION phrase, the ON SIZE ERROR phrase, and other language semantics. But in the event of an extraordinary condition like a machine check, normally your program will not regain control—it will be abnormally terminated. However, COBOL/VSE in conjunction with LE/VSE provides a mechanism whereby your program can gain control when such conditions occur. LE/VSE condition handling gives you the opportunity to write your own error-handling routines to handle conditions which can allow your program to resume executing.

In order to have LE/VSE pass control to your own user-written error routine, you must first identify and register its entry point to LE/VSE. Procedure-pointer data items allow you to pass the entry address of procedure entry points to LE/VSE services. For more information on procedure-pointer data items, see “Passing Entry Point Addresses with Procedure Pointers” on page 280.

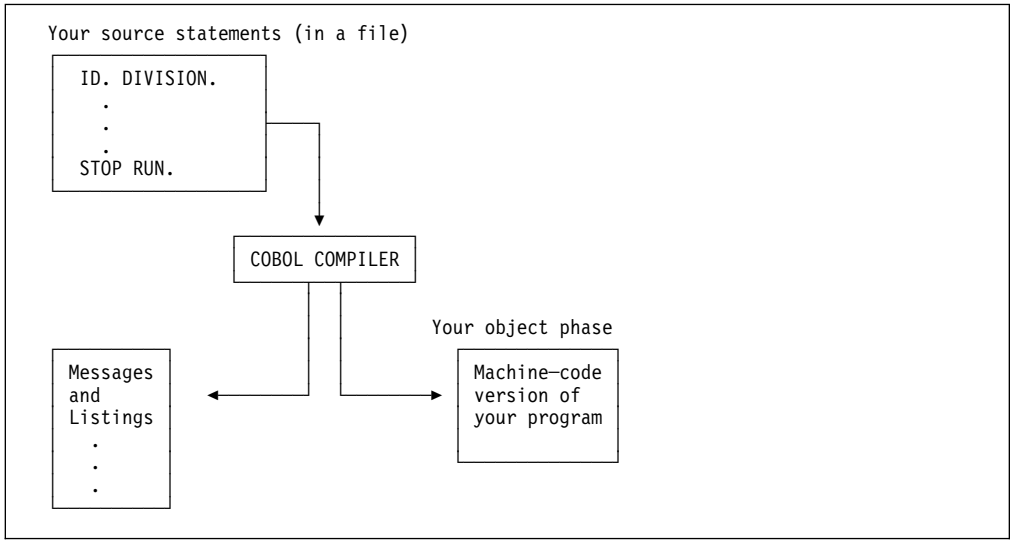
For more information on LE/VSE condition management and a COBOL/VSE example of handling conditions with a user-written condition handler, see *LE/VSE Programming Guide*.

Part 3. Compiling Your Program

This part of the book provides instructions for compiling your COBOL/VSE programs. More complex programming topics are discussed in Part 4, “Advanced Topics” on page 259.

Chapter 14. Methods of Compilation

Methods of compilation vary according to the system you are using. However, the compilation procedure is basically the same in all systems:



The system under which you can use the COBOL/VSE compiler is VSE/ESA batch.

You can compile a sequence of separate COBOL programs with a single invocation of the compiler by using a batch compile technique. See “Batch Compiling” on page 209 for more information.

Coding Compilation JCL

The JCL for compilation includes:

- A job identifier
- Definitions of the options to be used
- Definitions for the files needed
- A statement to execute the compiler

The easiest way to compile your program is to code JCL that uses a cataloged procedure, as shown in Figure 70 on page 207.

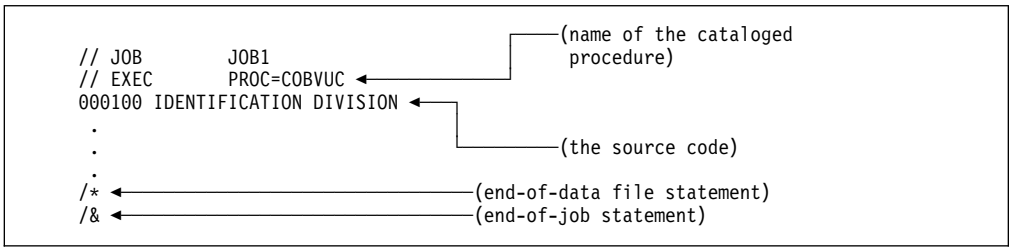


Figure 70. JCL for Compilation, Using a Cataloged Procedure

COBVUC is the name of a cataloged procedure that contains statements for compiling a program, including statements for defining the required files. (For information about how to set up cataloged procedures see your system programmer).

Cataloged procedures, however, may not give you the programming flexibility you need for more complex programs. You may need to specify your own job control statements. Figure 71 shows the general format of JCL used to compile a program.

```
// JOB    jobname
// LIBDEF PHASE,SEARCH=(lib.library)
// LIBDEF SOURCE,SEARCH=(lib.sublib,...)
// DLBL   IJSYS01,'COBOLVSE.WORKFILE.IJSYS01',0,SD
// EXTENT SYS001,volser,...
// ASSGN  SYS001,DISK,VOL=volser,SHR
// DLBL   IJSYS02,'COBVSE.WORKFILE.IJSYS02',0,SD
// EXTENT SYS002,volser,...
// ASSGN  SYS002,DISK,VOL=volser,SHR
// DLBL   IJSYS03,'COBVSE.WORKFILE.IJSYS03',0,SD
// EXTENT SYS003,volser,...
// ASSGN  SYS003,DISK,VOL=volser,SHR
// DLBL   IJSYS04,'COBVSE.WORKFILE.IJSYS04',0,SD
// EXTENT SYS004,volser,...
// ASSGN  SYS004,DISK,VOL=volser,SHR
// DLBL   IJSYS05,'COBVSE.WORKFILE.IJSYS05',0,SD
// EXTENT SYS005,volser,...
// ASSGN  SYS005,DISK,VOL=volser,SHR
// DLBL   IJSYS06,'COBVSE.WORKFILE.IJSYS06',0,SD
// EXTENT SYS006,volser,...
// ASSGN  SYS006,DISK,VOL=volser,SHR
// DLBL   IJSYS07,'COBVSE.WORKFILE.IJSYS07',0,SD
// EXTENT SYS007,volser,...
// ASSGN  SYS007,DISK,VOL=volser,SHR
// DLBL   IJSYSLN,'COBVSE.WORKFILE.IJSYSLN',0,SD
// EXTENT SYSLNK,volser,...
// ASSGN  SYSLNK,DISK,VOL=volser,SHR
// OPTION options
// EXEC   IGYCRCTL,SIZE=IGYCRCTL,PARM='options'
      .
      .
      .
/*
/&
```

Figure 71. JCL for Compiling a COBOL/VSE Program—General Format

Figure 72 shows a working example of JCL for compiling a program, using the VSE/VSAM Space Management for SAM Feature to define the required work files. The JCL in this example assumes that a default model for SAM ESDS files has been defined in the VSAM master catalog.

```
// JOB    JOB1
// LIBDEF PHASE,SEARCH=(PRD2.PROD)
// LIBDEF SOURCE,SEARCH=(PRIVATE.COPYLIB)
// DLBL   IJSYS01,'%COBVSE.WORKFILE.IJSYS01',0,VSAM,RECSIZE=4096,      X
               RECORDS=(50,100),DISP=(NEW,KEEP)
// DLBL   IJSYS02,'%COBVSE.WORKFILE.IJSYS02',0,VSAM,RECSIZE=4096,      X
               RECORDS=(50,100),DISP=(NEW,KEEP)
// DLBL   IJSYS03,'%COBVSE.WORKFILE.IJSYS03',0,VSAM,RECSIZE=4096,      X
               RECORDS=(50,100),DISP=(NEW,KEEP)
// DLBL   IJSYS04,'%COBVSE.WORKFILE.IJSYS04',0,VSAM,RECSIZE=4096,      X
               RECORDS=(50,100),DISP=(NEW,KEEP)
// DLBL   IJSYS05,'%COBVSE.WORKFILE.IJSYS05',0,VSAM,RECSIZE=4096,      X
               RECORDS=(50,100),DISP=(NEW,KEEP)
// DLBL   IJSYS06,'%COBVSE.WORKFILE.IJSYS06',0,VSAM,RECSIZE=4096,      X
               RECORDS=(50,100),DISP=(NEW,KEEP)
// DLBL   IJSYS07,'%COBVSE.WORKFILE.IJSYS07',0,VSAM,RECSIZE=4096,      X
               RECORDS=(50,100),DISP=(NEW,KEEP)
// DLBL   IJSYSLN,'%COBVSE.WORKFILE.IJSYSLN',0,VSAM,RECSIZE=322,       X
               RECORDS=(400,600)
// OPTION LINK
// EXEC   IGYCRCTL,SIZE=IGYCRCTL
000100 IDENTIFICATION DIVISION.
       .
       .
       .
/*
/&
```

Figure 72. Example of JCL for Compilation

JOB
Specifies the name of the job as JOB1. The JOB statement indicates the beginning of a job.

LIBDEF PHASE
Defines the sublibrary where the COBOL/VSE compiler resides.

LIBDEF SOURCE
Defines the sublibrary where the copy members reside.

IJSYS01, IJSYS02, IJSYS03, IJSYS04, IJSYS05, IJSYS06, IJSYS07
Defines compiler work files used by the compiler to process the source program. All work files must be on direct-access storage devices. VSE/VSAM will determine the volumes on which to allocate the compiler work files from the SAM ESDS default model in the VSAM catalog. The "%" prefix in the file-ID indicates that the work file is partition-unique.

IJSYSLN
Defines the SYSLNK file that receives output from the LINK option (the object phase).

OPTION LINK
Specifies that the generated object code be placed on disk or tape to be used later as input for the linkage editor.

EXEC
Specifies that the COBOL/VSE compiler (IGYCRCTL) is to be invoked.

/* The end-of-data file statement indicates the end of the input to the compiler (source code), and separates data from subsequent job control statements in the input stream.

/& The end-of-job statement indicates the end of the job.

Batch Compiling

A sequence of separate COBOL programs may be compiled with a single invocation of the compiler. The object programs produced from this compilation may be cataloged separately into a VSE Librarian sublibrary, with a member type of OBJ. The NAME compiler option may be used for this. Alternatively, the object programs from this compilation may be link-edited into a single phase.

Each program in the sequence must be terminated by an END PROGRAM header, except the last program in the batch (for which the END PROGRAM header is optional). CBL/PROCESS statements may optionally precede each program in the sequence.

If the END PROGRAM header is omitted from a program (other than the last program in a sequence of separate programs), the next program in the sequence will be nested within the preceding program. In this case, an intervening PROCESS statement will cause the generation of error diagnostics. An intervening CBL statement will also cause the generation of error diagnostics unless the CBL statement is coded entirely within the sequence number area (columns 1 through 6). In this case, no diagnostic message will be issued for the CBL statement because it is considered a label for the source statement line.

Note: If the CMPR2 compiler option is in effect, individual programs **must** be separated by the CBL form of the PROCESS/CBL statement (END PROGRAM headers **cannot** be used when CMPR2 is in effect). The PROCESS form of the PROCESS/CBL statement **cannot** be used as a program separator under CMPR2.

Options for each program in the sequence may be specified in the installation default macro, on the invocation of the compiler, and on CBL/PROCESS statements preceding a program. The following rules apply for options in a batch compile:

- The option settings used for each program in the sequence are based on the following hierarchy. See Figure 73 on page 210 for an example of compiler option hierarchy.
  1. Installation defaults, fixed at your site
  2. The values of the BUFSIZE, LIB, and SIZE compiler options that were in effect for the first program in the batch
  3. The CBL/PROCESS statements, if any, for the current program
  4. Options specified on the compiler invocation (for example, JCL PARM)
  5. Options specified on the JCL OPTION statement
  6. Nonfixed installation defaults

  If the current program does not contain CBL/PROCESS statements, then the settings of options that were in effect for the previous program are used.

If a CBL/PROCESS statement is specified in the current program, the CBL/PROCESS statements are resolved together with the options in effect prior to the first program.

- If the BUF, LIB, or SIZE options are required by any program in the sequence, they must be in effect for the first program of the batch sequence. (All programs in the batch will be treated as a single input file during BASIS, COPY, or REPLACE processing.)
- If the LIB option is specified for the batch, the NUMBER and SEQUENCE options cannot be changed during the batch compilation.

```
PP 5686-068 IBM COBOL for VSE/ESA 1.1.1                              Date 06/16/1998  Time 13:41:27   Page     1

JCL OPTION parameters:
NODECK,LINK,LIST,NOLISTX,NOSYM,TERM,NOXREF

Invocation parameters:
NOTERM

PROCESS(CBL) statements:
CBL FLAG(I,I)

Options in effect:                        All options are installation defaults unless otherwise noted:
    NOADATA
      ADV
      QUOTE
    NOAWO
      BUFSIZE(4096)
      ⋮
      FLAG(I,I)                           Process option PROGRAM 1
      ⋮
      OBJECT                              JCL OPTION statement option
      ⋮
    NOTERM                                INVOCATION option
      ⋮
End of compilation for program 1
      ⋮
PP 5686-068 IBM COBOL for VSE/ESA 1.1.1                              Date 06/16/1998  Time 13:41:27   Page    23

PROCESS(CBL) statements:
CBL APOST

Options in effect:
      ADV
    APOST                                 Process option in effect for PROGRAM 2
    NOAWO
      BUFSIZE(4096)
      ⋮
      FLAG(I)                             Returns to installation option for PROGRAM 2, and subsequent prog
      ⋮
      OBJECT                              JCL OPTION statement option remains in effect
      ⋮
    NOTERM                                INVOCATION option remains in effect
      ⋮
End of compilation for program 2
```

Figure 73. The Batch Compile Hierarchy For Compiler Options

If the NAME compiler option is in effect for a program in the batch sequence, in conjunction with the OBJECT compiler option, a linkage editor PHASE control statement is generated for that program.

Note: The VSE Linkage Editor does not support the link-editing of multiple separate phases with one invocation of the linkage editor. The NAME compiler option should only be specified for the first program in the batch sequence.

If the NAME compiler option is in effect for a program in the batch sequence, in conjunction with the DECK compiler option, a VSE Librarian CATALOG control statement is generated for that program. This simplifies the cataloging of separate object phases.

Figure 74 shows one invocation of the compiler, compiling three programs (PROG1, PROG2, and PROG3) and creating three object phases.

```
// JOB    jobname
// DLBL   IJSYSPH,'ijsysph.file-ID',0,SD
// EXTENT SYSPCH,volser,1,0,start,tracks
   ASSGN  SYSPCH,DISK,VOL=volser,SHR
// OPTION DECK,NOLINK
// EXEC   IGYCRCTL,SIZE=IGYCRCTL
 CBL NAME
010100    IDENTIFICATION DIVISION.
010200    PROGRAM-ID PROG1.
   ⋮
019000  END PROGRAM PROG1.
 CBL NAME
020100  IDENTIFICATION DIVISION.
020200  PROGRAM-ID PROG2.
   ⋮
029000  END PROGRAM PROG2.
030100  IDENTIFICATION DIVISION.
030200  PROGRAM-ID PROG3.
   ⋮
039000  END PROGRAM PROG3.
/*
   CLOSE  SYSPCH,cuu
// DLBL   IJSYSIN,'ijsysph.file-ID',0,SD
// EXTENT SYSIPT,volser
   ASSGN  SYSIPT,DISK,VOL=volser,SHR
// EXEC   LIBR,PARM='ACCESS SUBLIB=lib.sublib'
/*
   CLOSE  SYSIPT,SYSRDR
/&
```

Figure 74. Example of a Batch Compilation

Notes to Figure 74:

1. The JCL label information for the compiler work files and SYSLNK, if previously added to the system standard or partition standard label area, need not be specified in the job stream.
2. PROG1, PROG2 and PROG3 generate separate object phases on SYSPCH, each preceded by a VSE Librarian CATALOG control statement.
3. SYSPCH is assigned to a direct-access device. The SYSPCH file is then used as input to the VSE Librarian to catalog the object phases.
4. If the compiler does not reside in the SVA, the sublibrary where the compiler resides must be defined in a LIBDEF PHASE statement.

If the LANGUAGE option on the CBL/PROCESS statement is diagnosed as an error, the language selection will revert back to what was in effect prior to the first CBL/PROCESS statement encountered during compilation. The language in effect

during batch compilations will conform to the rules of CBL/PROCESS statement processing in that environment.

The following example illustrates the behavior of the LANGUAGE compiler option in a batch environment. The default installation option is uppercase ENGLISH (abbreviated to UE), and the invocation option is XX (a nonexistent language).

```
             Source                      Language in
                                           Effect
-------------------------------------------------------------------------
CBL   LANG(JP),FLAG(I,I),AP0ST,SIZE(MAX)  |UE|  Installation default -- UE
      IDENTIFICATION DIVISION.            |JP|  Invocation           -- XX
      PROGRAM-ID.  COMPILE1.              |: |
            .  .  .                       |: |
      END PROGRAM  COMPILE1.              |: |
CBL    LANGUAGE(YY)                       |UE|  CBL resets language
CBL    SIZE(2048K),LANGUAGE(JP),LANG(!!)  |: |  to UE. LANGUAGE(YY)
      IDENTIFICATION DIVISION.            |JP|  is ignored since it
      PROGRAM-ID.  COMPILE2.              |: |  is superseded by (JP).
            .  .  .                       |: |  (!!) is not alpha-
      END PROGRAM  COMPILE2.              |: |  numeric and is
      IDENTIFICATION DIVISION.            |: |  discarded.
      PROGRAM-ID.  COMPILE3.              |: |
            .  .  .                       |: |
      END PROGRAM  COMPILE3.              |: |
CBL   LANGUAGE(JP),LANGUAGE(YY)           |UE|  CBL resets language
            .  .  .                       |: |  to UE. LANGUAGE(YY)
            .  .  .                       |: |  supersedes (JP) but
            .  .  .                       |: |  is nonexistent.
```

Figure 75. LANGUAGE Compiler Option Example on a Batch Compile

For COMPILE1, the default language uppercase ENGLISH (UE) is in effect when scanning the invocation options. A diagnostic message is issued in uppercase English because XX is a nonexistent language identifier. The default UE remains in effect when scanning the CBL statement. The unrecognized option AP0ST in the CBL statement is diagnosed in uppercase English because the CBL statement has not completed processing and UE was the last valid language option. After the CBL options processing, the language in effect becomes Japanese (JP).

In COMPILE2, CBL statement errors are diagnosed in uppercase English because the language in effect prior to the first program is used (in this case uppercase English). If multiple LANGUAGE options are specified, only the last valid language specified is used. In this example the last valid language is Japanese (JP), and thus Japanese becomes the language in effect on completion of processing the CBL options. If diagnostics in Japanese are wanted for the options in the CBL/PROCESS statements, then the language in effect prior to COMPILE1 must be Japanese.

In COMPILE3, there is no CBL statement, and the language in effect, Japanese (JP), is inherited from the previous compilation.

Following COMPILE3, the CBL statement again resets the language in effect to uppercase English (UE). The language option in the CBL statement resolves the last specified two character alphanumeric language identifier, which is YY. Since YY is nonexistent, the language in effect remains uppercase English.

Input and Output Files

Figure 76 lists the function, and allowable device types for each file.

Figure 76 (Page 1 of 2). Files Used for Compilation

File	Type	Function	Allowable Device Types
SYSIPT (Required)[1]	Input	Reading the source program	Card reader Magnetic tape Direct access
SYSLST (Required)[1]	Output	Writing the storage map, listings, and messages	Printer Magnetic tape Direct access
SYSLOG (Optional)		Writing diagnostic and progress messages	Display console
SYSPCH (Optional)		Punching the object phase deck	Card punch Magnetic tape Direct access
SYSLNK (Optional)		Creating an object phase file as output from the compiler and input to the linkage editor	Direct access
SYSADAT (Optional)		Writing information about the program compile environment and program data elements	Direct access
IJSYS01 (Required)[2]	Work file	Work file needed by the compiler during compilation	Direct access
IJSYS02 (Required)[2]		Work file needed by the compiler during compilation	
IJSYS03 (Required)[2]		Work file needed by the compiler during compilation	
IJSYS04 (Required)[2]		Work file needed by the compiler during compilation	
IJSYS05 (Optional)[2]		Work file needed when LIB option is in effect	
IJSYS06 (Required)[2]		Work file needed by the compiler during compilation	
IJSYS07 (Required)[2]		Work file needed by the compiler for creating listing	

Figure 76 (Page 2 of 2). Files Used for Compilation

File	Type	Function	Allowable Device Types
Copy libraries (Optional)[1]	Library	Optional user source program libraries	Direct access

Note:

1. See “EXIT” on page 234 for additional information.
2. This file must be a single volume file unless it is defined using the VSE/VSAM Space Management for SAM Feature.

Required Compiler Files

Basic compilation requires the following files:

- SYSIPT—to provide the source program input
- SYSLST—for the compiler printed output
- IJSYS01, IJSYS02, IJSYS03, IJSYS04, IJSYS06, IJSYS07— for compiler work files. The logical unit numbers for these compiler work files are SYS001, SYS002, SYS003, SYS004, SYS006 and SYS007 respectively.
  (An optional work file, IJSYS05, with logical unit number SYS005, must be specified if you are using the LIB compiler option. The LIB compiler option is required if you have COPY, REPLACE, or BASIS statements in your program).

If you are requesting specific compilation features, specify the following files through JCL statements:

- SYSLOG—if you are using the TERMINAL compiler option, to get the compiler progress and diagnostic messages
- SYSPCH—if you are using the DECK compiler option, to request an object module be produced
- SYSLNK—if you are using the OBJECT compiler option, to request that an object module be produced for input to the linkage editor
- Copy libraries—if your source program uses the COPY or BASIS statement
- SYSADAT—use the ADATA option to generate the SYSADAT file (containing data produced by the compiler)

Source Code File: SYSIPT

This file is read by the compiler to obtain your source code. If you include your source code or BASIS statement in your job stream, it must immediately follow the EXEC statement that invokes the compiler, and be terminated with a /* statement. You can, however, use JCL statements to define a file that contains your source code. For example, to define a direct-access device file, use the DLBL, EXTENT, and ASSGN statements.

```
// DLBL   IJSYSIN,'file-ID',0,SD
// EXTENT SYSIPT,volser,1,0,start,tracks
// ASSGN  SYSIPT,DISK,VOL=volser,SHR
```

Output File: SYSLST

This file is used by the compiler to produce a listing. Output may be directed to a printer, a direct-access device, or a magnetic-tape device. The listing will include the results of the default or specified options of the PARM parameter (that is, diagnostic messages, the object code listing). For example:

```
// ASSGN  SYSLST,PRT1
```

Directing Compiler Messages to the Console: SYSLOG

SYSLOG can only be assigned permanently, and is usually assigned at system initialization.

Specifying Libraries: LIBDEF Job Control Statement

Add the LIBDEF job control statements if your program contains COPY or BASIS statements. These LIBDEF statements define the sublibraries that contain the data requested by COPY statements (in the source code) or by a BASIS statement in the input stream.

```
// LIBDEF SOURCE,SEARCH=(lib.sublib)
```

Note: You do not need the LIBDEF SOURCE statement if the NOLIB option is in effect.

Connect multiple sublibraries in the search chain if you have multiple copy or basis sublibraries. For example:

```
// LIBDEF SOURCE,SEARCH=(lib.sublib1,lib.sublib2)
```

Sublibraries are on direct-access storage devices.

Note: The sublibraries should be specified in the search chain in the order in which you want them to be searched.

Creating Object Code: SYSLNK or SYSPCH

When using the OBJECT compiler option, or DECK compiler option, you can store the object code on disk or tape. The compiler uses the SYSLNK or SYSPCH files you define in your JCL to store the object code.

You do not need to define SYSLNK in your JCL if the NOOBJECT option is in effect.

```
// DLBL   IJSYSLN,'file-ID',0,SD
// EXTENT SYSLNK,volser,1,0,start,tracks
// ASSGN  SYSLNK,DISK,VOL=volser,SHR
```

In the example above, the object phase is created ready to be passed to the linkage editor.

Note: Your installation may use the DECK option and JCL to define the SYSPCH file.

An example of defining SYSPCH as a direct-access device file follows.

```
// DLBL   IJSYSPH,'file-ID',0,SD
// EXTENT SYSPCH,volser,1,0,start,tracks
   ASSGN  SYSPCH,DISK,VOL=volser,SHR
```

Note: You do not need to define the SYSPCH file if the NODECK option is in effect.

Creating an Associated Data File : SYSADAT

When using the ADATA compiler option, the compiler uses the SYSADAT file you define in your JCL to store ADATA information.

An example of defining SYSADAT as a direct-access device file follows.

```
// DLBL   SYSADAT,'file-ID',0,SD
// EXTENT SYS008,volser,1,0,start,tracks
   ASSGN  SYS008,DISK,VOL=volser,SHR
```

Controlling Your Compilation

As noted before, the compiler's main job is to translate 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 direct and control your compilation. Each is discussed next.

Using Compiler-Directing Statements

You can put compiler-directing statements in your source program to help direct compilation. For a description of these statements see “Compiler-Directing Statements” on page 257. For the syntax of these statements, see *COBOL/VSE Language Reference*.

Using Compiler Options

The compiler is installed and set up with default compiler options. These default options will be used unless you replace them by specifying the wanted compiler options in one of the following ways, depending on which system you are using for compilation:

- For certain options, with the OPTION statement in your JCL
- With a PROCESS (or CBL) statement preceding the Identification Division header
- With the PARM parameter on the EXEC statement in your JCL

Note: You cannot replace any compiler options that your installation has set up as **fixed**.

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 such a listing, specify XREF. If you do not want one, specify NOXREF.

Some options have subparameters. For example, if you want 44 lines per page on your listings, specify LINECOUNT(44).

Precedence of Compiler Options

Compiler options are recognized in the order of precedence below:

Level 1: Installation defaults, fixed by your installation
Level 2: Those on PROCESS (or CBL) statements
Level 3: Those on JCL PARM= parameter on the EXEC statement
Level 4: Those on JCL OPTIONS statement
Level 5: Installation defaults, but not fixed

Options "fixed by your installation" refers to options customized using the options module IGYCOPT, and specified there as non-overridable. Level 5 refers to options customized using IGYCOPT but *not* specified there as non-overridable.

Level 4 refers to either the JCL OPTION statement or the STDOPT statement, and only applies to those options that can be specified in this manner (for example, DECK and XREF). The STDOPT JCL statement defines installation default options, and these can be temporarily overridden by the OPTIONS statement. For more information, see *VSE/ESA System Control Statements*.

Within this hierarchy of precedence, there are also rules for conflicting and mutually exclusive options as described under "Conflicting Compiler Options" on page 218.

Specifying Options on the PROCESS (CBL) Statement

Your programming installation can inhibit the use of PROCESS statements with the default options phase of the COBOL compiler. When PROCESS statements are found in a COBOL program where they are not allowed by the installation, error diagnostics are generated by the COBOL compiler.

You can code compiler options on the PROCESS statement. The PROCESS statement is placed before the Identification Division header and has the following format:

```
PROCESS option1 [,option2] ... [,optionn]
       IDENTIFICATION DIVISION.
```

One or more blanks must separate PROCESS and the first option. Separate options with a comma or a blank. The PROCESS statement must be placed before any comment lines or compiler-directing statements.

There must not be any embedded spaces within options. For example, FLAG(*x*,*y*) may **not** be written FLAG(*x y*).

PROCESS can start in columns 1 through 66. A sequence field is allowed in columns 1 through 6. When used with a sequence field, PROCESS can start in columns 8 through 66. If used, the sequence field must contain six characters, and the first character must be numeric.

You can use CBL as a synonym for PROCESS. CBL can start in columns 1 through 70. When used with a sequence field, CBL can start in columns 8 through 70.

You can use more than one PROCESS statement. If multiple PROCESS statements are used, they must follow one another with no intervening statement of any other type.

Options cannot be continued across multiple PROCESS statements.

Specifying Options Using JCL

An example of specifying compiler options using JCL follows:

```
   ⋮
// EXEC PGM=IGYCRCTL,SIZE=IGYCRCTL,PARM='LIST,NOCOMPILE(S),           *
               OBJECT,FLAG(E,E)'
```

Specifying Options, Using the JCL OPTION Statement

An example of specifying compiler options, using the JCL OPTION statement follows:

```
   ⋮
// OPTION LINK,SYM
// EXEC   IGYCRCTL
```

Compiler Options and their JCL OPTION Statement Equivalents

The following compiler options may be specified by using the equivalent option of the JCL OPTION statement.

Compiler Option	JCL Option Equivalent	Comments
DECK	DECK	The DECK compiler option is always specified using the JCL OPTION statement.
LIST	LISTX	
MAP	SYM	
OBJECT	LINK CATAL	The OBJECT compiler option is always specified using the LINK or CATAL option of the JCL OPTION statement.
SOURCE	LIST	
TERMINAL	TERM	
XREF	SXREF XREF	Use the XREF option of the JCL OPTION statement if you wish to specify the XREF(FULL) compiler option. Use the SXREF option of the JCL OPTION statement if you wish to specify the XREF(SHORT) compiler option.

Conflicting Compiler Options

The positive form of the compiler option and its negative form, for example, DECK and NODECK, are opposing options. If you specify both of them on the same level in the hierarchy listed above, the option specified last takes effect. For example, within your PROCESS (or CBL) statement or within your JCL PARM= statement, the option specified last takes effect.

In addition to the directly opposing options, a few of the compiler options are mutually exclusive. That is, when you specify one of the options in column A of Figure 77 on page 219, the option in column B is normally ignored, and the option in column C is forced on.

For example, if you specify both OFFSET and LIST in your PROCESS statement, in any order, OFFSET takes effect and LIST is ignored. However, results can vary, depending on the level at which you specify the option. For example, if you specify OFFSET in your JCL statement but LIST in your PROCESS statement, LIST will take effect because the options specified in the PROCESS statement and any options forced on by an option specified in the PROCESS statement have higher precedence.

Figure 77. Mutually Exclusive Options at the Same Level of Precedence

A: This is specified	B: These are ignored[1]	C: These are forced on[1]
TEST TEST(ALL) TEST(STMT) TEST(PATH) TEST(BLOCK)	OPTIMIZE	NOOPTIMIZE
OFFSET	LIST	NOLIST
CMPR2	FLAGSTD FLAGSAA DBCS DATEPROC	NOFLAGSTD NOFLAGSAA NODBCS NODATEPROC
NOCMPR2	FLAGMIG	NOFLAGMIG
WORD	FLAGSTD	NOFLAGSTD
FLAGSTD	FLAGSAA FLAGMIG DBCS	NOFLAGSAA NOFLAGMIG NODBCS
FLAGSAA	FLAGMIG	NOFLAGMIG
DBCS	FLAGMIG	NOFLAGMIG

Note: [1]Unless in conflict with a fixed installation default option.

What if one of the options from column A is set up as a default for your system and you want to use a conflicting option from column B? It is possible that an option may be set up at the installation level as a fixed option, in which case the options it conflicts with cannot be put into effect by individual programmers. But if the option from column A is a nonfixed default option, you can put a conflicting option from column B in effect by specifying it on your PROCESS statement or in your JCL PARM= statement.

For example:

> If OFFSET is your system default (nonfixed) but
> You want to use LIST for your program,

you can do so by specifying LIST in your PROCESS statement or JCL PARM= statement.

For more information on compiler options, including performance and ANSI considerations, see Chapter 15, “Compiler Options” on page 224.

Results of Compilation

When the compiler finishes processing your source program, it will have produced one or more of the following, depending on the compiler options you selected:

Figure 78. Possible Output Produced by the Compiler

Result	Option
Listing of your source program	SOURCE
List of errors the compiler discovered in your program	FLAG
Your object code	OBJECT and/or DECK with COMPILE
Listing of object code in machine and assembler language	LIST
Map of the data items in your program	MAP
Map of the relative addresses in your object code	OFFSET
Sorted cross-reference listing of procedure-, program-, and data-names	XREF
A system dump, if compilation ended with abnormal termination	DUMP
An Associated-Data File	ADATA

Listing output from a compilation will be in the file defined by SYSLST; object output will be in SYSLNK or SYSPCH. Progress and diagnostic messages may be directed to SYSLOG, as well as included in SYSLST. ADATA records will be in the file defined by SYSADAT.

Your immediate concern will be the errors the compiler found in your program. These are discussed briefly in Compiler-Detected Errors and Messages below.

If the compiler found no errors, you can go to the next step in the process: link-editing your program. See *LE/VSE Programming Guide* for information on this step. (If you used compiler options to suppress object code generation, you must recompile to obtain it.)

Save the listings you produced during compilation. Their use will come later, during the test-execution stage of your work, should you need to debug or tune.

Compiler-Detected Errors and Messages

As the compiler processes your source program, it checks for errors you might have made in violation of the rules of the COBOL/VSE language. For each such error discovered, the compiler issues a message. These messages are included in the compilation listing (subject to the FLAG option).

Each message does the following:

- Explains the nature of your error
- Identifies the compiler phase that detected the error
- Identifies the severity level of the error

Wherever possible, the message(s) provide(s) specific instructions for correcting the error.

Compiler Error Messages

The messages for errors found during processing of compiler options, CBL and PROCESS statements, or BASIS, COPY and REPLACE statements are displayed near the top of your listing.

The messages for compilation errors found in your program (ordered by line number) are displayed near the end of the listing for each program.

A summary of all errors found during compilation is displayed near the bottom of your listing. Each message issued by the compiler is of the following form:

Format

LineID	Message code	Message text
[*nnnnnn*]	IGY*ppxxxx-l*	*text of message*

nnnnnn The number of the source statement of the last line the compiler was processing. Source statement numbers are listed on the source printout of your program. If you specified the NUMBER option at compile time, these are your original source program numbers. If you specified NON-UMBER, the numbers are those generated by the compiler.

IGY The prefix that identifies this message as coming from the COBOL/VSE compiler.

pp Two characters that identify which phase of the compiler discovered the error. As an application programmer, you can ignore this information, unless you are diagnosing a suspected compiler error. In that case, see *COBOL/VSE Diagnosis Guide*.

xxxx A 4-digit number identifies the error message.

l Indicates the severity level of the error: I, W, E, S, or U (see "Compiler Error Message Codes").

Remember, if you used the FLAG option to suppress messages, there may be additional errors in your program.

Compiler Error Message Codes

Errors the compiler can catch fall into five categories of severity:

I	Informational (Return Code=0)	This is not an error that affects the running of the program; rather it is a coding inefficiency or other such condition that you can choose to change.
W	Warning (Return Code=4)	Although the statement in which the condition occurs is syntactically correct, it has the potential for causing an error when your program is run.
E	Error (Return Code=8)	The condition is definitely an error. However, the compiler has tried to correct it for you, and it is possible that your program will run properly.

S	Severe (Return Code=12)	The condition is a serious error. The compiler will not attempt to correct the error, but compilation will continue.
U	Unrecoverable (Return Code=16)	The error condition is of such magnitude that the compiler cannot continue.

In the following example, the part of the statement that caused the message to be issued is enclosed in quotes.

```
    .
    .
    .
LineID  Message code  Message text

     2  IGYDS0009-E   "PROGRAM" should not begin in area "A".  It was processed as if found in area "B".

     2  IGYDS1089-S   "PROGRAM" was invalid.  Scanning was resumed at the next area "A" item, level-number,
                      or the start of the next clause.

     2  IGYDS0017-E   "ID" should begin in area "A".  It was processed as if found in area "A".

     2  IGYDS1003-E   A "PROGRAM-ID" paragraph was not found.  Program name "CBLPGM01" was assumed.

     2  IGYSC1082-E   A period was required.  A period was assumed before "ID".

     2  IGYDS1102-E   Expected "DIVISION", but found "ALONGPRO".  "DIVISION" was assumed before "ALONGPRO".

     2  IGYDS1082-E   A period was required.  A period was assumed before "ALONGPRO".

     2  IGYDS1089-S   "ALONGPRO" was invalid.  Scanning was resumed at the next area "A" item, level-number,
                      or the start of the next clause.

     2  IGYDS1003-E   A "PROGRAM-ID" paragraph was not found.  Program name "CBLPGM02" was assumed.

     3  IGYPS0017-E   "PROCEDURE" should begin in area "A".  It was processed as if found in area "A".

    34  IGYSC0137-E   Program-name "ALONGPRO" did not match the name of any open program.  The "END PROGRAM" statement
                      was assumed to have ended program "CBLPGM02".

    34  IGYSC0136-E   Program "CBLPGM01" required an "END PROGRAM" statement at this point in the program.
                      An "END PROGRAM" statement was assumed.
Messages    Total    Informational    Warning    Error    Severe    Terminating

Printed:      12                                  10         2
   .
   .
   .
   .
```

Correcting Your Mistakes

Messages about source coding errors indicate where the error occurred (LineID) and the text of the message tells you what the problem is. With these, you can correct your source program and recompile.

Although you should try to correct your errors, it is not absolutely necessary to fix all of them. A W-level or I-level message could be left in a program without much risk, and you might well feel that the recoding and compilation needed to remove the error are not worth the effort. On the other hand, S-level and E-level errors are in the realm of probable program failure and ought to be corrected.

U-level errors are in a class by themselves. In this case, you have no choice but to correct the error, because the compiler is forced to terminate early and not produce complete object code and listing. In contrast with the four lower levels of errors, a

U-level error might not result from a mistake in your source program. It could arise from a flaw in the compiler itself, or in the system program.

If you decide to correct your compile-time errors, do so by editing your source file.

After correcting your errors, you need to recompile your program. If this second compilation is successful, you may then go on to the link-editing step. If the compiler still finds problems, you should repeat the above procedure until only informational messages or no messages at all are returned.

Generating a List of All Compiler Error Messages

You can generate a complete listing of compiler diagnostic messages, with their explanations, by compiling a program with a *program-name* of ERRMSG specified in the PROGRAM-ID paragraph. The rest of the program may be omitted. For example:

```
Identification Division.
Program-ID. ErrMsg.
```

Chapter 15. Compiler Options

You can direct and control compilation with the following:

- Compiler options
- Compiler-directing statements

Compiler options are listed and described in alphabetic order in “Compiler Option Descriptions” on page 226. Compiler-directing statements are listed at the end of this chapter on page 257.

Compiler Options Summary

Compiler options control compilation such that various aspects of your program are affected.

Figure 79 (Page 1 of 2). List of Compiler Options

Aspect of Your Program	Compiler Option	Abbreviations	Found on Page
Source language	APOST	None	246
	CMPR2	None	228
	CURRENCY	CURR/NOCURR	229
	DBCS	None	232
	LIB	None	240
	NUMBER	NUM/NONUM	242
	QUOTE	Q	246
	SEQUENCE	SEQ/NOSEQ	248
	WORD	WD/NOWD	255
Date processing	DATEPROC	DP	231
	INTDATE	None	238
	YEARWINDOW	YW	256
Maps and listings	LANGUAGE	LANG(EN|UE|JA|JP)	239
	LINECOUNT	LC	240
	LIST	None	240
	MAP	None	241
	OFFSET	OFF/NOOFF	245
	SOURCE	S/NOS	249
	SPACE	None	249
	TERMINAL	TERM/NOTERM	250
	VBREF	None	254
	XREF	X/NOX	255

Figure 79 (Page 2 of 2). List of Compiler Options

Aspect of Your Program	Compiler Option	Abbreviations	Found on Page
Object deck generation	COMPILE	C/NOC	228
	DECK	D/NOD	232
	NAME	None	242
	OBJECT	OBJ/NOOBJ	244
Object code control	ADV	None	227
	AWO	None	227
	FASTSRT	FSRT/NOFSRT	234
	NUMPROC	None	243
	OPTIMIZE	OPT/NOOPT	245
	OUTDD	OUT	246
	TRUNC	None	252
	ZWB	None	257
Virtual storage usage	BUFSIZE	BUF	227
	SIZE	SZ	248
	DATA	None	230
	DYNAM	DYN/NODYN	233
	RENT	None	247
	RMODE	None	247
Debugging and diagnostics	DUMP	DU/NODU	232
	FLAG	F/NOF	234
	FLAGMIG	None	235
	FLAGSAA	None	236
	FLAGSTD	None	236
	TEST	None	251
	SSRANGE	SSR/NOSSR	250
Other	ADATA	None	226
	EXIT	EX(INX,LIBX,PRTX, ADX)	415

Default Values for Compiler Options

The default options that were set up when your compiler was installed will be in effect for your program unless you replace them with other options. (In some installations, certain compiler options are set up as **fixed** so that you cannot replace 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 will list the default options specified by your installation.

Performance Considerations

There are several performance considerations you should be aware of when using compiler options. The DYNAM, FASTSRT, OPTIMIZE, NUMPROC(PFD), RENT, SSRANGE, TEST, and TRUNC compiler options can all affect run-time performance.

See Chapter 19, “Program Tuning” on page 326 for more details.

Option Settings for COBOL 85 Standard Compilation

The following compiler options are required to conform to the COBOL 85 Standard specification:

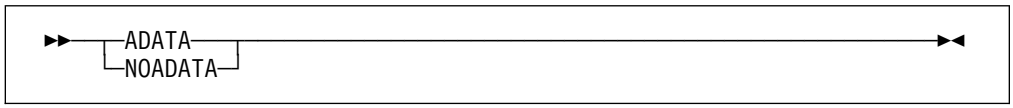
ADV	LIB
NOCMPR2	NAME(ALIAS) or NAME(NOALIAS)
NODATEPROC	NONUMBER
NODBCS	NUMPROC(NOPFD) or NUMPROC(MIG)
DYNAM	QUOTE
NOFASTSRT	NOSEQUENCE
NOFLAGMIG	TRUNC(STD)
NOFLAGSAA	NOWORD
FLAGSTD(H)	ZWB
INTDATE(ANSI)	

Compiler Option Descriptions

The compiler option descriptions that follow are given in alphabetic order. For a list of compiler options by effect, refer to Figure 79 on page 224.

Note: The defaults listed with the options below are the COBOL/VSE defaults shipped with the product. They may have been changed by your installation.

ADATA



Default is: NOADATA

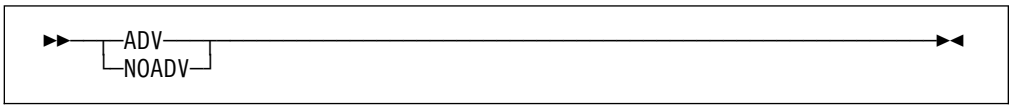
Abbreviations are: None

The ADATA option allows you to produce a file containing program data. From this file (SYSADAT) information about the compiled program can be extracted. The SYSADAT file provides a general-use programming interface to the compiler output. Utilities that previously parsed the compiler output list can now make use of the SYSADAT records.

Note: You cannot specify this option in a PROCESS (CBL) statement. It can only be specified:

- On the PARM field of JCL
- As an installation default

ADV



Default is: ADV

Abbreviation is: None

ADV is meaningful only if you use WRITE . . . ADVANCING in your source code.

With ADV in effect, the compiler adds one byte to the record length to account for the printer control character.

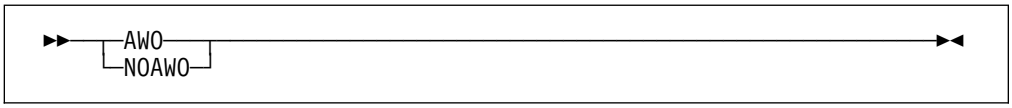
ADV conforms to the COBOL 85 Standard.

Use NOADV if you have already adjusted your record length to include one byte for the printer control character.

APOST

See “QUOTE/APOST” on page 246.

AWO

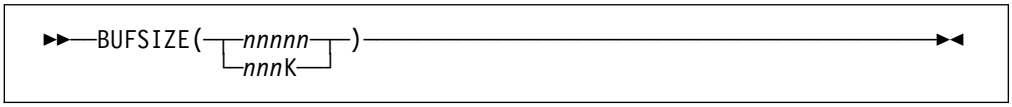


Default is: NOAWO

Abbreviation is: None

With AWO specified, the APPLY WRITE-ONLY clause will be in effect if any file within the program is physical sequential with blocked V-mode records. The clause will be in effect even if it was not specified within the program.

BUFSIZE



Default is: 4096

Abbreviation is: BUF

nnnnn
A decimal number that must be at least 256.

***nnn*K**
A decimal number in 1K increments.

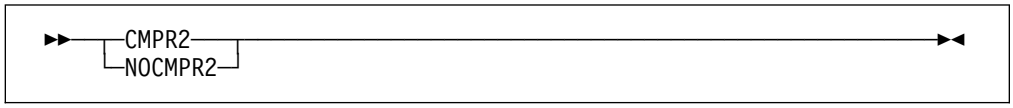
Use BUFSIZE to allocate an amount of main storage to the buffer for each compiler

work file (where 1K = 1024 bytes decimal). Usually, a large buffer size will enhance the performance of the compiler.

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.

CMPR2



Default is: NOCMPR2

Abbreviations are: None

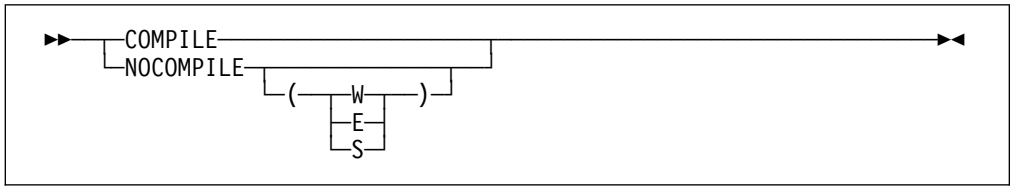
The CMPR2 option is provided for compatibility with VS COBOL II. Use CMPR2 when you want the compiler to generate code that is compatible with code generated by VS COBOL II Release 2.

Implementation of the COBOL 85 Standard created some instances where incompatibilities with VS COBOL II Release 2 can occur. Use of the CMPR2 and FLAGMIG options aid in the migration of programs written for VS COBOL II Release 2 to COBOL/VSE.

NOCMPR2 conforms to the COBOL 85 Standard.

Note: New COBOL/VSE language elements, such as intrinsic functions, are not supported under CMPR2. For more information, see *COBOL/VSE Migration Guide*.

COMPILE



Default is: NOCOMPILE(S)

Abbreviations are: C / NOC

Use the COMPILE option only if you want to force full compilation even in the presence of serious errors. All diagnostics and object code will be generated.

Note: You should not attempt to execute the object code generated if the compilation resulted in serious errors—the results could be unpredictable or an abnormal termination could occur.

Use NOCOMPILE without any subparameter to request a syntax check (only diagnostics produced, no object code).

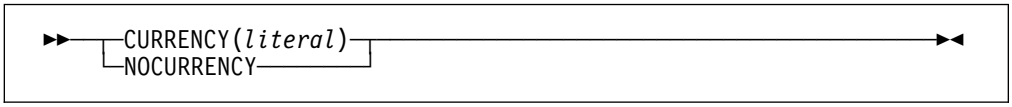
Use NOCOMPILE with W, E, or S for conditional full compilation. For meanings of error codes, see “Compiler-Detected Errors and Messages” on page 220. Full compilation (diagnostics and object code) will stop when the compiler finds an error of the level you specify (or higher), and only syntax checking will continue.

If you specify an unconditional NOCOMPILE, the following options have **no effect** because no object code will be produced:

DECK
LIST
OBJECT
OFFSET
OPTIMIZE
SSRANGE
TEST

Note: These options will be listed even though they will have no effect.

CURRENCY



Default is: NOCURRENCY

Abbreviations are: CURR / NOCURR

The default currency symbol is the dollar sign ($). You can use the CURRENCY option to specify an alternate default currency symbol to be used for the COBOL program.

NOCURRENCY specifies that no alternate default currency symbol will be used.

To change the default currency symbol, use the CURRENCY option with *literal* as follows:

literal
It must be a valid COBOL nonnumeric literal (including a hex literal) representing a single character which must not be any of the following:

- 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 DBCS literal

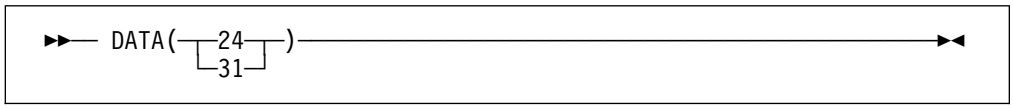
If your program processes only one currency type, you can use the CURRENCY option as an alternative to the CURRENCY SIGN clause for selecting the currency symbol you will use in the PICTURE clause of your program. If your program processes more than one currency type, you should use the CURRENCY SIGN clause with the WITH PICTURE SYMBOL phrase to specify the different currency sign types.

When both the CURRENCY option and the CURRENCY SIGN clause are used in a program, the CURRENCY option is ignored. Currency symbols specified in the CURRENCY SIGN clause(s) may be used in PICTURE clauses.

When the NOCURRENCY option is in effect and you omit the CURRENCY SIGN clause, you may only use the dollar sign ($) as the PICTURE symbol for the currency sign.

Note: The CURRENCY option literal can be delimited by either the quote or the apostrophe, regardless of the QUOTE/APOST compiler option setting.

DATA



Default is: DATA(31)

Abbreviation is: None

LE/VSE provides services that control the storage that is used at run time. COBOL/VSE uses these services for all storage requests.

For reentrant programs, the DATA(24|31) compiler option, in conjunction with the HEAP run-time option, controls whether storage for dynamic data areas (such as Working-Storage and FD record areas) is acquired from below the 16-megabyte line or from unrestricted storage.

When you specify the run-time option HEAP(BELOW), the DATA(24|31) compiler option has no effect; the storage for all dynamic data areas is allocated from below the 16-megabyte line. However, with HEAP(ANYWHERE) as the run-time option, storage for dynamic data areas is allocated from below the 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.

Specify the DATA(24) compiler option for programs running in 31-bit addressing mode that are passing data parameters to programs in 24-bit addressing mode. This ensures that the data will be addressable by the called program.

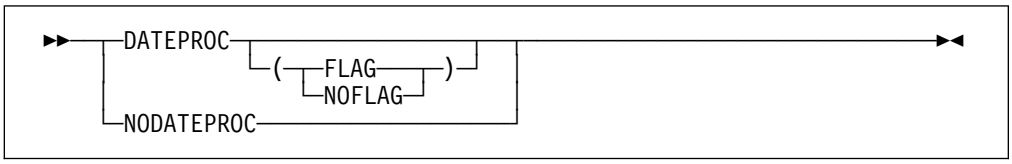
External Data: In addition to affecting how storage is acquired for dynamic data areas, the DATA(24|31) compiler option can also influence where storage for external data is obtained. Storage required for external data will be acquired from unrestricted storage if the following conditions are met:

- The program is compiled with the DATA(31) compiler option.
- The HEAP(ANYWHERE) run-time option is in effect.
- The ALL31(ON) run-time option is in effect.

In all other cases, the storage for external data will be obtained from below the 16-megabyte line. To specify the ALL31(ON) run-time option, all the programs in the run unit must be capable of running in 31-bit addressing mode.

For full details on run-time options, see the *LE/VSE Programming Reference*.

DATEPROC



Default is: NODATEPROC, or DATEPROC(FLAG) if only DATEPROC is specified

Abbreviations are: DP|NODP

Use the DATEPROC option to enable the millennium language extensions of the COBOL compiler. For information on using these extensions, see Chapter 22, “Using the Millennium Language Extensions” on page 366.

Note: VisualAge COBOL Millennium Language Extensions for VSE/ESA (program number 5686-MLE) must be installed on your system to specify anything other than NODATEPROC.

DATEPROC(FLAG)
With DATEPROC(FLAG), the millennium language extensions are enabled, and the compiler will produce a diagnostic message wherever a language element uses or is affected by the extensions. The message will usually be an information-level or warning-level message that identifies statements that involve date-sensitive processing. Additional messages may be generated that identify errors or possible inconsistencies in the date constructs. For information on how to reduce these diagnostic messages, see “Analyzing Date-Related Diagnostic Messages” on page 380.

Production of diagnostic messges, and their appearance in or after the source listing, is subject to the setting of the FLAG compiler option.

DATEPROC(NOFLAG)
With DATEPROC(NOFLAG), the millennium language extensions are in effect, but the compiler will not produce any related messages unless there are errors or inconsistencies in the COBOL source.

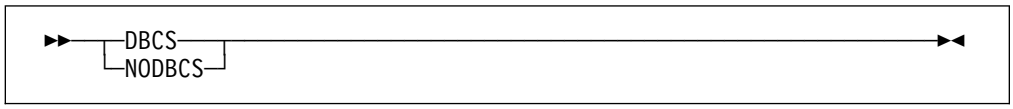
NODATEPROC
NODATEPROC indicates that the extensions are not enabled for this compilation unit. This affects date-related program constructs as follows:

- The DATE FORMAT clause is syntax-checked, but has no effect on the execution of the program.
- The DATEVAL and UNDATE intrinsic functions have no effect. That is, the value returned by the intrinsic function is exactly the same as the value of the argument.
- The YEARWINDOW intrinsic function returns a value of zero.

Notes:

1. Specification of the DATEPROC option requires that the NOCMPR2 option is also used.
2. NODATEPROC conforms to the COBOL 85 Standard.

DBCS



Default is: NODBCS

Abbreviations are: None

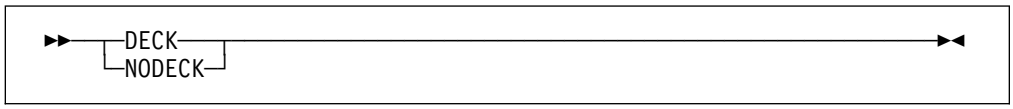
Use of DBCS causes the compiler to recognize X'0E' (SO) and X'0F' (SI) as shift codes for the double byte portion of a nonnumeric literal.

With DBCS selected, the double byte portion of the literal is syntax checked and the literal remains category alphanumeric.

DBCS is ignored if either CMPR2 or FLAGSTD is in effect.

NODBCS conforms to the COBOL 85 Standard.

DECK



Default is: NODECK

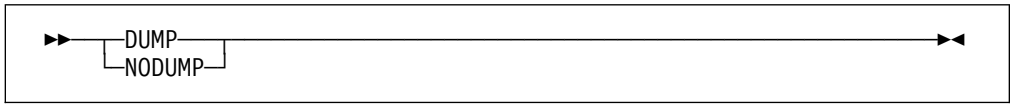
Abbreviations are: D / NOD

Use DECK to produce object code in the form of 80-column card images. If you use the DECK option, be certain that SYSPCH is defined in your JCL for compilation.

The DECK compiler option is specified by using the DECK job control option on the JCL OPTION statement. For more information on how to specify the DECK job control option, see *VSE/ESA System Control Statements*.

Note: For compatibility with COBOL/370™ under MVS and VM, the DECK or NODECK option may be specified on your PROCESS (or CBL) statement, or on the PARM parameter of your JCL EXEC statement. The option will be syntax-checked, but it will be ignored.

DUMP



Default is: NODUMP

Abbreviations are: DU / NODU

Note: This option is not intended for general use.

Use DUMP to produce a system dump at compile time for an internal compiler error. The DUMP option should only be used 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.

For information on how to interpret the user abend code, see the *COBOL/VSE Diagnosis Guide*. If you use the DUMP option, include the DUMP job control option or the PARTDUMP job control option on the JCL OPTION statement. For more information on how to specify DUMP and PARTDUMP job control options see *VSE/ESA System Control Statements*.

With DUMP, the compiler will not issue a diagnostic message before abnormal termination processing. Instead, a user abend will be issued with an IGYppnnnn message. In general, a message IGYppnnnn corresponds to a compile-time user abend nnnn. However, both IGYpp5nnn and IGYpp1nnn messages produce a user abend of 1nnn. You can usually distinguish whether the message is really a 5nnn or a 1nnn by recompiling with the NODUMP option.

Use NODUMP if you want normal termination processing, including:

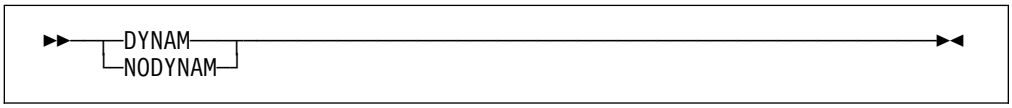
- Diagnostic messages produced so far in compilation
- A description of the error
- The name of the compiler phase currently executing
- The line number of the COBOL statement being processed when the error was found (if you have compiled with OPTIMIZE, the line number may not always be correct; for some errors it will be the last line in the program)
- The contents of the general purpose registers

Note: The use of the DUMP and OPTIMIZE compiler options together may 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. The use of the NODUMP option will allow the compiler to issue message `IGYOP3124-W` and continue processing.

DYNAM



Default is: NODYNAM

Abbreviations are: DYN / NODYN

Use DYNAM to cause separately compiled programs invoked through the CALL *literal* statement to be loaded dynamically at run time. DYNAM causes dynamic loads (for CALL) and deletes (for CANCEL) of separately compiled programs at

object time. Any CALL *identifier* statements that cannot be resolved in your program are also treated as dynamic calls.

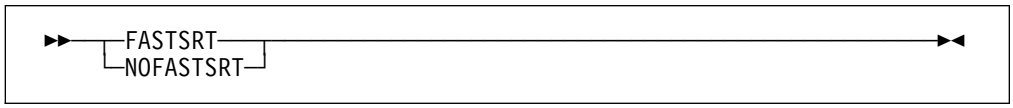
DYNAM conforms to the COBOL 85 Standard.

Note: Do not use DYNAM with CICS.

EXIT

The EXIT compiler option is discussed in Appendix D, “EXIT Compiler Option” on page 415.

FASTSRT



Default is: NOFASTSRT

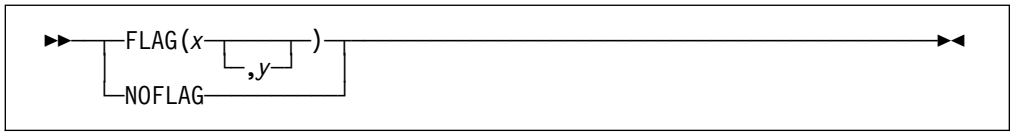
Abbreviations are: FSRT / NOFSRT

FASTSRT allows DFSORT/VSE, or its equivalent, to perform the input and output instead of COBOL/VSE.

NOFASTSRT conforms to the COBOL 85 Standard.

Refer to “Improving Sort Performance with FASTSRT” on page 184 for further information regarding the use of the FASTSRT option.

FLAG



Default is: FLAG(I)

Abbreviations are: F / NOF

x I, W, E, S, or U

y I, W, E, S, or U

(See “Compiler Error Message Codes” on page 221 for meanings of error codes.)

Use FLAG(*x*) to produce diagnostic messages for errors of a severity level *x* or above at the end of the source listing.

Use FLAG(*x*,*y*) to produce diagnostic messages for errors of severity level *x* or above at the end of the source listing, with error messages of severity *y* and above to be embedded directly in the source listing. The severity coded for *y* must not be lower than the severity coded for *x*. To use FLAG(*x*,*y*), you also need to specify the SOURCE compiler option.

Error messages in the source listing are set off by embedding the statement number within an arrow that points to the message code. The message code is then followed by the message text. For example:

```
  000413     MOVE CORR WS-DATE TO HEADER-DATE

==000413==>    IGYPS2121-S       " WS-DATE " was not defined as a data-name.  ...
```

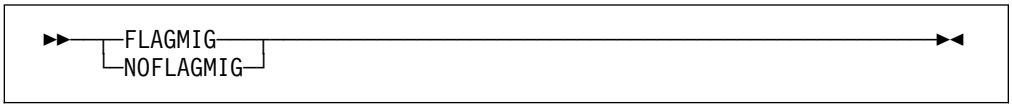
With FLAG(x,y) selected, messages of severity *y* and above will be embedded in the listing following the line that caused the message. (Refer to the notes below for exceptions.)

Use NOFLAG to suppress error flagging. NOFLAG will not suppress error messages for compiler options.

Notes:

1. Specifying embedded level-U messages is accepted, but will not produce any messages in the source. Embedding a level-U message is not recommended.
2. The FLAG option does not affect diagnostic messages produced before the compiler options are processed.
3. Diagnostic messages produced during processing of compiler options, CBL and PROCESS statements, or BASIS, COPY and REPLACE statements, are never embedded in the source listing. All such messages appear at the beginning of the compiler output.
4. Messages produced during processing of the *CONTROL (*CBL) statement are not embedded in the source listing.

FLAGMIG



Default is: NOFLAGMIG

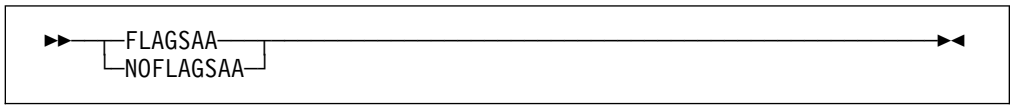
Abbreviations are: None

The FLAGMIG is provided for compatibility with VS COBOL II. Use FLAGMIG to identify language elements that may be implemented differently in VS COBOL II Release 2 than in COBOL/VSE. To use FLAGMIG, you also need to specify the CMPR2 compiler option.

NOFLAGMIG conforms to the COBOL 85 Standard.

Implementation of the ANSI 1985 Standard created some instances where incompatibilities with VS COBOL II Release 2 can occur. Use of the CMPR2 and FLAGMIG options aid in the migration of programs written for VS COBOL II Release 2 to COBOL/VSE. For further information on the items that are CMPR2 sensitive and their behavior under CMPR2, see *COBOL/VSE Migration Guide*.

FLAGSAA



Default is: NOFLAGSAA

Abbreviations are: None

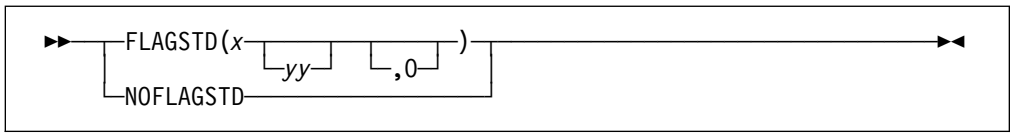
Use FLAGSAA to identify language elements that are **not** defined as part of the Systems Application Architecture COBOL Level 1 Programming Interface (SAA COBOL Level 1 CPI) provided by IBM. The purpose of this flagging is to identify elements that may restrict program portability across IBM systems. The elements will be flagged with warning (W) level messages at compile time.

To use FLAGSAA, the NOCMPR2 compiler option must be in effect, and FLAGSTD cannot be specified. If FLAGSTD and FLAGSAA are specified together under NOCMPR2, FLAGSAA is ignored.

NOFLAGSAA conforms to the COBOL 85 Standard.

See *Systems Application Architecture Common Programming Interface COBOL Reference* for more information about SAA and the COBOL interface.

FLAGSTD



Default is: NOFLAGSTD

Abbreviations are: None

x M, I, or H

Specifies the level or subset of Standard COBOL to be regarded as conforming.

M Specifies that language elements that are **not** from the minimum subset are to be flagged as “nonconforming standard.”

I Specifies that language elements that are **not** from the minimum or the intermediate subset are to be flagged as “nonconforming standard.”

H Specifies that the high subset is being used and elements will not be flagged by subset, and that elements in the IBM Extension category will be flagged as “nonconforming Standard, IBM extension.”

yy D, N, or S

Specifies, by a single character or combination of any two, the optional modules to be included in the subset.

D Specifies that elements from Debug module level 1 are **not** flagged as “nonconforming standard.”

N Specifies that elements from Segmentation module level 1 are **not** flagged as “nonconforming standard.”

S Specifies that 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.”

Use FLAGSTD to get informational messages about the Standard COBOL elements included in your program. To use FLAGSTD, the NOCMPR2 compiler option must be in effect. You can specify any of the following items for flagging:

- A selected FIPS (Federal Information Processing Standard) COBOL subset
- Any of the optional modules
- Obsolete language elements
- Any combination of subset and optional modules
- Any combination of subset and obsolete elements
- IBM extensions (these are flagged any time FLAGSTD is specified and are identified as “nonconforming nonstandard”)

The informational messages appear in the source program listing and contain the following information:

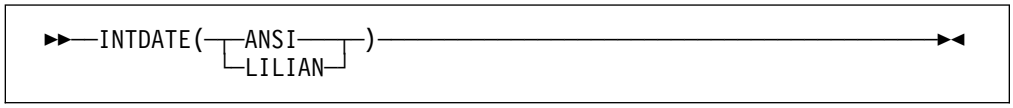
- Identify the element as “obsolete,” “nonconforming standard,” or “nonconforming nonstandard” (a language element that is both obsolete and nonconforming is flagged as obsolete only).
- Identify the clause, statement, or header that contains the element.
- Identify the source program line and beginning location of the clause, statement, or header that contains the element.
- Identify the subset or optional module to which the element belongs.

FLAGSTD requires the standard set of reserved words.

In the following example, the line number and column where a flagged clause, statement, or header occurred are shown, as well as the message code and text. At the bottom is a summary of the total of the flagged items and their type.

```
  LINE.COL CODE       FIPS MESSAGE TEXT

           IGYDS8211  Comment lines before "IDENTIFICATION DIVISION":
                      nonconforming nonstandard, IBM extension to
                      ANS/ISO 1985.

     11.14 IGYDS8111  "GLOBAL clause":  nonconforming standard, ANS/ISO
                      1985 high subset.

     59.12 IGYPS8169  "USE FOR DEBUGGING statement":  obsolete element
                      in ANS/ISO 1985.


FIPS MESSAGES TOTAL             STANDARD      NONSTANDARD      OBSOLETE

                3                    1              1              1
```

INTDATE



Default is: INTDATE(ANSI)

INTDATE(ANSI) instructs the compiler to use the ANSI COBOL Standard starting date for Integer date format dates used with date intrinsic functions. Day 1 is January 1, 1601.

With INTDATE(ANSI), the date intrinsic functions will return the same results as in COBOL/VSE Release 1 (without PTF UQ04360).

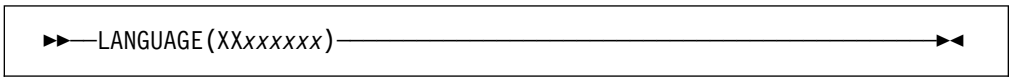
INTDATE(LILIAN) instructs the compiler to use the LE/VSE Lilian starting date for integer date format dates used with date intrinsic functions. Day 1 is October 15, 1582.

With INTDATE(LILIAN), the date intrinsic functions will return results compatible with the LE/VSE date callable services. These results will be different than in COBOL/VSE Release 1.

Notes:

1. When INTDATE(LILIAN) is in effect, CEECBLDY will not be usable since you will have no way to turn an ANSI integer into a meaningful date using either intrinsic functions or callable services. If you code a CALL *literal* statement with CEECBLDY as the target of the call with INTDATE(LILIAN) in effect, the compiler will diagnose this and convert the call target to CEEDAYS.
2. If your installation default option is INTDATE(LILIAN), you should recompile all of your COBOL/VSE programs that use Intrinsic Functions to ensure that all of your code will be using the LILIAN integer date standard. This method is the safest, because you can store integer dates and pass them between programs, even between PL/I, COBOL, and C programs, and know that the date processing will be consistent.

LANGUAGE



Default is: LANGUAGE(UENGLISH)

Abbreviation: LANG(EN|UE|JA|JP)

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).

XXxxxxxx
Specifies the language for compiler output messages. Entries for XXxxxxxx are shown in Figure 80 on page 239.

Figure 80. Entries for the LANGUAGE Compiler Option (2)

Entry	Abbreviation	Explanation
ENGLISH	EN	The output will be printed in mixed-case English.
JAPANESE (1)	JA, JP	The output will be printed in the Japanese language using the Japanese character set.
UENGLISH	UE	The output will be printed in uppercase English. This is the default selection.

Notes:

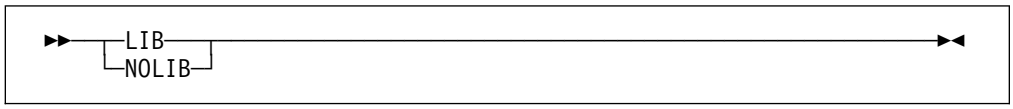
1. To specify a language other than UENGLISH, the appropriate language feature must be installed.
2. If your installation's system programmer has provided a language other than those described, you must specify at least the first two characters of this other language's name.

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 invoked.

Note: The NATLANG run-time option allows you to control the national language that is to be used for the run-time environment, including error messages, month names, and day-of-the-week names. The LANGUAGE compiler option and the NATLANG run-time option act independently of each other. They can be used together with neither taking precedence over the other.

For details on NATLANG, see the *LE/VSE Programming Reference*.

LIB



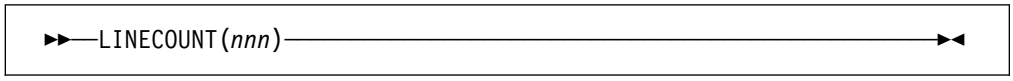
Default is: NOLIB

Abbreviations are: None

If your program uses COPY, BASIS, or REPLACE statements, you need to specify the LIB compiler option. In addition, for COPY and BASIS statements, include in your JCL the LIBDEF SOURCE statement for the sublibrary or sublibraries from which the compiler can take the copied code, and also include the JCL statements for the compiler work file IJSYS05.

LIB conforms to the COBOL 85 Standard.

LINECOUNT



Default is: LINECOUNT(60)

Abbreviation is: LC

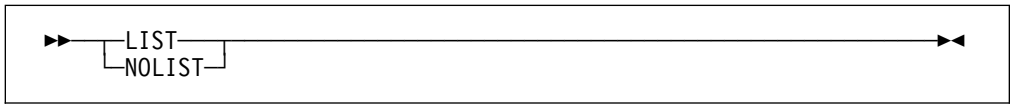
Use LINECOUNT(*nnn*) to specify the number of lines to be printed on each page of the compilation listing, or use LINECOUNT(0) to suppress pagination.

nnn must be an integer between 10 and 255, or 0.

If you specify LINECOUNT(0), no page ejects are generated within 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



Default is: NOLIST

Abbreviations are: None

Use LIST to produce a listing of the assembler-language expansion of your source code.

You will also get these in your output listing:

- Global tables

- Literal pools
- Information about Working-Storage
- Size of the program's Working-Storage, and its location in the object code if the program is compiled with the NORENT option

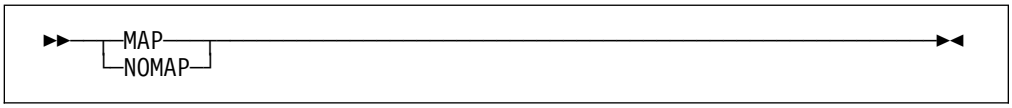
LIST and OFFSET are mutually exclusive. If you use both, LIST is ignored.

If you want to limit the assembler listing output, use *CONTROL LIST or NOLIST statements in your Procedure Division. Your source statements following a *CONTROL NOLIST are not included in the assembler listing at all, unless a *CONTROL LIST statement switches the output back to normal LIST format.

The LIST option may also be specified by using the LISTX job control option on the JCL OPTION statement. For more information on how to specify the LISTX job control option see *VSE/ESA System Control Statements*. For a description of the *CONTROL (*CBL) statement, see *COBOL/VSE Language Reference*.

For information on using LIST output, see “A Procedure Division Listing with Assembler Expansion (LIST Output)” on page 310.

MAP



Default is: NOMAP

Abbreviations are: None

Use MAP to produce a listing of the items you defined in the Data Division. Map output includes:

- Data Division map
- Global tables
- Literal pools
- Nested program structure map, and program attributes
- Size of the program's Working-Storage, 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 your Procedure Division. Your source statements following a *CONTROL NOMAP are not included in the listing until a *CONTROL MAP statement switches the output back to normal MAP format. For a description of the *CONTROL (*CBL) statement, see *COBOL/VSE Language Reference*.

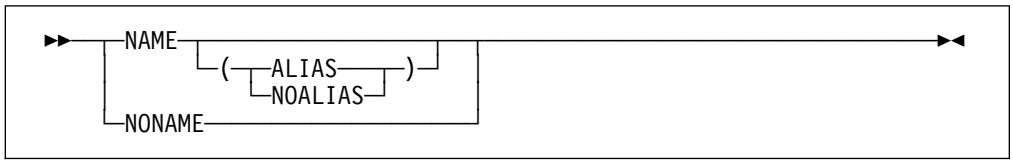
For information on using MAP output, see “Data Map Listing” on page 305.

By selecting the MAP option, you can also print an embedded MAP report in the source code listing. The condensed MAP information is printed to the right of data-name definitions in the File Section, Working-Storage Section, and Linkage Section of the Data Division.

The MAP option may also be specified by using the SYM job control option on the JCL OPTION statement. For more information on how to specify the SYM job control option see *VSE/ESA System Control Statements*. For a description of the *CONTROL (*CBL) statement, see *COBOL/VSE Language Reference*.

For information on using LIST output, see "A Procedure Division Listing with Assembler Expansion (LIST Output)" on page 310.

NAME



Default is: NONAME, or NAME(NOALIAS) if only NAME is specified

Abbreviations are: None

Use NAME to generate:

- A VSE Librarian CATALOG statement for each object module, when used in conjunction with the DECK compiler option
- A linkage editor PHASE statement, when used in conjunction with the OBJECT compiler option

When NAME is specified, in conjunction with the DECK compiler option, a VSE Librarian CATALOG statement is produced at the beginning of each object module deck written to SYSPCH. The format of the CATALOG statement produced is:

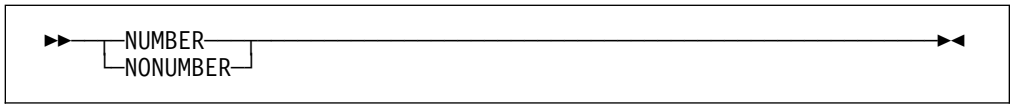
```
CATALOG module.OBJ REPLACE=YES
```

When NAME is specified, in conjunction with the OBJECT compiler option, a linkage editor PHASE statement is produced at the beginning of each object module deck written to SYSLNK. The format of the PHASE statement produced is:

```
PHASE module,*
```

Module names are formed using the rules for forming module names from PROGRAM-ID statements as described in *COBOL/VSE Language Reference*.

NUMBER



Default is: NONUMBER

Abbreviations are: NUM / NONUM

Use NUMBER if you have line numbers in your source code and want those numbers to be used in error messages and MAP, LIST, and XREF listings.

If you request NUMBER, columns 1 through 6 are checked to make sure that they contain only numbers, and the sequence is checked according to numeric collating

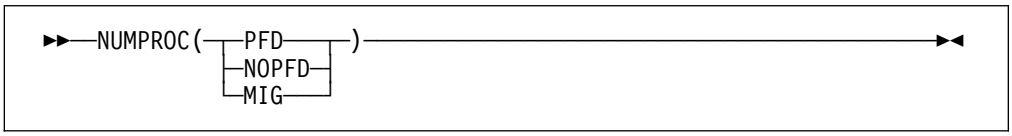
sequence. (In contrast, SEQUENCE checks them according to EBCDIC collating sequence.) When a line number is found to be out of sequence, the compiler assigns to it a line number with a value one number higher than the line number of the preceding statement. Sequence-checking continues with the next statement, based on the newly assigned value of the previous line.

If you use COPY statements and NUMBER is in effect, be sure that your source program line numbers and the COPY member line numbers are coordinated.

Use NONUMBER if you do not have line numbers in your source code, or if you want the compiler to ignore the line numbers you do have in your source code. With NONUMBER in effect, the compiler generates line numbers for your source statements and uses those numbers as references in listings.

NONUMBER conforms to the COBOL 85 Standard.

NUMPROC



Default is: NUMPROC(NOPFD)

Abbreviations are: None

Use NUMPROC(NOPFD) if you want the compiler to perform invalid sign processing. This option is not as efficient as NUMPROC(PFD); object code size will be increased, and there may be an increase in run-time overhead to validate all signed data.

NUMPROC(NOPFD) and NUMPROC(MIG) conform to the COBOL 85 Standard.

NUMPROC(PFD) is a performance option that can be used to bypass invalid sign processing. Use this option **only** if your program data agrees exactly with the following IBM system standards:

External decimal, unsigned—High-order 4 bits of the sign byte contain X'F'.

External decimal, signed overpunch—High-order 4 bits of the sign byte contain X'C' if the number is positive or 0, X'D' if it is not.

External decimal, separate sign—Separate sign contains the character '+' if the number is positive or 0, '-' 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, X'D' if it is not.

Data produced by COBOL/VSE arithmetic statements conforms to the above IBM system standards. However, the use of REDEFINES and group moves could change data so that it no longer conforms. If NUMPROC(PFD) is used, the INI-

TIALIZE statement should be used to initialize data fields, rather than using group moves.

The use of NUMPROC(PFD) can affect class tests for numeric data. NUMPROC(NOPFD), or NUMPROC(MIG), should be used if a COBOL program calls programs written in PL/I.

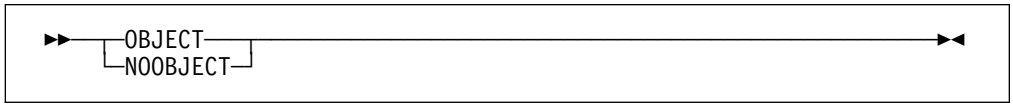
Sign representation is not only affected by the NUMPROC option, but also by the installation time option NUMCLS. See Figure 19 and Figure 20 on page 81 for the sign representations recognized by numeric class testing.

Use NUMPROC(MIG) to aid in migrating DOS/VS COBOL programs to COBOL/VSE. 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 may occur during conversion.
- Numeric comparisons are performed by a decimal compare, not a logical compare.

For more information on NUMPROC, see “Sign Representation and Processing” on page 79.

OBJECT



Default is: Installation dependent (see below)

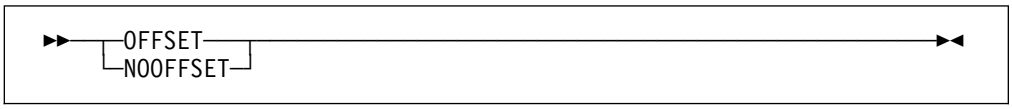
Abbreviations are: OBJ / NOOBJ

The OBJECT compiler option is used to produce object code to be used as input for the linkage editor. If you specify OBJECT, be certain that SYSLNK is defined in your JCL for compilation.

The OBJECT compiler option is specified by using either the LINK job control option or the CATAL job control option on the JCL OPTION statement. For more information on how to specify the LINK and CATAL job control options, see *VSE/ESA System Control Statements*.

For compatibility with COBOL for OS/390 & VM, the OBJECT or NOOBJECT option may be specified on your PROCESS (or CBL) statement, or on the PARM parameter of your JCL EXEC statement. The option will be syntax-checked, but it will be ignored.

OFFSET



Default is: NOOFFSET

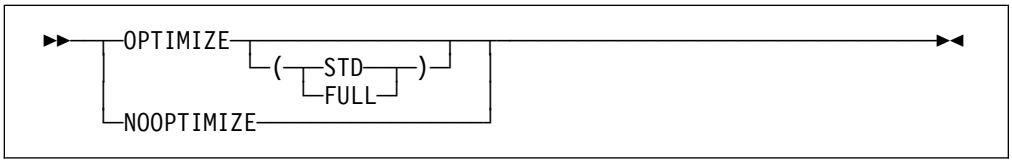
Abbreviations are: OFF / NOOFF

Use OFFSET to produce a condensed Procedure Division listing. With OFFSET, the procedure portion of the listing will contain line numbers, statement references, and the location of the first instruction generated for each statement. In addition, the following are produced:

- 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

OFFSET and LIST are mutually exclusive. If you use both, LIST is ignored.

OPTIMIZE



Default is: NOOPTIMIZE

Abbreviations are: OPT / NOOPT

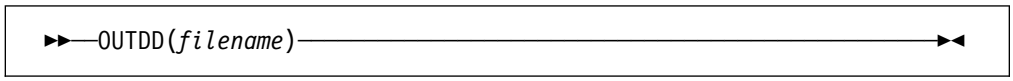
Use OPTIMIZE to reduce the run time of your object program; optimization may also reduce the amount of main storage your object program uses.

If OPTIMIZE is specified without any suboptions, then OPTIMIZE(STD) will be in effect.

The FULL suboption tells the compiler to discard any unused data items and not to generate code for any VALUE clauses for these data items. If the OPT(FULL) and MAP options are specified, then the Base Locator in the Data Division Map will have a number of XXXX if the data item is not used. The FULL suboption is mutually exclusive with the CMPR2 option. If OPTIMIZE(FULL) and CMPR2 are both specified, the OPTIMIZE(STD) and CMPR2 will be in effect.

The OPTIMIZE option is turned off in the case of a severe-level error or higher. OPTIMIZE and TEST without any suboptions or with the suboption ALL, STMT, PATH or BLOCK are mutually exclusive. If you use one of these combinations, OPTIMIZE is ignored.

OUTDD



Default is: OUTDD(SYSOUT)

Abbreviation is: OUT

filename The filename of the run-time diagnostics file.

Use OUTDD if you want run-time DISPLAY output (to the system logical output device) to go to a file other than SYSLST.

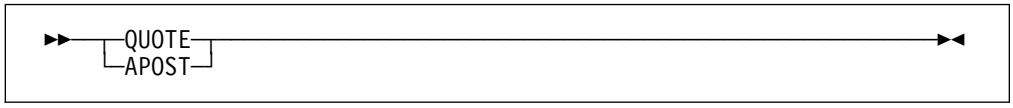
The filename specifies SYSLST or SYSOUT to direct the output to SYSLST, a DLBL filename to direct the output to a disk, or the programmer logical unit (SYS000 to SYS254) to direct the output to an unlabeled tape file or a printer.

Note: The MSGFILE run-time option allows you to specify the *filename* of the file to which all run-time diagnostics and reports generated by the RPTOPTS and RPTSTG run-time options are written. The default supplied by IBM is MSGFILE(SYSLST). If the OUTDD compiler option and the MSGFILE run-time option both specify the same filename, the DISPLAY output and error message information will be routed to the same destination.

LE/VSE does not check the validity of the MSGFILE *filename*. An invalid *filename* results in an error condition on the first attempt to issue a message.

For details on all the run-time options, see the *LE/VSE Programming Reference*.

QUOTE/APOST



Default is: QUOTE

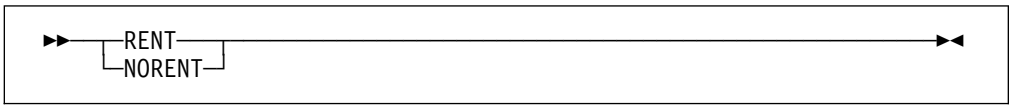
Abbreviations are: Q / APOST

Use QUOTE if you want the quotation mark (") to be the delimiter character for literals.

QUOTE conforms to the COBOL 85 Standard.

Use APOST if you want the apostrophe (') to be the delimiter character for literals.

RENT



Default is: NORENT

Abbreviation is: None

A program compiled as RENT is generated as a reentrant object phase; a program compiled as NORENT is generated as a nonreentrant object phase. Either may be invoked as a main program or subprogram.

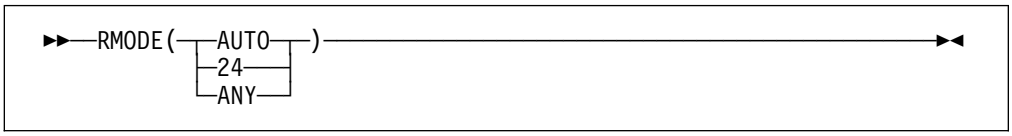
Note: You must use RENT for programs to be run under CICS.

When a reentrant program is to be run with extended addressing, the DATA(24|31) option may be used to control whether dynamic data areas are allocated in unrestricted storage or in storage acquired from below 16 megabytes. Programs must be compiled with RENT or RMODE(ANY) if they will be run with extended addressing in virtual storage above 16 megabytes.

Note: The DATA(24|31) compiler option has no effect for programs compiled with NORENT.

RENT/NORENT also affects the residency mode under which your program will run. See the description of the RMODE option. All COBOL/VSE programs have AMODE(ANY).

RMODE



Default is: AUTO

Abbreviation is: None

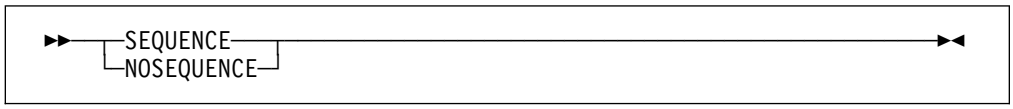
A program compiled with the RMODE(AUTO) option will have RMODE(24) if NORENT is specified, and RMODE(ANY) if RENT is specified.

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.

COBOL/VSE NORENT programs which are required to pass data to programs running in AMODE(24) must either be compiled with the RMODE(24) option, or link-edited with RMODE(24). The data areas for NORENT programs will be above the line or below the line depending on the RMODE of the program, even if DATA(24) has been specified. DATA(24) applies to programs compiled with the RENT option only.

SEQUENCE



Default is: SEQUENCE

Abbreviations are: SEQ / NOSEQ

When you use SEQUENCE, the compiler examines columns 1 through 6 of your source statements to check that the statements are arranged in ascending order according to their EBCDIC collating sequence. The compiler issues a diagnostic message if any statements are not in ascending sequence (source statements with blanks in columns 1 through 6 do not participate in this sequence check and do not result in messages).

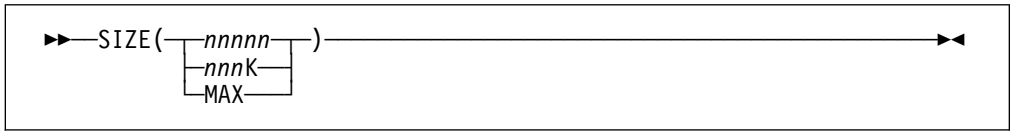
If you use COPY statements and SEQUENCE is in effect, be sure that your source program sequence fields and the COPY member sequence fields are coordinated.

If you use NUMBER and SEQUENCE, the sequence is checked according to numeric, rather than EBCDIC, collating sequence.

Use NOSEQUENCE to suppress this checking and the diagnostic messages.

NOSEQUENCE conforms to the COBOL 85 Standard.

SIZE



Default is: SIZE(MAX)

Abbreviation is: SZ

nnnnnn
A decimal number that must be at least 184320, if the compiler resides in shared storage (SVA); or 716800, if the compiler does not reside in shared storage. See your system programmer for guidance.

***nnn*K**
A decimal number in 1K increments. The minimum acceptable value is 180K, if the compiler resides in shared storage (SVA); or 700K, if the compiler does not reside in shared storage. See your system programmer for guidance.

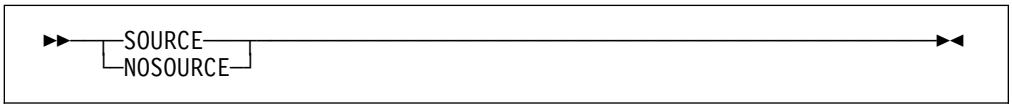
MAX
Requests the largest available block of GETVIS storage in the partition for use during compilation.

Use SIZE to specify amount of GETVIS storage available for compilation (where 1K = 1024 bytes decimal).

Do not use SIZE(MAX) if, when you invoke the compiler, you require it to leave a specific amount of unused storage available in the partition. If you specify SIZE(MAX) in an extended addressing environment, the compiler will use:

- Above the 16-megabyte line—all the storage in the partition
- Below the 16-megabyte line—storage for:
  - Work file buffers
  - Compiler phases that must be loaded below the line

SOURCE



Default is: SOURCE

Abbreviations are: S / NOS

Use SOURCE to get a listing of your source program. This listing will include any statements embedded by PROCESS or COPY statements.

SOURCE must be specified if you want embedded messages in the source listing.

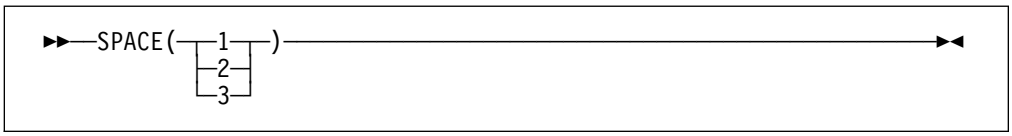
Use NOSOURCE to suppress the source code from the compiler output listing.

If you want to limit the SOURCE output, use *CONTROL SOURCE or NOSOURCE statements in your Procedure Division. Your source statements following a *CONTROL NOSOURCE are not included in the listing at all, unless a *CONTROL SOURCE statement switches the output back to normal SOURCE format.

The SOURCE option may also be specified by using the LIST job control option on the JCL OPTION statement. For more information on how to specify the LIST job control option, see *VSE/ESA System Control Statements*. For a description of the *CONTROL (*CBL) statement, see the *COBOL/VSE Language Reference*.

For information on using SOURCE output, see “Listing of Your Source Code—for Historical Records” on page 304.

SPACE



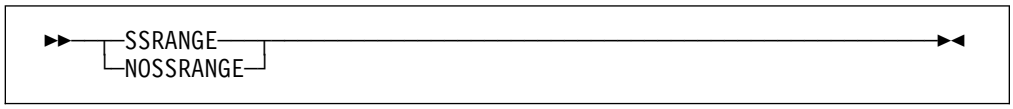
Default is: SPACE(1)

Abbreviation is: None

Use SPACE to select single, double, or triple spacing in your source code listing.

SPACE is meaningful only when SOURCE is in effect.

SSRANGE



Default is: NOSSRANGE

Abbreviations are: SSR / NOSSR

Use SSRANGE to generate code that checks if subscripts (including ALL subscripts) or indexes attempt to reference an area outside the region of the table. Each subscript or index is **not** individually checked for validity; rather, the effective address is checked to ensure that it does not cause a reference outside the region of the table. Variable-length items will also be checked to ensure that the reference is within their maximum defined length.

Reference modification expressions will be checked to ensure that:

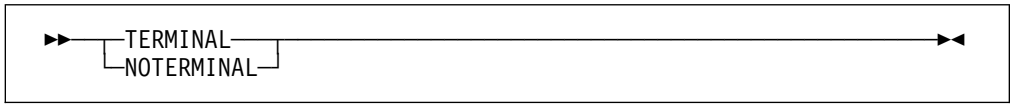
- The reference modification starting position is equal to or greater than 1.
- The reference modification starting position is not greater than the current length of the subject data item.
- The reference modification length value (if specified) is equal to or greater than 1.
- The reference modification starting position and length value (if specified) do not reference an area beyond the end of the subject data item.

If SSRANGE is in effect at compile time, the range-checking code is generated; range checking can be inhibited at run time by specifying CHECK(OFF) as a run-time option. This leaves range-checking code dormant within the object code. The range-checking code can then be optionally used to aid in resolving any unexpected errors without recompilation.

If an out-of-range condition is detected, an error message will be displayed and the program will be terminated.

Note: Remember you will only get range checking if you compile your program with the SSRANGE option and run it with the CHECK(ON) run-time option.

TERMINAL



Default is: NOTERMINAL

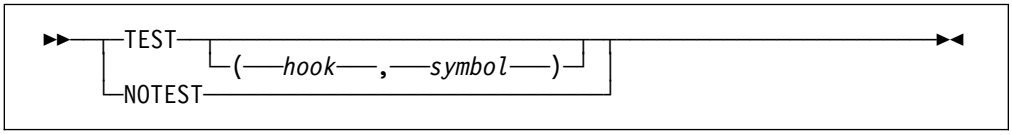
Abbreviations are: TERM / NOTERM

Use TERMINAL to send progress and diagnostic messages to the SYSLOG file.

Use NOTERMINAL if this additional output is not wanted.

The TERMINAL option may also be specified by using the TERM job control option on the JCL OPTION statement. For more information on how to specify the TERM job control option see *VSE/ESA System Control Statements*.

TEST



Default is: NOTEST

Abbreviations are: None

Use TEST to produce object code that enables Debug Tool/VSE to perfom batch and interactive debugging. 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 LE/VSE.

Use NOTEST if you do not want to generate object code with debugging information and do not want the formatted dump to include symbolic variables.

TEST has two suboptions; you can specify both, just one of the suboptions, or neither of the suboptions:

hook The hook-location suboption controls where compiled-in hooks will be generated to provide information to a debug tool.

- **NONE** Specifies that no hooks will be generated.
- **BLOCK** Specifies that hooks will be generated at all entry and exit points. A path point is anywhere 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.
- **PATH** Specifies that hooks will be generated at all path points, including program entry and exit points.
- **STMT** Specifies that hooks will be generated at every statement and label, as well as 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** Specifies that hooks will be generated at all statements, all path points, and at all program entry and exit points (both outermost and contained programs). In addition, if the DATEPROC option is in effect, hooks will be generated at all date processing statements.

symbol The symbol-table suboption controls whether dictionary tables will be generated.

- **SYM** Specifies that dictionary and calculation tables will be generated.

NOSYM Specifies that dictionary and calculation tables will not be generated.

When you specify both suboptions, they may appear in any order. The default values when TEST is specified without one or both suboptions are ALL and SYM. Therefore, TEST without any suboptions is equivalent to TEST(ALL,SYM).

Specify the SYM suboption of the TEST compiler option to have symbolic variables included in the formatted dump produced by LE/VSE.

COBOL/VSE uses the LE/VSE provided dump services to produce dumps that are consistent in content and format to those produced by other LE/VSE-conforming member languages. Whether LE/VSE produces a dump for unhandled conditions depends on the setting of the run-time option TERMTHDACT. If you specify TERMTHDACT(DUMP), a dump will be generated when a condition of severity 2 or greater goes unhandled.

For more information about dumps, see *LE/VSE Debugging Guide and Run-Time Messages*.

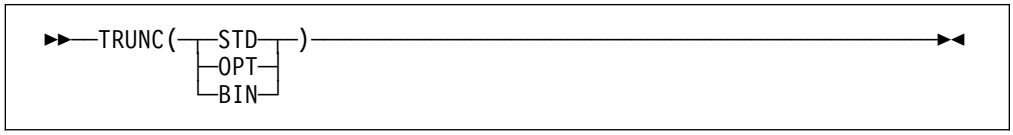
When you specify TEST without a hook-location value or with any one other than NONE, the NOOPTIMIZE compiler option goes into effect.

The TEST option will be deactivated if you use both the WITH DEBUGGING MODE clause and the USE FOR DEBUGGING statement, unless you specify the hook-location suboption NONE. The TEST option will appear in the list of options, but a diagnostic message will be issued to advise you that because of the conflict, TEST will not be in effect.

When the DATEPROC option is in effect, the TEST suboptions STMT and ALL cause hooks to be generated for all date processing statements. A date processing statement is any statement that references a date field, or any EVALUATE or SEARCH statement WHEN phrase that references a date field.

Note: Use of the TEST compiler option allows you to use Debug Tool/VSE to help debug your program. For information, see the *Debug Tool/VSE User's Guide and Reference*.

TRUNC



Default is: TRUNC(STD)

Abbreviations are: None

TRUNC(STD) conforms to the COBOL 85 Standard, while TRUNC(OPT) and TRUNC(BIN) are IBM extensions.

Use TRUNC(STD) to control the way arithmetic fields are shortened during MOVE and arithmetic operations. TRUNC(STD) applies only to USAGE BINARY receiving fields in MOVE statements and arithmetic expressions. When TRUNC(STD) is in

effect the final result of an arithmetic expression, or the sending field in the MOVE statement, is shortened to the number of digits in the PICTURE clause of the BINARY receiving field.

TRUNC(OPT) is a performance option. When TRUNC(OPT) is specified, the compiler assumes that the data conforms to PICTURE and USAGE specifications of the USAGE BINARY receiving fields in MOVE statements and arithmetic expressions. The results are manipulated in the most optimal way, either truncating to the number of digits in the PICTURE clause, or to the size of the binary field in storage (halfword, fullword, or doubleword).

You should use the TRUNC(OPT) option only if you are sure that the data being moved into the binary areas will not have a value with larger precision than that defined by the PICTURE clause for the binary item. Otherwise, truncation of high-order digits may occur. This truncation is performed in the most efficient manner possible; thus, the results will be dependent on the particular code sequence generated. It is not possible to predict the truncation without seeing the code sequence generated for a particular statement.

The TRUNC(BIN) option applies to all COBOL language elements that process USAGE BINARY data. When TRUNC(BIN) is in effect:

- BINARY receiving fields are cut off only at halfword, fullword, or doubleword boundaries.
- BINARY sending fields are treated as halfwords, fullwords, or doublewords when the receiver is numeric; TRUNC(BIN) has no effect when the receiver is not numeric.
- The full binary content of the field is significant.
- DISPLAY will convert the entire content of the binary field, with no truncation.

Note: TRUNC(BIN) is the recommended option when interfacing with other products that have System/370 format binary data (such as CICS, FORTRAN, and PLI). This is particularly true if there is a possibility of having more than 9 digits in a fullword, or more than 4 digits in a halfword.

Example 1:

```
01 BIN-VAR     PIC 99 USAGE BINARY.
⋮
MOVE 123451 to BIN-VAR
```

Figure 81. Values of the Data Items after the MOVE

	Decimal	Hex	Display
Sender	123451	00|01|E2|3B	
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 shortened to conform to the Picture clause. If the program is compiled with the TRUNC(BIN) option, the result is -7621.

The reason for the unusual looking answer in the TRUNC(BIN) version is that nonzero high-order digits were shortened. In this case, the generated code sequence would merely move the lower halfword quantity X'E23B' to the receiver. Because the new shortened value overflowed into the sign bit of the binary halfword, the value becomes a negative number.

This MOVE statement should not be compiled with the TRUNC(OPT) option because 123451 has greater precision than the PICTURE clause for BIN-VAR. If TRUNC(OPT) was used, however, the results again would be -7621. This is because the best performance was gained by not doing a decimal cut off.

Example 2:

```
01 BIN-VAR     PIC 9(6)  USAGE BINARY
⋮
MOVE 1234567891 to BIN-VAR
```

Figure 82. Values of the Data Items after the MOVE

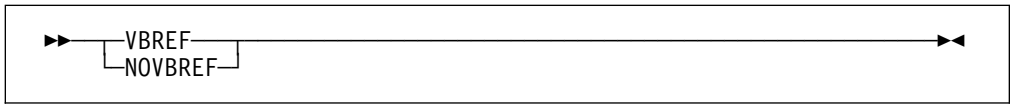
	Decimal	Hex	Display
Sender	1234567891	49|96|02|D3	
Receiver TRUNC(STD)	567891	00|08|AA|53	567891
Receiver TRUNC(OPT)	567891	00|08|AA|53	567891
Receiver TRUNC(BIN)	1234567891	49|96|02|D3	1234567891

When TRUNC(STD) is specified, the sending data is shortened to six integer digits to conform to the PICTURE clause of the BINARY receiver.

When TRUNC(OPT) is specified, the compiler assumes the sending data is not larger than the PICTURE clause precision of the BINARY receiver. The most efficient code sequence in this case performed truncation as if TRUNC(STD) had been specified.

When TRUNC(BIN) is specified, no truncation occurs because all of the sending data will fit into the binary fullword allocated for BIN-VAR.

VBREF



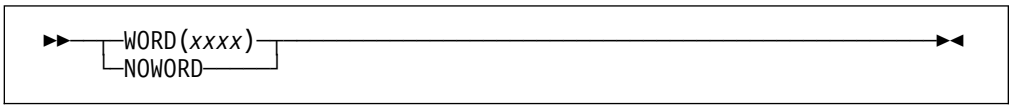
Default is: NOVBREF

Abbreviations are: None

Use VBREF to get a cross-reference among all verb types used in the source program and the line numbers in which they are used. VBREF also produces a summary of how many times each verb was used in the program.

Use NOVBREF for more efficient compilation.

WORD



Default is: NOWORD

Abbreviations are: WD / NOWD

xxxx
Are the ending characters of the name of the reserved word table (IGYC*xxxx*) to be used in your compilation. IGYC are the first 4 standard characters of the name, and *xxxx* may be 1 to 4 characters in length.

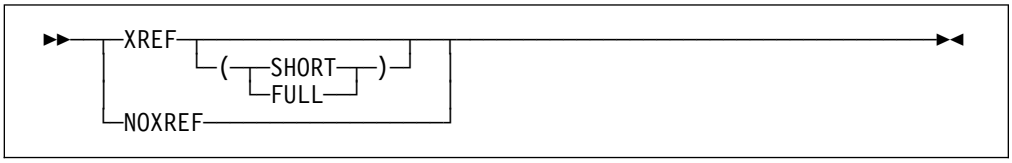
Use WORD(*xxxx*) to specify that an alternate reserved word table is to be used during compilation.

Alternate reserved word tables provide changes to the default reserved word table supplied by IBM. Your system programmer might have created one or more alternate reserved word tables for your site. See your system programmer for the names of alternate reserved word tables

COBOL/VSE provides an alternate reserved word table (IGYCCICS) specifically for CICS applications. It is set up so that COBOL words not supported under CICS are flagged with an error message. If you want to use this CICS reserved word table during your compilation, specify the compiler option WORD(CICS). For information on the CICS reserved word table, refer to “CICS Reserved Word Table” on page 390.

NOWORD conforms to the COBOL 85 Standard.

XREF



Default is: NOXREF

Abbreviations are: X / NOX

You can choose XREF, XREF(FULL), or XREF(SHORT).

Use XREF to get a sorted cross-reference listing. EBCDIC data-names and procedure-names will be listed in alphanumeric order. DBCS data-names and procedure-names will be listed based on their physical order in the program, and will appear before the EBCDIC data-names and procedure-names, unless the DBCSXREF installation option is selected with a DBCS ordering program.

Also included will be a section listing all the program names that are referenced within your program, and the line numbers where they are defined. External program names will be identified as such.

If you use XREF and SOURCE, cross-reference information will also be printed on the same line as the original source in the listing. Line number references or other information, will appear on the right hand side of the listing page. On the right of source lines that reference intrinsic functions, the letters 'INF' will appear with the line numbers of the location where the function's arguments are defined. Information included in the embedded references lets you know if an identifier is undefined or defined more than once. (UND or DUP will be printed); if an item is implicitly defined (IMP), as are special registers or figurative constants; and if a program name is external (EXT).

If you use XREF and NOSOURCE, you will get only the sorted cross-reference listing.

XREF(SHORT) will print only the explicitly referenced variables in the cross-reference listing. XREF(SHORT) applies to DBCS data names and procedure-names as well as EBCDIC names.

NOXREF suppresses this listing.

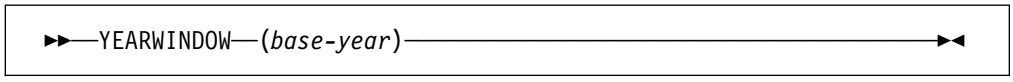
The XREF and XREF(SHORT) options may also be specified by using the XREF and SXREF job control option on the JCL OPTION statement. For more information on how to specify the XREF and SXREF job control options see *VSE/ESA System Control Statements*.

Notes:

1. Group names used in a MOVE CORRESPONDING statement will be listed in the XREF listing. The elementary names within those groups will also be listed.
2. In the data-name XREF listing, line numbers preceded by the letter “M” indicate that the data item is explicitly modified by a statement on that line.
3. XREF listings take additional storage

See Chapter 18, “Debugging” on page 292 for sample listings. Instructions on how to use listings and dumps for debugging are provided in *LE/VSE Debugging Guide and Run-Time Messages*.

YEARWINDOW



Default is: YEARWINDOW(1900)

Abbreviation is: YW

Use the YEARWINDOW option to specify the first year of the 100-year window (the century window) to be applied to windowed date field processing by the COBOL compiler. For information on using windowed date fields, see Chapter 22, “Using the Millennium Language Extensions” on page 366.

base-year represents the first year of the 100-year window, and must be specified as one of the following:

- An unsigned decimal number between 1900 and 1999.

This specifies the starting year of a fixed window. For example, YEARWINDOW(1930) indicates a century window of 1930–2029.

- A negative integer from -1 through -99.

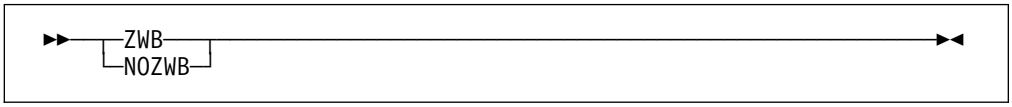
  This indicates a sliding window, where the first year of the window is calculated from the current run-time date. The number is subtracted from the current year to give the starting year of the century window. For example, YEARWINDOW(-80) indicates that the first year of the century window is 80 years before the current year at the time the program is run.

Notes:

1. The YEARWINDOW option has no effect unless the DATEPROC option is also in effect.
2. At run time, two conditions must be true:
   - The century window must have its beginning year in the 1900s
   - The current year must lie within the century window for the compilation unit

   For example, running a program in 1998 with YEARWINDOW(-99) violates the first condition, and would result in a run-time error.

ZWB



Default is: ZWB

Abbreviations are: None

With ZWB, the compiler removes the sign from a signed external decimal (DISPLAY) field when comparing this field to an alphanumeric elementary field during execution.

If the external decimal item is a scaled item (contains the symbol `'P'` in its PICTURE character string), its use in comparisons will not be affected by ZWB. Such items always have their sign removed before the comparison is made to the alphanumeric field.

ZWB affects program execution logic; that is, the same COBOL source program can give different results, depending on the option setting.

ZWB conforms to the COBOL 85 Standard.

Use NOZWB if you want to test input numeric fields for SPACES.

Compiler-Directing Statements

Several statements help you to direct the compilation of your program. For the definition of these statements, see *COBOL/VSE Language Reference*.

BASIS statement
This extended source program library statement provides a complete COBOL program as the source for a compilation.

***CONTROL (*CBL) statement**
This statement selectively suppresses or allows output to be produced. The names ***CONTROL** and ***CBL** are synonymous. This statement is described in "Controlling the Content of the Output Listing" on page 352.

COPY statement
This library statement places prewritten text in a COBOL program. (For more information on what you need to do at compile time to bring in copied code, see "Eliminating Repetitive Coding (the COPY Facility)" on page 343.)

DELETE statement
This extended source library statement removes COBOL statements from the BASIS source program.

EJECT statement
This statement specifies that the next source statement is to be printed at the top of the next page.

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, specifies which compiler options are to be used during compilation of the program. (See page 217 for the format of this statement.)

For details on specifying compiler options with the PROCESS (CBL) statement and with other methods, see the discussion under "Using Compiler Options" on page 216.

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. It is not intended for general use.

SKIP1/2/3 statement
These statements specify lines to be skipped in the source listing.

TITLE statement
This statement specifies that a title (header) be printed at the top of each page of the source listing. (See "Listing Header in the Identification Division" on page 17.)

USE statement
The USE statement provides **declaratives** to specify the following:

Error-handling procedures—EXCEPTION/ERROR
User label-handling procedures—LABEL
Debugging lines and sections—DEBUGGING

Part 4. Advanced Topics

This part of the book covers various advanced programming topics. Basic programming topics are covered in Part 2, “Coding Your Program” on page 15.

Chapter 16. Subprograms and Data Sharing

Sometimes an application is simple enough to be coded as a single, self-sufficient program. In many cases, however, an application's solution will consist of several, separately compiled programs bound together.

A **run unit** (the COBOL term synonymous with **enclave** in LE/VSE) includes one or more object programs and may include object programs written in other LE/VSE-conforming languages. Interlanguage communication (ILC) between COBOL and non-COBOL programs is discussed in *LE/VSE Writing Interlanguage Communication Applications*. If the first program to be executed in the run unit is a COBOL program, then that COBOL program is usually the **main program**. For information on exceptions to this rule, see *LE/VSE Writing Interlanguage Communication Applications*.

When a run unit consists of several, separately compiled programs that call each other, the programs must be able to communicate with each other. They need to transfer control and usually need to have access to common data. The following sections describe the methods that accomplish this interprogram communication between separately compiled programs.

Another method to achieve interprogram communication is to “nest” COBOL programs inside each other. This allows all the required subprograms for an application to be contained within one program and thereby require only a single compilation. This method is explained in “Nested Programs” on page 263.

Transferring Control to Another Program

In the Procedure Division, a program can call another program (generally called a subprogram in COBOL terms), and this called program may itself call yet another program. 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.

The called COBOL program starts executing at the top of the Procedure Division. (It is possible to specify another entry point where execution begins, using the ENTRY label in the called program. However, this is not a recommended practice in a structured program.)

When the called program processing is completed, the program can either transfer control back to the calling program or end the run unit.

A called program must not directly or indirectly execute its caller (such as program X calling program Y; program Y calling program Z; and program Z then calling program X). This is called a **recursive** call. If you attempt to execute a recursive call to a COBOL program, the run unit will end abnormally (abend).

Main Programs and Subprograms

No specific source code statements or options identify a COBOL program to be a main program or a subprogram. In an LE/VSE environment, if a COBOL program is the first program in the run unit, then it is the main program. Otherwise, it and all other COBOL programs in the run unit are subprograms.

Whether a COBOL program is a main program or a subprogram can be significant for either of two reasons:

- Effect of program termination statements.
- The state the main or subprogram is left in.

Program Termination Statements

The table below shows the action taken for each program termination statement in both a main program and a subprogram.

Figure 83. Effects of various 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 there is no next executable statement in a called program.
STOP RUN	Return to calling program[1] (Might be the system, and job 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.)	Return directly to the program that called the main program. (Might be the operating system, and job will end.)
GOBACK	Return to calling program[1] (Might be the operating system, and job will end.) Same effect as STOP RUN.	Return to calling program.

Note:

1. If the main program is called by a program written in another language that does not follow LE/VSE linkage conventions, return will be to this calling program.

State in Which Program is Left

A subprogram is usually left in its *last-used state* when it terminates with EXIT PROGRAM or GOBACK. The next time it is called in the run unit, its internal values will be as they were left, except that return values for PERFORM statements will be reset to their first values. In contrast, a main program is initialized each time it is called. There are two exceptions:

- A subprogram that is dynamically called and then canceled will be in the initial state the next time it is called.

- A program with the INITIAL attribute will be in the initial state each time it is called.

Note: The EXIT PROGRAM statement is sensitive to the CMPR2 compiler option. For VS COBOL II Release 2 compatibility and migration information, see the *COBOL/VSE Migration Guide*.

Making Calls between Programs

You will often want your COBOL/VSE programs to communicate with other COBOL and non-COBOL programs.

Calls between COBOL Programs

To transfer control from one COBOL/VSE program to another COBOL/VSE program, you can use one of these methods:

- Calls to nested programs
- Static calls
- Dynamic calls

In addition to making calls between COBOL/VSE programs, you can also make static and dynamic calls between COBOL/VSE and VS COBOL II programs and, in a non-CICS environment, between COBOL/VSE and DOS/VS COBOL programs. In a CICS environment, you must use EXEC CICS LINK to transfer control between COBOL/VSE and DOS/VS COBOL programs.

Dynamic calls from VS COBOL II and COBOL/VSE to DOS/VS COBOL programs are supported with the following restrictions:

- The DOS/VS COBOL program must be relinked with the COBOL/VSE Run-Time compatibility library.
- DOS/VS COBOL programs that are segmented cannot be loaded into GETVIS storage. These are programs that are compiled with LANGLVL(1) and specify the SEGMENT-LIMIT clause.
- The debug options STATE, FLOW, COUNT and SYMDMP are disabled.

Calls to nested programs allow you to create applications using structured programming techniques. They can also be used in place of PERFORM procedures to prevent unintentional modification of data items.

Calls to nested programs can be made using either the CALL *literal* or CALL *identifier* statement. For more information on nested programs, see “Nested Programs” on page 263.

A static call is used to invoke a separately compiled program that is link-edited into the same phase as the calling program. A dynamic call is used to invoke a separately compiled program that has been link-edited into a separate phase from the calling program. In this case, the subprogram phase is loaded into storage the first time it is called.

Note: Although they may use more storage than dynamic calls, static calls are executed more quickly.

A static call occurs when you use the CALL *literal* statement in a program that is compiled using the NODYNAM compiler option.

Use of the CALL *identifier* statement or CALL *literal* with the DYNAM compiler option results in a dynamic call. You should consider using dynamic calls when:

- The subprograms called with a CALL *literal* are used infrequently or are very large
- You want to call subprograms in their unused state
- You have an AMODE(24) program in the same run unit with COBOL/VSE programs that you want to execute in 31-bit addressing mode
- The name of the program to be called is not known until run time

For additional information on static and dynamic calls, see “Static and Dynamic Calls” on page 266.

Calls between COBOL/VSE and Non-COBOL Programs

LE/VSE provides interlanguage support which allows your COBOL/VSE programs to call and be called by PL/I VSE and assembler language programs. Whereas COBOL/VSE programs can only make static calls to PL/I VSE programs, PL/I VSE programs can make both static and dynamic calls to COBOL/VSE programs.

For full details on interlanguage communication (ILC) and information on the register conventions required for assembler calls, see *LE/VSE Writing Interlanguage Communication Applications*.

Nested Programs

Nested programs give you a method to create modular functions for your application and maintain structured programming techniques. They can be used as PERFORM procedures with the additional ability to protect “local” data items.

Nested programs allow for debugging a program before including it in the application. You can also compile your application with a single invocation of the compiler.

Structure of Nested Programs

A COBOL program may **contain** other COBOL programs. The **contained** programs may themselves contain yet other programs. A contained program may be **directly** or **indirectly** contained within a program.

Figure 84 on page 264 describes a nested program structure with directly and indirectly contained programs.

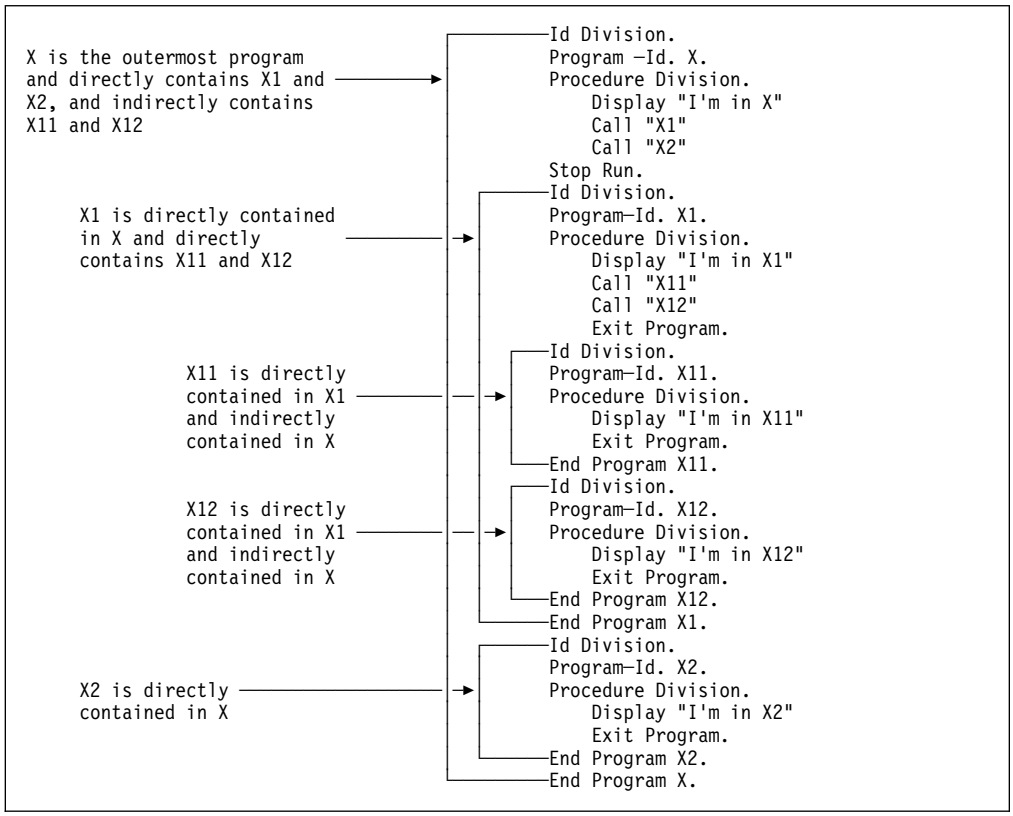


Figure 84. Nested Program Structure with Directly and Indirectly Contained Programs

Conventions for Using Nested Program Structure: There are several conventions that apply when using nested program structures.

1. The Identification Division is required in each program. All other divisions are optional.
2. Program names must be unique
3. Contained program names may be any valid COBOL word or a nonnumeric literal.
4. Contained programs cannot have a Configuration Section. The outermost program must specify any Configuration Section options that may be required.
5. Each contained program is included in the containing program immediately before its END PROGRAM header (see Figure 84).
6. Contained and containing programs must be terminated by an End Program header.

Calling Nested Programs: A contained program may only be called by its directly containing program, unless the contained program is identified as COMMON in its PROGRAM-ID clause. In that case, the COMMON program may also be called by any program that is contained (directly or indirectly) within the same program as the COMMON program. Only contained programs can be COMMON. Recursive calls are not allowed.

Figure 85 on page 265 shows the outline of a nested structure with some contained programs identified as COMMON.

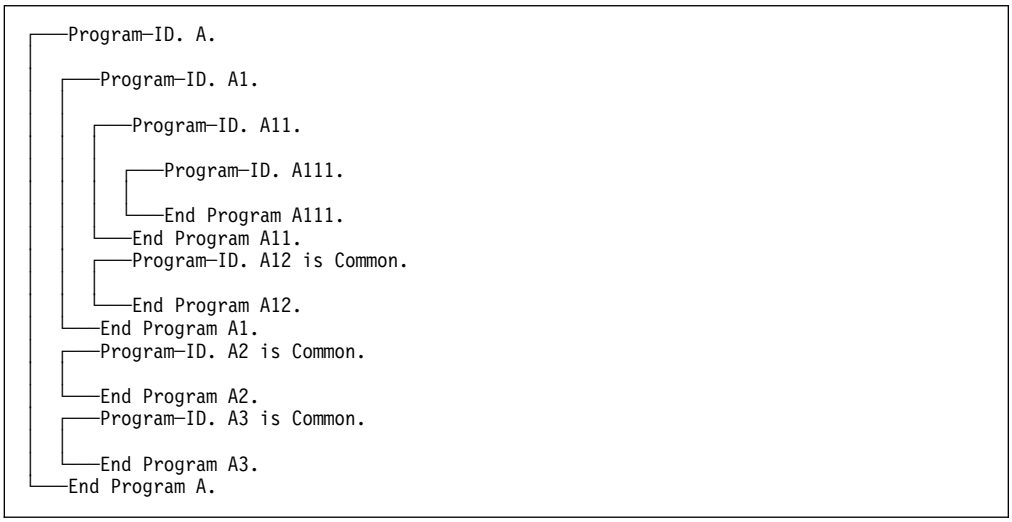


Figure 85. A Nested Structure with COMMON Programs

The following table describes the "calling hierarchy" for the structure that is shown in Figure 85. Notice that programs A12, A2, and A3 are identified as Common and the resulting differences in calls associated with them.

Figure 86. Calling Hierarchy for Nested Structures with COMMON programs

This Program	Can call these programs	And can be called by these programs
A	A1, A2, A3	None
A1	A11, A12, A2, A3	A
A11	A111, A12, A2, A3	A1
A111	A12, A2, A3	A11
A12	A2, A3	A1, A11, A111
A2	A3	A, A1, A11, A111, A12, A3
A3	A2	A, A1, A11, A111, A12, A2

You should note that:

- A2 cannot call A1 because A1 is not common and is not contained in A2.
- A111 cannot call A11 because that would be a recursive call
- A1 can call A2 because A2 is common
- A1 can call A3 because A3 is common

Scope of Names within a Nested Structure: There are two classes of names within nested structures— **local** and **global**. The class will determine whether a name is known beyond the scope of the program which declares it. There is also a specific search sequence for locating the declaration of a name after it is referenced within a program.

Local Names: Names are local unless declared to be otherwise (except the program name). These local names are not visible or accessible to any program outside of the one where they were declared; this includes both contained and containing programs.

Global Names: A name that is specified as global (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 within that program. This allows the contained programs to share common data and files from the containing program, simply by referencing the name of the item.

Any item that is subordinate to a global item (including condition names and indexes) is automatically global.

The same name may be declared with the GLOBAL clause multiple times, providing that each declaration occurs in a different program. Be aware that masking, or hiding, a name within a nested structure is possible by having the same name occur within different programs of the same containing structure. This could possibly cause some problems when a search for a name declaration is taking place.

Searching for Name Declarations: When a name is referenced within a program, a search is made to locate the declaration for that name. The search begins within the program that contains the reference and continues “outward” to containing programs until a match is found. The search follows this process:

1. Declarations within the program are searched first.
2. If no match is found, then **only** global declarations are searched in successive outer containing programs.
3. The search ends when the first matching name is found, otherwise an error exists if no match is found.

You should note that the search is for a global "name", not for a particular type of object associated with the name, such as a data item or file connector. The search stops when **any** match is found, regardless of the type of object. If the object declared is of a different type than what was expected, an error condition exists.

Static and Dynamic Calls

The following discussion applies to separately compiled subprograms only, not to nested (contained) programs. For information about calls within a nested structure, see “Nested Programs” on page 263.

When a subprogram is called, it may already be in main storage and can be link-edited in the same phase with the calling program (static call). Or it may be loaded only at the time it is called (dynamic call). With dynamic loading, the called program is loaded only when it is needed.

The link-edit process differs, depending on whether your program uses static calls or dynamic calls. For link-editing information, see the *LE/VSE Programming Guide*.

Static CALL Statement

A static call occurs when you use the CALL *literal* statement in a program that is compiled using the NODYNAM compiler option. With NODYNAM, all calls of the CALL *literal* format are handled as static calls.

In the static CALL statement, the calling COBOL program and all called programs are part of the same phase. When control is transferred to the called program, it is already resident in storage, and a branch to the called program 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. If the called program possesses the INITIAL attribute, it and each program directly or indirectly contained within it is placed into its initial state every time the called program is called within a run unit.

If alternate entry points are specified, a static CALL statement can use any alternate entry point to enter the called subprogram.

Dynamic CALL Statement

A dynamic call occurs when you use the CALL *literal* statement in a program that is compiled using the DYNAM compiler option, or when you use the CALL *identifier* statement. To dynamically call a COBOL program, the program name in the PROGRAM-ID paragraph or ENTRY statement must be identical to the corresponding phase name that contains the program, and the sublibrary that contains the phase must be available at run time.

In this form of the CALL statement, the called COBOL subprogram is not link-edited with the calling program, but is instead link-edited into a separate phase, and, at run time, is loaded only if and when it is required (that is, when called).

The execution of the dynamic CALL statement to a subprogram that is not resident in storage results in the loading of that subprogram from secondary storage into the partition containing the calling program, and a branch to the subprogram.

Thus, 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 by 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. Thus, the re-initialization of either of the following items is your responsibility:

- GO TO statements that have been altered
- Data items

If the same COBOL program is called under different run units, a separate copy of Working-Storage is allocated for each run unit.

When a CANCEL statement is issued for a subprogram, the storage occupied by the subprogram is freed, and a subsequent CALL to the subprogram will function as though it were the first. A CANCEL statement referring to a called subprogram can be issued by a program other than the original caller.

A dynamic call can only call the entry point for the phase. If a program has many entry points (specified using the ENTRY statement), more than one phase could be built, with a different entry point for each phase. If this is done, there will be one copy of the working storage for each phase.

When to use a Dynamic Call: Use a dynamic call statement when any of the following are true:

- You want to simplify maintenance tasks and take advantage of code reusability.

  When a subprogram is changed, all application phases that call it statically must be relinked. However, if the changed subprogram is called dynamically, then only the changed subprogram needs to be relinked. Thus, dynamic calls

make it easier to maintain one copy of a subprogram with a minimum amount of relinking.

- The subprograms called with CALL *literal* are used infrequently or are very large.

  If the subprograms are called only on a few conditions, dynamic calls can bring in the subprogram only when needed.

  If the subprograms are very large or there are many of them, use of static calls might require too much main storage. Less total storage would be required to call and cancel one, then call and cancel another, than to statically call both.

- You want to call subprograms in their unused state.

  This is most easily accomplished by identifying the subprogram with the INITIAL attribute. With this attribute, the subprogram will be placed in its initial (unused) state each time it is called.

  You can also selectively set the unused state by using the CALL and CANCEL procedure that is described next. (This is a more cumbersome procedure, but does provide control of the state, if that is essential). To do this, use a combination of dynamic CALL and CANCEL statements. When you CANCEL the subprogram that was initially called by a VS COBOL II or COBOL/VSE program, the next CALL will cause the subprogram to be reinitialized to its unused state. The CANCEL command does not take any action to release storage for subprograms that were dynamically loaded and branched to by non-COBOL programs.

- You have a DOS/VS COBOL or other AMODE(24) program in the same run unit with COBOL/VSE programs that you want to execute in 31-bit addressing mode.

  VS COBOL II and COBOL/VSE dynamic CALL processing include AMODE switching for AMODE(24) programs calling AMODE(31) programs, and vice versa. To have this implicit AMODE switching occur, you must use the LE/VSE run-time option, ALL31(OFF). AMODE switching is not performed when ALL31(ON) is specified. For details on the ALL31 run-time option, see the *LE/VSE Programming Reference*.

  When AMODE switching is performed, control is passed from the caller to an LE/VSE library routine. After the switching is performed, control is passed to the called program, and the library routine's save area will be positioned between the calling program's save area and the called program's save area.

- The program name to be called is not known until run time.

  – In this case, use the format CALL *identifier*, where *identifier* is a data item that will contain the name of the called program at run time. In terms of practical application, you might use CALL *identifier* when the program to be called is variable, depending on conditional processing in your program.

  – CALL *identifier* is always dynamic, even if you use the NODYNAM compiler option. To make all CALL *literal* calls in a program dynamic, use the compiler option DYNAM.

  When you use the NODYNAM option, do not mix a dynamic CALL *identifier* and a static CALL *literal* for the same subprogram. This wastes space because two copies of the subprogram are loaded into storage, and it does not guarantee that the subprogram will be left in its last-used state.

Performance Considerations of Static and Dynamic Calls

Because a statically called program is link-edited into the same phase 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 described above.

Statically called programs cannot be deleted (using CANCEL), so usage of 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. A statically called program is always loaded into storage, regardless of whether or not it is called; a dynamically called program is loaded only when it is called.
- You subsequently delete the dynamically called subprogram with a CANCEL statement.

  A statically called program cannot be deleted, but a dynamically called program can be deleted. 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.

CALL Statement Examples

A static CALL statement is illustrated in the following example:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RECORD-2                   PIC X.
01  RECORD-1.
  05  PAY                       PICTURE S9(5)V99.
  05  HOURLY-RATE              PICTURE S9V99.
  05  HOURS                     PICTURE S99V9.

PROCEDURE DIVISION.
     CALL "SUBPROG" USING RECORD-1.
     CALL "PAYMASTR" USING RECORD-1 RECORD-2.
     STOP RUN.
```

A dynamic CALL statement is illustrated in the following example:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  PGM-NAME                   PICTURE X(8).

01  RECORD-2                   PIC x.
01  RECORD-1.
  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.
     CANCEL PGM-NAME.
     MOVE "PAYMASTR" TO PGM-NAME.
     CALL PGM-NAME USING RECORD-1 RECORD-2.
     STOP RUN.
```

The following called subprogram is called by each of the two preceding calling programs:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUBPROG.
DATA DIVISION.
LINKAGE SECTION.
01  PAYREC.
  10  PAY                       PICTURE S9(5)V99.
  10  HOURLY-RATE               PICTURE S9V99.
  10  HOURS                     PICTURE S99V9.
77  PAY-CODE                   PICTURE 9.

PROCEDURE DIVISION USING PAYREC.
               .
               .
               .
      EXIT PROGRAM.

      ENTRY "PAYMASTR" USING PAYREC PAY-CODE.
               .
               .
               .
      GOBACK.
```

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.

When SUBPROG receives control, the values within RECORD-1 are made available to SUBPROG; however, in SUBPROG the structure is 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.

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.

Note: In a statically linked program, the CANCEL statement would not be valid.

In the example of a dynamically-linked program, two phases, PAYMASTR and SUBPROG, would need to be built, with entry points of PAYMASTR and SUBPROG respectively.

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. The values within RECORD-1 are again made available to SUBPROG. In addition, the value in RECORD-2 is now made available to SUBPROG through the corresponding USING operand PAY-CODE.

When processing reaches the GOBACK statement, control is returned to the calling program at the statement immediately following the second CALL statement.

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.

In any given execution of these two 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 the user wants to use the original values, they must be saved.

Subprogram Linkage

Called subprograms that are invoked at run time by the dynamic CALL statement must be members of the system phase sublibrary or of a user-supplied private sublibrary.

The static call statement results in the called subprogram being link-edited with the calling program into one phase.

Static and dynamic CALL statements can both be specified in the same program. The CALL *literal* statement results, in this case, in the subprogram invoked being link-edited with the calling program into one phase. The CALL *identifier* statement results in the dynamic invocation of a separate phase.

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 doesn't guarantee that the subprogram will be left in its last-used state, results can be unpredictable.

Converting Static Calls

You can convert static calls (CALL *literal* from a COBOL/VSE program that has been compiled with the NODYNAM option) to dynamic calls without recompiling your application program by using the IGZBRDGE assembler macro. This is useful, for example, if you would like an existing static COBOL/VSE program which

resides below the 16-megabyte line to call a new program which resides above the 16-megabyte line.

This support does not apply to the CICS environment.

The format of the macro is:

Format

```
modname IGZBRDGE ENTNMES=(name1,name2,...namen)
```

where:

modname
Is the name you give to the bridge object module

name1,name2,...namen
Are the names of the programs called by the static program. These names will be used as entry points in the assembled macro.

Assemble the macro and link-edit the resulting bridge object module with the object module containing the calling (static) program. (The INCLUDE statement for the bridge object module must precede the INCLUDE for the static calling program.) The linkage editor will resolve static calls in the calling (NODYNAM) program to the entry points in the bridge object module. This has the effect of removing the called programs from the phase.

When any COBOL/VSE program within the phase issues a static call to a program with an entry point in the bridge object module, control passes to the entry point in the bridge object module, and the appropriate program is **dynamically** loaded and executed.

For example, given the following programs and desired calling sequence:

```
COBOLA  (compiled NODYNAM)
  to call
    "COBOLB"  (compiled DYNAM)
      to call
        "COBOLC" (compiled NODYNAM)
          to call
            "COBOLD" (compiled NODYNAM)
              to call
                "COBOLE"
```

the macro would look like this:

```
MYNAME IGZBRDGE ENTNMES=(COBOLB,COBOLD)
```

In the example, `MYNAME` is the name for the bridge object module. Programs `COBOLB` and `COBOLD` are included in the macro because they are dynamically called from static programs. If you had not used the IGZBRDGE macro in this application, your phases would have looked like this:

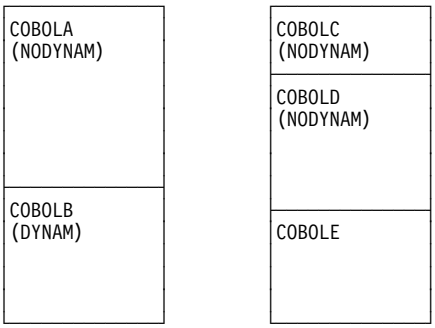


Figure 87. Example Application Using Static Calls

With the use of the IGZBRDGE macro, your phases will look like this:

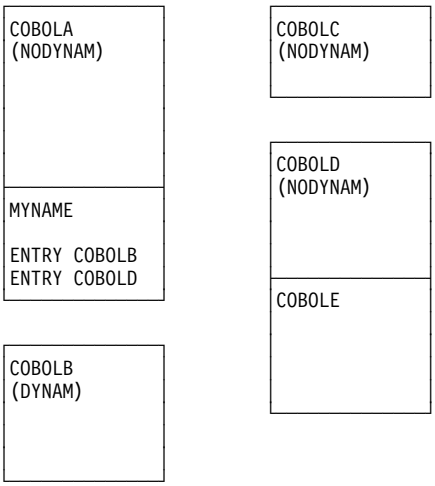


Figure 88. Example Application Using Static To Dynamic Call Conversion

If COBOLE had been included on the invocation of the IGZBRDGE macro, MYNAME would also have included an entry point for COBOLE, and COBOLE would not be loaded with COBOLD.

Note: Calls through the bridge may be made only by COBOL/VSE programs in a non-CICS environment.

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 to have access to common data.

This section discusses the manner in which programs can share data. For the purposes of this discussion, a “subprogram” is any program called by another program.

Passing Data BY REFERENCE or BY CONTENT

BY REFERENCE means that the subprogram is referring to and processing the data items in the calling program's storage, rather than working on a copy of the data.

BY CONTENT means that the calling program is passing only the **contents** of the *literal*, or *identifier*. With a CALL . . . BY CONTENT, the called program cannot change the value of the *literal* or *identifier* in the calling program, even if it modifies the variable in which it received the *literal* or *identifier*.

Whether you pass data items BY REFERENCE or BY CONTENT depends on what you want your program to do with the data:

- If you want the definition of the argument of the CALL statement in the calling program and the definition of the parameter in the called program to share the same memory, specify:

  ```
  CALL ... BY REFERENCE identifier.
  ```

  Any changes made by the subprogram to the parameter affects the argument in the calling program.

  An identifier in the USING phrase of the CALL . . . BY REFERENCE statement may be a file-name, in addition to a data-name. If the identifier is the file-name for a SAM file, the COBOL compiler passes the address of the Define The File (DTF) as this entry of the parameter list. The identifier may not be a VSAM file-name.

  File-names as CALL operands are allowed by the compiler as an extension. Any use of the extension generally depends on the specific internal implementation of the compiler. Control block field settings may change in future releases. Any changes made to the control block are the user's responsibility and not supported by IBM.

  Note: This mechanism cannot be used for file sharing between COBOL programs. This is only for passing DTFs to assembler programs. Use EXTERNAL or GLOBAL files to implement file sharing between COBOL programs.

- If you want to pass the address of a record area to a called program, specify:

  ```
  CALL ... BY REFERENCE ADDRESS OF record-name.
  ```

  The subprogram receives the ADDRESS special register for the record-name you specify.

  You must define the record-name as a level-01 or level-77 item in the Linkage Section of the called and calling programs. A separate ADDRESS special register is provided for each record in the Linkage Section.

- If you do not want the definition of the argument of the CALL statement in the calling program and the definition of the parameter in the called subprogram to share the same memory, specify:

  ```
  CALL ... BY CONTENT identifier.
  ```

- If you want to pass a literal value to a called program specify:

  ```
  CALL ... BY CONTENT literal.
  ```

  The called program cannot change the value of the literal.

- If you want to pass the length of a data item, specify:

  ```
  CALL ... BY CONTENT LENGTH OF identifier.
  ```

  The calling program passes the length of the *identifier* from its LENGTH special register. When literals are passed BY CONTENT, the called program cannot change the value.

- If you want to pass both a data item and its length to a subprogram, specify a combination of BY REFERENCE and BY CONTENT, for example:

```
CALL 'ERRPROC' USING BY REFERENCE A
                     BY CONTENT LENGTH OF A
```

Data items in a calling program can be described in the Linkage Section of all the programs it calls directly or indirectly. In this case, storage for these items is allocated in the highest calling program. That is, program A calls program B, which calls program C. Data items in program A can be described in the Linkage Sections of programs B and C, and the one set of data can be made available to all three programs.

Note: Do not pass parameters allocated in storage above 16 megabytes to AMODE(24) subprograms; use the DATA(24) option.

Describing Arguments in the Calling Program

In the calling program, the arguments are described in the Data Division in the same manner as other data items in the Data Division. Unless they are in the Linkage Section, storage is allocated for these items in the calling program. If you reference data in a file, the file must be open when the data is referenced. Code the USING clause of the CALL statement to pass the arguments.

Describing Parameters in the Called Program

In the called program, parameters are described in the Linkage Section. Code the USING clause after the PROCEDURE-DIVISION header to **receive** the parameters.

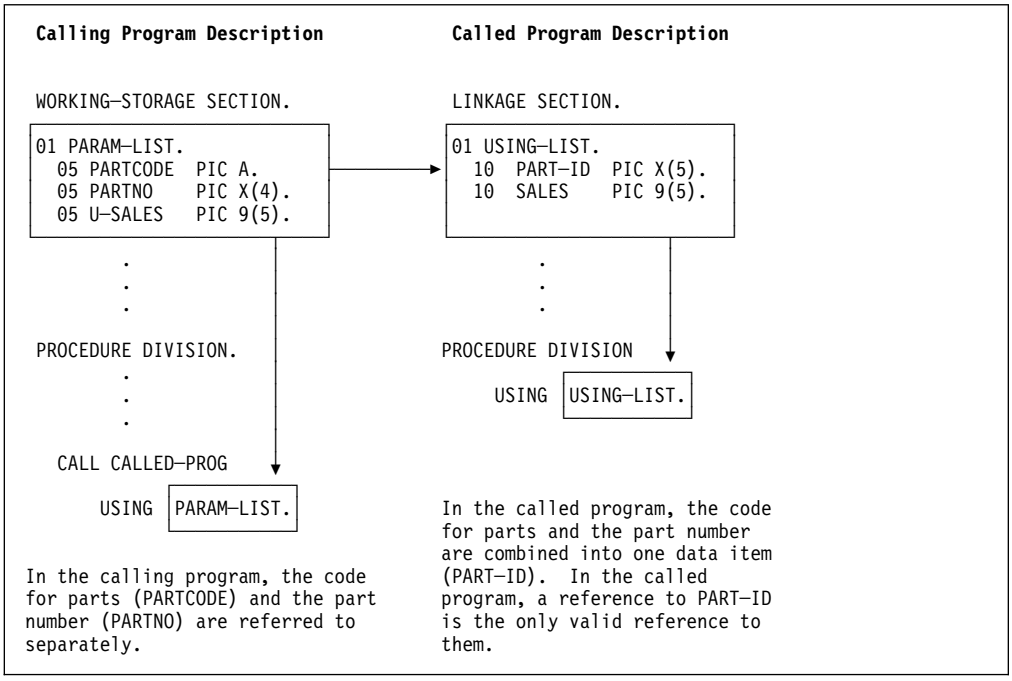


Figure 89. Common Data Items in Subprogram Linkage

Linkage Section

You must know what is being passed from the calling program and set up the Linkage Section in the called program to accept it. To the called program, it does not matter which clause of the CALL statement you use to pass the data (BY REFERENCE or BY CONTENT). In either case, the called program must describe the data it is receiving. It does this in the Linkage Section.

The number of *data-names* in the *identifier* list of a called program must not be greater than the number of *data-names* in the *identifier* list of the calling program. There is a one-to-one positional correspondence; that is, the first *identifier* of the calling program is passed to the first *identifier* of the called program, and so forth. The compiler makes no attempt to match arguments and parameters.

Grouping Data to Be Passed

Consider grouping all the data items you want to pass between programs and putting them under one level-01 item. If you do this, you can pass a single level-01 record between programs. For an example of this method, see Figure 89 on page 275.

To make the possibility of mismatched records even smaller, put the level-01 record in a copy library, and copy it in both programs. (That is, copy it in the Working-Storage Section of the calling program and in the Linkage Section of the called program.)

Using Pointers to Process a Chained List

You can manipulate pointer data items, which are a special type of data item to hold addresses, when you want to pass and receive addresses of a variably located record area. Pointer data items are data items that are either explicitly defined with the USAGE IS POINTER clause, or are ADDRESS special registers. A typical application for using pointer data items is in processing a chained list (a series of records where each one points to the next).

For this example, picture a chained list of data that contains individual salary records. Figure 90 shows one way to visualize how these records are linked in storage:

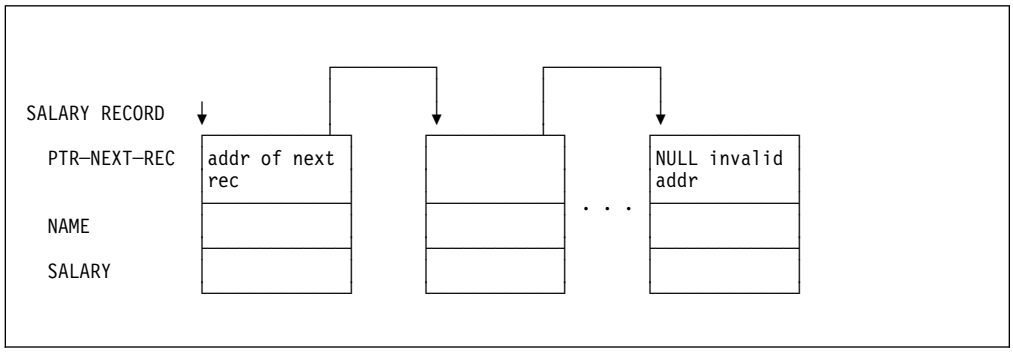


Figure 90. Representation of a Chained List Ending with NULL

The first item in each record points to the next record, except for the last record. The first item in the last record, in order to indicate that it is the last record, contains a null value instead of an address.

The high-level logic of an application that processes these records might look something like this:

```
OBTAIN ADDRESS OF FIRST RECORD IN CHAINED LIST FROM ROUTINE
CHECK FOR END OF THE CHAINED LIST
DO UNTIL END OF THE CHAINED LIST
   PROCESS RECORD
   GO ON TO THE NEXT RECORD
END
```

Figure 91 contains an outline of the processing 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.
 77  DEPT-TOTAL        PIC 9(4) VALUE IS 0.
******
 LINKAGE SECTION.
 01  SALARY-REC.
   02  PTR-NEXT-REC    POINTER.
   02  NAME            PIC X(20).
   02  DEPT            PIC 9(4).
   02  SALARY          PIC 9(6).
 01  DEPT-X            PIC 9(4).
******
 PROCEDURE DIVISION USING DEPT-X.
******
* FOR EVERYONE IN THE DEPARTMENT RECEIVED AS DEPT-X,
* GO THROUGH ALL THE RECORDS IN THE CHAINED LIST BASED ON THE
* ADDRESS OBTAINED FROM THE PROGRAM CHAIN-ANCH
* AND CUMULATE THE SALARIES.
* IN EACH RECORD, PTR-NEXT-REC IS A POINTER TO THE NEXT RECORD
* IN THE LIST; IN THE LAST RECORD, PTR-NEXT-REC IS NULL.
* DISPLAY THE TOTAL.
******
     CALL "CHAIN-ANCH" USING PTR-FIRST
     SET ADDRESS OF SALARY-REC TO PTR-FIRST
******
     PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL
      IF DEPT = DEPT-X
        THEN ADD SALARY TO DEPT-TOTAL
        ELSE CONTINUE
      END-IF
      SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC
     END-PERFORM
******
     DISPLAY DEPT-TOTAL
     GOBACK.
```

Figure 91. Program for Processing a Chained List

Passing Addresses between Programs

To obtain the address of the first SALARY-REC record area, program LISTS calls program CHAIN-ANCH:

```
CALL "CHAIN-ANCH" USING PTR-FIRST
```

PTR-FIRST is defined in WORKING-STORAGE in the calling program (LISTS) as a pointer data item:

```
WORKING-STORAGE SECTION.
01  PTR-FIRST         POINTER  VALUE IS NULL.
```

Upon return from the call to CHAIN-ANCH, PTR-FIRST contains the address of the first record in the chained list.

PTR-FIRST is initially defined as having a NULL value as a logic check. If something goes amiss with the call, and PTR-FIRST never receives the value of the address of the first record in the chain, a NULL value remains in PTR-FIRST and, according to the logic of the program, the records will not be processed.

NULL is a figurative constant used to assign the value of an invalid address (nonnumeric 0) to pointer items. It can be used in the VALUE IS NULL clause, in the SET statement, and as one of the operands of a relation condition with a pointer data item.

The LINKAGE SECTION of the calling program contains the description of the records in the chained list. It also contains the description of the department code that is passed, using the USING clause of the CALL statement.

```
LINKAGE SECTION.
01  SALARY-REC.
    02  PTR-NEXT-REC    POINTER.
    02  NAME            PIC X(20).
    02  DEPT            PIC 9(4).
    02  SALARY          PIC 9(6).
01  DEPT-X            PIC 9(4).
```

To “base” the record description SALARY-REC on the address contained in PTR-FIRST, use a SET statement:

```
CALL "CHAIN-ANCH" USING PTR-FIRST
SET ADDRESS OF SALARY-REC TO PTR-FIRST
```

Checking for the End of the Chained List

The chained list in this example is set up so the last record contains an invalid address. To do this, the pointer data item in the last record would be assigned the value NULL.

A pointer data item can be assigned the value NULL in two ways:

- A pointer data item can be defined with a VALUE IS NULL clause in its data definition.
- NULL can be the sending field in a SET statement.

In the case of a chained list in which the pointer data item in the last record contains a NULL value, the code to check for the end of the list would be:

```
IF PTR-NEXT-REC = NULL
   ⋮

(logic for end of chain)
```

If you have not reached the end of the list, process the record and move on to the next record.

In the program LISTS, this check for the end of the chained list is accomplished with a “DO WHILE” structure:

```
PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL
  IF DEPT = DEPT-X
    THEN ADD SALARY TO DEPT-TOTAL
    ELSE CONTINUE
  END-IF
  SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC
END-PERFORM
```

Continuing Processing the Next Record

To move on to the next record, set the address of the record in the LINKAGE-SECTION to be equal to the address of the next record. This is accomplished through the pointer data item sent as the first field in SALARY-REC:

```
SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC
```

Then repeat the record-processing routine, which will process the next record in the chained list.

A Variation: Incrementing Addresses Received from Another Program

The data passed from a calling program might contain header information that you want to ignore (for example, in data received from a CICS application that is not migrated to the command level).

Because pointer data items are not numeric, you cannot directly perform arithmetic on them. However, you can use the SET verb to increment the passed address in order to bypass header information.

You could set up the LINKAGE-SECTION like this:

```
LINKAGE SECTION.
01  RECORD-A.
  02  HEADER          PIC X(12).
  02  REAL-SALARY-REC PIC X(30).
⋮
01  SALARY-REC.
  02  PTR-NEXT-REC    POINTER.
  02  NAME            PIC X(20).
  02  DEPT            PIC 9(4).
  02  SALARY          PIC 9(6).
```

Within the Procedure Division, “base” the address of SALARY-REC on the address of REAL-SALARY-REC:

```
SET ADDRESS OF SALARY-REC TO ADDRESS OF REAL-SALARY-REC
```

SALARY-REC is now based on the address of RECORD-A + 12.

Passing Entry Point Addresses with Procedure Pointers

You can use procedure pointers, data items defined with the USAGE IS PROCEDURE-POINTER clause, to pass the entry address of a procedure entry point in a format required by certain LE/VSE callable services.

For example, to have a user-written error handling routine take control when an exception condition occurs during program execution, you must first pass the entry address of the routine to CEEHDLER, a condition management LE/VSE callable service, to have it registered.

Procedure-pointer data items can be set to contain the entry addresses for these entry points:

- Another non-nested COBOL program
- An alternate entry point in another COBOL program (as defined in an ENTRY statement)
- A program written in another language

A procedure-pointer data item can only be set using Format 6 of the SET statement; an LE/VSE entry variable is created for the literal or identifier specified and moved into the procedure-pointer data item. This form of the SET statement (Format 6) causes the program to be loaded according to the rules of the DYNAM/NODYNAM compiler option. Therefore, consider these factors when using procedure-pointer data items:

- If you compile your program with the NODYNAM option and you set your procedure-pointer item to a literal value (to an actual name of an entry point), then the value must refer to an entry point in the same phase as your program. (Otherwise the reference cannot be resolved.)
- If you compile your program with the DYNAM option, or if you set your procedure-pointer item to a variable that will contain the entry point at run time, then your procedure-pointer item, whether a literal or variable, must point to an entry point in a separate phase.
- If you set your procedure-pointer item to an entry address in a dynamically called phase and your program subsequently CANCELs that dynamically called phase, then your procedure-pointer item becomes undefined, and reference to it thereafter is not reliable.

For a complete definition of the USAGE IS PROCEDURE-POINTER clause and the SET statement, refer to the *COBOL/VSE Language Reference*.

Passing Return Code Information (RETURN-CODE Special Register)

You can use the RETURN-CODE special register to pass and receive return codes between programs.

When a COBOL/VSE 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 from a call, the contents of register 15 are stored into the calling program's RETURN-CODE special register. When control is returned from a COBOL/VSE program to the operating system, the special register contents are returned as a user return code.

You may need to take this treatment of the RETURN-CODE into consideration when control is returned to a COBOL/VSE program from a non-COBOL program. If the non-COBOL program does not use register 15 to pass back the return code, then the COBOL/VSE program's RETURN-CODE special register may be updated with an invalid value. Unless you set this special register to a meaningful value before your COBOL/VSE program returns to the operating system, an invalid return code will be passed back to the system.

Sharing Data Using the EXTERNAL Clause

Separately compiled programs (including programs within a batch sequence) may share data items by use of the EXTERNAL clause.

EXTERNAL is specified on the 01-level data description in the Working-Storage Section of a program, and the following rules apply:

1. Items subordinate to an EXTERNAL group item are themselves EXTERNAL.
2. The name used for the data item cannot be used on another EXTERNAL item within the same program.
3. The VALUE clause cannot be specified for any group item, or subordinate item, that is EXTERNAL.

Any COBOL program within the run unit, having the same data description for the item as the program containing the item, can access and process the data item. For example, if program A had the following data description:

```
01 EXT-ITEM1                PIC 99 EXTERNAL.
```

program B could access that data item by having the identical data description in its Working-Storage Section.

Remember, any program that has access to an EXTERNAL data item can change its value. Do not use this clause for data items you need to protect.

Sharing Files between Programs (EXTERNAL Files)

Using the EXTERNAL clause for files allows separately compiled programs within the run unit to have access to common files. The example on page 283, shows some of the advantages of using EXTERNAL files:

- The main program can reference the record area of the file, even though the main program does not contain any I/O statements.
- Each subprogram can control a single I/O function, such as OPEN, or READ.
- Each program has access to the file.

The following table gives the program (or subprogram) name for the example in Figure 93 on page 283 and describes its function.

Figure 92 (Page 1 of 2). Program Names for Input-Output Using EXTERNAL Files

Name	Function
`ef1`	This is the main program. It calls all the subprograms, and then verifies the contents of a record area.
`ef1openo`	This program opens the external file for output and checks the File Status Code.

Figure 92 (Page 2 of 2). Program Names for Input-Output Using EXTERNAL Files

Name	Function
ef1write	This program writes a record to the external file and checks the File Status Code.
ef1openi	This program opens the external file for input and checks the File Status Code.
ef1read	This program reads a record from the external file and checks the File Status Code.
ef1close	This program closes the external file and checks the File Status Code.

Additionally, COPY statements ensure that each subprogram contains an identical description of the file.

The sample program also uses the EXTERNAL clause for a data item in the Working-Storage Section. This item is used for checking File Status codes, and is also placed using the COPY statement.

The program uses three Copy Library members:

- The first is named efselect and is placed in the FILE-CONTROL paragraph.

```
Select ef1
   Assign To ef1
   File Status Is efs1
   Organization Is Sequential.
```

- The second is named effile and is placed in the File Section.

```
Fd ef1 Is External
          Record Contains 80 Characters
          Recording Mode F.
1 ef-record-1.
 2 ef-item-1 Pic X(80).
```

- The third is named efwrkstg and is placed in the Working-Storage Section.

```
1 efs1 Pic 99 External.
```

```
       Identification Division.
       Program-ID.
           ef1.
      *
      * This is the main program that controls the external file
      * processing.
      *
       Environment Division.
       Input-Output Section.
       File-Control.
           Copy efselect.
       Data Division.
       File Section.
           Copy effile.
       Working-Storage Section.
           Copy efwrkstg.
       Procedure Division.
           Call "ef1openo"
           Call "ef1write"
           Call "ef1close"
           Call "ef1openi"
           Call "ef1read"
           If ef-record-1 = "First record" Then
             Display "First record correct"
           Else
             Display "First record incorrect"
             Display "Expected: " "First record"
             Display "Found   : " ef-record-1
           End-If
           Call "ef1close"
           Goback.
       End Program ef1.
       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.
```

Figure 93 (Part 1 of 3). Input-Output Using EXTERNAL Files

```
       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.
```

Figure 93 (Part 2 of 3). Input-Output Using EXTERNAL Files

```
       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.
```

Figure 93 (Part 3 of 3). Input-Output Using EXTERNAL Files

Reentrant Programs

If you intend to have multiple users execute an application program at the same time, you must make your program **reentrant** by specifying the RENT option when you compile your program. (See page 247 for more information on the RENT option.) As a programmer, you do not need to worry about multiple copies of variables. The compiler creates the necessary reentrancy controls in your object module.

The following COBOL/VSE programs must be reentrant:

Programs to be used with CICS
Programs to be executed above 16 megabytes
Programs to be loaded and deleted by a non-COBOL program

For reentrant programs, the DATA(24|31) compiler option and the HEAP and ALL31 run-time options control whether dynamic data areas, such as Working-Storage, are obtained from storage below 16 megabytes or from unrestricted storage. For details on how to control where the storage is allocated from, see the description of the DATA compile-time option on page 230.

Calls to Alternative Entry Points

Static calls to alternative entry points work without restriction. Under VSE you cannot dynamically call alternate entry points because ALIASes are not supported. Only the entry point for the phase can be called dynamically.

Chapter 17. Interrupts and Checkpoint/Restart

The Checkpoint/Restart feature is designed to be used with programs running for an extended period of time when interruptions may halt processing before the end of the job. Because the Checkpoint/Restart feature causes a lot of extra processing, use it only when you anticipate interruptions caused by machine malfunctions, input/output errors, or intentional operator intervention. Checkpoint/Restart allows the interrupted program to be restarted at the job step or at a point other than the beginning of the job step.

The **checkpoint routine** is invoked from the COBOL phase containing your program. During execution of your program, the checkpoint routine creates checkpoint records at points you specify in your program. The checkpoint record contains information stored in registers and main storage when the program reached the checkpoint. You specify these checkpoints by using the COBOL RERUN clause in the Environment Division.

The **restart routine** restarts an interrupted program. Restart occurs at a specified checkpoint. The checkpoint record contains all information necessary to restart the program. Restart can be initiated at any time after the program has been interrupted, as long as the file containing the checkpoint records is available.

The COBOL RERUN clause provides linkage to the system checkpoint routine. Any cautions and restrictions on the use of the system Checkpoint/Restart feature also apply to the use of the RERUN clause during the execution of a SORT or MERGE statement. If restart is attempted with 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 Sort/Merge II and DFSORT/VSE are valid. For further information, see “Checkpoint/Restart During DFSORT/VSE” on page 190.

The Checkpoint/Restart feature is fully described in *VSE/ESA System Macros Reference* and *VSE/ESA Guide to System Functions*.

Getting a Checkpoint

To get a checkpoint, use job control statements and the RERUN EVERY *integer-1* RECORDS clause. Associate each RERUN clause with a particular COBOL file. The RERUN clause indicates that a checkpoint record is to be written onto a checkpoint file whenever a specified number of records on that file have been processed. The VSE Checkpoint/Restart feature does not provide support for the RERUN EVERY END REEL/UNIT clause. If you code this clause in your program, it will be treated as a comment.

The checkpoint records are written to the checkpoint file defined by your JCL. Checkpoints are recorded and numbered sequentially.

Designing a Checkpoint

Design your checkpoints at critical points in your program so that data can be easily reconstructed. Ensure that the contents of files are not changed between the time of the checkpoint and the time of the restart. This can be a problem with nonsequential files.

In a program using disk files, changes to records in these files will replace previous information. Design your program so that you can identify previously processed records. For example, consider a disk file containing loan records that are periodically updated for interest due. If a checkpoint is taken, records are updated, and then the program is interrupted; the program design should include a test so that the records updated after the last checkpoint will not be updated a second time when the program is restarted. To do this, you can set up a date field for each record and update the date each time the record is processed. Then, after the restart, test the date field to determine whether or not the record was previously processed.

For efficient repositioning of a print file, take checkpoints on that file only after printing the last line of a page.

The Checkpoint File

To define checkpoint files, specify the following JCL statements.

For Tape

```
// ASSGN SYSnnn,tape-unit
```

For Direct-Access Devices

```
// DLBL   system-name,'file-ID',0,SD
// EXTENT SYSnnn,volser,1,0,start,tracks
// ASSGN  SYSnnn,DISK,VOL=volser,SHR
```

SYS*nnn*
The same as the SYS*nnn* portion of the *assignment-name* used in the COBOL RERUN clause.

tape-unit
Identifies the magnetic tape unit.

system-name
The same as the *system-name* portion of the *assignment-name* used in the COBOL RERUN clause.

file-ID
The name given to the file used to record checkpoints, when the file resides on a direct-access device. This name identifies the checkpoint file to the restart procedure (see “Restarting a Program” on page 290).

volser
Identifies the direct-access volume by serial number.

start, tracks
Specifies the amount of track space needed for the checkpoint file on a direct-access device.

The following are examples that define checkpoint files.

Note: Checkpoint records for several COBOL files can be written into one checkpoint file.

```
// ASSGN  SYS000,TAPE
          ⋮
       ENVIRONMENT DIVISION.
           ⋮
           RERUN ON SYS000-S-CHKPT EVERY
                 5000 RECORDS OF ACCT-FILE.
```

Figure 94. Write Checkpoint Records, Using Tape

```
// DLBL   CHEK,'CHECK2',0,SD
// EXTENT SYS030,DB0003,1,0,3000,300
// ASSGN  SYS030,DISK,VOL=DB0003,SHR
          ⋮
       ENVIRONMENT DIVISION.
           ⋮
           RERUN ON CHEK EVERY
                 20000 RECORDS OF PAYCODE.
           RERUN ON SYS030-S-CHEK EVERY
                 30000 RECORDS OF IN-FILE.
```

Figure 95. Write Checkpoint Records, Using Disk

Restrictions

1. VSE/ESA does not support checkpoints being taken by programs executing in a partition that is larger than 16 megabytes, or in dynamic partitions
2. A checkpoint file must have sequential organization
3. Checkpoints cannot be written to VSAM files
4. Checkpoints cannot be written to 3540 Diskette files
5. Magnetic tape files used to record checkpoints must be unlabeled
6. Checkpoint records cannot be embedded in one of your program files. You must use a separate file exclusively for checkpoint records
7. Checkpoints during sort operations:

   If checkpoints are to be taken on a direct-access device during a sort operation, add a DLBL statement for SORTCKP in the job control for execution.

   If checkpoints are to be taken on a magnetic tape during a sort operation, add an ASSGN statement for SYS000 in the job control for execution.

   Checkpoint records on ASCII-collated sorts can be taken, but the *assignment-name* indicating the checkpoint file must not specify an ASCII 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 has been successfully completed, the following message is displayed:

0C00I CHKPT NO. *nnnn* WAS TAKEN ON *SYSnnn=cuu*

nnnn
is the 4-digit number which identifies the checkpoint.

SYSnnn
is the logical unit number and assigned to the device on which checkpoints are recorded.

cuu
is the physical address of the device on which checkpoints are recorded.

Restarting a Program

The system restart routine:

Retrieves the information recorded in a checkpoint record
Restores the contents of main storage and all registers
Repositions tape files
Restarts the program

In order to restart a program from a checkpoint, the RSTRT job control statement is used. To restart a program, you must do the following:

1. Have the operator rewind all magnetic tape files that were being used by the program when the checkpoint was recorded. If a multi-volume magnetic tape file was being used, have the operator mount, on the primary unit, the volume of the file that was in use at the time of the checkpoint.
2. Use the same JCL statements to restart the program as you used to run when the checkpoint was recorded. In place of the EXEC JCL statement, code a RSTRT JCL statement, specifying:
   - The logical unit number of the direct-access device or magnetic tape device on which the checkpoint records are recorded
   - The sequence number of the checkpoint to be used for restart
   - The system-name of the checkpoint file, if the checkpoint records were recorded on a direct-access device

When resubmitting a job, care should be taken that the program is restarted in the same partition in which it was executing when it was interrupted.

Sample Job Control Procedures for Checkpoint/Restart

Figure 96 illustrates a sequence of job control statements for restarting a job.

```
// JOB   CHECKP
// ASSGN SYS006,380           CHKPT TAPE
// DLBL  FILE1 ...
// DLBL  FILE2 ...
// ASSGN ...
// RSTRT SYS006,0013
/&
```

Figure 96. Restarting a Job at a Specific Checkpoint

The following are the characteristics of the checkpointed program that must be considered for the restart:

- The job name specified in the JOB statement was CHECKP; the same name must be used for restart.
- The checkpoint records were recorded on magnetic tape; therefore, no system-name needs to be specified in the RSTRT statement.
- The logical unit number SYS006 is used for the checkpoint file.
- The sequence number of the last checkpoint record was 0013; this or any previous checkpoint can be used for the restart.

Chapter 18. Debugging

COBOL/VSE provides several language elements and facilities to help you determine the cause of problems in program behavior. This chapter focuses on how to use source language for debugging and describes some alternative compiler options that enhance debugging.

If the problem with your program is not easily detected, you might need to analyze a storage dump of your program. See the *LE/VSE Debugging Guide and Run-Time Messages* for information on interpreting dumps. This book also contains examples demonstrating how to debug COBOL/VSE programs.

Using Source Language to Debug

You can use several COBOL language features to pinpoint the cause of a failure in your program. If the program is part of a large application already in production, you will not want to recompile and run the program again to debug. Instead, you can write a small test case to simulate the part of the program that failed and code some of these debugging features of the COBOL language in the test case to help detect the exact cause of the problem:

- DISPLAY statements
- USE EXCEPTION/ERROR declaratives
- Class Test
- INITIALIZE or SET Verbs
- Scope terminators
- File status keys
- USE FOR DEBUGGING declaratives

The rules for using each of these language features are explained in *COBOL/VSE Language Reference*.

Tracing Program Logic (DISPLAY Statements)

Adding DISPLAY statements can help you trace through the logic of the program. If, for example, you determine that the problem appears in an EVALUATE statement or in a set of nested IF statements, DISPLAY statements in each path will show you how the logic flow is working. If you determine that the problem is being caused by the way a numeric value is calculated, you can use DISPLAY statements to check the value of some of the interim results.

For example, you might insert code like this into your program:

```
DISPLAY "ENTER CHECK PROCEDURE"
    .
    .  (checking procedure routine)
    .
DISPLAY "FINISHED CHECK PROCEDURE"
```

to determine whether a particular routine started and finished. When you are sure that the routine works correctly, you can put asterisks in column 7 of the DISPLAY statement lines, which converts them to comment lines. Alternatively, you might put a 'D' in column 7 of your DISPLAY (or any other debugging) statements. If you include the WITH DEBUGGING MODE clause in the ENVIRONMENT DIVISION, the 'D' in column 7 will be ignored and the DISPLAY statements will be imple-

mented. Without the DEBUGGING MODE clause, the 'D' in column 7 makes the statement a comment.

Before you put the program into production, delete all the debugging aids you used and recompile the program. The program will run more efficiently and use less storage.

Note: The DISPLAY statement cannot be used in programs running under CICS.

Handling Input/Output Errors (USE EXCEPTION/ERROR Declaratives)

If you have determined that the problem lies in one of the I/O procedures in your program, you can include the USE EXCEPTION/ERROR declarative to help debug the problem.

If a file fails to open for some reason, the appropriate EXCEPTION/ERROR declarative will be activated. The appropriate declarative may be a specific one for the file or one specified for the different open attributes—INPUT, OUTPUT, I/O, or EXTEND.

Each USE AFTER STANDARD ERROR statement must be coded in a separate section. Each of these sections must be coded immediately after the Declarative Section keyword of the Procedure Division. The rules for coding the statements are provided in *COBOL/VSE Language Reference*.

Validating Data (Class Test)

If you suspect that your program is trying to perform arithmetic on nonnumeric data or is somehow receiving the wrong type of data on an input record, you can use the class test to validate the type of data. The class test checks whether data is alphabetic, alphabetic-lower, alphabetic-upper, DBCS, KANJI, or numeric.

Assessing Switch Problems (INITIALIZE or SET Statements)

Using INITIALIZE or SET statements to initialize a table or variable is useful when you suspect that the problem may be caused by residual data left in those fields. If your problem occurs intermittently and not always with the same data, the problem could be that a switch is not initialized but generally is set to the right value (0 or 1) by accident. By including a SET statement to ensure that the switch is initialized, you can either determine that the noninitialized switch is the problem or eliminate that as a possible cause.

Improving Program Readability (Explicit Scope Terminators)

Scope terminators can help you in debugging because they indicate clearly the end of a statement. The logic of your program will become more apparent, and therefore easier to trace, if you use scope terminators.

Finding Input/Output Errors (File Status Keys)

File status keys can help you determine if your program errors are due to the logic of your program or if they are I/O errors occurring on a storage media.

To use file status keys as a debugging aid, include a test after each I/O statement to check for a value other than zero in the status key. If the value is other than zero, you can expect that you will receive an error message. You can use a nonzero value as an indication that you should look at the way the I/O procedures

in the program were coded. You can also include procedures to correct the error based on the value of the status key.

The status key values and their associated meanings are described in Chapter 13, “Error Handling” on page 192.

Generating Information about Procedures (USE FOR DEBUGGING Declaratives)

The use of USE FOR DEBUGGING declaratives is another way to generate information about your program or test case and how it is executing. The declarative allows you to include statements in the program and specify when they should be executed when you run your compiled program. For example, if you want to check how many times a procedure is executed, you could include a debugging procedure in the USE FOR DEBUGGING declarative and use a counter to keep track of the number of times control passes to that procedure.

Each USE FOR DEBUGGING declarative must be coded in a separate section. This section, or these sections, must be coded in the Declaratives Section of the Procedure Division. The rules for coding them are provided in *COBOL/VSE Language Reference*.

You can have either debugging lines or debugging statements or both in your program. Debugging lines are statements within your program and are identified by a 'D' in column 7. Debugging statements are the statements coded in the Declaratives Section of the Procedure Division.

- The debugging statements in a USE FOR DEBUGGING declarative must:
  - Be only in the Declarative Section
  - Follow the header USE FOR DEBUGGING
- Debugging lines must:
  - Have a 'D' in column 7 to identify them

To use debugging lines and sections in your program, you must include both:

1. WITH DEBUGGING MODE on the SOURCE-COMPUTER line in the Environment Division
2. The DEBUG parameter on the EXEC statement of your JCL

Note: Remember that the TEST compiler option (with any suboption value other than NONE) and the run-time option DEBUG are mutually exclusive, with DEBUG taking precedence.

The example in Figure 97 on page 295 shows portions of a program to illustrate what kind of statements are needed to use a DISPLAY statement and a USE FOR DEBUGGING declarative to test a program. The DISPLAY statement is used to generate information on the output file. The USE FOR DEBUGGING declarative is used in conjunction with using a counter to show how many times a routine was actually executed.

Note: The adding-to-a-counter technique can be used to check:

- How many times a PERFORM was executed. You will know whether a particular routine is being used and whether the control structure you are using is correct.

- How many times a loop routine actually executes. This will tell you whether the loop is executing and whether the number you have used for the loop is accurate.

```
Environment Division
Source-Computer. IBM-370 With Debugging Mode.
   .
   .
   .
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     Value 1.
   .
   .
   .
Procedure Division.
Declaratives.
Debug-Declaratives Section.
    Use For Debugging On Some-Routine.
Debug-Declaratives-Paragraph.
    Display Trace-Msg, Debug-Name, Total.
End Declaratives.

Main-Program Section.
   .
   . (source program statements)
   .
  Perform Some-Routine.
   .
   . (source program statements)
   .
  Stop Run.

Some-Routine.
   .
   . (whatever statements you need in this paragraph)
   .
  Add 1 To Total.
Some-Routine-End
```

Figure 97. Example of Using the WITH DEBUGGING MODE Clause

In Figure 97, the DISPLAY statement specified in the Declaratives Section will issue this message:

```
Trace For Procedure-Name : Some-Routine 22
```

every time the procedure SOME-ROUTINE is executed. The number at the end of the message, 22, is the value accumulated in the data item, TOTAL; it shows the number of times SOME-ROUTINE has been executed. The statements in the debugging declarative are executed before the execution of the named procedure.

Note: Debugging sections are allowed only in the outermost program; they are not valid in nested programs. Debugging sections are also never triggered by procedures contained in nested programs.

You can also use the DISPLAY statement technique shown above to trace program execution and show the flow through your program. You do this by changing the USE FOR DEBUGGING declarative in the Declaratives Section to:

```
USE FOR DEBUGGING ON ALL PROCEDURES.
```

and dropping the word TOTAL from the DISPLAY statement. Now, a message will be displayed before each execution of every nondebugging procedure in the outermost program.

Using Compiler Options for Debugging

This section discusses the various compiler options that generate information to help you find coding mistakes and other errors in your program.

The FLAG Option

The FLAG option lets you select the level of error to be diagnosed during compilation and specify where the syntax-error messages appear in the listing. You should specify FLAG (I) or FLAG (I,I) to be notified of all errors in your program.

You specify in the **first parameter** the lowest severity level of the syntax-error messages to be issued. You can specify in the **second parameter**, which is optional, the lowest level of the syntax messages to be embedded in the source listing.

If you specify:

- I (informational) - you get all the messages. I-level messages generate a return code of zero; RC=0.
- W (warning) - you get all the warning messages and those of a higher severity. W-level errors generate a return code of four; RC=4.
- E (error) - you get all error messages and those of a higher severity. E-level errors generate a return code of eight; RC=8.
- S (severe) - you get all severe and U (unrecoverable) messages. S-level errors generate a return code of twelve; RC=12.
- U (unrecoverable) - you get only unrecoverable messages. U-level errors generate a return code of sixteen; RC=16.

When you specify the second parameter, the syntax-error messages are embedded in the source listing at the point where the compiler had enough information available to detect the error. All embedded messages, except those issued by the library compiler phase, will directly follow the statement to which they refer. The number of the statement containing the error is also included with the message. Embedded messages are repeated with the rest of the diagnostic messages following the source listing.

See Figure 98 on page 297 for an illustration of embedded messages and messages that appear in the source listing.

Notes:

1. If NOSOURCE is one of your options, the syntax-error messages are included only in the list at the end of the listing.
2. U-level errors are not embedded in the source listing, as an error of this severity terminates the compilation.

```
DATA VALIDATION AND UPDATE PROGRAM                          IGYCARPA  Date 06/16/1998  Time 13:41:27   Page    26
  LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Map and Cross Reference

   ⋮
  000971              ****************************************************************
  000972              ***          I N I T I A L I Z E   P A R A G R A P H          **
  000973              ***  Open files. Accept date, time and format header lines.    **
  000974              ***   Load location-table.                                     **
  000975              ****************************************************************
  000976                 100-initialize-paragraph.
  000977                   move spaces to ws-transaction-record
  000978                   move spaces to ws-commuter-record
  000979                   move zeros to commuter-zipcode
  000980                   move zeros to commuter-home-phone
  000981                   move zeros to commuter-work-phone
  000982                   move zeros to commuter-update-date
  000983                   open input update-transaction-file

==000983==> IGYPS2052-S An error was found in the definition of file "LOCATION-FILE". The reference to this file was discarded.

  000984                       location-file
  000985                       i-o commuter-file
  000986                       output print-file
  000987                   if commuter-file-status not = "00" and not = "97"
  000988      1              display "100-OPEN"
  000989      1              move 100 to comp-code
  000990      1              perform 500-vsam-error
  000991      1              perform 900-abnormal-termination
  000992                   end-if
  000993                   accept ws-date from date

==000993==> IGYPS2121-S "WS-DATE" was not defined as a data-name. The statement was discarded.

  000994                   move corr ws-date to header-date

==000994==> IGYPS2121-S "WS-DATE" was not defined as a data-name. The statement was discarded.

  000995                   accept ws-time from time

==000995==> IGYPS2121-S "WS-TIME" was not defined as a data-name. The statement was discarded.

  000996                   move corr ws-time to header-time

==000996==> IGYPS2121-S "WS-TIME" was not defined as a data-name. The statement was discarded.

  000997                   read location-file

==000997==> IGYPS2053-S An error was found in the definition of file "LOCATION-FILE". This input/output statement was discarded.

  000998                     at end
  000999      1                  set location-eof to true
  001000                   end-read
```

Embedded syntax message in the source listing.

Figure 98 (Part 1 of 2). FLAG(I,I) Output

```
DATA VALIDATION AND UPDATE PROGRAM                              IGYCARPA  Date 06/16/1998  Time 13:41:27   Page    50
LineID  Message code  Message text

   193  IGYDS1050-E   File "LOCATION-FILE" contained no data record descriptions. The file definition was discarded.

   889  IGYPS2052-S   An error was found in the definition of file "LOCATION-FILE". The reference to this file was discarded.

                      Same message on line:    983

   993  IGYPS2121-S   "WS-DATE" was not defined as a data-name. The statement was discarded.

                      Same message on line:    994

   995  IGYPS2121-S   "WS-TIME" was not defined as a data-name. The statement was discarded.

                      Same message on line:    996

   997  IGYPS2053-S   An error was found in the definition of file "LOCATION-FILE". This input/output statement was discarded.

                      Same message on line:   1009

  1008  IGYPS2121-S   "LOC-CODE" was not defined as a data-name. The statement was discarded.

  1219  IGYPS2121-S   "COMMUTER-SHIFT" was not defined as a data-name. The statement was discarded.

                      Same message on line:   1240

  1220  IGYPS2121-S   "COMMUTER-HOME-CODE" was not defined as a data-name. The statement was discarded.

                      Same message on line:   1241

  1222  IGYPS2121-S   "COMMUTER-NAME" was not defined as a data-name. The statement was discarded.

                      Same message on line:   1243

  1223  IGYPS2121-S   "COMMUTER-INITIALS" was not defined as a data-name. The statement was discarded.

                      Same message on line:   1244

  1233  IGYPS2121-S   "WS-NUMERIC-DATE" was not defined as a data-name. The statement was discarded.
Messages    Total    Informational    Warning    Error    Severe    Terminating
Printed:      19                                   1        18
* Statistics for COBOL program IGYCARPA:
*    Source records = 1765
*    Data Division statements = 277
*    Procedure Division statements = 513
End of compilation 1,  program IGYCARPA,  highest severity 12.
Return code 12

Some messages apply to more than one COBOL statement.
```

Figure 98 (Part 2 of 2). FLAG(I,I) Output

The NOCOMPILE Option

Use this option to produce a listing that will help you find your COBOL coding mistakes, such as missing definitions, improperly defined data names, and duplicate data names. You can use NOCOMPILE with or without parameters.

Using NOCOMPILE with Parameters

When you specify NOCOMPILE (x), where x is one of the error levels, your program will be compiled, if all the errors are of a lower severity than the x level. If an error of x level or higher occurs, the compilation stops and your program will be syntax-checked only. You will receive a source listing if you have specified the SOURCE option.

Using NOCOMPILE without Parameters

When you specify NOCOMPILE without parameters, the compiler only syntax-checks the source program. If you have also specified the SOURCE option, the compiler will produce a listing after the syntax checking is completed. The compiler does not produce object code when NOCOMPILE without parameters is in effect.

Note: The following compiler options are suppressed when you specify NOCOMPILE without parameters: DECK, LIST, OBJECT, OFFSET, OPTIMIZE, SSRANGE, and TEST.

The SEQUENCE Option

The SEQUENCE option tells the compiler to check your source program and flag statements that are out of sequence. You can use this option to tell you if a section of your source program was moved or deleted accidentally.

When you specify SEQUENCE, the compiler checks the source statement numbers you have supplied to see if they are in ascending order. Two asterisks are placed alongside any statement numbers out of sequence, and the total number of these statements is printed out as the first line of the diagnostics following the source listing.

The XREF Option

The XREF(FULL) option tells the compiler to generate a sorted cross-reference listing of data-names, procedure-names, and program-names. The cross-reference will include the line number where the data-name, procedure-name, or program-name was defined as well as the line numbers of all references.

You may use the cross-reference listing produced by the XREF option to find out where a data-name, procedure-name, or program-name was defined and referenced.

The XREF(SHORT) option allows you to control the cross-reference listing by printing only the explicitly referenced variables.

When you specify both the XREF (with FULL or SHORT) and the SOURCE options, you will get a modified cross-reference printed to the right of the source listing. This embedded cross-reference gives the line number where the data-name or procedure-name was defined.

For more information on the XREF option and some example listings, see “A Data-Name, Procedure-Name, and Program-Name Cross-Reference Listing” on page 322.

The MAP Option

Use the MAP option to produce a listing of the items you defined in the Data Division, plus all implicitly declared items. You can use the MAP output to locate the contents of a data item in a system dump.

For more information on the MAP option, see “Data Map Listing” on page 305.

Embedded Map Summary

When you specify the MAP option, an embedded MAP summary is generated to the right of the COBOL source data declaration. An embedded MAP summary contains condensed data MAP information. For more information, see “Embedded MAP Summary” on page 307.

The SSRANGE Option

You can use the SSRANGE compiler option to check subscripted or indexed data references, variable-length data references (a reference to a data item that contains an OCCURS DEPENDING ON clause), and reference-modified data references. If you specify the SSRANGE compiler option, additional code is generated to perform the checking at run time.

The subscripted or indexed data reference is checked to determine if the effective address of the wanted element is within the maximum boundary of the specified table.

The variable-length data reference is checked to determine if the actual length is positive and within the maximum defined length for the group data item.

The reference-modified data reference is checked to determine if the offset and length are positive and the sum of the offset and length are within the maximum length for the data item.

Remember that even when the SSRANGE option is specified, checking is not performed until run time and then, only if:

- The COBOL statement containing the indexed, subscripted, variable-length, or reference-modified data item is actually executed, and
- The CHECK(ON) run-time option is specified at run time.

If any check finds that an address is generated that is outside of the address range of the group data item containing the referenced data, an error message will be generated and the program will stop executing. The error message identifies the table or identifier that was being referenced and the line number in the program where the error occurred. Additional information is provided depending on the type of reference that caused the error.

If all subscripts, indices, or reference modifiers are literals in a given data reference and they result in a reference outside of the data item, the error will be diagnosed at compile time, regardless of what is specified for the SSRANGE compiler option.

Notes:

1. SSRANGE may cause your program's performance to diminish somewhat because of the extra overhead needed to check each subscripted or indexed item.
2. The subscripts, indexes, and variable-length items will be checked only if they are referenced as the object program runs.
3. The SSRANGE compiler option takes effect during run time, unless you have specified CHECK(OFF) as a run-time option.

The TEST Option

To obtain a formatted dump of working-storage, in case your program has ended abnormally, you should specify the TEST compiler option with the hook-location suboption of SYM. For details on the TEST suboptions, see the description of the TEST option on page 251.

Notes:

1. When you specify TEST with a hook-location value of ALL, STMT, PATH, or BLOCK or without any hook-location value, the OPTIMIZE compiler option is suppressed.
2. Remember that TEST (with any suboption value other than NONE) and USE FOR DEBUGGING/WITH DEBUGGING MODE statements are mutually exclusive, with USE FOR DEBUGGING/WITH DEBUGGING MODE taking precedence.

Getting Useful Listing Components

This section introduces the different types of compiler listings produced by COBOL/VSE. The type of listing produced by the compiler depends on which compiler options you specify.

After reading this section you should be familiar with each type of output; you will know how to request each type and what kind of information is provided in the output. The debugging procedures outlined in *LE/VSE Debugging Guide and Run-Time Messages* illustrate how the different types of output can be used to help you diagnose program failures.

A Short Listing—the Bare Minimum

If you do not specify any compiler options and the default options are NOSOURCE, NOXREF, NOVBREF, NOMAP, NOOFFSET, and NOLIST, or if all the compiler diagnostic options have been turned off, you will get a “short listing.”

The short listing contains diagnostic messages about the compilation, a list of the options in effect for the program, and statistics about the content of the program. Figure 99 on page 302 is an example of a short listing.

The listing is explained after Figure 99, and the numbers used in the explanation correspond to those in the figure. (For illustrative purposes, some errors that cause diagnostic messages to be issued were deliberately introduced.)

```
PP 5686-068 IBM COBOL for VSE/ESA 1.1.1                    1        Date 06/16/1998  Time 13:41:27   Page     1

JCL OPTION parameters:          2
NODECK,LINK,LIST,NOLISTX,NOSYM,NOTENM,SXREF

Invocation parameters:          3
OPTIMIZE

PROCESS(CBL) statements:
CBL RENT,NOSOURCE,TEST(ALL)     4

          5
        IGYOS4022-W   The "OPTIMIZE" option was discarded due to option conflict resolution. The "TEST" option from "PROCESS/CBL"
                      statement took precedence.

Options in effect:      6
    NOADATA
      ADV
      QUOTE
    NOAWO
      BUFSIZE(4096)
    NOCMPR2
    NOCOMPILE(S)
    NOCURRENCY
      DATA(31)
    NODATEPROC
    NODBCS
    NODECK
    NODUMP
    NODYNAM
    NOEXIT
    NOFASTSRT
      FLAG(I)
    NOFLAGMIG
    NOFLAGSAA
    NOFLAGSTD
      INTDATE(ANSI)
      LANGUAGE(EN)
    NOLIB
      LINECOUNT(60)
    NOLIST
    NOMAP
    NONAME
    NONUMBER
      NUMPROC(NOPFD)
      OBJECT
    NOOFFSET
    NOOPTIMIZE
      OUTDD(SYSOUT)
      RENT
      RMODE(AUTO)
      SEQUENCE
      SIZE(MAX)
    NOSOURCE
      SPACE(1)
    NOSSRANGE
      TERM
      TEST(ALL,SYM)
      TRUNC(STD)
    NOVBREF
PP 5686-068 IBM COBOL for VSE/ESA 1.1.1                                 Date 06/16/1998  Time 13:41:27   Page     2
    NOWORD
    NOXREF
      YEARWINDOW(1900)
      ZWB
```

Figure 99 (Part 1 of 2). Example of a Short Listing

```
DATA VALIDATION AND UPDATE PROGRAM     7                      IGYCARPA  Date 06/16/1998  Time 13:41:27   Page     3

LineID  Message code  Message text     8

        IGYDS0139-W   Diagnostic messages were issued during processing of compiler options. These messages are located at the
                      beginning of the listing.

   193  IGYDS1050-E   File "LOCATION-FILE" contained no data record descriptions. The file definition was discarded.

   889  IGYPS2052-S   An error was found in the definition of file "LOCATION-FILE". The reference to this file was discarded.

                      Same message on line:    983

   993  IGYPS2121-S   "WS-DATE" was not defined as a data-name. The statement was discarded.

                      Same message on line:    994

   995  IGYPS2121-S   "WS-TIME" was not defined as a data-name. The statement was discarded.

                      Same message on line:    996

   997  IGYPS2053-S   An error was found in the definition of file "LOCATION-FILE". This input/output statement was discarded.

                      Same message on line:   1009

  1008  IGYPS2121-S   "LOC-CODE" was not defined as a data-name. The statement was discarded.

  1219  IGYPS2121-S   "COMMUTER-SHIFT" was not defined as a data-name. The statement was discarded.

                      Same message on line:   1240

  1220  IGYPS2121-S   "COMMUTER-HOME-CODE" was not defined as a data-name. The statement was discarded.

                      Same message on line:   1241

  1222  IGYPS2121-S   "COMMUTER-NAME" was not defined as a data-name. The statement was discarded.

                      Same message on line:   1243

  1223  IGYPS2121-S   "COMMUTER-INITIALS" was not defined as a data-name.  The statement was discarded.

                      Same message on line:   1244

  1233  IGYPS2121-S   "WS-NUMERIC-DATE" was not defined as a data-name.  The statement was discarded.


Messages    Total    Informational    Warning    Error    Severe    Terminating     9

Printed:      21                          2         1        18


* Statistics for COBOL program IGYCARPA:     10
*    Source records = 1765
*    Data Division statements = 277
*    Procedure Division statements = 513


End of compilation 1,  program IGYCARPA,  highest severity 12.     11
Return code 12
```

Figure 99 (Part 2 of 2). Example of a Short Listing

1 COBOL/VSE default page header, including compiler level information from the LVLINFO installation time compiler option

2 Message about options passed to the compiler for options specified using the JCL OPTION statement, or the standard options for the partition.

3 Message about options passed to the compiler at compiler invocation. This message does not appear if no options were passed.

4 Options coded in the PROCESS (or CBL) statement.

RENT — The program was compiled to be reentrant to copy code from a library.

NOSOURCE — Turning SOURCE off eliminates the COBOL source code from the COBOL/VSE listing.

TEST(ALL) — The program was compiled for use with a debug tool

5 Deliberate option conflicts were forced by specifying the OPTIMIZE option on the compiler input parameter list. OPTIMIZE and the TEST(ALL) option specified on the CBL statement are mutually exclusive. As a result, the OPTIMIZE option is ignored.

6 Status of options at the start of this compilation.

7 Customized page header resulting from the COBOL program TITLE statement.

8 Program diagnostics. The first message will refer you to the library phase diagnostics, if there were any. Diagnostics for the library phase are always presented at the beginning of the listing.

9 Count of diagnostic messages in this program, grouped by severity level.

10 Program statistics for the program IGYCARPA.

11 Program statistics for the compilation unit. When you perform a batch compilation, the return code is the message severity level for the entire compilation.

Listing of Your Source Code—for Historical Records

By specifying the SOURCE compiler option, you request a copy of your source code to be included with the compiler output. You will want this output for testing and debugging your program—and as an historical record once the program is completely debugged. Figure 100 on page 305 shows an example of SOURCE output.

Using Your Own Line Numbers

The NUMBER compiler option tells the compiler to use your line numbers in the compiled program. When you specify the NUMBER option, the compiler does a sequence check of your source statement line numbers in columns 1 through 6 as the statements are read in. When a line number is found to be out of sequence, the compiler assigns to it a number with a value one higher than the line number of the preceding statement. The new value is flagged with two asterisks. A diagnostic message indicating an out of sequence error is also included in the compilation listing.

Figure 100 on page 305 shows an example of the output produced by the NUMBER compiler option. In the portion of the listing shown, the programmer numbered two of the statements out of sequence.

```
DATA VALIDATION AND UPDATE PROGRAM     1                      IGYCARPA  Date 06/16/1998  Time 13:41:27   Page    22
  LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Cross-Reference  2
    3     4     5
                087000/*****************************************************************
                087100***                D O   M A I N   L O G I C                 **
                087200***                                                          **
                087300*** Initialization. Read and process update transactions until **
                087400*** EOE. Close files and stop run.                          **
                087500*****************************************************************
                087600 procedure division.
                087700   000-do-main-logic.
                087800     display "PROGRAM IGYCARPA - Beginning"
                087900     perform 050-create-vsam-master-file.
                088150     display "perform 050-create-vsam-master finished".
  088151**      088125     perform 100-initialize-paragraph
                088200     display "perform 100-initialize-paragraph finished"
                088300     read update-transaction-file into ws-transaction-record
                088400         at end
              1 088500       set transaction-eof to true
                088600     end-read
                088700     display "READ completed"
                088800     perform until transaction-eof
              1 088900       display "inside perform until loop"
              1 089000       perform 200-edit-update-transaction
              1 089100       display "After perform 200-edit   "
              1 089200       if no-errors
              2 089300         perform 300-update-commuter-record
              2 089400         display "After perform 300-update "
              1 089650       else
  089651**    2 089600         perform 400-print-transaction-errors
              2 089700         display "After perform 400-errors "
              1 089800       end-if
              1 089900       perform 410-re-initialize-fields
              1 090000       display "After perform 410-reinitialize"
              1 090100       read update-transaction-file into ws-transaction-record
              1 090200           at end
              2 090300         set transaction-eof to true
              1 090400       end-read
              1 090500       display "After '2nd READ'   "
                090600     end-perform
```

Figure 100. Example of SOURCE and NUMBER Output

1 COBOL/VSE default page header

2 The scale line labels Area A, Area B, and source code column numbers

3 Source code line number assigned by the compiler

4 Program (PL) and statement (SL) nesting level

5 Columns 1 through 6 of program (the sequence number area)

Data Map Listing

The MAP compiler option provides you with a mapping of all Data Division items, plus all implicitly declared variables, of your program. From the MAP output you can locate specific data items within a storage dump. You can see an example of MAP output in Figure 101 on page 306. The numbers used in the explanation below the figure correspond to the numbers used in Figure 101. The terms and symbols used in MAP output are described in Figure 103 on page 308 and Figure 104 on page 308, respectively.

```
DATA VALIDATION AND UPDATE PROGRAM                          IGYCARPA  Date 06/16/1998  Time 13:41:27   Page    49

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         10
Source   Hierarchy and                               Base      Hex-Displacement  Asmblr Data                 Data Def
LineID   Data Name                                   Locator   Blk   Structure   Definition   Data Type      Attributes
     4  PROGRAM-ID IGYCARPA ----------------------------------------------------------------------------------------*
   181   FD COMMUTER-FILE . . . . . . . . . . . . .                                          VSAM           9
   183   01 COMMUTER-RECORD . . . . . . . . . . . . BLF=0000   000               DS 0CL80     Group
   184     02 COMMUTER-KEY. . . . . . . . . . . . . BLF=0000   000   0 000 000   DS 16C       Display
   185     02 FILLER. . . . . . . . . . . . . . . . BLF=0000   010   0 000 010   DS 64C       Display
   187   FD COMMUTER-FILE-MST . . . . . . . . . . .                                          VSAM           F
   189   01 COMMUTER-RECORD-MST . . . . . . . . . . BLF=0001   000               DS 0CL80     Group
   190     02 COMMUTER-KEY-MST. . . . . . . . . . . BLF=0001   000   0 000 000   DS 16C       Display
   191     02 FILLER. . . . . . . . . . . . . . . . BLF=0001   010   0 000 010   DS 64C       Display
   193   FD LOCATION-FILE . . . . . . . . . . . . .                                          SAM            FB
   198   01 LOCATION-RECORD . . . . . . . . . . . . BLF=0002   000               DS 0CL80     Group
   199     02 LOC-CODE. . . . . . . . . . . . . . . BLF=0002   000   0 000 000   DS 2C        Display
   200     02 LOC-DESCRIPTION . . . . . . . . . . . BLF=0002   002   0 000 002   DS 20C       Display
   201     02 FILLER. . . . . . . . . . . . . . . . BLF=0002   016   0 000 016   DS 58C       Display
   204   FD UPDATE-TRANSACTION-FILE . . . . . . . .                                          SAM            FB
   209   01 UPDATE-TRANSACTION-RECORD . . . . . . . BLF=0003   000               DS 80C       Display
   217   FD PRINT-FILE. . . . . . . . . . . . . . .                                          SAM            FB
   222   01 PRINT-RECORD. . . . . . . . . . . . . . BLF=0004   000               DS 121C      Display
   229   01 WORKING-STORAGE-FOR-IGYCARA . . . . . . BLW=0000   000               DS 1C        Display
   231   77 COMP-CODE . . . . . . . . . . . . . . . BLW=0000   008               DS 2C        Binary
   232   77 WS-TYPE . . . . . . . . . . . . . . . . BLW=0000   010               DS 3C        Display
   235   01 I-F-STATUS-AREA . . . . . . . . . . . . BLW=0000   018               DS 0CL2      Group
   236     02 I-F-FILE-STATUS . . . . . . . . . . . BLW=0000   018   0 000 000   DS 2C        Display
   237     88 I-O-SUCCESSFUL. . . . . . . . . . . .
   240   01 STATUS-AREA . . . . . . . . . . . . . . BLW=0000   020               DS 0CL8      Group
   241     02 COMMUTER-FILE-STATUS. . . . . . . . . BLW=0000   020   0 000 000   DS 2C        Display
   242     88 I-O-OKAY. . . . . . . . . . . . . . .
   243     02 COMMUTER-VSAM-STATUS. . . . . . . . . BLW=0000   022   0 000 002   DS 0CL6      Group
   244       03 VSAM-R15-RETURN-CODE. . . . . . . . BLW=0000   022   0 000 002   DS 2C        Binary
   245   77 UNUSED-DATA-ITEM  . . . . . . . . . . . BLW=XXXX   022   0 000 002   DS 10C       Display        11
   ⋮
```

Figure 101. Example of Map Output

1 Explanations of the data definition attribute codes

2 Source line number where the data item was defined

3 Level definition or number. The compiler generates this number in the following way:

- First level of any hierarchy is always 01. Increase 1 for each level— any item you coded as 02 through 49.
- Level numbers 66, 77, and 88, and the indicators FD and SD, are not changed.

4 Data-name that is used in the source file

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 Pseudo-assembler code showing how the data is defined.

9 Data definition attribute codes. The definitions are explained at the top of the Data Division Map.

10 The data type and use. These terms are explained in Figure 103 on page 308.

11 OPT(FULL) was specified and UNUSED-DATA-ITEM was not referred in the Procedure Division. Therefore UNUSED-DATA-ITEM was deleted, resulting in the base locator being set to 'XXXX'.

Embedded MAP Summary

An embedded MAP summary is printed by specifying the MAP option when generating a listing. The summary appears in the listing's right margin for lines within the Data Division that specify data declarations. Figure 102 describes the fields included in the embedded map summary.

Note: When both XREF data and an embedded MAP summary exist on the same line, the embedded MAP summary is printed first.

```
000002    Identification Division.
000003
000004    Program-ID.    IGYCARPA.
   ⋮
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=0003+000        121C
   ⋮
000228    Working-storage section.
000229      01 Working-storage-for-IGYCARA   pic x.                      BLW=0000+000        1C
000230
000231      77 comp-code                     pic S9999 comp.             BLW=0000+008        2C
000232      77 ws-type                       pic x(3)   value spaces.    BLW=0000+010        3C
000233
000234
000235      01 i-f-status-area.                                          BLW=0000+018        0CL2
000236        05 i-f-file-status             pic x(2).                   BLW=0000+018,0000000 2C
000237          88 i-o-successful            value zeros.
000238
000239
000240      01 status-area.                                              BLW=0000+020        0CL8
000241        05 commuter-file-status        pic x(2).                   BLW=0000+020,0000000 2C
000242          88 i-o-okay                  value zeros.
000243        05 commuter-vsam-status.                                   BLW=0000+022,0000002 0CL6
000244          10 vsam-r15-return-code      pic 9(2) comp.              BLW=0000+022,0000002 2C
000245          10 vsam-function-code        pic 9(1) comp.              BLW=0000+024,0000004 2C
000246          10 vsam-feedback-code        pic 9(3) comp.              BLW=0000+026,0000006 2C
000247
000248      77 update-file-status            pic xx.                     BLW=0000+028        2C
000249
000250      01 flags.                                                    BLW=0000+030        0CL3
000251        05 transaction-eof-flag        pic x value space.          BLW=0000+030,0000000 1C
000252            88 transaction-eof         value "Y".
000253        05 location-eof-flag           pic x value space.          BLW=0000+031,0000001 1C
000254            88 location-eof            value "Y".
000255        05 transaction-match-flag      pic x.                      BLW=0000+032,0000002 1C
   ⋮
000876    procedure division.
000877      000-do-main-logic.
000878        display "PROGRAM IGYCARPA - Beginning"
000879        perform 050-create-vsam-master-file.
   ⋮
```

Figure 102. Example of an Embedded MAP Summary

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 Abbreviated pseudo-assembler code showing how the data is defined

Figure 103. Terms Used in MAP Output

Use	Definition	Description
GROUP	DS 0CLn[1]	Group Fixed-Length
ALPHABETIC	DS nC	Alphabetic
ALPHA-EDIT	DS nC	Alphabetic-Edited
DISPLAY	DS nC	Alphanumeric
AN-EDIT	DS nC	Alphanumeric-Edited
GRP-VARLEN	DS VLC=n	Group Variable-Length
NUM-EDIT	DS nC	Numeric-Edited
DISP-NUM	DS nC	External Decimal
BINARY	DS 1H[2], 1F[2], 2F[2], 2C, 4C, or 8C	Binary (Computational)
COMP-1	DS 4C	Internal floating-point (single precision)
COMP-2	DS 8C	Internal floating-point (double precision)
PACKED-DEC	DS nP	Internal Decimal (Computational-3)
DBCS	DS nC	DBCS (Display-1)
DBCS-EDIT	DS nC	DBCS Edited
INDX-NAME		Index-name
INDEX		Index
POINTER		Pointer
File processing method (VSAM or SAM)		File (FD)
Level name for RENAMES		Condition (66)
Level name		Condition (77)
Level name for condition name		Condition (88)
Sort file definition		Sort Definition (SD)

Note:

1 **n** is the size in bytes, except in variable-length groups, where it is a variable-length cell number.

2 If the SYNCHRONIZED clause appears, these fields are used.

Figure 104 (Page 1 of 2). Symbols Used in LIST and MAP Output

Symbol	Definition
BL=n[1]	Base locator for special registers
BLF=n[1]	Base locator for files
BLS=n[1]	Base locator for sort items
IDX=n[1]	Base locator for index names
BLA=n[1]	Base locator for function/evaluated data
BLV=n[1]	Base locator for variably located data
BLX=n[1]	Base locator for external data
BLL=n[1]	Base locator for linkage section
BLW=n[1]	Base locator for working storage

Figure 104 (Page 2 of 2). Symbols Used in LIST and MAP Output

Symbol	Definition
BLA=n[1]	Base locator for alphanumeric temporaries
CBL=n[1]	Base locator for CGT
TGT FDMP TEST INFO. AREA + nnnn[3]	FDUMP/TEST information area
SYSLIT AT + nnnn[3]	Displacement for system literal from beginning of system literal pool
RBKST=n[1]	Register backstore cell
PSV=n[1]	Perform save cell number
TGTFIXD + nnnn[3]	Offset from beginning of fixed portion of TGT
CLLE@=	Load list entry address in TGT
TOV=n[1]	TGT overflow cell number
EVALUATE=n[1]	Evaluate boolean cell in TGT
TS1=aaaa	Temporary storage cell number in subpool 1
FCB=n[1]	FCB address
TS2=aaaa	Temporary storage cell number in subpool 2
GN=n(hhhhh)[2]	Generated procedure name and its offset in hexadecimal
TS3=aaaa	Temporary storage cell number in subpool 3
IDX=n[1]	Index cell number
TS4=aaaa	Temporary storage cell number in subpool 4
ODOSAVE=n[1]	ODO save cell number
V(routine name)	Assembler VCON for external routine
OPT=nnnn[3]	Optimizer temporary storage cell
VLC=n[1]	Variable-length name cell number (ODO)
PBL=n[1]	Base locator for procedure code
VNI=n[1]	Variable name initialization
PFM=n[1]	PERFORM n times cells
WHEN=n[1]	Evaluate WHEN cell number in TGT
PGMLIT AT + nnnn[3]	Displacement for program literal from beginning of literal pool

Note:

1 n is the number of the entry

2 (hhhhh) is the program offset in hexadecimal

3 nnnn is the offset in decimal from the beginning of the entry

Nested Program Map

The MAP compiler option also supplies you with a nested program map if your program contains nested programs. The nested program map shows where the programs are defined and provides program attribute information.

```
PP 5686-068 IBM COBOL for VSE/ESA 1.1.1                    NESTED    Date 06/16/1998  Time 13:41:27   Page     5
Nested Program Map

    1

Program Attribute codes (rightmost column) have the following meanings:
   C = COMMON
   I = INITIAL
   U = PROCEDURE DIVISION USING...

   2    3       4                                                    5

Source Nesting                                                      Program
LineID Level   Program Name from PROGRAM-ID paragraph                Attributes
     2         NESTED. . . . . . . . . . . . . . . . . . . . . . . .
    12     1     X1. . . . . . . . . . . . . . . . . . . . . . . . .
    20     2       X11 . . . . . . . . . . . . . . . . . . . . . . .
    27     2       X12 . . . . . . . . . . . . . . . . . . . . . . .
    35     1     X2. . . . . . . . . . . . . . . . . . . . . . . . .
```

Figure 105. Example of Nested Program Map

1. Explanations of the program attribute codes
2. The source line number where the program was defined
3. Depth of program nesting
4. The program name
5. The program attribute codes

A Procedure Division Listing with Assembler Expansion (LIST Output)

The LIST compiler option provides you with a listing of the Procedure Division along with the assembler coding produced by the COBOL/VSE compiler. This type of output can be especially helpful when you are trying to find the failing verb in a program. You can also use this output to find the address in storage of a data item that was moved during the program.

Getting LIST Output

You request LIST output from the compiler by specifying the LIST compiler option when you compile your program. You will receive the output if:

- You have specified the COMPILE option (or the NOCOMPILE(x) option is in effect and an error level x or higher does not occur)
- You did not specify the OFFSET option. OFFSET and LIST are mutually exclusive options with OFFSET taking precedence.

Reading LIST Output

The LIST compiler option produces eight pieces of output:

1. An assembler listing of the initialization code for the program
2. Information on the Program Global Table
3. Information on the Constant Global Table
4. An assembler listing of the source code for the program
5. The location of compiler-generated tables in the object phase
6. A map of the Task Global Table (TGT)
7. Information on the location and size of working storage and control blocks
8. Information on the location of literals and code for dynamic storage use

The following examples highlight the parts of the LIST compiler output that may be useful to you for debugging your program. You do not need to be able to program in assembler language to understand the output produced by LIST. The comments which accompany most of the assembler code will provide you with a conceptual understanding of the functions performed by the code. However, if you find that you need to interpret a particular instruction and you are unfamiliar with assembler instructions, you can refer to *Enterprise Systems Architecture/370 Reference Summary* for help.

The symbols used in LIST output are defined in Figure 104 on page 308.

Program Initialization Code

A listing of the program initialization code can help you to debug your program. This piece of output also gives you information on the characteristics of the COBOL/VSE source program; you can interpret the program signature information bytes to verify such characteristics as: the compiler options in effect, types of data items present, and the verbs used in the Procedure Division.

Figure 106 shows an example of program initialization code. Explanations of some of the fields in the listing follow the figure.

```
DATA VALIDATION AND UPDATE PROGRAM                          IGYCARPA  Date 06/16/1998  Time 13:41:27   Page    49

     1    2                    3                                4
    000000                IGYCARPA DS   0H
                                   USING *,15
    000000  47F0 F0E4             B    228(,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  00000000             DC   F'0'                   STACK FRAME SIZE
    00000C  00000014             DC   A(@PPA1-IGYCARPA)      OFFSET TO PPA1 FROM PRIMARY ENTRY
    000010  47F0 F001             B    1(,15)                 RESERVED
    000014                @PPA1  DS   0H                     PPA1 STARTS HERE
    000014  4A                   DC   X'4A'                  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  000000AC             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  00000000             DC   F'0'                   OFFSET FOR STACK OVERFLOW RETURN
    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   H'0'                   AVAILABLE HALF-WORD
```

Figure 106 (Part 1 of 2). LIST Output—Program Initialization with Program Signature Highlighted

```
000038                 @MAINENT DS   0H                    PRIMARY ENTRY POINT ADDRESS
000038  00000000                DC   A(IGYCARPA)     @PARMS: 1) PRIMARY ENTRY POINT ADDRESS
00003C  000000F4                DC   A(PGT)                 2) PGT ADDRESS                    6
000040  000058D0                DC   A(TGT)                 3) TGT ADDRESS                    7
000044  00000060                DC   A(*+28)                4) A(@EPNAM)= E. P. NAME ADDRESS
000048  00000000                DC   A(IGYCARPA)            5) CURRENT ENTRY POINT ADDRESS
00004C  00002322                DC   A(START)       @BRVAL: 6) PROCEDURE CODE ADDRESS
000050  00000000                DC   V(IGZCBSN)             7) INITIALIZATION ROUTINE
000054  000000C0                DC   A(@CEEPARM)            8) ADDRESS OF PARM LIST FOR CEEINT
000058  00000000                DC   AL4'0'                 AVAILABLE WORD
00005C  0000                    DC   AL2'0'                 AVAILABLE HALF-WORD
00005E  0008                    DC   X'0008'                LENGTH OF PROGRAM NAME
000060                 @EPNAM   DS   0H                    ENTRY POINT NAME
000060  C9C7E8C3C1D9D7C1        DC   C'IGYCARPA'            PROGRAM NAME                      8
000068  F1F9F9F8                DC   CL4'1998'     @TIMEVRS: YEAR OF COMPILATION               9
00006C  F0F6F1F6                DC   CL4'0616'               MONTH/DAY OF COMPILATION          10
000070  F1F3F4F1                DC   CL4'1341'               HOURS/MINUTES OF COMPILATION      11
000074  F2F7                    DC   CL2'27'                SECONDS FOR COMPILATION DATE
000076  F0F1F0F1F0F1            DC   CL6'010101'            VERSION/RELEASE/MOD LEVEL OF PROD 12
00007C  00000000                DC   AL4'0'                 AVAILABLE WORD
000080  0000                    DC   X'0000'                INFO. BYTES 28-29
000082  076C                    DC   X'076C'                SIGNED BINARY YEARWINDOW OPTION VALUE
000084  A0483D4400A1            DC   X'A0483D4400A1'        INFO. BYTES 1-6                   13
00008A  0080D2A8090B            DC   X'0080D2A8090B'        INFO. BYTES 7-12
000090  8D3060000400            DC   X'8D3060000400'        INFO. BYTES 13-18
000096  0000018000              DC   X'0000018000'          INFO. BYTES 19-23
00009B  00                      DC   X'00'                  COBOL SIGNATURE LEVEL
00009C  00000125                DC   X'00000125'            # DATA DIVISION STATEMENTS       14
0000A0  00000201                DC   X'00000201'            # PROCEDURE DIVISION STATEMENTS  15
0000A4  200400                  DC   X'200400'              INFO. BYTES 24-26                 13
0000A7  00                      DC   X'00'                  RESERVED
0000A8  40404040                DC   C'    '              USER LEVEL INFO (LVLINFO)         16
0000AC                 @PPA2    DS   0H                    PPA2 STARTS HERE
0000AC  05                      DC   X'05'                  CEL MEMBER IDENTIFIER
0000AD  00                      DC   X'00'                  CEL MEMBER SUB-IDENTIFIER
0000AE  00                      DC   X'00'                  CEL MEMBER DEFINED BYTE
0000AF  01                      DC   X'01'                  CONTROL LEVEL OF PROLOG
0000B0  00000000                DC   V(CEESTART)            VCON FOR LOAD MODULE
0000B4  00000000                DC   F'0'                   OFFSET TO THE CDI (NONE)
0000B8  FFFFFFBC                DC   A(@TIMEVRS-@PPA2)      OFFSET TO TIMESTAMP/VERSION INFO
0000BC  00000000                DC   A(IGYCARPA)            ADDRESS OF CU PRIMARY ENTRY POINT
0000C0                 @CEEPARM DS   0H                    PARM LIST FOR CEEINT
0000C0  00000038                DC   A(@MAINENT)            POINTER TO PRIMARY ENTRY PT ADDR
0000C4  00000008                DC   A(@PARMCEE-@CEEPARM)   OFFSET TO PARAMETERS FOR CEEINT
0000C8                 @PARMCEE DS   0H                    PARAMETERS FOR CEEINT
0000C8  00000006                DC   F'6'                   1) NUMBER OF ENTRIES IN PARM LIST
0000CC  00000038                DC   A(@MAINENT)            2) POINTER TO PRIMARY ENTRY PT ADDR
0000D0  00000000                DC   V(CEESTART)            3) ADDRESS OF CEESTART
0000D4  00000000                DC   V(CEEBETBL)            4) ADDRESS OF CEEBETBL
0000D8  00000005                DC   F'5'                   5) CEL MEMBER IDENTIFIER
0000DC  00000000                DC   F'0'                   6) FOR CEL MEMBER USE
0000E0  00000000                DC   F'0'                   AVAILABLE WORD
0000E4  00000000                DC   AL4'0'                 AVAILABLE WORD
0000E8  00000000                DC   AL4'0'                 AVAILABLE WORD
0000EC  00000000                DC   AL4'0'                 AVAILABLE WORD
0000F0  0000                    DC   AL2'0'                 AVAILABLE HALF-WORD
```

Figure 106 (Part 2 of 2). LIST Output—Program Initialization with Program Signature Highlighted

1 Offset from start of the COBOL program

2 Hexadecimal representation of assembler instruction

3 Pseudo assembler code generated for the COBOL program

4 Comments explaining assembler code

5 The "eye catcher" indicating COBOL/VSE compiler used to compile this program

6 The address of the Program Global Table (PGT)

7 The address of the Task Global Table (TGT), or the address of the Dynamic Access Block (DAB) if the program is re-entrant

8 The program name as used in the Identification Division of the program

9 The 4-digit year that the program was compiled

10 The month and the day that the program was compiled

11 The time that the program was compiled

12 The version, release, and modification level of the COBOL/VSE compiler used to compile this program (each represented in two digits)

13 The program signature information bytes. These provide information about compiler options, the Data Division, the Environment Division, and the Procedure Division for this program. See Figure 107, Figure 108 on page 314, Figure 109 on page 315, Figure 110 on page 315, and Figure 111 on page 316 for bit mappings of the program signature bytes.

14 The number of statements in the Data Division

15 The number of statements in the Procedure Division

16 A 4-byte user-controlled level information field. The value of this field is controlled by the LVLINFO.

Program Signature Information Bytes

Compiler Options in Effect

Figure 107 (Page 1 of 2). Compiler Options In Effect

Byte	Bit	On	Off
1	0	ADV	NOADV
1	1	APOST	QUOTE
1	2	DATA(31)	DATA(24)
1	3	DECK	NODECK
1	4	DUMP	NODUMP
1	5	DYNAM	NODYNAM
1	6	FASTSRT	NOFASTSRT
1	7	FDUMP	NOFDUMP
2	0	LIB	NOLIB
2	1	LIST	NOLIST
2	2	MAP	NOMAP
2	3	NUM	NONUM
2	4	OBJ	NOOBJ
2	5	OFFSET	NOOFFSET
2	6	OPTIMIZE	NOOPTIMIZE
2	7	Filename supplied in OUTDD option will be used	Default filename for OUTDD will be used
3	0	NUMPROC(PFD)	NUMPROC(NOPFD)
3	1	RENT	NORENT
3	2	RES	NORES
3	3	SEQUENCE	NOSEQUENCE
3	4	SIZE(MAX)	SIZE(value)
3	5	SOURCE	NOSOURCE
3	6	SSRANGE	NOSSRANGE
3	7	TERM	NOTERM
4	0	TEST	NOTEST
4	1	TRUNC(STD)	TRUNC(OPT)
4	2	User-Supplied Reserved Word List	Installation Default Reserved Word List

Figure 107 (Page 2 of 2). Compiler Options In Effect

Byte	Bit	On	Off
4	3	VBREF	NOVBREF
4	4	XREF	NOXREF
4	5	ZWB	NOZWB
4	6	NAME	NONAME
4	7	CMPR2	NOCMPR2
5	0	NUMPROC(MIG)	
5	1	NUMCLASS	NONUMCLASS
5	2	DBCS	NODBCS
5	3	AWO	NOAWO
5	4	TRUNC(BIN)	not TRUNC(BIN)
5	5	ADATA	NOADATA
5	6	CURRENCY	NOCURRENCY
26	0	RMODE(ANY)	RMODE(24)
26	1	TEST(STMT)	not TEST(STMT)
26	2	TEST(PATH)	not TEST(PATH)
26	3	TEST(BLOCK)	not TEST(BLOCK)
26	4	OPT(FULL)	OPT(STD) or NOOPT
26	5	INTDATE(LILIAN)	INTDATE(ANSI)
27	4	DATEPROC	NODATEPROC

Note: The FDUMP compiler option (see byte 2) provides compatibility with VS COBOL II.

Items Present in the Data Division

Figure 108. Items Present in the Data Division

Byte	Bit	Item
6	0	SAM file descriptor
6	1	VSAM sequential file descriptor
6	2	VSAM indexed file descriptor
6	3	VSAM relative file descriptor
6	4	CODE-SET clause (ASCII files) in file descriptor
6	5	Spanned Records
6	6	PIC G (DBCS item)
6	7	OCCURS DEPENDING ON clause in data description entry
7	0	SYNCHRONIZED clause in data description entry
7	1	JUSTIFIED clause in data description entry
7	2	USAGE IS POINTER item
7	3	Complex OCCURS DEPENDING ON clause
7	4	External floating-point items in the Data Division
7	5	Internal floating-point items in the Data Division
7	6	Reserved
7	7	USAGE IS PROCEDURE-POINTER item

Items Present in the Environment Division

Figure 109. Items Present in the Environment Division

Byte	Bit	Item
8	0	FILE STATUS clause in FILE-CONTROL paragraph
8	1	RERUN clause in I/O-CONTROL paragraph of I/O section.
8	2	UPSI switch defined in SPECIAL-NAMES paragraph

Verbs Present in the Procedure Division

Figure 110 (Page 1 of 2). Verbs Present in the Procedure Division

Byte	Bit	Item
9	0	ACCEPT
9	1	ADD
9	2	ALTER
9	3	CALL
9	4	CANCEL
9	6	CLOSE
10	0	COMPUTE
10	2	DELETE
10	4	DISPLAY
10	5	DIVIDE
11	1	END-PERFORM
11	2	ENTER
11	3	ENTRY
11	4	EXIT
11	6	GO TO
11	7	IF
12	0	INITIALIZE
12	2	INSPECT
12	3	MERGE
12	4	MOVE
12	5	MULTIPLY
12	6	OPEN
12	7	PERFORM
13	0	READ
13	2	RELEASE
13	3	RETURN
13	4	REWRITE
13	5	SEARCH
13	7	SET
14	0	SORT
14	1	START
14	2	STOP
14	3	STRING

Figure 110 (Page 2 of 2). Verbs Present in the Procedure Division

Byte	Bit	Item
14	4	SUBTRACT
14	7	UNSTRING
15	0	USE
15	1	WRITE
15	2	CONTINUE
15	3	END-ADD
15	4	END-CALL
15	5	END-COMPUTE
15	6	END-DELETE
15	7	END-DIVIDE
16	0	END-EVALUATE
16	1	END-IF
16	2	END-MULTIPLY
16	3	END-READ
16	4	END-RETURN
16	5	END-REWRITE
16	6	END-SEARCH
16	7	END-START
17	0	END-STRING
17	1	END-SUBTRACT
17	2	END-UNSTRING
17	3	END-WRITE
17	4	GOBACK
17	5	EVALUATE
17	7	SERVICE statement

More Procedure Division Information

Figure 111 (Page 1 of 2). More Procedure Division Information

Byte	Bit	Item
21	0	Hexadecimal literal
21	1	Altered GO TO
21	2	I/O error declarative
21	3	LABEL declarative
21	4	DEBUGGING declarative
21	5	Program segmentation
21	6	OPEN...EXTEND
21	7	EXIT PROGRAM
22	0	CALL literal
22	1	CALL identifier
22	2	CALL...ON OVERFLOW
22	3	CALL...LENGTH OF
22	4	CALL...ADDRESS OF

Figure 111 (Page 2 of 2). More Procedure Division Information

Byte	Bit	Item
22	5	CLOSE...REEL/UNIT
22	6	Exponentiation used
22	7	Floating point items used
23	0	COPY
23	1	BASIS
23	2	DBCS name in program
23	3	SHIFT-OUT and SHIFT-IN in program
23	4-7	Highest error severity at entry to ASM2 phase IGYBINIT
24	0	DBCS literal
24	1	REPLACE
24	2	Reference modification was used
24	3	Nested program
24	4	INITIAL
24	5	COMMON
24	6	SELECT ... OPTIONAL
24	7	EXTERNAL
25	0	GLOBAL
25	1	RECORD IS VARYING
25	2	ACCEPT FROM SYSIPT used in Label Declarative
25	3	DISPLAY UPON SYSLST used in Label Declarative
25	4	DISPLAY UPON SYSPCH used in Label Declarative
25	5	Intrinsic function was used

Note: A return code of greater than 4 from the compiler could mean that some of the verbs shown as being in the program in information bytes may have been discarded because of an error.

Assembler Code for Source Program

If, in the course of debugging, you find the address in storage of the instruction that was executing when the abend occurred, you will want to find the COBOL verb which corresponds to that instruction. Once you have found the address of the failing instruction, you can go to the assembler listing and find the verb for which that instruction was generated.

```
DATA VALIDATION AND UPDATE PROGRAM                          IGYCARPA  Date 06/16/1998  Time 13:41:27   Page    52

000433  MOVE  1
000435  READ
000436  SET

     2      3          4         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(CEE5ABN)
    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               GN=14   EQU  *
    000F64  5820 D154            L    2,340(0,13)           VN=3
    000F68  07F2                 BCR  15,2
```

Figure 112. LIST Output—Assembler Code Generated from Source Code

1 Source line-number and COBOL verb, paragraph-name or section-name

In line 000436, SET is the COBOL verb. An * 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 phase, in hexadecimal notation

3 The object code instruction, in hexadecimal notation

The first two or four hexadecimal digits are the instruction, while 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

TGT Memory Map

The Task Global Table (TGT), which is described in greater detail in the *LE/VSE Debugging Guide and Run-Time Messages*, contains information about the environment in which your program is running. Assume you are looking for the number of file control blocks (FCBs) for your program. You can go to the TGT Memory Map and find the name of the TGT field that contains the number of FCBs. The offset into the TGT (number of bytes past the beginning of the TGT) is listed to the left of the field name in the memory map. You can then go to the start of the TGT, count off the number of bytes equal to the offset, and find the number of FCBs for your program.

```
DATA VALIDATION AND UPDATE PROGRAM                          IGYCARPA  Date 06/16/1998  Time 13:41:27   Page   132


               *** TGT MEMORY MAP ***
          PGMLOC  TGTLOC
            1       2       3
          010EA0  000000  72 BYTE SAVE AREA
          010EE8  000048  TGT IDENTIFIER
          010EEC  00004C  NEXT AVAILABLE BYTE ADDRESS FOR CEL
          010EF0  000050  TGT LEVEL INDICATOR
          010EF1  000051  RESERVED - 3 SINGLE BYTE FIELDS
          010EF4  000054  32 BIT SWITCH
          010EF8  000058  POINTER TO RUNCOM
          010EFC  00005C  POINTER TO COBVEC
          010F00  000060  POINTER TO PROGRAM DYNAMIC BLOCK TABLE
          010F04  000064  NUMBER OF FCB'S
          010F08  000068  WORKING STORAGE LENGTH
          010F0C  00006C  POINTER TO PREVIOUS TGT IN TGT CHAIN
          010F10  000070  ADDRESS OF IGZESMG WORK AREA
          010F14  000074  ADDRESS OF 1ST GETMAIN BLOCK (SPACE MGR)
          010F18  000078  FULLWORD RETURN CODE
          010F1A  00007A  RETURN CODE SPECIAL REGISTER
          010F1C  00007C  SORT-RETURN SPECIAL REGISTER
          010F1E  00007E  MERGE FILE NUMBER
          010F20  000080  ADDRESS OF CEL COMMON ANCHOR AREA
          010F24  000084  LENGTH OF TGT
          010F28  000088  RESERVED - 1 SINGLE BYTE FIELD
          010F29  000089  PROGRAM MASK USED BY THIS PROGRAM
          010F2A  00008A  RESERVED - 2 SINGLE BYTE FIELDS
          010F2C  00008C  NUMBER OF SECONDARY FCB CELLS
          010F30  000090  LENGTH OF THE VN(VNI) VECTOR
          010F34  000094  COUNT OF NESTED PROGRAMS IN COMPILE UNIT
          010F38  000098  DDNAME FOR DISPLAY OUTPUT
          010F40  0000A0  SORT-CONTROL SPECIAL REGISTER
          010F48  0000A8  POINTER TO COM-REG SPECIAL REGISTER
          010F4C  0000AC  CALC ROUTINE REGISTER SAVE AREA
          010F80  0000E0  ALTERNATE COLLATING SEQUENCE TABLE PTR.
          010F84  0000E4  ADDRESS OF SORT G.N. ADDRESS BLOCK
          010F88  0000E8  ADDRESS OF PGT
          010F8C  0000EC  CURRENT INTERNAL PROGRAM NUMBER
          010F90  0000F0  POINTER TO 1ST IPCB
          010F94  0000F4  ADDRESS OF THE CLLE FOR THIS PROGRAM
          010F98  0000F8  POINTER TO ABEND INFORMATION TABLE
          010F9C  0000FC  POINTER TO TEST INFO FIELDS IN THE TGT
          010FA0  000100  ADDRESS OF START OF COBOL PROGRAM
          010FA4  000104  POINTER TO VN'S IN CGT
          010FA8  000108  POINTER TO VN'S IN TGT
          010FAC  00010C  POINTER TO FIRST PBL IN THE PGT
          010FB0  000110  POINTER TO FIRST FCB CELL
          010FB4  000114  WORKING STORAGE ADDRESS
          010FB8  000118  POINTER TO FIRST SECONDARY FCB CELL

               *** VARIABLE PORTION OF TGT ***

          010FBC  00011C  BACKSTORE CELL FOR SYMBOLIC REGISTERS
          011044  0001A4  BASE LOCATORS FOR SPECIAL REGISTERS
          01104C  0001AC  BASE LOCATORS FOR WORKING-STORAGE     4
          011054  0001B4  BASE LOCATORS FOR LINKAGE-SECTION
          011058  0001B8  BASE LOCATORS FOR FILES
          01106C  0001CC  BASE LOCATORS FOR ALPHANUMERIC TEMPS
          011070  0001D0  CLLE ADDR. CELLS FOR CALL LIT. SUB-PGMS.
          01108C  0001EC  TEST INFORMATION AREA
          0110D0  000230  VARIABLE NAME (VN) CELLS
          01113C  00029C  INDEX CELLS
          011160  0002C0  PERFORM SAVE CELLS
          011288  0003E8  VARIABLE LENGTH CELLS
          011294  0003F4  ODO SAVE CELLS
          0112A0  000400  FCB CELLS
          0112B4  000414  ALL PARAMETER BLOCK
          011318  000478  INTERNAL PROGRAM CONTROL BLOCKS
          011328  000488  TEMPORARY STORAGE-1
          011338  000498  TEMPORARY STORAGE-2
```

Figure 113. LIST Output—TGT Memory Map

1 Hexadecimal offset of the TGT field from the start of the COBOL program

2 Hexadecimal offset of TGT field from the start of the TGT

3 Explanation of the contents of the TGT field

4 TGT fields for the base locators of COBOL data areas

Location and Size of Working Storage

You can use this piece of LIST output to find the location in a storage dump of data items defined in working storage.

```
  1              2          3
WRK-STOR LOCATED AT 0066D8 FOR 00001598 BYTES
```

Figure 114. LIST Output—Working Storage

1 Working storage identification

2 The hexadecimal offset of working storage from the start of the COBOL program

3 Length of working storage in hexadecimal

A Condensed Procedure Division Listing

The OFFSET compiler option allows you to request a condensed version of the Procedure Division. This listing is helpful when you need to verify that you still have a valid logic path after you move or add Procedure Division sections.

When you specify OFFSET, the compiler generates a condensed verb listing, global tables, working storage information, and literals. Note that OFFSET and LIST are mutually exclusive compiler options with OFFSET taking precedence.

```
DATA VALIDATION AND UPDATE PROGRAM                          IGYCARPA  Date 06/16/1998  Time 13:41:27   Page    54

  1     2     3

LINE #  HEXLOC  VERB                    LINE #  HEXLOC  VERB                    LINE #  HEXLOC  VERB
000878 002388 DISPLAY                   000879 00239A PERFORM                   000880 0023B6 DISPLAY
000881 0023C8 PERFORM                   000882 0023E4 DISPLAY                   000883 0023F6 READ
000885 002446 SET                       000887 00244A DISPLAY                   000888 002454 PERFORM
000889 002460 DISPLAY                   000890 002472 PERFORM                   000891 00248E DISPLAY
000892 0024A0 IF                        000893 0024AE PERFORM                   000894 0024CA DISPLAY
000896 0024E4 PERFORM                   000897 002500 DISPLAY                   000899 002512 PERFORM
000900 002532 DISPLAY                   000901 002544 READ                      000903 002594 SET
000905 002598 DISPLAY                   000907 0025A6 DISPLAY                   000908 0025B8 CLOSE
000910 00260C DISPLAY                   000916 002616 IF                        000917 002620 DISPLAY
000918 00262A MOVE                      000919 002630 PERFORM                   000920 00264C PERFORM
000926 00266C PERFORM                   000927 00268C DISPLAY                   000928 00269E PERFORM
000929 0026BE DISPLAY                   000930 0026D0 DISPLAY                   000931 0026DA DISPLAY
000932 0026E4 DISPLAY                   000933 0026EE STOP                      000944 0026F8 OPEN
```

Figure 115. Example of OFFSET Compiler Output

The following numbers refer to the numbers in Figure 115.

1 Line Number

Your line numbers or compiler-generated line numbers are listed.

2 Offset

The offset, in hexadecimal, from the start of the program, of the code generated for this verb.

The verbs are listed in the order in which they occur, and once for each time they are used.

3 Verb used

A Verb Cross-Reference Listing

The VBREF compiler option produces an alphabetic listing of all the verbs in your program and shows where each is referenced. The output includes each verb used, a count of the number of times it is used, and the line numbers where the verb is used. You can use VBREF output as a handy lookup when you need to find an instance of a particular verb.

```
DATA VALIDATION AND UPDATE PROGRAM                          IGYCARPA  Date 06/16/1998  Time 13:41:27   Page    49

    1        2                          3

 Count   Cross-reference of verbs       References

    2   ACCEPT . . . . . .  1012 1014
    2   ADD. . . . . .  1320 1336
    2   CALL . . . . . .  1436 1437
    5   CLOSE. . . . . .  908 958 983 1556 1565
   20   COMPUTE. . . . . .  1536 1670 1674 1687 1690 1693 1694 1695 1708 1712 1716 1721 1726 1731 1739 1743
                            1748 1753 1758 1763
    2   CONTINUE . . . . . .  1068 1077
    2   DELETE . . . . . .  977 1223
   83   DISPLAY. . . . . .  878 880 882 887 889 891 894 897 900 905 907 910 917 927 929 930 931 932 946 953
                            955 960 966 973 979 985 1007 1039 1053 1057 1061 1063 1070 1072 1079 1083 1087
                            1090 1092 1094 1096 1098 1100 1102 1104 1107 1110 1112 1113 1118 1127 1130 1138
                            1141 1143 1147 1198 1201 1215 1225 1417 1418 1419 1420 1421 1422 1423 1431 1432
                            1433 1434 1435 1515 1516 1522 1527 1528 1550 1551 1558 1559 1654
    2   EVALUATE . . . . . .  1191 1587
    1   EXIT . . . . . .  1464
   46   IF . . . . . .  892 916 945 952 959 965 972 978 984 1006 1038 1054 1058 1080 1084 1089 1111 1126
                            1137 1144 1155 1158 1161 1164 1167 1171 1175 1178 1181 1197 1214 1224 1270 1277
                            1295 1302 1319 1351 1360 1369 1381 1391 1514 1526 1549 1557
  183   MOVE . . . . . .  918 950 970 996 997 998 999 1000 1001 1008 1013 1015 1027 1040 1055 1059 1066
                            1081 1085 1091 1093 1095 1097 1099 1101 1114 1128 1139 1156 1159 1162 1165 1169
                            1173 1176 1179 1182 1190 1193 1199 1205 1207 1210 1211 1216 1221 1226 1231 1238
                            1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1259 1260
                            1261 1262 1263 1264 1265 1269 1271 1274 1278 1280 1281 1283 1284 1285 1287 1288
                            1289 1290 1294 1296 1299 1303 1305 1306 1308 1309 1321 1324 1329 1331 1333 1337
                            1343 1344 1345 1346 1347 1348 1349 1350 1352 1353 1357 1358 1361 1363 1364 1366
                            1368 1371 1372 1373 1374 1378 1379 1382 1384 1385 1387 1392 1394 1398 1404 1405
                            1406 1407 1408 1409 1410 1411 1445 1448 1453 1456 1475 1476 1477 1478 1480 1481
                            1482 1487 1494 1519 1532 1537 1538 1539 1547 1581 1591 1596 1601 1606 1611 1616
                            1621 1626 1631 1636 1641 1646 1651 1656 1657 1709 1713 1718 1723 1728 1733 1740
                            1745 1750 1755 1760 1765
    5   OPEN . . . . . .  944 964 1002 1473 1513
   61   PERFORM. . . . . .  879 881 888 890 893 896 899 919 920 926 928 947 948 954 956 961 962 967 968 974
                            975 980 981 986 987 1009 1010 1025 1041 1115 1116 1145 1194 1195 1200 1202 1206
                            1208 1209 1212 1217 1218 1222 1227 1228 1232 1276 1286 1301 1310 1359 1370 1380
                            1389 1395 1399 1534 1540 1541 1562 1563
    8   READ . . . . . .  883 901 971 1016 1028 1105 1520 1544
    1   REWRITE. . . . . .  1213
    4   SEARCH . . . . . .  1064 1073 1444 1452
   46   SET. . . . . .  885 903 1018 1030 1043 1062 1071 1103 1108 1393 1443 1451 1523 1529 1546 1552
                            1580 1589 1590 1594 1595 1599 1600 1604 1605 1609 1610 1614 1615 1619 1620 1624
                            1625 1629 1630 1634 1635 1639 1640 1644 1645 1649 1650 1669 1673
    1   STOP . . . . . .  933
    4   STRING . . . . . .  1266 1291 1354 1375
   33   WRITE. . . . . .  951 1196 1322 1323 1325 1326 1327 1328 1330 1332 1335 1484 1489 1492 1495 1497
                            1501 1542 1684 1685 1697 1698 1699 1770 1772 1774 1775 1776 1777 1778 1779 1780
```

Figure 116. Example of VBREF Compiler Output

The numbers in the explanation below correspond to Figure 116.

1 Number of times the verb is used in the program

2 Verb

3 Line numbers where verb is used

A Data-Name, Procedure-Name, and Program-Name Cross-Reference Listing

The XREF compiler option provides you with sorted cross-reference listings of data-names, procedure-names, and program-names. The listings also tell you the location of all references to a particular data-, procedure-, or program-name. This output will help you find, quickly, a reference to a particular data-, procedure-, or program-name in your program.

If your program contains DBCS user-defined words, these user-defined words will be listed **before** the alphabetic list of EBCDIC user-defined words. The DBCS user-defined words are listed in physical order according to their appearance in the COBOL/VSE source program.

Note: Group names in a MOVE CORRESPONDING statement are listed in the XREF listing. The cross-reference listing includes the group names and all the elementary names involved in the move.

Using a Sorted Cross-Reference Listing

You can use XREF output to find where you have used a particular data- or procedure-name. If you need to find all of the statements that modify a particular data item, you can use the output to determine what line(s) referenced or modified a data item. With the XREF output, you can also determine the context in which a procedure is referenced in your program. For example, you can determine whether a verb was used in a PERFORM block or as part of a USE FOR DEBUGGING declarative. (The context of the procedure reference is indicated by the characters preceding the line number.)

```
DATA VALIDATION AND UPDATE PROGRAM                          IGYCARPA  Date 06/16/1998  Time 13:41:27   Page    50


An "M" preceding a data-name reference indicates that the data-name is modified by this reference.

    1         2                            3

 Defined   Cross-reference of data names   References

     264   ABEND-ITEM1
     265   ABEND-ITEM2
     347   ADD-CODE . . . . . . . . . . .  1126 1192
     381   ADDRESS-ERROR. . . . . . . . .  M1156
     280   AREA-CODE. . . . . . . . . . .  1266 1291 1354 1375
     382   CITY-ERROR . . . . . . . . . .  M1159

       4

Context usage is indicated by the letter preceding a procedure-name reference.
These letters and their meanings are:
    A = ALTER (procedure-name)
    D = GO TO (procedure-name) DEPENDING ON
    E = End of range of (PERFORM) through (procedure-name)
    G = GO TO (procedure-name)
    P = PERFORM (procedure-name)
    T = (ALTER) TO PROCEED TO (procedure-name)
    U = USE FOR DEBUGGING (procedure-name)

    5         6                            7

 Defined   Cross-reference of procedures   References

     877   000-DO-MAIN-LOGIC
     943   050-CREATE-VSAM-MASTER-FILE. .  P879
     995   100-INITIALIZE-PARAGRAPH . . .  P881
    1471   1100-PRINT-I-F-HEADINGS. . . .  P926
    1511   1200-PRINT-I-F-DATA. . . . . .  P928
    1573   1210-GET-MILES-TIME. . . . . .  P1540
    1666   1220-STORE-MILES-TIME. . . . .  P1541
    1682   1230-PRINT-SUB-I-F-DATA. . . .  P1562
    1706   1240-COMPUTE-SUMMARY . . . . .  P1563
    1052   200-EDIT-UPDATE-TRANSACTION. .  P890
    1154   210-EDIT-THE-REST. . . . . . .  P1145
    1189   300-UPDATE-COMMUTER-RECORD . .  P893
    1237   310-FORMAT-COMMUTER-RECORD . .  P1194 P1209
    1258   320-PRINT-COMMUTER-RECORD. . .  P1195 P1206 P1212 P1222
    1318   330-PRINT-REPORT . . . . . . .  P1208 P1232 P1286 P1310 P1370 P1395 P1399
    1342   400-PRINT-TRANSACTION-ERRORS .  P896
```

Figure 117. Example of XREF Output—Data-Name Cross-References

The numbers used in the explanation below correspond to the numbers in Figure 117.

Cross-Reference of Data-Names

1 Line number where the name was defined

2 Data-name

3 Line numbers where the name was used. If an “M” precedes the line number, the data item was explicitly modified at the location

Cross-Reference of Procedure References

4 Explanations of the context usage codes for procedure references

5 Line number where the procedure-name is defined

6 Procedure-name

7 Line numbers where the procedure is referenced and the context usage code for the procedure

The XREF compiler option also provides you with a sorted cross-reference listing of program names in your main program.

```
PP 5686-068 IBM COBOL for VSE/ESA 1.1.1                      NESTED    Date 06/16/1998  Time 13:41:27   Page     4

   1        2                          3

 Defined   Cross-reference of programs     References


 EXTERNAL   EXTERNAL1. . . . . . . . . . . 25
       2   X. . . . . . . . . . . . . . .  41
      12   X1 . . . . . . . . . . . . . .  33 7
      20   X11. . . . . . . . . . . . . .  25 16
      27   X12. . . . . . . . . . . . . .  32 17
      35   X2 . . . . . . . . . . . . . .  40 8
```

Figure 118. Example of XREF Output - Program Cross-Reference

The numbers used in the explanation below correspond to the numbers in Figure 118.

1 The line number where the program-name was defined. If the program is external, the word EXTERNAL will be displayed instead of a definition line number

2 The program name

3 Line numbers where the program is referenced

Using an Embedded Cross-Reference

The XREF compiler option also provides you with a modified cross-reference embedded in the source listing. This embedded cross-reference provides the line number where the data-name or procedure-name was defined.

```
DATA VALIDATION AND UPDATE PROGRAM                          IGYCARPA  Date 06/16/1998  Time 13:41:27   Page     2
  LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Map and Cross Reference
   ⋮
  000876               procedure division.
  000877                 000-do-main-logic.
  000878                   display "PROGRAM IGYCARPA - Beginning"
  000879                   perform 050-create-vsam-master-file.                                  943
  000880                   display "perform 050-create-vsam-master finished".
  000881                   perform 100-initialize-paragraph                                      995
  000882                   display "perform 100-initialize-paragraph finished"                        1
  000883                   read update-transaction-file into ws-transaction-record               204 338
  000884                       at end
  000885     1               set transaction-eof to true                                       252
  000886                   end-read

   ⋮

  000995                 100-initialize-paragraph.
  000996                   move spaces to ws-transaction-record                                IMP   2
  000997                   move spaces to ws-commuter-record                                   IMP 314
  000998                   move zeros to commuter-zipcode                                      IMP 325
  000999                   move zeros to commuter-home-phone                                   IMP 326
  001000                   move zeros to commuter-work-phone                                   IMP 327
  001001                   move zeros to commuter-update-date                                  IMP 331
  001002                   open input update-transaction-file                                  204
  001003                       location-file                                                   193
  001004                       i-o commuter-file                                               181
  001005                       output print-file                                               217

   ⋮

  001471               1100-print-i-f-headings.
  001472
  001473                   open output print-file.                                             217
  001474
  001475                   move function when-compiled to when-comp.                           IFN 696
  001476                   move when-comp (5:2) to compile-month.                              696 638
  001477                   move when-comp (7:2) to compile-day.                                696 640
  001478                   move when-comp (3:2) to compile-year.                               696 642
  001479
  001480                   move function current-date (5:2) to current-month.                  IFN   2
  001481                   move function current-date (7:2) to current-day.                    IFN 649
  001482                   move function current-date (3:2) to current-year.                   IFN 651
  001483
  001484                   write print-record from i-f-header-line-1                           222 633
  001485                         after new-page.                                               139
  001486

   ⋮
```

Figure 119. Example of an Embedded Cross-Reference

The numbers used in the explanation below correspond to the numbers in Figure 119.

1 The line number of the definition of the data-name or procedure-name in the program.

2 Special definition symbols. These symbols are:

- **UND** The user name is undefined
- **DUP** The user name is defined more than once
- **IMP** An implicitly defined name, such as special registers and figurative constants
- **IFN** An intrinsic function reference
- **EXT** An external reference
- ***** The program-name is unresolved because the NOCOMPILE option is in effect

Chapter 19. Program Tuning

Improving a program is always possible, but no program deserves limitless effort.

Before getting involved in COBOL details, examine the underlying algorithms for your program. For top performance, a sound algorithm is essential. The classic example is sorting, where a simple technique to sort a million items can take hundreds of thousands times longer than one using a sophisticated algorithm.

After deciding on the algorithm, 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. After you have improved the algorithm and data structures, consider other details of the COBOL source code.

The best COBOL programs are those that are easily understood. When a program is comprehensible, you can assess its performance. If the program has a tangled control flow, then it will be difficult to both understand and maintain. The optimizer will also be limited when trying to improve the code.

Note: Although COBOL/VSE allows segmentation language, you will not improve storage allocation by using it, because COBOL/VSE does not perform overlay.

The information in this chapter will help you write programs that result in better generated code sequences and that use system services better. This chapter describes three general areas that affect program performance:

- Coding techniques and considerations
- Optimization
- Compiler options

Coding Techniques and Considerations

The performance of your program can generally be improved through careful use of coding techniques. Some of these techniques also have an influence on the actions the optimizer takes when trying to improve code efficiency.

Careful consideration and implementation of these techniques will generally have a positive influence on your program's performance.

Programming Style

The coding style you use can, in certain circumstances, affect how the optimizer treats your code.

Structured Programming

The structured programming statements, such as EVALUATE and in-line PERFORM, make the program more comprehensible and also generate a linear control flow. This allows the optimizer to operate over larger regions of the program.

Avoid using the following constructs:

- The ALTER statement

- Backward branches (except as needed for loops for which PERFORM is unsuitable)
- PERFORM procedures that involve irregular control flow; for example, a PERFORM procedure such that control cannot pass to the end of the procedure and therefore cannot return to the PERFORM statement

Use top-down programming constructs. Out-of-line PERFORM statements are a natural means of implementing top-down programming techniques. With the optimizer, out-of-line PERFORM statements can be as efficient as in-line PERFORM statements in many cases, since the linkage code may be simplified or eliminated altogether.

Factoring Expressions

Factoring can save a lot of computation; for example, this 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
```

is more efficient than this code:

```
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** do factoring for you.

Symbolic Constants

If you want the optimizer to recognize a data item as a constant throughout the program, initialize it with a VALUE clause and do not modify it anywhere in the program.

If you pass a data item to a subprogram BY REFERENCE, the optimizer considers it to be an external data item and assumes that it is modified at every subprogram call.

If you move a literal to a data item, the optimizer recognizes it as a constant, but only in a limited region of the program following the MOVE statement.

Recognizing Constant Calculations

When several items of an expression are constant, ensure that the optimizer is permitted to optimize them. For evaluating expressions, the compiler is bound by the left-to-right evaluation rules of COBOL. Consequently, either move all the constants to the left end of the expression, or group them inside parentheses. For example, given that `V1`, `V2`, and `V3` are variables and `C1`, `C2`, and `C3` are constants, the expressions:

```
V1 * V2 * V3 * (C1 * C2 * C3)
C1 + C2 + C3 + V1 + V2 + V3
```

that contain the constant calculations:

```
C1 * C2 * C3 and C1 + C2 + C3
```

respectively, are preferable to those that contain none:

```
V1 * V2 * V3 * C1 * C2 * C3
V1 + C1 + V2 + C2 + V3 + C3
```

Often, in production programming, there is a tendency to place invariant factors on the right-hand end of expressions. This can result in less efficient code because optimization is lost.

Recognizing Duplicate Calculations

When several components of different expressions are duplicates, make sure the compiler is permitted to optimize them. For evaluating arithmetic expressions, the compiler is bound by the left-to-right evaluation rules of COBOL. Consequently, either move all the duplicates to the left ends of the expressions, or group them inside parentheses. Given that `V1` through `V5` are all variables, the calculation, `V2 * V3 * V4`, is a duplicate (known as a common subexpression) between the following two statements:

```
COMPUTE A = V1 * (V2 * V3 * V4)
COMPUTE B = V2 * V3 * V4 * V5
```

In the following example, the common subexpression is `V2 + V3`:

```
COMPUTE C = V1 + (V2 + V3)
COMPUTE D = V2 + V3 + V4
```

There are no common subexpressions in these examples:

```
COMPUTE A = V1 * V2 * V3 * V4
COMPUTE B = V2 * V3 * V4 * V5
COMPUTE C = V1 + (V2 + V3)
COMPUTE D = V4 + V2 + V3
```

Given that the optimizer can eliminate duplicate calculations, there is no need for you to introduce artificial temporaries. The program is often more comprehensible without them.

Use of Data

In certain circumstances, the use of one data type over another can be more efficient. Using consistent data types can reduce the need for conversions when performing operations on data items.

Computational Data Items

When a data item is used mainly for arithmetic or as a subscript, specify USAGE BINARY on the data description entry for the item. The hardware operations for manipulating binary data are faster than those for manipulating decimal data.

Note, however, that if a fixed-point arithmetic statement has intermediate results with a large precision (number of significant digits), the compiler will use decimal arithmetic anyway, after converting the operands to PACKED-DECIMAL form. For fixed-point arithmetic statements, the compiler will normally use binary arithmetic for simple calculations with binary operands, if the precision remains at 8 digits or fewer. Above 18 digits, the compiler always uses decimal arithmetic. With a precision of 9 to 18 digits, the compiler may use either form.

For a BINARY data item, the most efficient code is generated if the item has:

A sign (an S in its PICTURE clause), and

8 digits or fewer

When a data item is used for arithmetic, but is larger than 8 digits or is also used with DISPLAY data items, then PACKED-DECIMAL is a good choice. 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.

For a PACKED-DECIMAL data item, the most efficient code is generated if the item 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 digits or fewer, because a library routine call must be used to multiply or divide larger items

Consistent Kinds of Data

When performing 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. Also, a substantial amount of code may have to be executed to make operands conformable; for example, scaling one of the operands to give it the appropriate number of decimal places. This can largely be avoided by giving them both the same usage and also appropriate PICTURE specifications. That is, two numbers to be compared, added, or subtracted should not only have the same usage but also the same number of decimal places (9s after the V in the PICTURE clause).

Implications of External Data

External data items contain every data item in the Linkage Section, together with any data items that are passed BY REFERENCE to subprograms. External data items also refer to any data items that are defined with the EXTERNAL clause, and also to external (UPSI) switches. Each level-01 data item in the Linkage Section is separately addressed. Consequently, the program must load a base register each time one of these data items is referenced. By contrast, you can reference each 4K block of data items in the Working-Storage Section from the same base register. Also, the optimizer must assume that any called subprogram can modify any external data item and inhibit the optimization (across subprogram calls) of expressions containing external data items.

Coding Data Files

When using SAM files, use blocking whenever possible (the BLOCK CONTAINS clause). This will significantly reduce the file processing time. See “Block Sizes” on page 139 for additional information.

When writing to variable-length blocked sequential files, use the APPLY WRITE-ONLY clause for the file or use the AWO compiler option. With AWO specified, APPLY WRITE-ONLY will be in effect for all files within the program that are physically sequential with V-mode records. This can reduce the number of calls to Data Management Services. For additional information, see “APPLY WRITE-ONLY Clause” on page 24.

When using VSAM files, increase the number of data buffers for sequential access or index buffers for random access. Also, select a

control interval size (CISZ) appropriate for the application (smaller CISZ results in faster retrieval for the random processing at the expense of inserts, whereas a larger CISZ is more efficient for sequential processing). If you use alternate indexes, using access method services to build them is more efficient than using the AIXBLD run-time option. For a complete description of this run-time option. see the *LE/VSE Programming Reference*.

For better performance, access the records sequentially and avoid using multiple alternate indexes when possible. For more suggestions, see "Considerations for VSAM Performance" on page 173.

Coding Data Types

Avoid using USAGE DISPLAY data items in areas that are heavily used for calculations. See "Use of Data" on page 328 for additional information.

When using COMP-3 data items in calculations, use 15 or fewer digits in the PICTURE specification to avoid the use of library routines for multiplication and division. See "Use of Data" on page 328 for additional information.

Plan the use of fixed-point and floating-point data types. For more information, see "Planning the Use of Fixed-Point and Floating-Point Data Types."

Using indexes to address a table is more efficient than using subscripts since the index already contains the displacement from the start of the table and does not have to be calculated at run time.

When using subscripts (as opposed to indexes) to address a table, use a binary (COMP) signed data item with 9 or fewer digits. Additionally, in some cases, using 4 or fewer digits for the data item may also reduce CPU time. For additional information, see "Referring to an Item in a Table" on page 96, "Efficient Coding for Tables" on page 114, and "Use of Data" on page 328.

When using OCCURS DEPENDING ON (ODO) data items, ensure that the ODO objects are binary (COMP) to avoid unnecessary conversions each time the variable-length items are referenced. When using ODO data items, you may experience performance degradation since special code must be executed each time a variable length data item is referenced. If you do use variable-length data items, copying them into a fixed-length data item prior to a period of high-frequency use can reduce some of this overhead. See page 333 for additional ODO information.

Planning the Use of Fixed-Point and Floating-Point Data Types

You can enhance program performance by carefully determining when to use fixed-point and floating-point data types.

Arithmetic Expressions

Calculation of arithmetic expressions that are evaluated in floating-point mode are most efficient when the operands involved require the least amount of conversion. Using operands that are COMP-1 or COMP-2 produces the most efficient code.

Integer items declared as BINARY or PACKED DECIMAL, with 9 or fewer digits, can be quickly converted to floating-point data. Also, con-

version from a COMP-1 or COMP-2 item to a fixed-point integer with 9 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.

Exponentiations

Exponentiations with large exponents can be evaluated more quickly and with more accurate results using floating point. For example,

```
COMPUTE fixed-point1 = fixed-point2 ** 100000
```

could be computed more quickly and accurately if coded as

```
COMPUTE fixed-point1 = fixed-point2 ** 100000.E+00
```

since the presence of a floating-point exponent would cause floating-point arithmetic to be used to compute the exponentiation.

Table Handling

Table handling operations can be a major part of an application. There are several techniques that can improve the efficiency of these operations and also influence the effects of the optimizer.

Subscript Calculations

For the table element reference:

```
ELEMENT(S1 S2 S3)
```

where `S1`, `S2`, and `S3` are subscripts, the compiler must evaluate the following expression:

```
comp_s1 * d1 + comp_s2 * d2 + comp_s3 * d3 + base_address
```

where:

comp_s1
The value of S1 after conversion to binary, and so on.

d1, d2, and d3
The **strides** for each dimension.

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).

Although the expression could be evaluated in any order, the compiler evaluates it in left-to-right order. Thus, the optimizer will find the most opportunities for eliminating calculations if the constant or duplicate subscripts are the leftmost.

You can further optimize table element references by declaring any subscripts as COMPUTATIONAL.

Recognizing Constant Subscript calculations

Suppose that `C1`, `C2`,. . . are constant data items and that `V1`, `V2`,. . . are variable data items. Then, in the following table element reference, only the individual terms `comp_c1 * d2` and `comp_c2 * d3` are eliminated as constant:

```
ELEMENT(V1 C1 C2)
```

In **this** table element reference, however, the entire subexpression `comp_c1 * d1 + comp_c2 * d2` is constant.

```
ELEMENT(C1 C2 V1)
```

Consequently, try to use your tables so that constant subscripts are the leftmost subscripts. If all the subscripts are constant, then no subscript calculation is done at object time, as in the following table element reference:

```
ELEMENT(C1 C2 C3)
```

With the optimizer, the reference can be as efficient as a reference to a scalar (nontable) item.

Recognizing Duplicate Subscript Calculations

Similar considerations apply to duplicate subscript calculations. Between the following two table element references, only the individual terms `comp_v3 * d2` and `comp_v4 * d3` are common subexpressions:

```
ELEMENT(V1 V3 V4)
ELEMENT(V2 V3 V4)
```

However, the entire subexpression `comp_v1 * d1 + comp_v2 * d2` is common between **these** two table element references:

```
ELEMENT(V1 V2 V3)
ELEMENT(V1 V2 V4)
```

So, even when all the subscripts are variables, try to use your tables so that it is the rightmost subscript that varies most often for references that occur close to each other in the program. This will also improve the pattern of storage reference as well as paging.

If all the subscripts are duplicates, then the entire subscript calculation is a common subexpression, as in the two references:

```
ELEMENT(V1 V2 V3)
ELEMENT(V1 V2 V3)
```

So, 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.

Recognizing Constant and Duplicate Index Calculations

Index calculations are similar to subscript calculations, except that no multiplication need be done, since index values have the stride factored into them. The recommendations for optimizing subscripted references apply unchanged to indexed references. Index calculations involve loading the indexes into registers, and these data transfers can be optimized, much as the individual subscript calculation terms are optimized.

Tables with Identical Element Specifications

When subscripting or indexing different tables, it is most efficient if all tables have the same element length. With equal element lengths, the stride for the last dimension of the tables will be equal. The rightmost subscript or index computed for one table can then be reused by the optimizer for the others. So, consider defining tables to match the element length of the other tables with which they are involved.

If both the element lengths, **and** the number of occurrences in each dimension are equal, then the strides for dimensions other than the last will also be equal, resulting in greater commonality between their sub-

script calculations. The optimizer can then reuse subscripts or indexes other than the rightmost.

Indexing Preferred to Subscripting

The optimizer can eliminate duplicate subscript (and index) calculations so that repeated references to an element of a table are equally efficient with subscripting and indexing. However, the original reference to a particular table element is more efficient with indexes than with subscripts, even if the subscripts are COMPUTATIONAL. This is because 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.

Introduction of Artificial Indexing Temporaries

Relative indexing can be more efficient than direct indexing, so there is no merit in keeping alternative indexes with the offset factored in. This is because:

```
ELEMENT (I + 5, J - 3, K + 2)
```

is no more costly than:

```
ELEMENT (I5, J3, K2)
```

except that the second example requires this prerequisite processing:

```
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 makes the direct indexing less efficient than the relative indexing.

Implications of OCCURS DEPENDING ON (ODO)

A group item that contains a subordinate OCCURS DEPENDING ON data item has a variable length.

The program must execute special code every time a variable-length data item is referenced.

Because this code is out-of-line, it may interrupt optimization. Furthermore, the code to manipulate variable-length data items is substantially less efficient than that for fixed-size data items and may result in a significant increase in processing time. For instance, the code to compare or move a variable-length data item may involve calling a library routine, and is significantly slower than the equivalent code for fixed-length data items.

To avoid unnecessary conversions when variable-length items are referenced, specify BINARY for OCCURS . . . DEPENDING objects.

Copy variable-length data items into fixed-length data items prior to a period of high-frequency access.

See “Subscripting Using Index-Names (Indexing)” on page 97 for information on subscripting and indexing.

Optimization

This section discusses the benefits of the OPTIMIZE compiler option as well as other compiler features affecting optimization.

The OPTIMIZE Compiler Option

The COBOL/VSE optimizer is activated by specifying the OPTIMIZE compiler option. The following optimization takes place if you specify OPTIMIZE or OPTIMIZE(STD):

- Eliminate unnecessary transfers of control or simplify inefficient branches, including those generated by the compiler that are not evident from looking at the source program.
- Simplify the compiled code for both a PERFORM statement and a CALL statement to a contained (nested) program. Where possible, the optimizer places the statements in-line, eliminating the need for linkage code. This optimization is known as “procedure integration”, and is further discussed below. 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 calculations such as subscript calculations by saving the results for later reuse.
- Eliminate constant calculations by performing them when the program is compiled.
- 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 executed (unreachable code elimination).

You can request the following conditions/optimization if you specify the OPTIMIZE(FULL) option:

- Delete from the program any unused items

To see how the optimizer works on your program, compare the generated code with and without the OPTIMIZE option. (You can request the assembler language listing of the generated code by specifying the LIST compiler option.)

For unit testing your programs, you may find it easier to debug code that has not been optimized. But when the program is ready for final test, specify OPTIMIZE, so that the tested code and the production code are identical. You may also want to use the option during development, if a program is used frequently without recompilation. However, the overhead for OPTIMIZE may outweigh its benefits if you recompile frequently, unless you are using the assembler language expansion (LIST option) to fine tune your program.

PERFORM Procedure Integration

PERFORM procedure integration is the process whereby a PERFORM statement is replaced by its performed procedures. The advantage here 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 PERFORM statement, then the program may become larger. The optimizer limits this increase to no more than 50 percent, after which it no longer uses procedure integration. However, if you are concerned about program size, you may want to prevent procedure integration in specific instances. You can do this 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.

For example, all the PERFORM statements in the following program will be transformed by procedure integration:

```
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:

```
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.
```

Whereas in this 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.
```

Unreachable code elimination: Because of procedure integration, one PERFORM procedure may be repeated several times. As further optimization proceeds on each copy of the procedure, portions may be found to be unreachable, depending on the context into which the code was copied. A warning message is generated for each such occurrence.

Contained Program Procedure Integration

Contained program procedure integration is the process where a CALL to a contained program is replaced by the program code. The advantage here is that the resulting program runs faster without the overhead of CALL linkage and with more linear control flow.

Program size: If the contained programs are invoked by several CALL statements and replace each such CALL statement, then the program may become larger. The optimizer limits this increase to no more than 50 percent, after which it no longer uses procedure integration. The optimizer then chooses the next best optimization for the CALL statement; the linkage overhead can be as few as two instructions.

Unreachable code elimination: As a result of procedure integration, one contained program may be repeated several times. As further optimization proceeds on each copy of the program, portions may be found to be unreachable, depending on the context into which the code was copied. When this happens, a warning message is generated.

Other Compiler Features that Affect Optimization

Another compiler feature that may have a significant influence on the effects of the optimizer option is the USE FOR DEBUGGING ON ALL PROCEDURES statement.

The ON ALL PROCEDURES option of the USE FOR DEBUGGING statement generates extra code at each transfer to every procedure-name. It can be very useful for debugging, but can make the program significantly larger, and can substantially inhibit optimization.

Additionally, compiler options can also have an effect (see “Compiler Options” on page 337 for details).

Compiler Options

You may have a customized system that requires certain options for optimum performance. Check with your system programmer to ensure that installed options are not required before changing defaults. You can see what your system defaults are by issuing ERRMSG. See "Generating a List of All Compiler Error Messages" on page 223 for instructions on issuing ERRMSG.

The tuning methods and performance information discussed here are intended to help you select from various COBOL/VSE options for compiling your programs.

Important

Make sure that COBOL/VSE serves your needs. Please confer with system programmers on COBOL/VSE tuning considerations. Doing so will ensure that the options you choose are appropriate for programs being developed at your site.

A brief description of each item is followed by performance advantages and disadvantages, reference information, and usage notes where applicable. Refer to specified pages for additional information.

AWO The AWO compiler option allows the file buffer to be written to the output device. When you use AWO, the APPLY WRITE-ONLY clause is in effect for all SAM files in the program with V-mode records.

Performance advantages

Using the AWO option can result in a performance savings because this will generally result in fewer calls to the system file management routines to handle the I/O.

Performance disadvantages

In general, no performance disadvantages.

Reference information

For a description of the APPLY WRITE-ONLY clause, see "APPLY WRITE-ONLY Clause" on page 24. For syntax of the AWO option, see "AWO" on page 227.

DYNAM The DYNAM compiler option dynamically loads subprograms invoked through the CALL statement at run time.

Performance advantages

Using DYNAM means easier subprogram maintenance since the application will not have to be relink-edited if the subprogram is changed.

When using the DYNAM option, you can free virtual storage that is no longer needed by issuing the CANCEL statement.

Performance disadvantages

You pay a slight performance penalty using DYNAM since the call must go through an LE/VSE routine.

Reference information

For a description of the DYNAM option, see "DYNAM" on page 233.

FASTSRT The FASTSRT compiler option specifies that the DFSORT/VSE product (or equivalent) will handle all of the I/O.

Performance advantages

FASTSRT eliminates the overhead of returning to COBOL/VSE after each record is processed.

Performance disadvantages

No performance disadvantages.

Reference information

For more information on improving sort performance, see "Improving Sort Performance with FASTSRT" on page 184. The FASTSRT syntax appears in "FASTSRT" on page 234.

Usage notes

Using FASTSRT is recommended when direct work files are used for the sort work files. Not all sorts are eligible for this option.

NUMPROC(PFD), (NOPFD), (MIG) Use this compiler option for sign processing when coding numeric comparisons.

Performance advantages

NUMPROC(PFD) generates significantly more efficient code for numeric comparisons.

Performance disadvantages

For most references to COMP-3 and DISPLAY numeric data items, using NUMPROC(MIG) and NUMPROC(NOPFD) causes extra code to be generated because of sign "fix up" processing. This extra code may also inhibit some other types of optimizations.

Reference information

See "Sign Representation and Processing" on page 79 for sign representation information, and "NUMPROC" on page 243 for the NUMPROC syntax.

Usage notes

When using 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, using NUMPROC(PFD) may not be applicable for all programs. For performance-sensitive applications, the use of NUMPROC(PFD) is recommended where possible.

For noneligible programs, using NUMPROC(MIG) has less sign fixup than NUMPROC(NOPFD).

OPTIMIZE Use the OPTIMIZE compiler option to ensure your code is optimized for better performance.

Performance advantages

Generally results in more efficient run-time code.

Performance disadvantages

OPTIMIZE requires more processing time for compiles than NOOPTIMIZE.

Reference information

For further description of OPTIMIZE, see “The OPTIMIZE Compiler Option” on page 334. See “OPTIMIZE” on page 245 for the OPTIMIZE syntax.

Usage notes

NOOPTIMIZE is generally used during program development when frequent compiles are necessary, and it also allows for easier debugging. For production runs, however, the use of OPTIMIZE is recommended.

RENT Use the RENT compiler option to generate a reentrant program.

Performance advantages

Using RENT enables the program to be placed in the Shared Virtual Area (SVA) for running above the 16-megabyte line on an extended addressing system.

Performance disadvantages

Using RENT generates additional code to ensure that the program is reentrant.

Reference information

The syntax for the RENT option appears in “RENT” on page 247.

RMODE(ANY) The RMODE(ANY) compiler option allows the program to be loaded anywhere.

Performance advantages

Using the RMODE(ANY) option with NORENT programs allows the program and its Working-Storage to be located above the 16-MB line, relieving storage below the line.

Performance disadvantages

In general, no performance disadvantages.

Reference information

For a description of the RMODE compiler option, see “RMODE” on page 247.

SSRANGE The SSRANGE compiler option verifies that all subscripts, indexes, and reference modification expressions are within proper bounds.

Performance advantages

No performance advantages.

Performance disadvantages

SSRANGE generates additional code for verifying subscripts.

Reference information

The syntax for the SSRANGE option appears in “SSRANGE” on page 250.

Usage notes

In general, if you only need to verify the subscripts a few times in the application instead of at every reference, coding your own checks may be faster than using the SSRANGE compiler option. SSRANGE may be turned off at run time with the CHECK(OFF) run-time option. For performance-

sensitive applications, the use of the NOSSRANGE compiler option is recommended.

TEST The TEST compiler option with any hook-location suboption other than NONE (i.e. ALL, STMT, PATH, BLOCK) produces object code that can take full advantage of and be run under a Debug Tool under MVS and VM.

Performance advantages

No performance advantages.

Performance disadvantages

Since the TEST compiler option with any hook-location sub-option other than NONE 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 the performance degradation may be).

Reference information

The syntax for TEST appears in its description under “TEST” on page 251.

Usage notes

TEST without a hook-location suboption or with any one other than NONE forces the NOOPTIMIZE compiler option into effect. For production runs, the use of NOTEST is recommended. However, if during the production run, you want a symbolic dump of the variables in a formatted dump if the program abends, compile with TEST(NONE,SYM).

TRUNC(STD), (OPT), (BIN) This compiler option creates code that will shorten the receiving fields of arithmetic operations.

Performance advantages

TRUNC(OPT) does not generate extra code and generally improves performance.

Performance disadvantages

Both TRUNC(BIN) and TRUNC(STD) generate extra code whenever a BINARY data item is changed. TRUNC(BIN) is usually the slowest of these options.

Reference information

For a more detailed explanation of the TRUNC option, see “TRUNC” on page 252.

Usage notes

TRUNC(STD) conforms to the COBOL 85 Standard, whereas TRUNC(BIN) and TRUNC(OPT) do not. When using TRUNC(OPT), the compiler assumes that the data conforms to the PICTURE and USAGE specifications. For performance-sensitive applications, the use of TRUNC(OPT) is recommended where possible.

Other Product Considerations

CICS

If the application is running under the Customer Information Control System (CICS), converting EXEC CICS LINKs to COBOL CALLs can improve transaction response time.

See “COBOL/VSE Programming Considerations for CICS” on page 388 for additional information on coding COBOL programs that run under CICS.

Performance Tuning Worksheet

This worksheet will help you evaluate your program's performance. If you answer **yes** to each question, you are probably improving your performance. However, be sure you understand the function of each option before considering the performance trade-off. You may prefer function over improved performance in many instances.

Figure 120. Performance Tuning Worksheet

AWO	Are you using the AWO option when possible? Refer to page 337.	_____
DYNAM	Are you using NODYNAM? Consider the performance trade-offs. Refer to page 337.	_____
FASTSRT	When using direct work files for the sort work files, have you selected the FASTSRT option? Refer to page 338.	_____
NUMPROC	Are you using NUMPROC(PFD) where possible? Refer to page 338.	_____
OPTIMIZE	Are you using OPTIMIZE for production runs? Refer to page 338.	_____
RENT	Are you using NORENT? Consider the performance trade-offs. Refer to page 339.	_____
RMODE(ANY)	Are you using RMODE(ANY) with your NORENT programs? Consider the performance trade-offs with storage usage. Refer to page 339.	_____
SSRANGE	Are you using NOSSRANGE for production runs? Refer to page 339.	_____
TEST	Are you using NOTEST for production runs? Refer to page 340.	_____
TRUNC	Are you using TRUNC(OPT) where possible? Refer to page 340.	_____

Run-Time Performance Considerations

In addition to coding techniques and compiler options, the characteristics of your run-time environment also influence program performance. For information on how the various run-time options and other run-time considerations control the execution of your compiled program, see the *LE/VSE Programming Guide* and *LE/VSE Programming Reference*.

Chapter 20. Techniques to Improve Programmer Productivity

Improving your programming productivity can be as valuable to your installation's resource management as coding effective and efficient programs. The techniques discussed in this chapter can help reduce some tedious and time-consuming coding tasks and allow you to use your time and system resources more effectively.

Using these suggestions may not always be possible or practical. Refer to your organization's guidelines and requirements before using the following techniques.

The techniques described in this chapter are:

- Eliminate repetitive coding using the COPY facility
- Make program changes using the REPLACE statement
- Simplify complex coding using intrinsic functions and callable services
- Find coding errors
- Control the content of the output listing
- Use the Debug Tool to help debug your programs

Eliminating Repetitive Coding (the COPY Facility)

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 sublibrary. Then, you can retrieve these code sequences from this library and have them included in your program at compile time by using the COPY or BASIS statements.

You can use INSERT or DELETE statements, or the REPLACING phrase of the COPY statement to modify the included text.

Note: If you use the EXIT compiler option to specify a LIBEXIT phase, your results may differ from those presented in this chapter.

Source books are searched for in the sublibraries specified by the user in the LIBDEF SOURCE chain.

At compile time, the librarian searches all sublibraries identified by the LIBDEF JCL statements; for example:

```
// LIBDEF SOURCE,SEARCH=(lib.sublib)
```

Retrieving Source Statements

Members of the COPY sublibrary can be retrieved using COPY or BASIS statements.

COPY Statement

The COPY statement allows you to include stored source statements in any part of your program. For example, if 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 member CFILEA by using the COPY statement in the Data Division of your source program code as follows:

```
FD FILEA
            COPY CFILEA.
```

The library entry is copied into your program, and the resulting program listing looks as follows:

```
    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 is printed on a separate line, and copied lines are preceded by a "C".

Assume that a member named DOWORK was stored with the following statements:

```
   COMPUTE QTY-ON-HAND = TOTAL-USED-NUMBER-ON-HAND
   MOVE QTY-ON-HAND to PRINT-AREA
```

To retrieve the stored member, DOWORK, write:

```
paragraph-name. COPY DOWORK.
```

The statements included in the DOWORK procedure will follow the *paragraph-name*.

Note: In order for the text copied to have a D inserted in column 7 (debugging line indicator), the D must appear on the first line of the COPY statement. A COPY statement itself can never be a debugging line; if it contains a D and WITH DEBUGGING mode is not specified, the COPY statement will nevertheless be processed.

BASIS Statement

Frequently used source programs, such as payroll programs, can be inserted into the COPY sublibrary. The BASIS statement brings in an entire source program at compile time. Calling in a program eliminates the need to handle a program each time you compile it.

You may modify any statement in the source program by referring to its COBOL sequence number with an INSERT or DELETE statement. The use of BASIS to support INSERT and DELETE requires that sequence fields contain only numeric characters.

- INSERT adds new source statements after the sequence number indicated.
- DELETE eliminates the statements indicated by the sequence numbers.

You may delete a single statement with one sequence number, or you may delete more than one statement by indicating the first and last sequence numbers to be deleted, separated by a hyphen.

Note: The COBOL sequence number is the 6-digit number that you assign in columns 1 through 6 of the source statements. The COBOL sequence numbers are used to update COBOL source statements at compile time. Such changes are in effect for the one compilation only.

For example, assume that a company payroll program is kept as a source program in the COPY library. The name of the program is PAYROLL. During a particular year, retirement tax is taken out at a rate of 2.5% each week for all personnel until earnings exceed $15,000. The coding for accomplishing this is shown in Figure 121.

```
000730   Evaluate True
000735     When Annual-Pay Greater Than 15000
000740       Move Zero To Tax-Pay
000745     When Annual-Pay Greater Than 15000 - Base-Pay
000750       Compute Tax-Pay = (15000 - Annual-Pay) * .025
000755     When Other
000760         Compute Tax-Pay = Base-Pay * .025
000765     End-Evaluate
000770     Add Base-Pay To Annual-Pay
            ⋮
000850     Stop Run.
```

Figure 121. COBOL Statements for Deducting Retirement Tax

Because of a change in the law, tax is now to be taken out until earnings exceed $17,800 and a new percentage is to be applied. You can code these changes, as shown in Figure 122.

```
   ⋮
Basis Payroll
Delete 000735-000760
000735      When Annual-Pay Greater Than 17800
000740        Move Zero To Tax-Pay
000745      When Annual-Pay Greater Than 17800 - Base-Pay
000750        Compute Tax-Pay = (17800 - Annual-Pay) * .044
000755      When Other
000760        Compute Tax-Pay = Base-Pay * .044
```

Figure 122. Changes to Source Program

The changed program will contain the coding shown in Figure 123 on page 346. The listing will have “I” in column 7 for inserted lines.

```
000730   Evaluate True
000735I    When Annual-Pay Greater Than 17800
000740I      Move Zero To Tax-Pay
000745I    When Annual-Pay Greater Than 17800 - Base-Pay
000750I      Compute Tax-Pay = (17800 - Annual-Pay) * .044
000755I    When Other
000760I      Compute Tax-Pay = Base-Pay * .044
000765   End-Evaluate
000770   Add Base-Pay To Annual-Pay.
            ⋮
000850   Stop Run.
```

Figure 123. COBOL Statements Changed to Source COPY Library Statements

Changes made through use of the INSERT and DELETE statements remain in effect for the one compilation only. The copy of PAYROLL in the library is not changed.

Making a Change to Your Program (the REPLACE Statement)

The REPLACE statement provides you with a means of applying a change to sections of COBOL source programs without having to manually find all places that need to be changed. It is an easy method of doing simple string substitutions. It is similar in action to the COPY statement, except that it acts on the entire source program, not just on the text in COPY libraries. See *COBOL/VSE Language Reference* for the format and syntax of the statement.

You can use the REPLACE statement to resolve any conflict between new reserved COBOL words and names you have used in your program. For example, if you have used the name DAY-OF-WEEK for a data item in your program, you will need to change all the occurrences of that name. To do this, insert the following REPLACE statement before the first line of your program:

```
REPLACE ==DAY-OF-WEEK== BY ==WEEKDAY==.
```

This will replace all occurrences of DAY-OF-WEEK in your program with the new name WEEKDAY. Because REPLACE statements are processed after any COPY statements are processed, it will also change any occurrences of DAY-OF-WEEK that were included in text from COPY libraries.

The rules for using the REPLACE statement are:

- The LIB compiler option must be specified
- REPLACE statements may not introduce COPY statements (although COPY statements may introduce REPLACE statements)
- REPLACE statements may not modify or introduce other REPLACE statements
- Any REPLACE statement is in effect from the point at which it is specified until:
  - it is explicitly turned off
  - the occurrence of another REPLACE statement
  - the end of the compiled program is reached

Simplifying Complex Coding and Other Programming Tasks

By using COBOL/VSE intrinsic functions and LE/VSE Callable Services, you can avoid having to code a lot of arithmetic or other complex tasks.

Intrinsic Functions

COBOL/VSE provides various string- and number-manipulation capabilities that can help you simplify your coding. For more information, see “Built-in (Intrinsic) Functions” on page 49 and “Numeric Intrinsic Functions” on page 83.

LE/VSE Callable Services

LE/VSE callable services simplify performing arithmetic and many other types of programming tasks. Invoked with standard CALL statements, there are many LE/VSE services that help you perform:

- Condition handling
- Dynamic storage management
- Date and time calculations
- Mathematical calculations
- Message handling
- National language support
- General services such as obtaining an LE/VSE formatted dump

To invoke an LE/VSE service, use a CALL statement with the appropriate parameters for that particular service:

```
Call "CEESSSQT" using argument, feedback-code, result
```

Where you define the variables in the CALL statement, in the Data Division of your program with appropriate data definitions required by the particular function you are calling:

```
77  argument          comp-1.
77  feedback-code     pic  x(12)  display.
77  result            comp-1.
```

In this example, LE/VSE service CEESSSQT calculates the value of the square root of the variable `argument` and returns this value in the variable `result`. The value returned in `feedback-code` indicates whether the service completed successfully. After a call to an LE/VSE service, the RETURN-CODE special register is set to 0 regardless of whether or not the service completes successfully.

The parameter for the return-code is generally the last parameter in the list, except for mathematical services where it is the second to last parameter.

For a complete list of, and for detailed information on, the syntax, parameter descriptions, and usage notes for each LE/VSE callable service, see the *LE/VSE Programming Reference*.

Condition Handling

LE/VSE condition handling provides facilities that allow COBOL/VSE applications to react to unexpected errors.

Note that you can use language constructs or run-time options to select the level at which you want to handle each condition. For example, you can decide to handle a

particular error in your COBOL program, let the LE/VSE condition handler take care of it, or percolate the error so that it is handled by the operating system. Only a truly catastrophic failure need disrupt your application environment.

In support of LE/VSE condition handling, COBOL/VSE adds support for procedure-pointer data items, as described under "Passing Entry Point Addresses with Procedure Pointers" on page 280.

Dynamic Storage Services

These services enable you to get, free, and reallocate storage. In addition, you can create your own user-defined storage pools.

Date and Time Calculations

With the date and time services, you can get the current local time and date in several formats, as well as perform date and time conversions. Two callable services, CEEQCEN and CEESCEN, provide a predictable way to handle 2-digit years, such as 95 for 1995 or 02 for 2002.

Mathematical Calculations

Calculations that are easy to perform with this type of callable service include logarithmic, exponential, trigonometric, square root, and integer functions.

Note: COBOL/VSE also supports a set of intrinsic functions that include some of the same mathematical and date functions. The LE/VSE callable services and intrinsic functions provide equivalent results for the same functions. See "Numeric Intrinsic Functions" on page 83 for an overview of numeric intrinsic functions and "LE/VSE Callable Services" on page 86 for an explanation of the differences between COBOL/VSE intrinsic functions and LE/VSE date and mathematical services.

Message Handling

Message handling services include getting, dispatching, and formatting messages. Messages for non-CICS applications can be directed to files or printers, while CICS messages are directed to a CICS transient data queue. LE/VSE takes care of splitting the message to accommodate the record length of the destination, as well as presenting the message in the correct national language, such as Japanese or English.

National Language Support

These services make it easy for your applications to support the language wanted 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 99, or 6,23,99.

General Callable Services

LE/VSE also offers a set of general callable services, which include the capability to get an LE/VSE formatted dump.

Depending upon the options you select, the LE/VSE formatted dump may contain the names and values of variables, as well as information about conditions, program tracebacks, control blocks, storage, and files. All LE/VSE dumps have a common, well-labeled, and easy-to-read format.

Sample List of LE/VSE Callable Services

The following table gives examples of a few callable services available with LE/VSE.

Note: Many more services are available than those listed in the table. For a complete list, see the *LE/VSE Programming Reference*.

Figure 124 (Page 1 of 2). LE/VSE Callable Services Available from COBOL/VSE.

Function Type	For Example:	
Date and Time	**CEECBLDY**	To convert a string representing a date into a COBOL integer date format. The COBOL integer date format represents a date as the number of days since 31 December 1600. The service is compatible with ANSI COBOL intrinsic functions.
	CEEQCEN, CEESCEN	To query and set the LE/VSE century window. These two callable services are valuable when one or more programs use two digits to express a year. That is, 03, can easily be interpreted as 2003 and not 1903.
	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.
Mathematical Services	**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.
Dynamic Storage Services	**CEEGTST**	To get storage.
	CEECZST	To change the size of a previously allocated storage block.
	CEEFRST	To free storage.
Condition Handling Services	**CEEHDLR**	To register a user condition handler.
	CEESGL	To raise or signal a condition.
	CEEMRCR, CEEMRCE	To indicate where the program will resume running after the condition handler has completed.
Message Handling Services	**CEEMOUT**	To dispatch a message.
	CEEMGET	To retrieve a message.
National Language Support Services	**CEE5LNG**	To change or query the current national language.
	CEE5CTY	To change or query the current national country.
	CEE5MCS	To obtain the default currency symbol for a given country.

Figure 124 (Page 2 of 2). LE/VSE Callable Services Available from COBOL/VSE.

Function Type	For Example:	
General Services	**CEE5DMP**	To obtain an LE/VSE formatted dump.
	CEETEST	To start a debug tool, such as that provided by Debug Tool/VSE.

Using LE/VSE Callable Services—An Example

Many callable services offer the COBOL programmer entirely new function that would require extensive coding using previous versions of COBOL. Two such services are CEEDAYS and CEEDATE, which you can use effectively when you want to format dates for output.

Figure 125 shows a sample COBOL program that uses LE/VSE services to format and display a date from the results of a COBOL ACCEPT statement.

```
 ID DIVISION.
 PROGRAM-ID. HOHOHO.
*************************************************************
* FUNCTION:  DISPLAY TODAY'S DATE IN THE FOLLOWING FORMAT: *
*           WWWWWWWWW, MMMMMMMM DD, YYYY                    *
*                                                           *
*           I.E.   SUNDAY, DECEMBER 25, 1994                *
*                                                           *
*************************************************************
 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).
 77   DAYSFC               PIC X(12).
 77   DATEFC               PIC X(12).
```

Figure 125 (Part 1 of 2). Example with LE/VSE Callable Services

```
 PROCEDURE DIVISION.
*************************************************************
*    USE  LE DATE/TIME CALLABLE SERVICES TO PRINT OUT       *
*    TODAY'S DATE FROM COBOL ACCEPT STATEMENT.              *
*************************************************************
     ACCEPT CHRDATE-STRING FROM DATE.

     MOVE "YYMMDD" TO PICSTR-STRING.
     MOVE 6 TO PICSTR-LENGTH.
     CALL "CEEDAYS" USING CHRDATE , PICSTR , LILIAN , DAYSFC.

     MOVE " WWWWWWWWWZ, MMMMMMMMMZ DD, YYYY " TO PICSTR-STRING.
     MOVE 50 TO PICSTR-LENGTH.
     CALL "CEEDATE" USING LILIAN , PICSTR , FORMATTED-DATE ,
           DATEFC.

     DISPLAY "******************************".
     DISPLAY FORMATTED-DATE.
     DISPLAY "******************************".

     STOP RUN.
```

Figure 125 (Part 2 of 2). Example with LE/VSE Callable Services

Using CEEDAYS and CEEDATE reduces the code required without LE/VSE.

Finding Coding Errors

Errors fall into two broad classes: those that the compiler can identify when it examines your source program, and those that surface later when you run the program. The second type of error, the run-time error, is often harder to track down.

For ideas on diagnosing and correcting run-time errors, see *LE/VSE Debugging Guide and Run-Time Messages*.

Checking for Errors, Using NOCOMPILE

After you have completed your design reviews and desk-checked your coding, you can check your program for syntax errors with the NOCOMPILE option. For information on how to specify a compiler option such as NOCOMPILE, see “Using Compiler Options” on page 216.

Errors the Compiler Can Identify

The compiler detects errors that are not dependent on run-time logic. While it can correct some of the errors it finds, the compiler cannot correct all of them. If your compilation fails, you need to fix the source program and compile it again.

Common coding mistakes include:

- Misspellings
- Faulty punctuation (especially missing, extra, or misplaced periods)
- Not observing COBOL formatting rules (Area A and Area B)
- Incorrect or incomplete syntax
- Using reserved words in data-definitions or paragraph headers

Sometimes mistakes go unnoticed at compile time because the statement makes sense to the compiler, even though it is not what you intended. These errors surface at run time.

Errors the Compiler Cannot Identify

Check for these mistakes in your program logic:

- Failing to match the record description in your source program with the record format on the file to be read. For example, a numeric field that should contain zero (numeric) actually contains blanks (alphanumeric).
- Trying to perform calculations on invalid data.
- Moving data into an item that is too small for it.
- Moving a group item to another group item when the subordinate data descriptions are incompatible.
- Specifying a USAGE clause for a redefining data item that is different from the USAGE specified for the item redefined, and not keeping track of the changed USAGE.
- Specifying subscript values that are not within the bounds of the table. (You can use the SSRANGE compiler option to check for this type of error.)
- Attempting an illogical I/O operation, such as reading a file that is opened OUTPUT, or closing a file that is not opened. (Test the file status code after each input/output.)
- Not defining a sign field for items that may hold negative values. (The sign is lost and what should have been a negative number becomes a positive number.)
- Not initializing the items in the Working-Storage Section with a value before using them.
- Not initializing counters and indexes.
- In a called program, incorrectly matching the data descriptions in the Linkage Section with those of the calling program.
- In the calling program, incorrectly identifying the data to be passed.

Controlling the Content of the Output Listing

The *CONTROL (*CBL) statement is an output listing control statement. With the SOURCE, MAP, and LIST compiler options, this statement selectively suppresses or allows production of output. The names ***CONTROL** and ***CBL** are synonymous; wherever one appears in the explanation that follows, the other may be substituted.

The characters ***CONTROL** or ***CBL** may start in any column, beginning with column 7, followed by at least one space or comma and one or more option key words. The option keywords must be separated by one or more spaces or commas. The *CONTROL statement must be the only statement on the line and may be terminated with a period. Continuation is not allowed.

Note: *CONTROL does **not** turn options on and off. It only allows listing control for those listing options in effect for the compilation. The keywords that may appear are: SOURCE and NOSOURCE, MAP and NOMAP, LIST and NOLIST.

The source line containing the *CONTROL statement does not appear in the source listing.

Your installation may set certain options to have fixed values that cannot be replaced for individual applications. These fixed options take precedence over options specified using:

PARM statement in JCL
JCL OPTION statement
PROCESS statement (synonym: CBL)
*CONTROL (*CBL)

(See “Using Compiler Options” on page 216 for more information on the order of precedence of compiler options.)

The requested options are handled in the following manner:

- If an option or its negation appears more than once in a *CONTROL statement, the last occurrence of the option keyword is used.
- If the corresponding option has been requested as a parameter to the compiler, then a *CONTROL statement with the negation of the option keyword must precede the portions of the source program for which listing output is to be inhibited. Listing output then resumes when a *CONTROL statement with the affirmative option keyword is encountered.
- If the negation of the corresponding option has been requested as a parameter to the compiler, the output of that type is **always** inhibited.

Selective Source Listing

Production or suppression of a listing of the input source program lines is controlled by the *CONTROL SOURCE and *CONTROL NOSOURCE statements (or *CBL SOURCE and *CBL NOSOURCE).

If any *CONTROL NOSOURCE or *CBL NOSOURCE statement is encountered and SOURCE has been requested as a parameter to the compiler, the following informational (I-level) message is issued:

```
Printing of the source code has been suppressed.
```

The source program text is displayed as follows:

- If NUM **is not** requested, the sequence field is printed to the left of the listing.
- If NUM **is** requested, the sequence field is blank, unless sequence is violated (in which case the sequence field appears to the left of the listing).

The entire line is displayed in the body of the listing.

Storage Mapping in the Data Division

Parts of the MAP listing and embedded MAP summary can be selected or inhibited by use of *CONTROL MAP or *CONTROL NOMAP statements (*CBL MAP or *CBL NOMAP statements) interspersed throughout the source.

For example, if the Data Division listing has the following format:

```
*CONTROL NOMAP            *CBL NOMAP
    01  A                     01  A
    02  B                     02  B
*CONTROL MAP              *CBL MAP
```

then A and B will not appear in the MAP listing, and the embedded MAP summaries for A and B will not appear in the right margin of the source listing. For examples of MAP and embedded MAP listings, see “Data Map Listing” on page 305.

Object Code in the Procedure Division

You can control the selective listing of generated object code with the *CONTROL LIST and *CONTROL NOLIST statements (*CBL LIST and *CBL NOLIST).

- User-defined names are displayed
- The sequence field and COBOL statement are placed on a separate line to make room for the user-defined names.
- One or two operands participating in the machine instructions are displayed on the right. An asterisk immediately follows the data-names that are defined in more than one structure (thus, made unique by qualification in the source program).
- The relative location of any generated label appearing as an operand is displayed in parentheses. (This is done because the target label may be far removed from the reference, especially in structured programming.)
- Internal clauses of COBOL statements are displayed in the statement line format.

Debug Tool/VSE

Debug Tool/VSE provides these productivity enhancements:

- Debug Tool/VSE sessions can be in interactive full-screen mode, or in batch mode.

  During an interactive full-screen mode session, you can use Debug Tool/VSE's full-screen services and session panel windows on a 3270 device to debug your program as 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 different languages. Debug Tool/VSE automatically determines the language of the program or subprogram being run.

For more information, see the *Debug Tool/VSE User's Guide and Reference*.

Chapter 21. The "Year 2000" Problem

This chapter provides some information on date processing problems associated with the year 2000, and recommends some solutions that you can adopt to help resolve them.

Date Processing Problems

Many applications use two digits rather than four to represent the year in date fields, and assume that these values represent years from 1900 to 1999. This compact date format works well for the 1900s, but it does not work for the year 2000 and beyond because these applications interpret "00" as 1900 rather than 2000, producing incorrect results.

This chapter outlines a number of approaches you can adopt to resolve problems of this nature, and points to facilities available in the COBOL compiler and in the LE/VSE callable services that can assist you.

For more information on the new features of the COBOL language that can help resolve date-related problems, see Chapter 22, "Using the Millennium Language Extensions" on page 366.

For more information about Year 2000 issues, and IBM software products that can help you identify and resolve their related problems, visit the website at: http://www.software.ibm.com/year2000.

Year 2000 Solutions

There are several solutions to the Year 2000 problem. Many of these solutions refer to a "century window". A century window is a 100-year interval, such as 1950–2049, within which any 2-digit year is unique. For example, with a century window of 1930–2029, 2-digit years would be interpreted as follows:

Year values from 00 through 29 are interpreted as years 2000–2029
Year values from 30 through 99 are interpreted as years 1930–1999

The solutions outlined in this chapter are:

- The Full Field Expansion Solution (the long-term approach)
- The Internal Bridging Solution
- The Century Window Solution
- The Mixed Field Expansion and Century Window Solution
- The Century Encoding/Compression Solution
- The Integer Format Date Solution

Each of these is discussed in more detail below.

The Full Field Expansion Solution

To take your programs through to the year 9999, you *must* eventually rewrite applications and rebuild databases and files to use 4-digit year fields rather than 2-digit year fields.

The field expansion method is a long-term solution and is the recommended approach to addressing the Year 2000 problem. To achieve this field expansion, you need to develop a program to read in the old data, convert it, and write it back into a copy of the original file or data base that has been expanded to hold the 4-digit year data. All new data would then go into the new file or database. All of your application programs that use those files and databases need to be changed to act on the new 4-digit year date fields instead of the 2-digit year fields.

Your conversion program needs to use a century window when expanding 2-digit years to 4 digits, to ensure that the output dates are correct.

There are several ways to use COBOL/VSE to help convert your databases or files from 2-digit year dates to 4-digit year dates, with a century window being taken into account:

DATEPROC processing

Use the DATEPROC compiler option and the DATE FORMAT clause to define date fields, and use MOVE statements to expand the dates based on the century window specified by the yearwindow compiler option. For example:

```
05  Date-Short      Pic x(6) Date Format yyxxxx.
⋮
05  Date-Long       Pic x(8) Date Format yyyyxxxx.
⋮
Move Date-Short to Date-Long.
```

For more information, see Chapter 22, “Using the Millennium Language Extensions” on page 366.

COBOL coding

You can move a 2-digit year date field to an expanded receiving field, and “hard code” a century component as part of the move. For example:

```
05  Date-Short      Pic x(6) Date Format yyxxxx.
⋮
05  Date-Long       Pic x(8) Date Format yyyyxxxx.
⋮
String "19" Date-Short Delimited by Size
       Into Date-Long.
```

The hard-coded century component assumes a century window of 1900–1999 in this example, but you can add code to recognize different date ranges and assign a different century based on the value of Date-Short. For example, the following code expands the date based on a century window of 1930–2029:

```
       05  Date-Short      Pic x(6) Date Format yyxxxx.
       ⋮
       05  Date-Long       Pic x(8) Date Format yyyyxxxx.
       ⋮
   77  Century             Pic x(2).
       ⋮
       If Date-Short Less than "300000" Then
          Move "20" to Century
       Else
          Move "19" to Century
       End-If.
       String Century Date-Short Delimited by Size
              Into Date-Long.
```

Intrinsic functions

Three intrinsic functions are available to expand 2-digit year dates into 4-digit year dates, with the window being specified as an argument to the function. The functions are:

DATE-TO-YYYYMMDD
Expand a Gregorian date with a 2-digit year to the same date with a 4-digit year.

DAY-TO-YYYYDDD
Expand a Julian date with a 2-digit year to the same date with a 4-digit year.

YEAR-TO-YYYY
Expand a 2-digit year to a 4-digit year.

With these functions, you specify the century window to be used when expanding the year. For full details and syntax of these functions, see *COBOL/VSE Language Reference*.

Callable services

LE/VSE provides a number of callable services to manipulate and convert dates. Some of these services can accept a date with a 2-digit year as input, and in this case, the callable services will apply the LE/VSE century window. The following services either affect or can be affected by this century window:

CEECBLDY Convert a date to a COBOL integer number of days.

CEEDAYS Convert a date to a Lilian integer number of days.

CEEQCEN Query the LE/VSE century window.

CEESCEN Change the LE/VSE century window.

CEESECS Convert a date and time stamp into a number of Lilian seconds.

For full details on these and other callable services, see the *LE/VSE Programming Reference*.

For additional information about the century window feature of the LE/VSE callable services, see rhe *LE/VSE Programming Guide*.

Advantages:

- The code changes are straightforward.
- Minimum testing is required and possibly no need for simulation of future dates on dedicated machines.

- Faster resulting code.
- The issue is addressed once and for all.
- Maintenance will become cheaper.

Disadvantages:

- Databases and files must be changed.

The Internal Bridging Solution

This solution involves keeping the dates in your files and databases as 2-digit year dates, and expanding them into other data items in your program.

In your application progams, you need to add some data items to hold the 4-digit year dates, and some processing logic to expand and contract the date fields. The resultant program will be structured like this:

1. Read the input files with 2-digit year dates.
2. Declare “shadow” data items that contain 4-digit year dates, and expand the 2-digit year fields into these work fields.
3. Use the 4-digit year dates for all date processing in the program.
4. Copy (window) the 4-digit year date fields back to 2-digit format for the output process.
5. Write the 2-digit year dates to the output files.

There are several ways to use COBOL/VSE to achieve the field expansion and windowing needed for this solution.

For date field expansion:

- Use the DATEPROC compiler option and the DATE FORMAT clause to define the dates in the input records as windowed date fields, and the work fields as expanded date fields. Perform expanded MOVEs or stores using MOVE or COMPUTE statements.
- Use the intrinsic functions DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD, and YEAR-TO-YYYY to copy and expand date fields from the input records to work fields.
- Use the LE/VSE callable services CEEDAYS and CEEDATE.

For date windowing:

- Simply MOVE the last 2 digits of the year back to the 2-digit year date fields. You should also add some code to check that the date is still within the century window, and take some error action if it is not. For example, if the 4-digit year field contains 2010 and the century window is 1905–2004, the date is outside the century window, and to simply store the last 2 digits would be incorrect.
- With the DATEPROC compiler option and the DATE FORMAT clause, copy the expanded date fields back to windowed date fields. If you use a COMPUTE statement to do this, you can use the ON SIZE ERROR phrase to ensure that the date remains within the century window, or to take some action if it doesn't. For details, see “ON SIZE ERROR Phrase” on page 376.

Advantages:

- Databases and files need not be changed.
- The code changes are straightforward.
- Good interim step towards a full field expansion solution.
- Faster resulting code.

Disadvantages:

- Some risk of data corruption.
- Short- to medium-term solution only.

The Century Window Solution

The century window solution allows 2-digit years to be interpreted in a 100-year window (because each 2-digit number can only occur once in any 100-year period).

There are several ways to use COBOL/VSE to help you achieve this:

- Use the DATEPROC compiler option and the DATE FORMAT clause to define date fields. This provides an automated windowing capability using the century window defined by the YEARWINDOW compiler option.

  For more information, see Chapter 22, “Using the Millennium Language Extensions” on page 366.

- Use the intrinsic functions DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD, and YEAR-TO-YYYY to interpret date fields based on a century window. The century window is specified as an argument to the intrinsic function. For example:

  ```
  If Function YEAR-TO-YYYY (Current-Year, 48) Greater Than
     Function YEAR-TO-YYYY (Due-Year, 48) Then
     Display "Due date has passed."
  End-If.
  ```

  In this example, the century window begins 48 years prior to the year at the time the program is being run. If the program is running in 1998, then the century window is 1950–2049. This would allow a Current-Year value of 00 to be “greater” than a Due-Year value of 99.

- Insert IF statements around the references to date fields in your program, to determine how to apply a century component. For example, the following code implements a century window of 1940–2039:

  ```
  If YY-1 less than "40" Then
     Move "20" to CC-1
  Else
     Move "19" to CC-1
  End-If.
  ```

- Use the LE/VSE callable services to manipulate date fields using a century window defined by the CEESCEN service.

You cannot use the century window forever because a 2-digit year can only be unique in a given 100-year period. Over time you will need more than 100 years for your data window—in fact, many companies need more than 100 years now. For example, the century window cannot solve the problem of trying to figure out how old a customer is if the customer is older than 100 years and you only have

2-digit year dates. For these issues and others you need to adopt The Full Field Expansion Solution.

Advantages:

- No database or file changes.

Disadvantages: The following disadvantages apply to the Century Window solution regardless of which method you use to implement it:

- Performance will be slower due to increased logic.
- More testing is required to validate changes, and simulation of future dates on dedicated machines is essential.
- Very difficult to manage across applications.
- The problem is not permanently solved and it will become necessary to change date programs and databases to use 4-digit years eventually.

In addition, if you do not use the DATEPROC and DATE FORMAT method, the following disadvantages apply:

- Risk of performing incorrect translations.
- Code changes are more error-prone and require more expertise.
- Increased maintenance costs.

The Mixed Field Expansion and Century Window Solution

You don't have to convert all of your files and databases at one time. Where a data base is shared by many applications, it might be more convenient to keep any dates that it contains in 2-digit year form. But where a file is used by a limited number of programs, it is best to eliminate the 2-digit year constraint as soon as possible.

For those dates that are still in 2-digit year form, you can use internal bridging or century windowing, both of which are described in detail in “The Internal Bridging Solution” on page 359, and “The Century Window Solution” on page 360, respectively.

You change the data descriptions for dates that you have expanded to 4-digit year form, and then use any of the techniques described in “The Full Field Expansion Solution” for processing them.

The DATEPROC compiler option is a particularly convenient way of implementing this solution, since it directly supports the use of both expanded and windowed date fields within a single statement.

The mixed solution has the advantages and disadvantages of the individual techniques that are discussed in these sections. In addition, the mixed solution has the advantage that files and databases can be changed as convenient, and kept unmodified otherwise.

The Century Encoding/Compression Solution

The century encoding/compression solution involves encoding/compressing numbers greater than 99 into existing 2-byte date fields. (For example, using hexadecimal rather than decimal digits.) This means rewriting applications to correctly interpret encoded/compressed values in the database.

This solution is the least desirable way to address the Year 2000 problem.

Advantages:

- Uses existing 2-byte date fields.

Disadvantages:

- Performance will be slower due to increased logic.
- More testing is required to validate changes and simulation of future dates on dedicated machines is essential.
- Very difficult to manage across applications.
- Code changes are more error-prone and require more expertise.
- Increased maintenance costs.
- The problem is not permanently solved and it will become necessary to change date programs and databases to use 4-digit years eventually.
- Cannot be read in dumps or listings.
- Must be translated whenever externalized.
- Risk of performing incorrect translations.

The Integer Format Date Solution

Integer dates specify a number of days from some point in the past.

Integer dates are provided by COBOL intrinsic functions and by the LE/VSE callable services.

This solution is *not* the recommended way to address the Year 2000 problem. Instead, use the The Full Field Expansion Solution described on page 357.

Advantages:

- Uses only 4 bytes to store a date.

Disadvantages:

- Performance will be slower due to increased logic.
- More testing is required to validate changes and simulation of future dates on dedicated machines is essential.
- Very difficult to manage across applications.
- Code changes are more error-prone and require more expertise.
- Increased maintenance costs.
- The problem is not permanently solved and it will become necessary to change date programs and databases to use 4-digit years eventually.

- Cannot be read in dumps or listings.
- Must be translated whenever externalized.
- Risk of performing incorrect translations.
- There are too many different integer format starting dates:
  - CICS and SQL/DS start with January 1, 1900
  - PL/I does not support integer date values
  - C starts with January 1, 1970
  - COBOL starts with January 1, 1601
  - LE/VSE callable services start with October 15, 1582 (Lilian integer dates)

  There will be no problems with integer dates if conversion to and from integer is done using the same method in the same program. There will only be problems if the integer values are stored or passed between programs. These problems could still be avoided by:

  - Not using the value returned by CEECBLDY as input to other LE/VSE callable services; CEECBLDY returns an ANSI COBOL integer date that can be used with COBOL intrinsic functions.
  - Only using LE/VSE callable services, or only COBOL intrinsic functions, for getting and manipulating 4-digit year dates.
  - Using the INTDATE(LILIAN) compiler option. This will cause the intrinsic functions to return Lilian integer dates that will be compatible with LE/VSE date callable services (and different from the results returned in COBOL/VSE Release 1).

    When INTDATE(LILIAN) is in effect, CEECBLDY will not be usable, because you will have no way to turn an ANSI integer into a meaningful date using either intrinsic functions or callable services. If you code a CALL '*literal*' statement with CEECBLDY as the target of the call, and INTDATE(LILIAN) is in effect, the compiler will diagnose this and convert the call target to CEEDAYS.

    This method is the safest of the three, because you can store integer dates, pass them between programs, and even pass them from PL/I to COBOL to C programs and have no problems, as long as all programs use LE/VSE callable services for date processing.

    For more information on the INTDATE compiler option, see “INTDATE” on page 238.

Performance Considerations

Any implementation of a solution to the year 2000 problem will have some impact on the performance of your application. This section discusses some of the performance aspects that you should consider.

Performance Comparison

The following implementation methods are listed in order of least performance impact to most performance impact.

Full field expansion
The best performance can be obtained by expanding all of the dates in your files as a one-time task, and thereafter using the 4-digit year date fields in all processing.

Mixed field expansion and DATEPROC windowing
If the dates in your files have not yet been expanded, the best performance can be obtained by expanding the date fields as they are read from the files, and using these expanded dates in the main processing body of the program. In this way, the expansion process is only performed once for each date field.

Mixed field expansion and manual windowing
You can expand your input date fields manually, using combinations of COBOL coding, intrinsic functions, and callable services to apply the century window. This has more performance impact than DATEPROC windowing, even though the expansion process is still only performed once for each date field.

DATEPROC windowing
The millennium language extensions provide a windowing method that is designed to be efficient. However, the action of viewing a windowed date field for a COBOL IF or MOVE statement still imposes some processor overhead.

Manual windowing
Date windowing using COBOL IF statements adds a level of complexity to the program, and adds some processor overhead because of the extra COBOL statements. Typically the overhead of an IF statement of this form is more than the overhead of the automatic DATEPROC windowing process.

How to Get 4-digit Year Dates

Many COBOL programs need to obtain the date at the time of execution, to use as "Date-Of-..." fields in output files or reports, or to compare against other dates read from input files. COBOL provides a number of methods of obtaining the current date with a 4-digit year. The simplest of these are:

The intrinsic function CURRENT-DATE
Retrieves the date in Gregorian form, and can also retrieve the current time and the offset from Greenwich Mean Time.

ACCEPT *identifier* FROM DATE YYYYMMDD
Retrieves the date in Gregorian form.

ACCEPT *identifier* FROM DAY YYYYDDD
Retrieves the date in Julian form.

The CEELOCT callable service
Retrieves the date in three different forms, including Gregorian with a 4-digit year.

Using Callable Services with DOS/VS COBOL and VS COBOL II

If you are using DOS/VS COBOL or VS COBOL II, support for the LE/VSE date/time callable services is limited, as follows:

DOS/VS COBOL There is no support for LE/VSE date/time callable services.

VS COBOL II You cannot use static calls from VS COBOL II programs to any LE/VSE callable services. You can use DYNAMIC calls from VS COBOL II programs to any of the following date/time callable services to process dates:

CEECBLDY	CEEISEC
CEEDATE	CEELOCT
CEEDATM	CEEQCEN
CEEDAYS	CEESCEN
CEEDYWK	CEESECI
CEEGMT	CEESECS
CEEGMTO	CEE5CTY

Chapter 22. Using the Millennium Language Extensions

This chapter provides information on the millennium language extensions that have been incorporated into COBOL/VSE to assist with Year 2000 processing.

Description

The term “Millennium Language Extensions” refers collectively to the features of COBOL/VSE that are activated by the DATEPROC compiler option to help with Year 2000 date logic problems.

Note: The millennium language extensions can only be enabled if your system has the product VisualAge COBOL Millennium Language Extensions for VSE/ESA (5686-MLE) installed with your compiler.

The DATEPROC compiler option enables special date-oriented processing of identified date fields, and the YEARWINDOW compiler option specifies the 100-year window (the century window) to be used for the interpretation of 2-digit windowed years. For a description of the DATEPROC compiler option, see “DATEPROC” on page 231. For a description of the YEARWINDOW compiler option, see “YEARWINDOW” on page 256.

The extensions, when enabled, include:

- The DATE FORMAT clause. This is added to items in the Data Division to identify date fields, and to specify the location of the year component within the date.
- The reinterpretation of the function return value as a date field, for the following intrinsic functions:

  DATE-OF-INTEGER
  DATE-TO-YYYYMMDD
  DAY-OF-INTEGER
  DAY-TO-YYYYDDD
  YEAR-TO-YYYY

- The reinterpretation as a date field of the conceptual data items DATE, DATE YYYYMMDD, DAY, and DAY YYYYDDD in the following forms of the ACCEPT statement:

  ACCEPT *identifier* FROM DATE
  ACCEPT *identifier* FROM DATE YYYYMMDD
  ACCEPT *identifier* FROM DAY
  ACCEPT *identifier* FROM DAY YYYYDDD

- The intrinsic functions UNDATE and DATEVAL, used for selective reinterpretation of date fields and non-dates.
- The intrinsic function YEARWINDOW, which retrieves the starting year of the century window set by the YEARWINDOW compiler option.

This chapter describes how you can use these new facilities to help solve date logic problems in your COBOL programs.

Getting Started

With the millennium language extensions, you can make simple changes to your COBOL programs to define date fields, and the compiler recognizes and acts on those dates using a century window to ensure consistency.

A century window is a 100-year interval, such as 1950–2049, within which any 2-digit year is unique. For windowed date fields, the century window start date is specified by the YEARWINDOW compiler option. When the DATEPROC option is in effect, the compiler applies this window to 2-digit year, or windowed, date fields in the program. For example, with a century window of 1930–2029, COBOL interprets 2-digit years as:

Year values from 00 through 29 are interpreted as years 2000–2029
Year values from 30 through 99 are interpreted as years 1930–1999

To implement date windowing using COBOL date processing, you define the century window with the YEARWINDOW compiler option, and identify the date fields in your program with DATE FORMAT clauses. The compiler then automatically applies the century window to operations on those dates. It is often possible to implement a solution in which the windowing process is fully automatic; that is, you simply identify the fields that contain windowed dates, and you do not need any extra program logic to implement the windowing.

Implementing Date Processing

Following is a list of simple steps that you need to follow in order to implement automatic date recognition in a COBOL program:

- Use the DATEPROC compiler option to enable the process. You specify this as either DATEPROC(FLAG) to get some helpful diagnostic messages, or DATEPROC(NOFLAG). For full information, see “DATEPROC” on page 231.
- Use the YEARWINDOW compiler option to set the century window. There are two ways of doing this:
  - For a *fixed window*, specify a 4-digit year between 1900 and 1999 as the YEARWINDOW option value. For example, YEARWINDOW(1950) defines a fixed window of 1950–2049.
  - For a *sliding window*, specify a negative integer from -1 through -99 as the YEARWINDOW option value. For example, YEARWINDOW(-48) defines a sliding window that starts 48 years before the year that the program is running. So if the program is running in 1998, the century window is 1950–2049, and in 1999 it automatically becomes 1951–2050, and so on. For a full description and syntax, see “YEARWINDOW” on page 256.
- Add the DATE FORMAT clause to the data description entries of those data items in the program that contain dates that you want the compiler to recognize as windowed or expanded dates. For a full description of the DATE FORMAT clause, see the *COBOL/VSE Language Reference*.
- To expand dates, use MOVE or COMPUTE statements to copy the contents of windowed date fields to expanded date fields.
- If necessary, use the DATEVAL and UNDATE intrinsic functions, to convert between date fields and non-dates. For a full description of these functions, see the *COBOL/VSE Language Reference*.

- Compile the program with the DATEPROC(FLAG) option, and review the diagnostic messages to see if date processing has produced any unexpected side effects (see “Analyzing Date-Related Diagnostic Messages” on page 380). When the compilation has only Information-level diagnostics, you can recompile with the DATEPROC(NOFLAG) option to produce a “clean” listing.

This provides a simple implementation of date windowing and expansion capabilities in a COBOL program.

Resolving Date-Related Logic Problems

This section discusses three approaches that you can adopt to assist with date-related processing problems, and shows how you can use the millennium language extensions with each approach to achieve a solution.

These and other approaches are discussed in conceptual terms in “Year 2000 Solutions” on page 356. The description here concentrates on the application of COBOL date processing capabilities as a tool to implement the solutions.

The approaches outlined here are:

- Basic Remediation (the century window solution)
- Internal Bridging
- Full Field Expansion

Basic Remediation

The simplest method of ensuring that your programs will continue to function through the year 2000 is to implement a century window solution.

With this method, you define a century window, and specify the fields that contain windowed dates. The compiler then interprets the 2-digit years in those date fields according to the century window.

The following sample code shows how a program can be modified to use this automatic date windowing capability. The program checks whether a video tape was returned on time:

```
CBL  DATEPROC(FLAG),YEARWINDOW(-60)
        ⋮
    01  Loan-Record.
        05  Member-Number   Pic X(8).
        05  Tape-ID         Pic X(8).
        05  Date-Due-Back   Pic X(6) Date Format yyxxxx.
        05  Date-Returned   Pic X(6) Date Format yyxxxx.
        ⋮
        If Date-Returned Greater than Date-Due-Back Then
           Perform Fine-Member.
```

In this example, there are no changes to the Procedure Division from the program's previous version. The addition of the DATE FORMAT clause on the two date fields means that the compiler recognizes them as windowed date fields, and therefore applies the century window when processing the IF statement. For example, if Date-Due-Back contains “000102” (January 2, 2000) and Date-Returned contains

"991231" (December 31, 1999), Date-Returned is less than (earlier than) Date-Due-Back, so the program does not perform the Fine-Member paragraph.

Advantages:

- Fast and easy to implement.
- No change to the program's logic, therefore less testing required.
- This solution will allow your programs to function into and beyond the year 2000.

Disadvantages:

- This should be regarded as a short-term solution, not as a permanent fix.
- There may be some performance degradation introduced by the date windowing functions.
- Implementation of this solution is application-dependent. It will not suit all applications.

Internal Bridging

If your files and databases have not yet been converted to 4-digit year dates, but you prefer to use 4-digit expanded-year logic in your programs, you can use an internal bridge technique to process the dates as 4-digit years. Your program will be structured as follows:

1. Read the input files with 2-digit year dates.
2. Declare these 2-digit dates as windowed date fields and move them to expanded date fields, so that the compiler automatically expands them to 4-digit year dates.
3. In the main body of the program, use the 4-digit year dates for all date processing.
4. Window the dates back to 2-digit years.
5. Write the 2-digit year dates to the output files.

This process provides a convenient migration path to a full expanded-date solution, and also may have performance advantages over using windowed dates. For more information, see "Performance Considerations" on page 363.

Using this technique, your changes to the program logic are minimal. You simply add statements to expand and contract the dates, and change those statements that refer to dates to use the 4-digit year date fields in Working-Storage instead of the 2-digit year fields in the records.

Because you are converting the dates back to 2-digit years for output, you should allow for the possibility of the year being outside the century window. For example, if a date field contains the year 2005, but the century window is 1905–2004, then the date is outside the window, and simply moving it to a 2-digit year field would be incorrect. To protect against this, you can use a COMPUTE statement to store the date, with the ON SIZE ERROR phrase to detect whether or not the date is within the century window. For more details, see "ON SIZE ERROR Phrase" on page 376.

The following example shows how a program can be changed to implement an internal bridge method:

```
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.
```

Advantages:

- Straightforward changes to the program logic, therefore testing is easy.
- This solution will allow your programs to function into and beyond the year 2000.
- This is a good incremental step towards a full expanded-year solution.
- Good performance.

Disadvantages:

- Some risk of data corruption.

Full Field Expansion

The full field expansion solution involves explicitly expanding 2-digit year date fields to contain full 4-digit years in your files and databases, and then using those fields in expanded form in your programs. This is the only method by which you can be assured of reliable date processing for all applications.

The millennium language extensions allow you to progressively move towards a full date field expansion solution, using the following steps:

1. Apply the short-term (basic remediation) solution, and use this until you have the resources to implement a more permanent solution.
2. Apply the internal bridging scheme. This allows you to use expanded dates in your programs while your files continue to hold dates in 2-digit year form. This in turn will allow you to progress more easily to a full field expansion solution,

because there will be no further changes to the logic in the main body of the programs.

3. Change the file layouts and database definitions to use 4-digit year dates.
4. Change your COBOL copybooks to reflect these 4-digit year date fields.
5. Run a utility program (or special-purpose COBOL program) to copy from the old format files to the new format. For a sample program, see Figure 126.
6. Recompile your programs and perform regression testing and date testing.

After you have completed the first two steps, the remaining steps in the sequence can be repeated any number of times. You do not need to change every date field in every file at the same time. Using this method, you can select files for progressive conversion based on criteria such as business needs or interfaces with other applications.

When you use this method, you will need to write special-purpose programs to convert your files to expanded-date form. Figure 126 shows a simple program that copies from one file to another while expanding the date fields. Note that the record length of the output file is larger than that of the input file because the dates are expanded.

```
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.
```

Figure 126 (Part 1 of 2). Expanding File Dates

```
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.
```

Figure 126 (Part 2 of 2). Expanding File Dates

The following notes apply to Figure 126 on page 371.

1 The fields DUE-DATE and REMINDER-DATE in the input record are both Gregorian dates with 2-digit year components. They have been defined with a DATE FORMAT clause in this program so that the compiler will recognize them as windowed date fields.

2 The output record contains the same two fields in expanded date format. They have been defined with a DATE FORMAT clause so that the compiler will treat them as 4-digit year date fields.

3 The MOVE CORRESPONDING statement moves each item in INPUT-RECORD individually to its matching item in OUTPUT-RECORD. When the two windowed date fields are moved to the corresponding expanded

date fields, the compiler will expand the year values using the current century window.

Advantages:

- This is a permanent solution; no more changes are required. This solution will allow your programs to function into and beyond the year 2000.
- Best performance.
- Maintenance will be easier.

Disadvantages:

- Need to ensure that changes to databases, copybooks, and programs are all synchronized.

Programming Techniques

This section describes the techniques you can use in your programs to take advantage of date processing, and the effects of using date fields on COBOL language elements.

For full details of COBOL syntax and restrictions, see the *COBOL/VSE Language Reference*.

Date Comparisons

When you compare two date fields, the two dates must be compatible; that is, they must have the same number of non-year characters (see “Compatible Dates” on page 384). The number of digits for the year component need not be the same. For example:

```
77  Todays-Date         Pic X(8) Date Format yyyyxxxx.
01  Loan-Record.
    05  Date-Due-Back   Pic X(6) Date Format yyxxxx.
    ⋮
    If Date-Due-Back Greater than Todays-Date Then...
```

In this example, a windowed date field is compared to an expanded date field, so the century window is applied to Date-Due-Back.

Note that Todays-Date must have a DATE FORMAT clause in this case to define it as an expanded date field. If it did not, it would be treated as a non-date field, and would therefore be considered to have the same number of year digits as Date-Due-Back. The compiler would apply the assumed century window to it, which would create an inconsistent comparison. For more information, see “The Assumed Window” on page 386.

Level 88 Condition-Name

If a windowed date field has an 88-level condition-name associated with it, the literal in the VALUE clause is windowed against the century window for the compilation unit rather than the assumed century window of 1900–1999. For example:

```
05  Date-Due        Pic 9(6) Date Format yyxxxx.
88  Date-Target              Value 051220.
```

If the century window is 1950–2049 and the contents of Date-Due is 051220 (representing December 20, 2005), then the condition

```
If Date-Target
```

would evaluate to TRUE, but the condition

```
If Date-Due = 051220
```

would evaluate to FALSE. This is because the literal 051220 is treated as a non-date, and therefore windowed against the assumed century window of 1900–1999 to represent December 20, 1905. But where the same literal is specified in the VALUE clause of an 88-level condition-name, it becomes part of the data item to which it is attached. Because this data item is a windowed date field, the century window is applied whenever it is referenced.

You can also use the DATEVAL intrinsic function in a comparison expression to convert a literal to a date field, and the output from the intrinsic function will then be treated as either a windowed or expanded date field to ensure a consistent comparison. For example, using the above definitions, both of these conditions

```
If Date-Due = Function DATEVAL (051220 "YYXXXX")
If Date-Due = Function DATEVAL (20051220 "YYYYXXXX")
```

would evaluate to TRUE. For more information on the DATEVAL intrinsic function, see “DATEVAL” on page 379.

Restriction: With a level-88 condition name, you can also specify the THRU option on the VALUE clause, for example:

```
05  Year-Field      Pic 99 Date Format yy.
88  In-Range               Value 98 Thru 06.
```

With this form, the windowed value of the second item must be greater than the windowed value of the first item. However, the compiler can only verify this if the YEARWINDOW compiler option specifies a fixed century window (for example, YEARWINDOW(1940) rather than YEARWINDOW(-60)).

For this reason, if the YEARWINDOW compiler option specifies a sliding century window, you cannot use the THRU option on the VALUE clause of a level-88 condition name.

Sign Condition

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, this comparison is not valid because the literal value Zero is a non-date, and is therefore windowed against the assumed century window to give a value of 1900000 (for more information, see “Treatment of Non-Dates” on page 385).

In this case, you can use a sign condition, as follows:

```
If Order-Date Is Zero Then...
```

instead of a literal comparison. With a sign condition, Order-Date is treated as a non-date, and the century window is not considered.

Notes:

1. This only applies 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 to the results of the expression.
2. You could also use the UNDATE intrinsic function to achieve the same result. For details, see "UNDATE" on page 380.

Arithmetic Expressions

You can perform arithmetic operations on numeric date fields in the same manner as any numeric data item, and, where appropriate, the century window will be used in the calculation. However, there are some restrictions on where date fields can be used in arithmetic expressions.

Arithmetic operations that include date fields are restricted to:

- Adding a non-date to a date field
- Subtracting a non-date from a date field
- Subtracting a date field from a compatible date field to give a non-date result

The following arithmetic operations are not allowed:

- Any operation between incompatible date fields
- Adding two date fields
- Subtracting a date field from a non-date
- Unary minus, applied to a date field
- Multiplication, division, or exponentiation of or by a date field

Windowed Date Fields

Where a windowed date field participates in an arithmetic operation, it is processed as if the value of the year component of the field were first incremented by 1900 or 2000, depending on the century window. For example:

```
01  Review-Record.
    03  Last-Review-Year    Pic 99 Date Format yy.
    03  Next-Review-Year    Pic 99 Date Format yy.
    ⋮
    Add 10 to Last-Review-Year Giving Next-Review-Year.
```

If the century window is 1910–2009, and the value of Last-Review-Year is 98, then the computation proceeds as if Last-Review-Year is first incremented by 1900 to give 1998. Then the ADD operation is performed, giving a result of 2008. This is stored in Next-Review-Year as 08.

Order of Evaluation

Because of the restrictions on date fields in arithmetic expressions, you may find that programs that previously compiled successfully now produce diagnostic messages when some of the data items are changed to date fields.

Consider the following example:

```
01  Dates-Record.
    03  Start-Year-1        Pic 99 Date Format yy.
    03  End-Year-1          Pic 99 Date Format yy.
    03  Start-Year-2        Pic 99 Date Format yy.
    03  End-Year-2          Pic 99 Date Format yy.
    ⋮
    Compute End-Year-2 = Start-Year-2 + End-Year-1 - Start-Year-1.
```

In this example, the first arithmetic expression evaluated is:

```
Start-Year-2 + End-Year-1
```

However, this is the addition of two date fields, which is not permitted. To resolve this, you should use parentheses to isolate those parts of the arithmetic expression that are allowed. For example:

```
Compute End-Year-2 = Start-Year-2 + (End-Year-1 - Start-Year-1).
```

In this case, the first arithmetic expression evaluated is:

```
End-Year-1 - Start-Year-1
```

This is the subtraction of one date field from another, which is permitted, and gives a non-date result. This non-date result is then added to the date field End-Year-1, giving a date field result which is stored in End-Year-2.

ON SIZE ERROR Phrase

In the example in “Windowed Date Fields” on page 375, the result of 2008 falls within the century window of 1910–2009, so a value of 08 in Next-Review-Year will be recognized as 2008 by subsequent statements in the program.

However, the statement:

```
Add 20 to Last-Review-Year Giving Next-Review-Year.
```

would give a result of 2018. As this falls outside the range of the century window, if the result is stored in Next-Review-Year it would be incorrect, because later references to Next-Review-Year would interpret it as 1918. In this case, the result of the operation depends on whether the ON SIZE ERROR phrase is specified on the ADD statement, as follows:

- If SIZE ERROR is specified, the receiving field is not changed, and the SIZE ERROR imperative statement is executed.
- If SIZE ERROR is not specified, the result is stored in the receiving field with the left-hand digits truncated.

This is an important consideration when developing an internal bridging solution to resolve a date processing problem (see “Internal Bridging” on page 369). When you contract a 4-digit year date field back to 2 digits to write it to the output file, you need to ensure that the date falls within the century window, and that therefore the 2-digit year date will be represented correctly in the field.

You can achieve this using a COMPUTE statement to do the contraction, with a SIZE ERROR phrase to handle the out-of-window condition. For example:

```
Compute Output-Date-YY = Work-Date-YYYY
        On Size Error Go To Out-of-Window-Error-Proc.
```

Note: SIZE ERROR processing for windowed date receivers recognizes *any* year value that falls outside the century window. That is, a year value less than the starting year of the century window raises the SIZE ERROR condition, as does a year value greater than the ending year of the century window.

Sorting and Merging

Note: DFSORT/VSE is the IBM sort/merge licensed program. Wherever DFSORT/VSE is mentioned, any other equivalent sort/merge product can be used.

If your version of DFSORT/VSE supports the Y2PAST option and the field identifiers Y2C, Y2D, and Y2Z, you can perform sort and merge operations using windowed date fields as sort keys. The fields will be sorted on their windowed date sequence, with the century window being taken into account for the sort process.

The following example shows a sort of a transaction file, with the transaction records being sorted by date and time within account number. The field 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/VSE in order for it to perform the sorting operation properly. In addition to the information that is always passed to DFSORT/VSE, COBOL also passes:

- The century window as the Y2PAST sort option
- The windowed year field and date format of Trans-Date

DFSORT/VSE then uses this information to perform the sorting process.

For information on DFSORT/VSE and the Y2PAST option, see the *DFSORT/VSE Application Programming Guide*.

Other Date Formats

To be eligible for automatic windowing, a date field should contain a 2-digit year as the first or only part of the field. The remainder of the field, if present, must be between 2 and 4 characters, but its content is not important. For example, it can contain a 3-digit Julian day, or a 2-character identifier of some event specific to the enterprise.

If there are date fields in your application that do not fit these criteria, then you may 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 3-character field consisting of a 2-digit year and a single character to represent the month (A–L representing 1–12). This is not supported because date fields can have only zero, 2, 3, or 4 non-year characters.
- A Gregorian date of the form DDMMYY. This is not supported because the year component is not the first part of the date.

If you need to use date windowing in cases like these, you will need to add some code to isolate the year portion of the date.

In the following example, the two date fields contain dates of the form DDMMYY:

```
03  Last-Review-Date Pic 9(6).
03  Next-Review-Date Pic 9(6).
⋮
Add 1 to Last-Review-Date Giving Next-Review-Date.
```

In this example, if Last-Review-Date contains 230197 (January 23, 1997), then Next-Review-Date will contain 230198 (January 23, 1998) after the ADD statement is executed. This is a simple method of setting the next date for an annual review. However, if Last-Review-Date contains 230199, then adding 1 gives 230200, which is not the desired result.

Because the year is not the first part of these date fields, the DATE FORMAT clause cannot be applied without some code to isolate the year component. In the next example, the year component of both date fields has been isolated so that COBOL can apply the century window and maintain consistent results:

```
03  Last-Review-Date.
    05  Last-R-DDMM  Pic 9(4).
    05  Last-R-YY    Pic 99 Date Format yy.
03  Next-Review-Date Pic 9(6).
    05  Next-R-DDMM  Pic 9(4).
    05  Next-R-YY    Pic 99 Date Format yy.
⋮
Move Last-R-DDMM to Next-R-DDMM.
Add 1 to Last-R-YY Giving Next-R-YY.
```

Controlling Date Processing Explicitly

There may be times when you want COBOL data items to be treated as date fields only under certain conditions, or only in specific parts of the program. Or your application may contain 2-digit year date fields that cannot be declared as windowed date fields because of some interaction with another software product. For example, if a date field is used in a context where it is only recognized by its true

binary contents without further interpretation, the date in this field cannot be windowed. This includes:

- A key on a VSAM file
- A search field in a database system such as DL/I or SQL/DS
- A key field in a CICS command

Conversely, there may be times when you want a date field to be treated as a non-date in specific parts of the program.

COBOL provides two intrinsic functions to cater for these conditions:

DATEVAL Converts a non-date into a date field.

UNDATE Converts a date field into a non-date.

DATEVAL

You can use the DATEVAL intrinsic function to convert a non-date into a date field, so that COBOL will apply the relevant date processing to the field. The first argument to the function is the non-date to be converted, and the second argument specifies the date format. The second argument is a literal string with a specification similar to that of the date pattern in the DATE FORMAT clause.

As an example, assume that a program contains a field Date-Copied, and that this field is referenced many times in the program, but most of these references simply move it between records or reformat it for printing. Only one reference relies on it containing a date, for comparison with another date.

In this case, it is better to leave the field as a non-date, and use the DATEVAL intrinsic function in the comparison statement. For example:

```
03  Date-Distributed Pic 9(6) Date Format yyxxxx.
03  Date-Copied      Pic 9(6).
⋮
If FUNCTION DATEVAL(Date-Copied "YYXXXX") Less than
                    Date-Distributed ...
```

In this example, the DATEVAL intrinsic function converts Date-Copied into a date field so that the comparison will be meaningful.

In most cases, the compiler makes the correct assumption about the interpretation of a non-date, but accompanies this assumption with a warning-level diagnostic message. This typically happens when a windowed date is compared to a literal:

```
03  When-Made        Pic x(6) Date Format yyxxxx.
⋮
If When-Made = "850701" Perform Warranty-Check.
```

The literal is assumed to be a compatible windowed date but with a century window of 1900–1999, thus representing July 15, 1985. You can use the DATEVAL intrinsic function to make the year of the literal date explicit, and eliminate the warning message:

```
If When-Made = Function Dateval("19850701" "YYYYXXXX")
    Perform Warranty-Check.
```

For a full description and syntax of the DATEVAL intrinsic function, see the *COBOL/VSE Language Reference*.

UNDATE

The UNDATE intrinsic function converts a date field to a non-date, so that it can be referenced without any date processing.

In the following example, the field Invoice-Date in Invoice-Record is a windowed Julian date. In some records, it contains a value of "00999" to indicate that this is not a “true” invoice record, but a record containing file control information.

Invoice-Date has been given a DATE FORMAT clause because most of its references in the program are date-specific. However, in the instance where it is checked for the existence of a control record, the value of "00" in the year component will lead to some confusion. A year of "00" in Invoice-Date will represent a “true” year of either 1900 or 2000, depending on the century window. This is compared to a non-date (the literal "00999" in the example), which will always be windowed against the assumed century window and will therefore always represent the year 1900.

To ensure a consistent comparison, you should use the UNDATE intrinsic function to convert Invoice-Date to a non-date. This means that the IF statement is not comparing any date fields, so it does not need to apply any windowing. For example:

```
01  Invoice-Record.
    03  Invoice-Date    Pic x(5) Date Format yyxxx.
    ⋮
    If FUNCTION UNDATE(Invoice-Date) Equal "00999" ...
```

For a full description and syntax of the UNDATE intrinsic function, see the *COBOL/VSE Language Reference*.

Analyzing 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 non-date 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 run-time results are unpredictable.
- Severe-level, to indicate that the usage of the date field is incorrect. The statement that generated this error is discarded from the compilation.

You should always eliminate error-level and severe-level messages from your program by correcting the use of the date fields in the affected statements. Warning-level messages deserve special attention because the assumptions that the compiler makes may not be correct.

Your approach to warning-level diagnostic messages should be:

1. Avoidance. Be specific in your program changes, so that the compiler does not need to make assumptions about your intended use of date fields (see “Avoiding Warning-Level Messages” on page 381).
2. Analysis. Examine each diagnostic message, and either eliminate it, or, where you consider it to be unavoidable, ensure that the compiler's assumptions are correct (see “Analyzing Warning-Level Messages”).

Avoiding Warning-Level Messages

To avoid warning-level diagnostic messages, follow these simple guidelines:

- Don't specify a date field in a context where a date field doesn't make sense, such as a FILE STATUS, PASSWORD, ASSIGN USING, LABEL RECORD, or LINAGE item. If you do, you'll get a warning-level message and the date field will be treated as a non-date.
- Ensure that implicit or explicit aliases for date fields are compatible, such as in a group item that consists solely of a date field.
- Ensure that if a date field is defined with a VALUE clause, the value is compatible with the date field definition.
- Use the DATEVAL intrinsic function if you want a non-date treated as a date field, such as when moving a non-date to a date field, or comparing a windowed date field with a non-date and you want a windowed date comparison. If you don't use DATEVAL, the compiler will make an assumption about the use of the non-date, and produce a warning-level diagnostic message. Even if the assumption is correct, you may want to use DATEVAL just to eliminate the message. For more information on the DATEVAL intrinsic function, see “DATEVAL” on page 379.
- Use the UNDATE intrinsic function if you want a date field treated as a non-date, such as moving a date field to a non-date, or comparing a non-date and a windowed date field and you don't want a windowed comparison. For more information on the UNDATE intrinsic function, see “UNDATE” on page 380.

Analyzing Warning-Level Messages

The following guidelines will help you to analyze date-related warning-level diagnostic messages:

- The diagnostic messages may indicate some date data items that should have had a DATE FORMAT clause but were missed. You should either add DATE FORMAT clauses to these items, or use the DATEVAL intrinsic function in references to them.
- Pay particular attention to the usage of literals in relation conditions involving date fields or in arithmetic expressions that include date fields. Note that the DATEVAL function may be used on literals (as well as non-date data items) to specify a DATE FORMAT pattern to be used, and the UNDATE function may be used to enable a date field to be used in a context where date-oriented behavior is not desired.
- With the REDEFINES and RENAMES clauses, the compiler may produce a warning-level diagnostic message if a date field and a non-date occupy the same storage location. You should check these cases carefully to confirm that all uses of the various aliased data items are correct, and that none of the perceived non-date redefinitions actually are dates or can adversely affect the date logic in the program.

Other Potential Problems

When you change a COBOL program to make use of the millennium language extensions, you may find that some parts of the program need special attention to resolve unforeseen changes in behavior. This section outlines some of those areas that you may need to consider.

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. This is because the internal representation of packed decimal numbers always allows for an odd number of digits (for details of internal representation, see “Internal Representation of Numeric Items” on page 77).

A field that holds a 6-digit Gregorian date, for example, may be declared as PIC S9(6) COMP-3, and this will reserve four bytes of storage. But the programmer may have declared the field as PIC S9(7), knowing that this would reserve the same four bytes, with the high-order digit always containing a zero.

If you simply add a DATE FORMAT YYXXXX clause to this field, the compiler will give you 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 check carefully 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. Other actions you can take include:

- Using a REDEFINES clause to define the field as both a date and a non-date (this 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

Contracting Moves

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, this means that the leading two digits of the year are truncated.

Depending on the contents of the sending field, the results of such a move may 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.

Principles

To gain the most benefit from the millennium language extensions, it is important to understand the reasons for their introduction into the COBOL language, and the rationale behind their design. In particular, there are some apparent inconsistencies that only make sense with an understanding of what the extensions are, and what they are not.

You should not consider using the extensions in new applications, or in enhancements to existing applications, unless the applications are using old data that cannot be expanded until later.

The extensions do not provide fully-specified or complete date-oriented data types, with semantics that recognize, for example, the month and day parts of Gregorian dates. They do however provide special semantics for the year part of dates.

The millennium language extensions focus on a few key principles:

1. Programs to be re-compiled with date semantics are fully-tested and valuable assets of the enterprise. Their only relevant limitation is that any 2-digit years in the programs are restricted to the range 1900–1999.
2. No special processing is done for the non-year part of dates. That is why the non-year part of the supported date formats is denoted by Xs. To do otherwise might change the meaning of existing programs. The only date-sensitive semantics that are provided involve automatically expanding (and contracting) the 2-digit year part of dates with respect to the century window for the program.
3. Dates with 4-digit year parts are generally only of interest when used in combination with windowed dates. Otherwise there is little difference between 4-digit year dates and non-dates.

Objectives

Based on these principles, the millennium language extensions are designed to meet a number of objectives. You should evaluate the objectives that you need to meet in order to resolve your date processing problems, and compare them against the objectives of the millennium language extensions, to determine how your application can benefit from them.

The objectives of the millennium language extensions are as follows:

1. The primary objective is to extend the useful life of your application programs, as they are currently specified, into the twenty-first century.
2. Source changes to accomplish this must be held to the bare minimum, preferably limited to augmenting the declarations of date fields in the Data Division. To implement basic remediation of date problems, you should not be required to make any changes to the program logic in the Procedure Division.
3. The existing semantics of the programs will not be changed by the addition of date fields. For example, where a date is expressed as a literal, as in:

   ```
   If Expiry-Date Greater Than 980101 ...
   ```

   the literal is considered to be compatible (windowed or expanded) with the date field to which it is compared. Further, because the existing program assumes that 2-digit year dates expressed as literals are in the range 1900–1999, the

extensions do not change this assumption (see “The Assumed Window” on page 386).

4. The windowing feature is not intended for long-term use. Its intention is to extend the useful life of applications through the year 2000, as a start towards a long-term solution that can be implemented later.
5. The expanded date field feature *is* intended for long-term use, as an aid for expanding date fields in files and databases.

Concepts

With these principles and objectives in mind, you can better understand some of the concepts of the millennium language extensions, and how they interact with other parts of COBOL. This section describes some of these concepts.

Date Semantics

All arithmetic, whether performed on date fields or not, acts only on the numeric contents of the fields; date semantics for the non-year parts of date fields are not provided. For example, adding 1 to a windowed Gregorian date field that contains the value 980831 gives a result of 980832, not 980901.

However, date semantics *are* provided for the year parts of date fields. For example, if the century window is 1950–2049, and the value of windowed date field TwoDigitYear is 49, then the following ADD statement will result in the SIZE ERROR imperative statement being executed:

```
Add 1 to TwoDigitYear
  on Size Error Perform CenturyWindowOverflow
End-Add
```

Compatible Dates

The meaning of the term *compatible dates* depends on the COBOL division in which the usage occurs, as follows:

- The Data Division usage is concerned with the declaration of date fields, and the rules governing COBOL language elements such as subordinate data items and the REDEFINES clause. In the following example, Review-Date and Review-Year are compatible because Review-Year can be declared as a subordinate data item to Review-Date:

  ```
  01  Review-Record.
      03  Review-Date            Date Format yyxxxx.
          05  Review-Year Pic XX Date Format yy.
          05  Review-M-D  Pic XXXX.
  ```

  For full details, see the *COBOL/VSE Language Reference*.

- The Procedure Division usage is concerned with how date fields can be used together in operations such as comparisons, moves, and arithmetic expressions. To be considered compatible, date fields must have the same number of non-year characters. For example, a field with DATE FORMAT YYXXXX is compatible with another field that has the same DATE FORMAT, and with a YYYYXXXX field, but not with a YYXXX field.

The remainder of this discussion relates to the Procedure Division usage of compatible dates.

You can perform operations on date fields, or on a combination of date fields and non-dates, provided that the date fields in the operation are compatible. For example, with the following definitions:

```
01  Date-Gregorian-Win  Pic 9(6) Packed Date Format yyxxxx.
01  Date-Julian-Win     Pic 9(5) Packed Date Format yyxxx.
01  Date-Gregorian-Exp  Pic 9(8) Packed Date Format yyyyxxxx.
```

The statement:

```
If Date-Gregorian-Win Less than Date-Julian-Win ...
```

is inconsistent because the number of non-year digits is different between the two fields. The statement:

```
If Date-Gregorian-Win Less than Date-Gregorian-Exp ...
```

is accepted because the number of non-year digits is the same for both fields. In this case the century window is applied to the windowed date field (Date-Gregorian-Win) to ensure that the comparison is meaningful.

Where a non-date is used in conjunction with a date field, the non-date is either assumed to be compatible with the date field, or treated as a simple numeric value, as described in the following section.

Treatment of Non-Dates

The simplest kind of non-date is just a literal value. The following items are also non-dates:

- A data item whose data description does not include a DATE FORMAT clause.
- The results (intermediate or final) of some arithmetic expressions. For example, the difference of two date fields is a non-date, wheras the sum of a date field and a non-date is a date field.
- The output from the UNDATE intrinsic function.

When you use a non-date in conjunction with a date field, the compiler interprets the non-date as either a date whose format is compatible with the date field, or a simple numeric value. This interpretation depends on the context in which the date field and non-date are used, as follows:

Comparison

Where a date field is compared to a non-date, the non-date is considered to be compatible with the date field in the number of year and non-year characters. In the following example:

```
01  Date-1              Pic 9(6) Date Format yyxxxx.
    ⋮
    If Date-1 Greater than 971231 ...
```

Because the non-date literal 971231 is being compared to a windowed date field, it is treated as if it had the same DATE FORMAT as Date-1, but with a base year of 1900.

Arithmetic operations

In all supported arithmetic operations, non-date fields are treated as simple numeric values. In the following example:

```
01  Date-2              Pic 9(6) Date Format yyxxxx.
    ⋮
    Add 10000 to Date-2.
```

the numeric value 10000 is added to the Gregorian date in Date-2, effectively adding one year to the date.

MOVE statement

Moving a date field to a non-date is not supported. However, you can use the UNDATE intrinsic function to achieve this. For more information, see "UNDATE" on page 380.

When you move a non-date to a date field, the sending field is assumed to be compatible with the receiving field in the number of year and non-year characters. For example, when you move a non-date to a windowed date field, the non-date field is assumed to contain a compatible date with a 2-digit year.

The Assumed Window

Where the program operates on windowed date fields, the compiler applies the century window for the compilation unit; that is, the one defined by the YEARWINDOW compiler option. Where a windowed date field is used in conjunction with a non-date, and the context demands that the non-date also be treated as a windowed date, the compiler uses an assumed century window to resolve the non-date field.

The assumed century window is 1900–1999, which is typically not the same as the century window for the compilation unit.

In many cases, particularly for literal non-dates, this assumed century window will be the correct choice. For example, in the construct:

```
01  Manufacturing-Record.
    03  Makers-Date Pic X(6) Date Format yyxxxx.
    ⋮
    If Makers-Date Greater than "720101" ...
```

the literal should retain its original meaning of January 1, 1972, and not change to 2072 if the century window is, for example, 1975–2074. Even if the assumption is correct, it is better to make the year explicit, and also eliminate the warning-level diagnostic message that accompanies application of the assumed century window, by using the DATEVAL intrinsic function:

```
    If Makers-Date Greater than
                 Function Dateval("19720101" "YYYYXXXX") ...
```

In other cases however, the assumption may not be correct. For example:

```
01  Project-Controls.
    03  Date-Target     Pic 9(6).
    ⋮
01  Progress-Record.
    03  Date-Complete   Pic 9(6) Date Format yyxxxx.
    ⋮
    If Date-Complete Less than Date-Target ...
```

For this example, assume that Project-Controls is in a COPY member that is used by other applications that have not yet been upgraded for Year 2000 processing,

and therefore Date-Target cannot have a DATE FORMAT clause. In the example, if:

- The century window is 1910–2009,
- Date-Complete is 991202 (Gregorian date: December 2, 1999), and
- Date-Target is 000115 (Gregorian date: January 15, 2000),

then:

- Date-Complete is earlier than (less than) Date-Target.

However, because Date-Target does not have a DATE FORMAT clause, it is a non-date, so the century window applied to it is the assumed century window of 1900–1999, which means that it is processed as January 15, 1900. So Date-Complete will be greater than Date-Target, which is not the desired result.

In this case, you should use the DATEVAL intrinsic function to convert Date-Target to a date field for this comparison. For example:

```
If Date-Complete Less than
   Function Dateval (Date-Target "YYXXXX") ...
```

For more information on the DATEVAL intrinsic function, see “DATEVAL” on page 379.

Chapter 23. Target Environment Considerations

For certain COBOL/VSE functions and features, there are requirements based on the particular environment in which your program will run. These are outlined briefly in the sections that follow.

For detailed information on developing applications for specialized subsystems, you should refer to the publication for the particular system of interest, as listed in the “Bibliography” on page 448.

COBOL/VSE Programming Considerations for CICS

When developing COBOL/VSE programs that run under CICS, certain considerations and restrictions apply. A discussion of these coding considerations and restrictions follows. For additional information on developing programs to run under CICS, consult the CICS application programming guide appropriate for your CICS environment (see “Bibliography” on page 448 for a list of CICS books).

For information on the run-time considerations and restrictions of running COBOL/VSE programs under CICS, see the *LE/VSE Programming Guide*.

If you have written DOS/VS COBOL programs that run under CICS, see *COBOL/VSE Migration Guide* for information on differences between DOS/VS COBOL and COBOL/VSE under CICS.

Developing a COBOL/VSE Program for CICS

COBOL/VSE programs written for CICS can run under CICS/VSE Version 2 Release 3 or later.

The following list summarizes the unique considerations of developing and running COBOL/VSE programs for and under CICS, respectively. Each of these is discussed in greater detail in subsequent sections.

- Coding input/output in CICS

  CICS handles all input/output between the application program and devices (including terminals). Therefore, instead of using COBOL input/output statements to perform input/output, use CICS commands.

- Compiler Options

  When coding for CICS, certain compiler options are either required or recommended.

- CICS Reserved Word Table

  The CICS reserved word table supplied by IBM can be used during compilation to flag certain COBOL language elements not supported under CICS.

- Using CICS HANDLE with COBOL/VSE Programs

  There are certain considerations to be aware of when using a CICS HANDLE command to handle conditions, aids, and abends caused by a COBOL/VSE subprogram.

- CICS coding restrictions

Certain COBOL statements are not allowed under CICS.

- Translating CICS commands into COBOL

  The CICS translator interprets CICS commands and generates COBOL code.

- Compiling CICS Code

  The COBOL/VSE compiler compiles the code generated by the CICS translator. COBOL/VSE programs that use CICS commands must be link-edited with the CICS stub.

- Calls under CICS

  Certain CALL restrictions and requirements must be observed if your program is to run under CICS.

After compilation and link-edit, there are other steps needed before the COBOL program can run under CICS. For example, CICS tables must be updated. However, these CICS topics are beyond this book's focus. More information can be found in the appropriate CICS application programming guide (see “Bibliography” on page 448 for a list of CICS books).

CICS Commands and the Procedure Division

To code your program, you need to know how to code CICS commands within the Procedure Division. Processing logic must be written in COBOL.

CICS commands are statements you include in the Procedure Division of your application program. They have the following basic format:

```
EXEC CICS command name and command options
END-EXEC
```

Coding Input/Output in CICS

All input/output is handled through CICS commands. Therefore, you do not describe files or code any OPEN, CLOSE, READ, START, REWRITE, WRITE, DELETE, ACCEPT or DISPLAY statements. Instead, you use CICS commands to retrieve, update, insert, and delete data.

Compiler Options

Required Options: When coding for CICS, these compiler options are required:

RENT
NODYNAM
LIB (if the program has a COPY or BASIS statement in it)

Note: The CICS translator always inserts a line into COBOL/VSE programs that specifies:

```
CBL RES,RENT,NODYNAM,LIB
```

There is no longer a RES compiler option for COBOL/VSE. If you specify it, you will receive an informational message.

Recommended Options: TRUNC(BIN) is recommended for those applications that use binary data items that may contain more than 9 digits in a fullword or more than 4 digits in a halfword.

WORD(CICS) is recommended if you want those COBOL language elements not supported under CICS to be flagged at compile time.

Options with No Effect: These compiler options have no effect under CICS:

ADV
FASTSRT
OUTDD

For detailed information on individual compiler options, see pages 224–257.

CICS Reserved Word Table

COBOL/VSE provides an alternate reserved word table (IGYCCICS) specifically for CICS application programs. It is set up so that COBOL words not supported under CICS are flagged by the compiler with an error message.

Contents of the table: In addition to the COBOL words restricted by the default reserved word table supplied by IBM, the CICS reserved word table restricts as supplied by IBM the following COBOL words:

ACCEPT	FILE-CONTROL	RERUN
CLOSE	INPUT-OUTPUT	REWRITE
DELETE	I-O-CONTROL	SD
DISPLAY	MERGE	SORT
FD	OPEN	START
FILE	READ	WRITE

SORT Users

If you intend to use the SORT statement under CICS (COBOL/VSE supports an interface for the SORT statement under CICS), you must modify the CICS reserved word table before using it. The words underlined above must be removed from the list of words marked as restricted, because they are required for the SORT function.

For instructions on how to customize the CICS reserved word table, refer to the *COBOL/VSE Installation and Customization Guide* or see your system programmer.

How to Use the Table: Specify the compiler option WORD(CICS) if you want the CICS reserved word table to be used during your compilation. For a description of the WORD compiler option, refer to page 255.

Using CICS HANDLE with COBOL/VSE Programs

Effect of the CBLPSHPOP Run-time Option

The setting of the CBLPSHPOP run-time option affects the state of the HANDLE specifications when calling a COBOL/VSE subprogram. For more information about the CBLPSHPOP run-time option see the *LE/VSE Programming Reference*.

CBLPSHPOP(ON): When CBLPSHPOP is ON, the LE/VSE run time performs the following when a COBOL/VSE subprogram is called:

- As part of program initialization, the run time does an EXEC CICS PUSH HANDLE.
- As part of program termination, the run time does an EXEC CICS POP HANDLE.

Therefore with CBLPSHPOP(ON), when a COBOL/VSE program calls another COBOL/VSE program, the calling program's HANDLE specifications are suspended. The default actions for HANDLE apply until the called program issues its own HANDLE commands. The effects of the calling program's HANDLE specifications are reinstated upon return.

Note: HANDLE conditions are not suspended when calling a nested program.

It is recommended that you run with CBLPSHPOP(ON) if any of your called COBOL/VSE subprograms uses one or more of the following CICS commands:

- CICS HANDLE CONDITION
- CICS HANDLE AID
- CICS HANDLE ABEND
- CICS IGNORE CONDITION
- CICS PUSH HANDLE
- CICS POP HANDLE

CBLPSHPOP(OFF): When CBLPSHPOP is OFF, the LE/VSE run time will not perform the CICS PUSH/POP on a call to a COBOL/VSE subprogram.

In this case, when a COBOL/VSE program calls another COBOL/VSE program, the calling program's HANDLE specifications are **not** suspended.

CICS HANDLE Restrictions

COBOL/VSE does not support the use of the CICS HANDLE command with the LABEL option to handle conditions, aids, and abends in a program which were caused by another program invoked using the COBOL CALL statement. Attempts to perform cross program branching due to the use of the CICS HANDLE command with the LABEL option will result in a transaction abend.

Note: If a condition, aid, or abend occurs in a nested program, then the LABEL for the condition, aid, or abend must be in the same nested program; otherwise unpredictable results will occur.

Figure 127 on page 392 illustrates 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.

The following illustrates how a condition is handled in three example scenarios.

1 CBLPSHPOP(ON): If the CICS READ command in program B causes a condition, the condition will not be handled by program A (the handle specifications have been suspended because the run time performed a CICS PUSH HANDLE); the condition will turn into a transaction abend.

2 CBLPSHPOP(OFF): If the CICS READ command in program B causes a condition, the condition will not be handled by program A (the run time will diagnose the attempt to perform cross program branching due to the use of a CICS HANDLE command with the LABEL option); the condition will turn 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 will 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' USING DFHEIBLK DFHCOMMAREA.
      .
      .
 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.
```

Figure 127. CICS HANDLE in COBOL Programs

Coding Restrictions

The following restrictions should be followed when coding COBOL/VSE programs that run under CICS:

- Do not use EXEC, CICS, DLI, and END-EXEC for variable names.
- Do not use the FILE-CONTROL entry in the Environment Division, unless it is being used for a SORT statement.
- Do not use the FILE SECTION of the Data Division, unless it is being used for a SORT statement.
- Do not use user-specified parameters to the main program.
- Do not use USE declaratives (except USE FOR DEBUGGING).
- Do not use these COBOL language statements:

  ACCEPT (Format 1 or 2—see “System Date under CICS” on page 394)
  CLOSE
  DELETE
  DISPLAY
  MERGE
  OPEN
  READ
  RERUN
  REWRITE
  START
  STOP *literal*
  WRITE

- The following restrictions apply to a COBOL/VSE program running above the 16-megabyte line:

  1. BMS (Basic Mapping Support) maps, map sets, and partition sets resident above the 16-megabyte line are not supported.
  2. If the receiving program is link-edited with AMODE=31, addresses passed must be 31 bits long, or 24 bits long with the leftmost byte set to zeros.
  3. If the receiving program is link-edited with AMODE=24, addresses passed must be 24 bits long.

- The ON OVERFLOW phrase and ON EXCEPTION phrase of the CALL statement are supported under CICS with the following exception:

  – If the COBOL/VSE program has been compiled with the CMPR2 compiler option, there are no conditions under CICS which will cause the statement specified by the ON OVERFLOW to be executed.

- If you use the CICS HANDLE CONDITION or 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.
- Calls

  – For the list of CALL restrictions and requirements, see “Calls under CICS” on page 395.

- REPLACE statements that contain EXEC commands must occur after the PROCEDURE DIVISION statement of the program for the EXEC commands to be translated.
- When coding nested (contained) programs, you must pass the DFHEIB and DFHCOMMAREA parameters to any nested programs that contain EXEC commands and/or references to the EIB (EXEC Interface Block). The same parameters must also be passed to any program that forms part of the control hierarchy between such a program and its top level program.
- The space character is **not** interchangeable with a comma or semicolon within EXEC commands: You must use the space as a word separator.

COBOL 85 Standard Considerations

CICS/VSE features the translator option, ANSI85, that supports most of the new language features introduced by the COBOL 85 Standard. The ANSI85 translator option supports the following language features:

- 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

See the *CICS/VSE Application Programming Guide* for detailed information on the ANSI85 translator support.

Translating CICS Commands into COBOL

The CICS translator takes your source program, and converts the EXEC CICS commands to COBOL code. The translator replaces each EXEC CICS command with one or more COBOL statements, one of which is a CALL statement.

Compiling and Link-Editing CICS Code

The CICS translator generates a CBL statement to force the following compiler options for your program: RES, RENT, NODYNAM, and LIB. You cannot replace these compiler options with options passed when you invoke the compiler (for example, with PARM=...).

The input file for the compiler is the file you received as a result of the translation, which is SYSPCH by default.

For information on link-edit considerations, see the *LE/VSE Programming Guide*.

System Date under CICS

You should not use a Format 1 ACCEPT statement in a CICS program. Format 2 ACCEPT is supported with the four-digit year options; that is:

```
ACCEPT identifier FROM DATE YYYYMMDD
ACCEPT identifier FROM DAY YYYYDDD
```

The recommended ways of retrieving the system date in a CICS program are these forms of the ACCEPT statement, and the CURRENT-DATE intrinsic function. These methods work in both CICS and non-CICS environments.

Note: The following forms of the ACCEPT statement to receive 2-digit year dates are not supported under CICS:

```
ACCEPT identifier FROM DATE
ACCEPT identifier FROM DAY
```

Calls under CICS

If your COBOL/VSE program runs under CICS, observe these CALL restrictions and requirements:

- You can make calls to and from VS COBOL II and/or COBOL/VSE programs.
- You can call a pre LE-conforming assembler-language program from a COBOL/VSE program. You cannot call a COBOL/VSE or a VS COBOL II program from a pre LE-conforming assembler-language program.
- Calls to LE-conforming assembler-language programs are supported. Calls from LE-conforming assembler-language programs are supported with the restriction that the assembler-language program is not a main program.
- The NODYNAM compiler option must be used if the COBOL/VSE program has been translated by the CICS translator.
- CALL *identifier* can be used with the NODYNAM compiler option to dynamically call a program. Called programs can contain any function supported by CICS for the language.
- If you are calling a COBOL/VSE program that has been translated, you must pass the EIB and COMMAREA as the first two parameters in the CALL statement.
- COBOL/VSE and VS COBOL II programs cannot CALL or be CALLed by DOS/VS COBOL programs. EXEC CICS LINK must be used instead.
- Support for interlanguage communication (ILC) with other HLL languages is available. For more detailed information on ILC, see *LE/VSE Writing Interlanguage Communication Applications*. Where ILC is not supported, you can use CICS LINK, XCTL, and RETURN instead.

COBOL/VSE Programming Considerations for DL/I

Although much of the coding of a COBOL/VSE program will be the same when running under DL/I, you should be aware of the following recommendations and restrictions.

Using CEETDLI to Interface to DL/I

With COBOL/VSE you can invoke DL/I facilities using the following interfaces:

- CBLTDLI Call
- The LE/VSE callable service CEETDLI

Calls to CEETDLI are coded the same way as calls to CBLTDLI, and CEETDLI performs essentially the same function as CBLTDLI. However, CEETDLI does offer increased condition handling capabilities.

For a complete description of CEETDLI, including its syntax, see the *LE/VSE Programming Reference*. For considerations on condition handling under DL/I, see the *LE/VSE Programming Guide*.

For Mixed COBOL/VSE, VS COBOL II, and DOS/VS COBOL Applications

In an application with any mixture of COBOL/VSE, VS COBOL II, and DOS/VS COBOL programs, the following compiler options are recommended:

- RENT for COBOL/VSE programs
- RENT and RES for VS COBOL II programs

COBOL/VSE Programming Considerations for SQL/DS

In general, most of the coding for your COBOL/VSE programs will be the same when you want to use SQL/DS. However, to retrieve, update, insert, and delete SQL/DS services, you must use SQL statements.

When SQL/DS completes processing an SQL statement, it passes the return code back in the communications area and not in Register 15. Therefore, the COBOL/VSE RETURN-CODE special register may contain an invalid value. Because a COBOL/VSE program stores its RETURN-CODE special register into Register 15 before it returns to its caller, your COBOL/VSE program should set the RETURN-CODE special register to a meaningful value before returning to its caller.

For details on embedding SQL statements in COBOL programs, see *SQL/DS Application Programming Guide*.

Part 5. Appendixes

Appendix A. COBOL/VSE Compiler Limits

The following table lists the compiler limits for COBOL/VSE programs. Other operating systems may impose further limits. The numbers are **guidelines** to the limits.

Figure 128 (Page 1 of 3). COBOL/VSE Compiler Limits

Language Element	Limit(s)
Size of program	999,999 lines
Number of literals	4,194,303[1]
Total length of literals	4,194,303 bytes[1]
Reserved Word Table entries	1536
COPY REPLACING ... BY ... (items per COPY statement)	N/A
Number of COPY libraries	N/A
Block size of COPY library	32,767 bytes
Identification Division	
Environment Division	
Configuration Section	
SPECIAL-NAMES paragraph	
function-name IS	18
UPSI-n ... (switches)	0-7
alphabet-name IS ...	N/A
literal THRU/ALSO ...	256
Input-Output Section	
FILE-CONTROL paragraph	
SELECT file-name ...	65,535
ASSIGN system-name ...	N/A[2]
ALTERNATE RECORD KEY data-name ...	253
RECORD KEY length	N/A[3]
RESERVE integer (buffers)	255[4]
I-O-CONTROL paragraph	
RERUN ON system-name ...	32,767
integer RECORDS	16,777,215
SAME RECORD AREA	255
FOR file-name ...	255
SAME SORT/MERGE AREA	N/A[2]
MULTIPLE FILE ... file-name	N/A[2] **Note:** The MULTIPLE FILE TAPE phrase is ignored.
Data Division	
File Section	

Figure 128 (Page 2 of 3). COBOL/VSE Compiler Limits

Language Element	Limit(s)
FD file-name ...	65,535
LABEL data-name ...	255
(if no optional clauses)	
Label record length	80 bytes
DATA RECORD dnm ...	N/A[2]
BLOCK CONTAINS integer	1,048,575[5]
RECORD CONTAINS integer	1,048,575[5]
Item length	1,048,575 bytes[5]
SD file-name ...	65,535
DATA RECORD dnm ...	N/A[2]
Sort record length	32,751 bytes
Working-Storage Section	
items without the EXTERNAL attribute	134,217,727 bytes
items with the EXTERNAL attribute	134,217,727 bytes
77 data-names	16,777,215 bytes
01-49 data-names	16,777,215 bytes
88 condition-name ...	N/A
VALUE literal ...	N/A
66 RENAMES ...	N/A
PICTURE character string	30
Numeric item digit positions	18
Num-edit character positions	249
PICTURE replication ( )	16,777,215
PIC repl (editing)	32,767
DBCS Picture replication ( )	8,388,607
Group item size:	
File section	1,048,575 bytes
Elementary item size	16,777,215 bytes
VALUE initialization	16,777,215 bytes
(Total length of all value literals)	
OCCURS integer	16,777,215
Total number of ODOs	4,194,303[1]
Table size	16,777,215 bytes
Table element size	8,388,607 bytes
ASC/DES KEY ...	
(per OCCURS clause)	12 KEYS
Total length	256 bytes
INDEXED BY ... (index names)	
(per OCCURS clause)	12
Total num of indexes (index names)	65,535
Size of relative index	32,765
Linkage Section	134,217,727 bytes
Total 01 + 77 (data items)	N/A
Procedure Division	

Figure 128 (Page 3 of 3). COBOL/VSE Compiler Limits

Language Element	Limit(s)
Procedure + constant area	4,194,303 bytes[1]
USING identifier ...	32,767
Procedure-names	1,048,575[1]
Subscripted data-names per verb	32,767
Verbs per line (TEST)	7
ADD identifier ...	N/A
ALTER pn1 TO pn2 ...	4,194,303[1]
CALL ... BY CONTENT id	2,147,483,647 bytes
CALL id/lit	
USING id/lit...	16380
CALL literal ...	4,194,303[1]
Active programs in run unit	32,767
number of names called (DYN)	N/A
CANCEL id/lit ...	N/A
CLOSE file-name ...	N/A
COMPUTE identifier ...	N/A
DISPLAY id/lit ...	N/A
DIVIDE identifier ...	N/A
ENTRY USING id/lit ...	N/A
EVALUATE ... subjects	64
EVALUATE ... WHEN clauses	256
GO pn ... DEPENDING	255
INSPECT TALLY/REPL clauses	N/A
MERGE file-name ASC/DES KEY ...	N/A
Total key length	3,072 bytes[6]
USING file-name ...	9[7]
MOVE id/lit TO id ...	N/A
MULTIPLY identifier ...	N/A
OPEN file-name	N/A
PERFORM	4,194,303
SEARCH ... WHEN ...	N/A
SET index/id ... TO	N/A
SET index ... UP/DOWN	N/A
SORT file-name ASC/DES KEY	N/A
Total key length	3,072 bytes[6]
USING file-name ...	9[7]
STRING identifier ...	N/A
DELIMITED id/lit ...	N/A
UNSTRING	
DELIMITED id/lit OR id/lit ...	255
INTO id/lit ...	N/A
USE ... ON file-name ...	N/A

Note:

1 Items included in 4,194,303 byte limit for procedure plus constant area.
2 Treated as comment; there is no limit.
3 No compiler limit, but VSAM limits it to 255 bytes.
4 The SAM limit is 2.
5 Compiler limit shown, but SAM limits it to 32,767 bytes.
6 For DFSORT/VSE, the limit is 3,072 bytes.
7 For DFSORT/VSE, the limit is 9 files.

Appendix B. Intermediate Results and Arithmetic Precision

The compiler treats arithmetic statements as a succession of operations, performed according to operator precedence, and sets up an intermediate field to contain the results of these operations.

Intermediate results are possible in the following cases:

- In an ADD or SUBTRACT statement containing multiple operands immediately following the verb
- In a COMPUTE statement specifying a series of arithmetic operations or multiple result fields
- In arithmetic expressions contained in conditional statements and reference modification specifications
- In the GIVING option with multiple result fields for the ADD, SUBTRACT, MULTIPLY, or DIVIDE statements
- In a statement with an intrinsic function used as an operand

For a discussion on when the compiler uses fixed-point or floating-point arithmetic, refer to “Fixed-Point versus Floating-Point Arithmetic” on page 88.

Calculating Precision of Intermediate Results

The compiler uses algorithms to determine the number of **integer** and **decimal** places reserved for intermediate results.

In the following discussion of how the compiler determines the number of integer and decimal places reserved for intermediate results, these abbreviations are used:

i The number of **integer** places carried for an intermediate result.

If the ROUNDED option is used, one more integer may be added for accuracy, if necessary.

d The number of **decimal** places carried for an intermediate result.

dmax In a particular statement, the largest of:

- The number of decimal places needed for the final result field(s)
- The maximum number of decimal places defined for any operand, except divisors or exponents
- The outer-dmax for any function operand

inner-dmax

The *inner-dmax* for a function is the largest of:

- The number of decimal places defined for any of its elementary arguments
- The *dmax* for any of its arithmetic expression arguments
- The *outer-dmax* for any of its embedded functions

outer-dmax
The number that determines how a function result contributes to operations outside of its own evaluation (e.g., if the function is an operand in an arithmetic expression or an argument to another function).

op1 The first operand in a generated arithmetic statement. For division, ***op1*** is the divisor.

op2 The second operand in a generated arithmetic statement. For division, ***op2*** is the dividend.

i1,i2 The number of integer places in ***op1*** and ***op2***, respectively.

d1,d2 The number of decimal places defined for ***op1*** and ***op2***, respectively.

ir The **I**ntermediate **R**esult field obtained from the execution of a generated arithmetic statement or operation. ***ir1***, ***ir2***,. . ., represent successive intermediate results. These intermediate results are generated either in registers or in storage locations. Successive intermediate results may have the same location.

The compiler treats each statement as a succession of operations. For example, the following statement:

```
COMPUTE Y = A + B * C - D / E + F ** G
```

is calculated as:

** *F*	BY *G*	yielding *ir1*
MULTIPLY *B*	BY *C*	yielding *ir2*
DIVIDE *E*	INTO *D*	yielding *ir3*
ADD *A*	TO *ir2*	yielding *ir4*
SUBTRACT *ir3*	FROM *ir4*	yielding *ir5*
ADD *ir5*	TO *ir1*	yielding *Y*

Fixed-Point Data and Intermediate Results

The number of integer and decimal places in an **intermediate result** can be determined by using the following guidelines:

Figure 129. Determining the Precision of an Intermediate Result

Operation	Integer Places	Decimal Places
+ or -	(*i1* or *i2*) + 1, whichever is greater	*d1* or *d2*, whichever is greater
*	*i1* + *i2*	*d1* + *d2*
/	*i2* + *d1*	(*d2* - *d1*) or *dmax*, whichever is greater

Note: You must define the operands of any arithmetic statements with enough decimal places to give the required accuracy in the final result.

Figure 130 indicates the action of the compiler when handling intermediate results for fixed-point numbers.

Figure 130. Determining When the Compiler May Shorten Intermediate Results

Value of i + d	Value of d	Value of i + dmax	Action Taken
<30 =30	Any value	Any value	*i* integer and *d* decimal places are carried for *ir*.
>30	<*dmax* =*dmax*	Any value	30-*d* integer and *d* decimal places are carried for *ir*.
	>*dmax*	<30 =30	*i* integer and 30-*i* decimal places are carried for *ir*.
		>30	30-*dmax* integer and *dmax* decimal places are carried for *ir*.

Exponentiations Evaluated in Fixed-Point Arithmetic

Exponentiation is represented by the expression **op1** ** **op2**. Based on the characteristics of **op2**, the compiler handles exponentiation of fixed-point numbers in one of three ways.

- When **op2** is expressed with decimals, floating-point rules (see "Floating-Point Data and Intermediate Results" on page 406) are used to calculate the exponentiation.
- When **op2** is an integral literal or constant, the value **d** is computed as

  *d = d1 * |op2|*

  When **op1** is a data-name or variable, the value **i** is computed as

  *i = i1 * |op2|*

  When **op1** is a literal or constant, the actual value of **op1** ** |**op2**| is computed and **i** is set equal to the number of integers in that value.

  Having calculated **i** and **d**, the compiler takes the action indicated in the following figure to handle intermediate results:

Figure 131. Determining When the Compiler May Shorten Intermediate Results for Exponentiation

Value of i + d	Other Conditions	Action Taken
<30	Any	*i* integer and *d* decimal places are carried for *ir*.
=30	*op1* has an odd number of digits	*i* integer and *d* decimal places are carried for *ir*.
	op1 has an even number of digits	The exponentiation is treated the same as it is when **op2** is an integral data-name or a variable, except in the case of a 30-digit integer raised to the power of literal 1, where the calculation is done according to the rules for *op1* with an odd number of digits.
>30	Any	The exponentiation is treated the same as it is when **op2** is an integral data-name or a variable.

If **op2** is negative, then the value of 1 is divided by the result produced by the preliminary calculation described above. The values of **i** and **d** that are used are calculated using the rules for division found on page 402.

- When **op2** is an integral data-name or a variable, then **dmax** decimals and **30-dmax** integers are used. In this case, **op1** is multiplied by itself (|*op2*| - 1) times. For example, the following statement:

  COMPUTE *Y = A ** B*, where B has a value of 4

  is calculated as:

MULTIPLY A	BY *A*	yielding *ir1*
MULTIPLY ir1	BY *A*	yielding *ir2*
MULTIPLY ir2	BY *A*	yielding *ir3*
MOVE ir3	TO *ir4*	which has *dmax* decimals

  The values of **i** and **d** that are used for the above multiplications are calculated using the rules for multiplication found on page 402.

  If *B* is positive, *Y = ir4*.

  If *B* is negative, however,

DIVIDE ir4	INTO *1*	yielding *ir5*, which has *dmax* decimals
Y = ir5		

  In the case where *op2 = 0*, the answer is 1. (Note that division-by-0 and exponentiation SIZE ERROR conditions apply. See *COBOL/VSE Language Reference* for specific information on the SIZE ERROR option.)

Fixed-point exponents with more than 9 significant digits are always shortened to 9 digits. If the exponent is a literal or constant, an E-level compiler diagnostic message is issued; otherwise, an informational message is issued at run time.

Shortened Intermediate Results

Whenever the number of digits in a decimal number is greater than 30, the field is shortened to 30 digits. You will get a warning message when you compile the program. If truncation occurs at run time, a message is issued and execution continues.

If you think an intermediate result field might exceed 30 digits, you can use floating-point operands (COMP-1 and COMP-2) to avoid truncation.

Binary Data and Intermediate Results

If an operation involving binary operands requires intermediate results greater than 18 digits, the compiler converts the operands to internal decimal before performing the operation. If the result field is binary, the result will be converted from internal decimal to binary.

Binary items are used most efficiently when the intermediate result is not greater than 9 digits.

Intrinsic Functions Evaluated in Fixed-Point Arithmetic

Integer functions and mixed functions can both return an integer result. The *inner-dmax* and *outer-dmax* values are determined by the characteristics of the function.

Integer Functions: These functions always return an integer, and the *outer-dmax* will always be zero. For those functions whose arguments must be integer, the *inner-dmax* will also always be zero. The following table summarizes the precision of the function results:

Figure 132. Precision of Integer Intrinsic Functions

Function	Inner-Dmax	Outer-Dmax	Function Result
DATE-OF-INTEGER	0	0	8-digit integer
DAY-OF-INTEGER	0	0	7-digit integer
FACTORIAL	0	0	fixed-point, 30-digit integer
INTEGER-OF-DATE	0	0	7-digit integer
INTEGER-OF-DAY	0	0	7- digit integer
LENGTH	n/a	0	9- digit integer
MOD	0	0	integer with as many digits as min(i1 i2)
ORD	n/a	0	3-digit integer
ORD-MAX		0	9-digit integer
ORD-MIN		0	9-digit integer
INTEGER		0	With a fixed-point argument, result will be fixed-point integer with one more integer digit than the argument. With a floating-point argument, result will be fixed point, 30-digit integer.
INTEGER-PART		0	With a fixed-point argument, result will be fixed-point integer with the same number of integer digits as the argument. With a floating-point argument, result will be fixed-point, 30-digit integer.

Mixed Functions: When the compiler treats a mixed function as fixed-point arithmetic, the result will be either integer or fixed point with decimals (when any argument is floating point, the function becomes a floating-point function and will follow floating-point rules). For MAX, MIN, RANGE, REM, and SUM, the *outer-dmax* is always equal to the *inner-dmax*. To determine the precision of the result returned for these functions, apply the rules for fixed-point arithmetic to each step in the algorithm used to calculate the function result.

MAX and MIN

1. Assign the first argument to your function result.
2. For each remaining argument:
   a. Compare the algebraic value of your function result with the argument.
   b. For MAX assign the greater of the two, and for MIN assign the lesser of the two to your function result.

RANGE

1. Use the steps for MAX to select your maximum argument
2. Use the steps for MIN to select your minimum argument
3. Subtract the minimum argument from the maximum
4. Assign the difference to your function result

REM

1. Divide argument-1 by argument-2
2. Remove all noninteger digits from the result of step 1
3. Multiply the result of step 2 by argument-2
4. Subtract the result of step 3 from argument-1
5. Assign the difference to your function result

SUM

1. Assign the value 0 to your function result
2. For each argument:
   a. Add the argument to your function result
   b. Assign the sum to your function result

Floating-Point Data and Intermediate Results

Floating-point instructions are used to compute an arithmetic expression if any of the following conditions are true:

- A receiver or operand in the expression is COMP-1, COMP-2, external floating-point data, or a floating-point literal
- An exponent contains decimal places
- An exponent is an expression that contains an exponentiation or divide operator **and** ***dmax*** is greater than zero
- An intrinsic numeric function is a floating-point function

If any operation in an arithmetic expression is computed in floating point, the entire expression is computed as though all operands were converted to floating point and the operations evaluated using floating-point instructions.

If an expression is computed in floating point, double precision floating point is used if any receiver or operand in the expression is not COMP-1, or if a multiplication or exponentiation operation appears in the expression. Whenever double precision floating point is used for one operation in an arithmetic expression, then all operations in the expression are computed as though double precision floating-point instructions were used.

Note: If a floating-point operation has an intermediate result field in which exponent overflow occurs, the job will be abnormally terminated.

Exponentiations Evaluated in Floating-Point Arithmetic

Floating-point exponentiations are always evaluated using double precision floating-point arithmetic

The value of a negative number raised to a fractional power is undefined. For example, (-2) ** 3 is equal to -8, but (-2) ** (3.000001) is not defined. When an exponentiation is evaluated in floating point and there is a possibility that the value of the exponentiation will be undefined (as in the example above), then the value of the exponent is evaluated at run time to determine whether or not it is actually an integer.

Intrinsic Functions Evaluated in Floating-Point Arithmetic

The floating-point numeric functions will always return a double precision floating-point value. For a list of the floating-point, fixed-point and mixed functions see "Numeric Intrinsic Functions" on page 83.

Note: Remember that mixed functions with floating-point arguments will be evaluated using floating-point arithmetic.

ON SIZE ERROR and Intermediate Results

When the CMPR2 compiler option is in effect, the ON SIZE ERROR option for MULTIPLY and DIVIDE applies to both intermediate and final results. For the other arithmetic operations, the ON SIZE ERROR option applies only to the final calculated results.

Arithmetic Expressions in Nonarithmetic Statements

Arithmetic expressions may appear in contexts other than arithmetic statements, for example, the IF verb. In such statements, the rules for intermediate results, floating point, and double precision floating point apply, with the following modifications:

- Abbreviated IF statements are treated as though the statements were unabbreviated.
- An explicit relation condition exists when a required relational operator is used to define the comparison between two operands (herein referred to as comparands). In an explicit relation condition where one or both of the comparands is an arithmetic expression, the rules for intermediate results are determined taking into consideration the attributes of both comparands. That is to say, **dmax** is defined to be the maximum number of decimal places defined for any operand of either comparand, except divisors and exponents. The rules for floating point and double precision floating point apply if any of the following conditions are true:
  - Any operand in either comparand is COMP-1, COMP-2, external floating-point data, or a floating-point literal
  - An exponent contains decimal places
  - An exponent is an expression that contains an exponentiation or divide operator and **dmax** is greater than zero

  For example, in the statement:

  ```
  IF operand-1 = expression-1 THEN . . .
  ```

  where `operand-1` is a data-name defined to be COMP-2, and `expression-1` contains only fixed-point operands, the rules for floating-point arithmetic apply to `expression-1` since it is being compared to a floating-point operand,
- When the comparison between an arithmetic expression and either a data item or another arithmetic expression is defined without the use of a relational operator, then no explicit relation condition is said to exist. In this case, the arithmetic expression is evaluated without regard to the attributes of the operand with which the comparison is being made. For example, in the statement

```
EVALUATE expression-1
  WHEN expression-2 THRU expression-3
  WHEN expression-4
     ⋮
END-EVALUATE
```

each arithmetic expression is evaluated in fixed-point or floating-point arithmetic based on its own characteristics.

Appendix C. Coding Your Program for Cross-System Portability

Compatible Compilers

This appendix outlines the level of compatibility between:

IBM COBOL for VSE/ESA Version 1 Release 1, and

COBOL/370 Version 1 Release 1.

Subsequent changes to either of these products, whether by PTF, new release or modification levels, or replacement products, may have affected the level of compatibility outlined here.

COBOL/VSE Release 1 allows you to compile a COBOL/VSE program under VSE/ESA, and run the object program under VSE/ESA, OS/390®, MVS, or VM. Similarly, the object program produced when a COBOL program is compiled by COBOL/370 under OS/390, MVS, or VM can be executed under OS/390, MVS, VM, or VSE/ESA.

When you migrate a COBOL object program from one operating system environment to another, you must:

- Link-edit the object program using the target system linkage editor
- Run the program with the appropriate Language Environment installed on the target system

The appropriate Language Environment products are:

- For VSE/ESA, IBM Language Environment for VSE/ESA (LE/VSE)
- For OS/390, MVS, and VM, IBM SAA AD/Cycle® Language Environment/370

This appendix contains information that will help you prepare your program for cross-system execution.

Compiling under VSE and Running under OS/390, MVS, or VM

Programs compiled using COBOL/VSE Release 1 can be executed in the following systems and subsystems with Language Environment/370 Version 1 Release 2 and later releases. The operating systems are:

OS/390
MVS/ESA™
VM/ESA®

Within this appendix, the generic term MVS is used to indicate an OS/390 or MVS/ESA system. The generic term VM is used to indicate a VM/ESA system.

COBOL/VSE programs can be executed under the following subsystems:

CICS/VSE
CICS/ESA®
IMS/ESA®

COBOL/370 under OS/390, MVS, and VM provides all the features provided by COBOL/VSE under VSE/ESA.

Compiler Options that Affect Portability

If you intend to run your COBOL/VSE programs under MVS or VM, you should carefully consider the compiler options you specify. The following compiler options will affect the way your program will run under MVS or VM. They will also help you to prepare your object programs for execution under MVS or VM. For a description of how to specify these options, see “Using Compiler Options” on page 216.

DECK

Use the DECK compiler option to produce an object program in a format that is suitable for migration to MVS or VM. The object program produced when the OBJECT compiler option is specified is not suitable for migration from VSE to MVS or VM. For information on how to migrate the object program, see “Migrating Object Programs to MVS or VM” or “Migrating Object Programs to VSE/ESA” on page 413.

OUTDD

When you run a COBOL/VSE program under MVS or VM, the OUTDD compiler option is used to specify the name of the file for run-time DISPLAY output.

If you do not specify the OUTDD compiler option, the default is SYSOUT.

Migrating Object Programs to MVS or VM

“Compiler Options that Affect Portability” describes the compiler options that you should specify if you wish to produce an object program that can be migrated from VSE to MVS or VM. Once you have produced an object program, you need to copy it from your VSE system to the target system. How you do this depends upon the environment in which you work and the communications facilities available.

The procedure described in the following examples, uses a tape file to copy the object program. This procedure allows you to copy an object program from a stand-alone VSE system to a stand-alone MVS or VM system. If your VSE system runs as a guest operating system under VM, or if you have communications facilities between your VSE system and your target system, your site may have different procedures for copying object programs. See your system programmer for more information.

The following examples illustrate general job control procedures for saving and copying your object program.

Saving Your Object Program

The object program produced by the COBOL/VSE compiler can be saved in a VSE Librarian sublibrary. You can do this by specifying the following statements:

```
// JOB    jobname
  ⋮
// DLBL   IJSYSPH,'ijsysph.file-ID',0,SD
// EXTENT SYSPCH,volser,1,0,start,tracks
   ASSGN  SYSPCH,DISK,VOL=volser,SHR
  ⋮
// OPTION DECK
// EXEC   IGYCRCTL,SIZE=IGYCRCTL,PARM='NAME(NOALIAS)'
   CLOSE  SYSPCH,PUNCH
// DLBL   IJSYSIN,'ijsysph.file-ID',0,SD
// EXTENT SYSIPT,volser,1,0,start,tracks
   ASSGN  SYSIPT,DISK,VOL=volser,SHR
// EXEC   LIBR,PARM='ACCESS SUBLIB=lib.sublib'
/*
   CLOSE  SYSIPT,SYSRDR
/&
```

Copying Your Object Program to Tape

The object program saved in the previous example can now be copied to a tape file. In order to remove the control characters from the first character position of the SYSPCH output, you should use a VSE system utility such as DITTO. The following job steps produce a blocked output tape file containing the object program.

```
// JOB    jobname
*
*  STEP 1:  ASSIGN SYSPCH TO A DISK FILE
*           WRITE THE REQUIRED DITTO CT COMMAND TO SYSPCH
*
// DLBL   IJSYSPH,'ijsysph.file-ID',0,SD
// EXTENT SYSPCH,volser,1,0,start,tracks
   ASSGN  SYSPCH,DISK,VOL=volser,SHR
// UPSI   1
// EXEC   DITTO
$$DITTO   SET HEADERPG=NO
$$DITTO   CC
$$DITTO   CT  OUTPUT=SYSnnn,BLKFACTOR=blkfac
/*
$$DITTO   EOJ
*
*  STEP 2:    PUNCH THE OBJECT PROGRAM TO SYSPCH
*            (USE FORMAT=NOHEADER TO REMOVE THE LIBRARIAN
*             CATALOG STATEMENT)
*
// EXEC   LIBR
   ACCESS SUBLIB=lib.sublib
   PUNCH  progname.OBJ FORMAT=NOHEADER EOF=YES
/*
   CLOSE  SYSPCH,PUNCH
```

```
*
*  STEP 3:  ASSGN SYSIPT TO THE SYSPCH DISK FILE
*           COPY THE OBJECT PROGRAM TO TAPE (WITHOUT CC)
*           (THE DITTO COMMAND IS ON SYSIPT DISK FILE)
*
// DLBL   IJSYSIN,'ijsysph.file-ID',0,SD
// EXTENT SYSIPT,volser,1,0,start,tracks
   ASSGN  SYSIPT,DISK,VOL=volser,SHR
// ASSGN  SYSnnn,cuu
// MTC    REW,SYSnnn
// MTC    WTM,SYSnnn
// MTC    REW,SYSnnn
// UPSI   1
// EXEC   DITTO
/*
// MTC    RUN,SYSnnn
   CLOSE  SYSIPT,SYSRDR
/&
```

Copying Your Object Program to VM

Once you have copied the object program to a tape, you can transfer the program to your target system. You can use the following CMS commands to define the input and output files, and to copy the object program to your A disk.

```
FILEDEF  INMOVE  TAPn (RECFM FB LRECL 80 BLKSIZE nnn
FILEDEF  OUTMOVE DISK fn TEXT A1 (RECFM F LRECL 80
MOVEFILE
```

Copying Your Object Program to MVS

Use an MVS system utility such as IEBGENER to copy the object program into a partitioned data set (PDS). This JCL example assumes the PDS was previously allocated.

```
//jobname  JOB
//         EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD UNIT=TAPE,DISP=(OLD),LABEL=(,NL),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=nnn)
//SYSUT2   DD DSN=dataset(progname),DISP=SHR
//SYSIN    DD DUMMY
//*
```

Once you have copied the object program to a tape, you can transfer the program to your target system.

Compiling under MVS or VM and Running under VSE

COBOL/370 programs that are compatible with Language Environment/370 Release 2 are also compatible with IBM Language Environment for VSE/ESA.

COBOL/370 programs, compiled under MVS or VM, can be executed under VSE/ESA and under the CICS/VSE subsystem.

COBOL/370 under VSE/ESA offers all the features of COBOL/370 under MVS or VM.

Compiler Options that Affect Portability

If you intend to run your COBOL/370 programs under VSE/ESA, you should carefully consider the compiler options you specify. The following compiler options will help you to prepare your object programs for migration to VSE/ESA. For a description of how to specify these options, see “Using Compiler Options” on page 216.

OUTDD

When you run a COBOL/370 program under VSE, the OUTDD compiler option is used to specify the name of the file for run-time DISPLAY output.

If you do not specify the OUTDD compiler option, the default is SYSOUT.

NONAME

The link-edit NAME statement generated by the NAME compiler option is system specific, and is not suitable for migration. Specify the default option NONAME.

Migrating Object Programs to VSE/ESA

Once you have produced an object program, you need to copy it from your MVS or VM system to the target system. How you do this depends upon the environment in which you work and the communications facilities available.

The procedure described in the following examples, uses a tape file to copy the object program. This procedure allows you to copy an object program from a stand-alone MVS or VM system to a stand-alone VSE/ESA system. If your VSE system runs as a guest operating operating system under VM, or if you have communications facilities between your MVS or VM system and your target system, your site may have different procedures for copying object programs. See your system programmer for more information.

Copying Your Object Program from VM

You can use the following CMS commands to copy the object program from your A disk to tape.

```
FILEDEF  INMOVE  DISK fn TEXT A1 (LRECL 80 BLKSIZE 80 RECFM FB
FILEDEF  OUTMOVE TAPn (LRECL 80 RECFM FB BLKSZIE nnn
MOVEFILE
```

Once you have copied the object program to tape, you can transfer the object program to the target VSE/ESA system.

Copying Your Object Program from MVS

Use an MVS system utility such as IEBGENER to copy the object program from a partitioned data set (PDS) to tape.

```
//jobname  JOB
//         EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD DSN=dataset(progname),DISP=SHR
//SYSUT2   DD UNIT=TAPE,DISP=NEW,LABEL=(,NL),
//            DCB=(RECFM=F,LRECL=80)
//SYSIN    DD DUMMY
//*
```

Once you have copied the object program to a tape, you can transfer the program to your target VSE/ESA system.

Copying Your Object Program from Tape

The object program saved in the previous example can now be copied from the tape file.

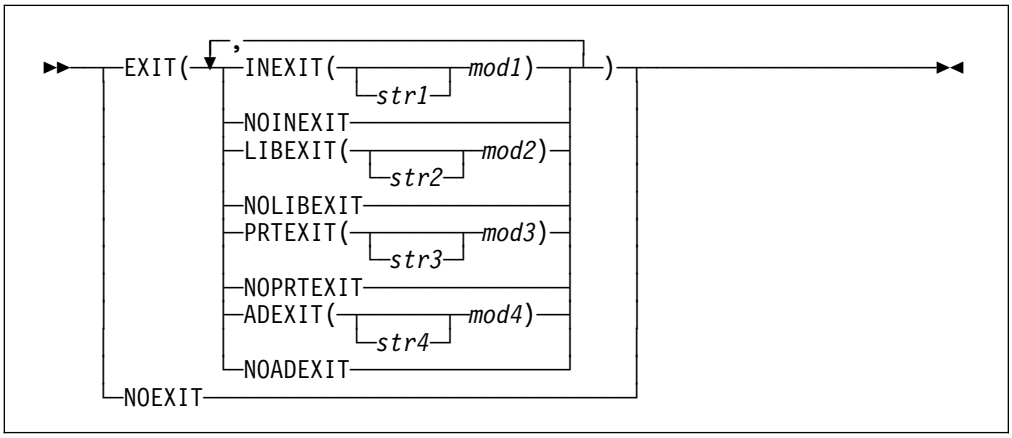
```
// JOB jobname
*
* ASSIGN SYSIPT TO A TAPE FILE
* CATALOG THE OBJECT PROGRAM IN A VSE SUBLIBRARY
*
// ASSGN SYSIPT,TAPE
// EXEC  LIBR,PARM='ACCESS S=lib.sublib;CATALOG name.OBJ REPLACE=YES'
/*
/&
```

Appendix D. EXIT Compiler Option

This appendix gives the syntax of the EXIT compiler option, and shows how you can write compiler exit routines. These routines can be used to replace the compiler's standard input and output routines for:

- Reading primary input (COBOL source, normally from SYSIPT)
- Reading secondary input (COPY or BASIS processing, normally read from a VSE Librarian sublibrary)
- Writing printed output (normally to SYSLST)
- Writing the SYSADATA file (normally to the filename specified in the SYSADAT DLBL statement)

Syntax and Parameters



Default is: NOEXIT

Abbreviations are: EX(INX,LIBX,PRTX,ADX)

Negative abbreviations are: EX(NOINX,NOLIBX,NOPRTX,NOADX)

For the EXIT option, the suboptions INEXIT, LIBEXIT, PRTEXIT, and ADEXIT also have a negative form with associated abbreviations. They are:

NOINEXIT (NOINX)
NOLIBEXIT (NOLIBX)
NOPRTEXIT (NOPRTX)
NOADEXIT (NOADX)

Use the EXIT option to allow the compiler to accept user-supplied phases in place of SYSIPT, copy sublibraries, and SYSLST, and a user-supplied phase to inspect ADATA records. Any combination of exits may be specified.

Note: The EXIT option **cannot** be specified in a PROCESS (CBL) statement. It can only be specified at invocation in the PARM field of JCL, or at installation time.

INEXIT(['str1',]mod1)
The compiler obtains source code from a user-supplied phase (where “mod1” is the phase name), instead of SYSIPT.

LIBEXIT(['str2',]mod2)
The compiler obtains copy code from a user-supplied phase (where “mod2” is the phase name), instead of a VSE Librarian sublibrary. For use with either COPY or BASIS statements.

PRTEXIT(['str3',]mod3)
The compiler passes printer destined output to the user-supplied phase (where “mod3” is the phase name), instead of SYSLST.

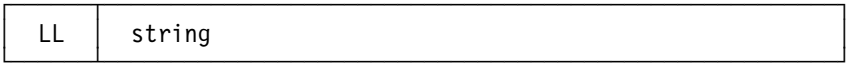
ADEXIT(['str4',]mod4)
Specifies that a user-supplied associated-data (SYSADAT) exit is to be used for the compilation, (where *mod4* is the phase name).

The phase names **mod1, mod2, mod3, and mod4**, can refer to the same phase.

The suboptions **'str1', 'str2', 'str3', and 'str4'**, are optional. They are character strings up to 64 characters in length and enclosed in apostrophes, that are passed to the exit phase. Any character is allowed, but included apostrophes must be doubled, and lowercase characters are folded to uppercase.

Character String Formats

The format of the string, as it appears to the user exit phase, is:



where **LL** is a halfword (on a halfword boundary) containing the length of the string. If the string suboption is not specified, LL contains zero.

User-Exit Work Area

When an exit is used, the compiler provides a **user-exit work area** that can be used to save the address of GETVIS storage obtained by the exit phase. This allows the phase to be reentrant.

The **user-exit work area** is four fullwords, residing on a fullword boundary, that is initialized to binary zeros before the first exit routine is invoked. The address of the work area is passed to the exit phase in a parameter list. After initialization, the compiler makes no further reference to the work area, so you will need to establish your own conventions for the use of the work area if more than one exit is active during the compilation. For example, the INEXIT phase uses the first word in the work area, the LIBEXIT phase uses the second word, the PRTEXIT phase uses the third word, and the ADEXIT phase uses the fourth word.

Linkage Conventions

Your EXIT phases should use COBOL/VSE standard linkage conventions between COBOL programs, between library routines, and between COBOL programs and library routines. You need to be aware of these conventions in order to trace the call chain correctly.

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.

Using INEXIT

When INEXIT is specified, the compiler loads the exit phase (`mod1`) during initialization, and invokes the phase using the OPEN operation code (op code). This allows the phase to prepare its source for processing and then pass the status of the OPEN request back to the compiler. Subsequently, each time the compiler requires a source statement, the exit phase is invoked with the GET op code. The exit phase then returns either the address and length of the next statement or the end-of-data indication (if no more source statements exist). When end-of-data is presented, the compiler invokes the exit phase with the CLOSE op code so that the phase can release any resources that are related to its input.

The compiler uses a parameter list to communicate with the exit phase. The parameter list consists of 10 fullwords containing addresses, and Register 1 contains the address of the parameter list. Notice that `Return Code`, `Data Length`, and `Data`, are placed by the exit phase for return to the compiler; and the other items are passed from the compiler to the exit phase. Figure 133 shows the contents of the parameter list and a description of each item.

Figure 133 (Page 1 of 2). Parameter List for INEXIT

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 phase, indicating 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 user exit phase.
16	Data length	Fullword, placed by the exit phase, specifying the length of the record being returned by the GET operation (must be 80).
20	Data or 'str1'	Fullword, placed by the exit phase, 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	
28	Not used	

Figure 133 (Page 2 of 2). Parameter List for INEXIT

Offset	Contains Address of	Description of Item
32	Not used	
36	Not used	

Using LIBEXIT

When LIBEXIT is specified, the compiler loads the exit phase (`mod2`) during initialization. The exit phase is used in place of the VSE Librarian sublibraries. Calls are made to the phase by the compiler to obtain copy text whenever COPY or BASIS statements are encountered.

Note: If LIBEXIT is specified, the LIB compiler option must be in effect.

The initial call invokes the phase with an OPEN op code. This allows the phase to prepare the specified library-name for processing. The OPEN op code is also issued the first time a new library-name is specified. The exit phase returns the status of the OPEN request to the compiler by passing a return code.

Once a library-name has successfully opened, the exit phase is then invoked with a FIND op code. The exit phase establishes positioning at the requested text-name (or basis-name) in the specified library-name. This becomes the “active copy source.” When positioning is complete, the exit phase passes an appropriate return code to the compiler.

The compiler then invokes the exit phase with a GET op code, and the exit phase passes the compiler the length and address of the record to be copied from the “active copy source.” The GET operation is repeated until the end-of-data indicator is passed to the compiler.

When end-of-data is presented, the compiler will issue a CLOSE op code request so that the exit phase can release any resources related to its input.

Nested COPY Statements

Any record from the “active copy source” can contain a COPY statement. When this occurs, the compiler issues a request based on the following:

- If the requested library-name from the nested COPY statement was not previously opened, the compiler invokes the exit phase with an OPEN op code, followed by a FIND op code for the new text-name.
- If the requested library-name is already open, then the compiler issues the FIND op code for the newly requested text-name (an OPEN is **not** issued in this case).

Note that the compiler will not allow recursive calls to text-name.

When the exit phase receives the OPEN or FIND request, it should “push” its control information concerning the “active copy source” onto a stack and then complete the requested action (OPEN or FIND). The newly requested text-name (or basis-name) now becomes the “active copy source.”

Processing continues in the normal manner with a series of GET requests until the end-of-data indicator is passed to the compiler.

At end-of-data for the nested “active copy source,” the exit phase should “pop” its control information from the stack. The next request from the compiler will be a FIND, so that the exit phase can reestablish positioning at the previous “active copy source.”

The compiler now invokes the exit phase with a GET request, and the exit phase must pass the **same record** that was passed previously from this copy source. The compiler verifies that the same record was passed, and then the processing continues with GET requests until the end-of-data indicator is passed.

Figure 134 shows the contents of the parameter list used for LIBEXIT, and a description of each item.

Figure 134 (Page 1 of 2). Parameter List for LIBEXIT

Offset	Contains Address of	Description of Item
00	User-exit type	Halfword identifying which user-exit is to perform the operation. 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 phase, indicating 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 user exit phase.
16	Data length	Fullword, placed by the exit phase, specifying the length of the record being returned by the GET operation (must be 80).
20	Data or 'str2'	Fullword, placed by the exit phase, 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.
24	System library-name	8-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	8-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.

Figure 134 (Page 2 of 2). Parameter List for LIBEXIT

Offset	Contains Address of	Description of Item
32	Library-name	30-character area containing the full library-name from the COPY statement. Padded with blanks if required, and used as-is (not folded to upper-case). Applies to OPEN, CLOSE, and FIND.
36	Text-name	30-character area containing the full text-name from the COPY statement. Padded with blanks if required, and used as-is (not folded to upper-case). Applies only to FIND.

Using PRTEXIT

When PRTEXIT is specified, the compiler loads the exit phase (`mod3`) during initialization. The exit phase is used in place of the SYSLST file.

The compiler invokes the phase using the OPEN operation code (op code). This allows the phase to prepare its output destination for processing and then pass the status of the OPEN request back to the compiler. Subsequently, each time the compiler has to print a line, the exit phase is invoked with the PUT op code. The compiler supplies the address and length of the record that is to be printed, and the exit phase returns the status of the PUT request to the compiler by a return code. The first byte of the record to be printed contains an ANSI printer control character.

Before the compilation is ended, the compiler invokes the exit phase with the CLOSE op code so that the phase can release any resources that are related to its output destination.

Figure 135 shows the contents of the parameter list used for PRTEXIT, and a description of each item.

Figure 135 (Page 1 of 2). Parameter List for PRTEXIT

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 phase, indicating 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 user exit phase.
16	Data length	Fullword, specifying the length of the record being supplied by the PUT operation (the compiler sets this value to 133).

Figure 135 (Page 2 of 2). Parameter List for PRTEXIT

Offset	Contains Address of	Description of Item
20	Data buffer or	Fullword, containing the address of the data buffer where the compiler has placed the record to be printed by the PUT operation.
	'str3'	'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	
28	Not used	
32	Not used	
36	Not used	

Using ADEXIT

When ADEXIT is specified, the compiler loads the exit phase (`mod4`) during initialization.

The compiler invokes the phase using the OPEN operation code (op code). This allows the phase to prepare for processing and then pass the status of the OPEN request back to the compiler. Subsequently, each time the compiler has written a record to the SYSADAT file, the exit phase is invoked with the PUT op code. The compiler supplies the address and length of the record that has been written to the SYSADAT file, and the exit phase returns the status of the PUT request to the compiler by a return code.

The ADATA exit is a 'read-only' exit. That is, the exit may only inspect the data record passed to it. It may not change the contents of the record written to the SYSADAT file.

The exit may be written to select data from the ADATA records presented to the exit, and to write that data to a file that is maintained by the exit.

Before the compilation is ended, the compiler invokes the exit phase with the CLOSE op code so that the phase can release any resources that it may hold.

Figure 136 shows the contents of the parameter list used for ADEXIT, and a description of each item.

Figure 136 (Page 1 of 2). Parameter List for ADEXIT

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

Figure 136 (Page 2 of 2). Parameter List for ADEXIT

Offset	Contains Address of	Description of Item
08	Return code	Fullword, placed by the exit phase, indicating 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 user exit phase.
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 that may be inspected through 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	
28	Not used	
32	Not used	
36	Not used	

Error Handling

The compiler will report error messages whenever an exit phase cannot be loaded, or if an exit phase returns an “operation failed” or inappropriate return code.

Message IGYSI5008 is written to the operator and the compiler terminates with a return code = 16, when any of the following occur:

- An exit phase 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
- A nonzero return code is received from ADEXIT during an OPEN request

The exit type and operation (OPEN or LOAD) is identified in the message.

Any other error from INEXIT, PRTEXIT, or ADEXIT will cause the compiler to terminate.

The following conditions are detected and reported by the compiler:

5203 PUT request to SYSLST user exit failed with return code "nn".

5204 Record address not set by "exit-name" user-exit.

5205 GET request from SYSIPT user-exit failed with return code "nn".

5206 Record length not set by "exit-name" user-exit.

5224 A PUT request to the ADEXIT user-exit failed with return code "nn".

An Example SYSIPT User-Exit

Figure 137 shows an example of a SYSIPT user-exit phase written using COBOL/VSE.

```
      ************************************************************
      *                                                          *
      * Name:  SKELINX                                           *
      *                                                          *
      * Function:  Example of a SYSIPT user-exit written         *
      *            in the COBOL language.                        *
      *                                                          *
      *----------------------------------------------------------*
      *                                                          *
      * LINKAGE NOTE:  Link with run-time options phase,         *
      *                CEEUOPT CSECT, that contains at least     *
      *                the following:                            *
      *                                                          *
      *                CEEUOPT  CSECT                            *
      *                CEEUOPT  AMODE ANY                        *
      *                CEEUOPT  RMODE ANY                        *
      *                CEEXOPT  RTEREUS=(ON)                     *
      *                         END                              *
      *                                                          *
      ************************************************************

       Identification Division.
          Program-ID.  Skelinx.

       Environment Division.

       Data Division.

         Working-Storage Section.

      *   ********************************************************
      *   *                                                      *
      *   *  Local variables.                                    *
      *   *                                                      *
      *   ********************************************************

          77 Operation         Pic 9(4)   Comp.

      *   ********************************************************
      *   *                                                      *
      *   *  Definition of the User-Exit Parameter List, which  *
      *   *  is passed from the COBOL compiler to the user-exit *
      *   *  passed from the COBOL compiler to the user-exit    *
      *   *  phase.                                              *
      *   *                                                      *
      *   ********************************************************

         Linkage Section.
          01 Exit-Type        Pic 9(4)   Comp.
          01 Exit-Operation   Pic 9(4)   Comp.
          01 Exit-ReturnCode  Pic 9(5)   Comp.
```

Figure 137 (Part 1 of 4). Example SYSIPT User-Exit

```
          01 Exit-WorkArea.
             05 Sysipt-Slot   Pic 9(5)   Comp.
             05 Libexit-slot  Pic 9(5)   Comp.
             05 Syslst-slot   Pic 9(5)   Comp.
             05 Reserved-Slot Pic 9(5)   Comp.
          01 Exit-DataLength  Pic 9(5)   Comp.
          01 Exit-DataArea    Pic 9(9)   Comp.
          01 Exit-Open-Parm   Redefines Exit-DataArea.
             05 String-Len    Pic 9(4)   Comp.
             05 Open-String   Pic X(64).
          01 Exit-Print-Line  Redefines Exit-DataArea  Pic X(133).
          01 Exit-Library     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-Library    Exit-Systext
                                Exit-CBLLibrary Exit-CBLText.

           Add 1 To Exit-Operation Giving Operation
           Go To Handle-Sysipt
                 Handle-Library
                 Handle-Syslst
                 Handle-Adata
              Depending On Exit-Type.
           Move 16 To Exit-ReturnCode
           Goback.
```

Figure 137 (Part 2 of 4). Example SYSIPT User-Exit

```
************************************************
*    S Y S I N   E X I T   P R O C E S S O R   *
************************************************
 Handle-Sysipt.

     Go To Sysipt-Open
           Sysipt-Close
           Sysipt-Get
        Depending On Operation.
     Move 16 To Exit-ReturnCode
     Goback.

 Sysipt-Open.
*    ------------------------------------------------------
*    Prepare for reading source
*    ------------------------------------------------------
     Goback.

 Sysipt-Close.
*    ------------------------------------------------------
*    Release resources
*    ------------------------------------------------------
     Goback.

 Sysipt-Get.
*    ------------------------------------------------------
*    Retrieve next source record
*    ------------------------------------------------------

*    ------------------------------------------------------
*    The following can be used to return the address of the
*    record to the compiler.
*    ------------------------------------------------------
     Call "GETADDRESS" Using
                       By Reference Record-Variable
                       By Reference Exit-DataArea

*    ------------------------------------------------------
*    Set length of record in User-Exit Parameter List
*    ------------------------------------------------------
     Move 80 To Exit-DataLength

     Goback.

****************************************************
*    L I B R A R Y   E X I T   P R O C E S S O R   *
****************************************************
 Handle-Library.
     Display "**** This phase for SYSIPT only"
     Move 16 To Exit-ReturnCode
     Goback.
```

Figure 137 (Part 3 of 4). Example SYSIPT User-Exit

```
*******************************************************
*    S Y S P R I N T   E X I T   P R O C E S S O R    *
*******************************************************
 Handle-Syslst.
     Display "**** This phase for SYSIPT only"
     Move 16 To Exit-ReturnCode
     Goback.
*******************************************************
*    A D A T A         E X I T   P R O C E S S O R    *
*******************************************************
 Handle-Adata.
     Display "**** This phase for SYSIPT only"
     Move 16 To Exit-ReturnCode
     Goback.
*******************************************************
*******************************************************
**                                                   **
**   Internal program to obtain the address of an    **
**   item in the caller's WORKING-STORAGE SECTION    **
**                                                   **
*******************************************************
*******************************************************
 Identification Division.
    Program-ID.  GetAddress.
 Environment Division.
 Data Division.
   Linkage Section.
    01 The-Item         Pic X(80).
    01 Its-Address      Pointer.
 Procedure Division Using The-Item Its-Address.
     Set Its-Address To Address Of The-Item.
     Goback.
 End Program GetAddress.

 End Program Skelinx.
```

Figure 137 (Part 4 of 4). Example SYSIPT User-Exit

Appendix E. Sample Programs

This appendix contains information about the sample programs which are included on your product tape:

- Overview of the program
- Hierarchy chart or program chart
- Format and sample of the input data
- Sample of reports produced
- Information on how to run the program
- List of the language elements and concepts that are illustrated

Pseudocode and other comments regarding these programs are included in the program prologue which you can obtain in a program listing.

The sample programs in this section demonstrate many language elements and concepts of COBOL/VSE:

- **IGYTCARA** -provides an example of using SAM files and VSAM indexed files and illustrates using many COBOL intrinsic functions.
- **IGYTSALE** -illustrates many of the LE/VSE callable services features.

Overview of the IGYTCARA

A company with several local offices wants to establish employee carpools. This batch application needs to perform two tasks:

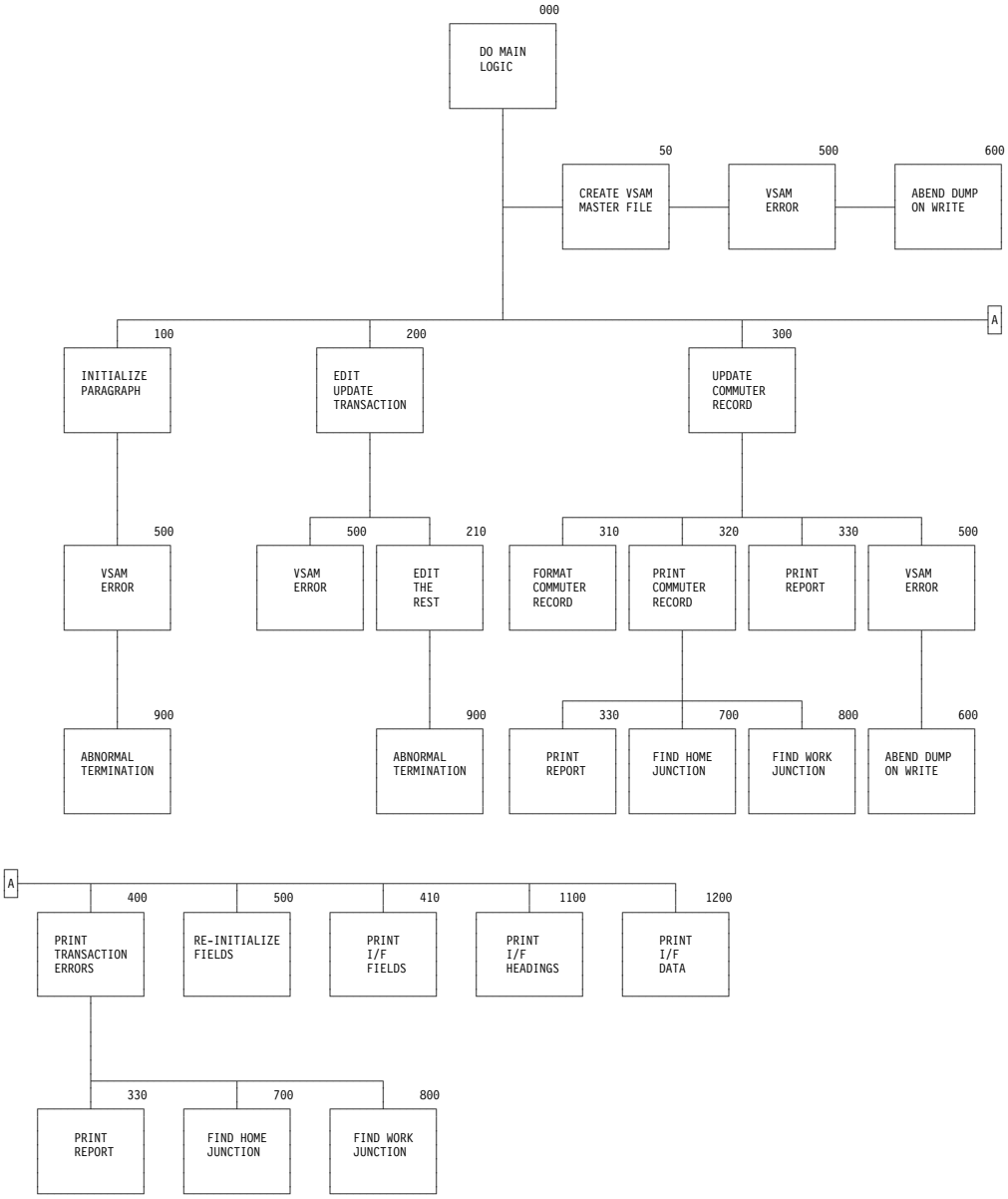
1. Produce reports of employees who can share rides from the same home location to the same work location.
2. Update the carpool data to:
   - Add data for new employees
   - Change information for participating employees
   - Delete employee records
   - List invalid update requests

Data Validation and Update

Using SAM files and VSAM indexed files, this program:

- Validates transaction file entries (sequential file processing)
- Updates a master file (indexed file processing)

Hierarchy Chart for IGYTCARA



Input Data for IGYTCARA

As input to our program, the company collected information from interested employees, coded the information, and produced an input file. (Spaces between fields are left out, just as they would be in your input file.)

Figure 138. Format of Input File for IGYTCARA

Where:

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 junction
10. Work location junction
11. Driving status code

Figure 139 shows a sample section of the input file.

```
A10111ROBERTS  AB1021 CRYSTAL COURTSAN FRANCISCOCA9990141555501904155551387H1W1D

A20212KAHN     DE789 EMILY LANE    SAN FRANCISCOCA9992141555518904155552589H2W2D

P48899                                        99ASDFG0005557890123ASDFGHJ    T

R10111ROBERTS  AB1221 CRYSTAL COURTSAN FRANCISCOCA9990141555501904155551387H1W1D

A20212KAHN     DE789 EMILY LANE    SAN FRANCISCOCA9992141555518904155552589H2W2D

D20212KAHN     DE

D20212KAHN     DE

A20212KAHN     DE789 EMILY LANE    SAN FRANCISCOCA9992141555518904155552589H2W2D

A10111BONNICK  FD1025 FIFTH AVENUE SAN FRANCISCOCA9990541555595904155557895H8W3

A10111PETERSON SW435 THIRD AVENUE  SAN FRANCISCOCA9990541555546904155553717H3W4
```

Figure 139. Example of IGYTCARA Input Data

Report Produced by IGYTCARA

Figure 140 shows a sample of what the output report produced by IGYTCARA might look like. Your actual output may vary slightly in appearance, depending on your system.

```
REPORT  #: IGYTCARA                              COMMUTER FILE UPDATE LIST                                  PAGE #:    1
PROGRAM #: IGYTCARA       RUN TIME: 18:57                                                  RUN DATE: 04/21/95
=============================================================================================================================
     | 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
     .
     .
     .
```

Figure 140. Example of IGYTCARA Output Report

Running IGYTCARA

The job control statements described later in this section will perform a combined compile, link-edit, and run of the IGYTCARA program. If you wish to only compile, or compile and link-edit the program, then you will need to modify these job control statements.

All the files required by the IGYTCARA program, and the source programs, are supplied on the product installation tape. The files (IGYTCODE and IGYTRANX), and the source program (IGYTCARA) are members in the sublibrary into which COBOL/VSE was installed.

Note: You should check with your system programmer to get copies of these members.

Compiler Options

These options must be in effect, either by installation default, or in addition to the CBL statement in the source file for IGYTCARA:

NOADV
NOCMPR2
NODYNAM
NONAME
NONUMBER
QUOTE
SEQUENCE

With these options in effect, the program will not cause any diagnostic messages to be issued. The sequence number string provided in the source file will be useful for searching for the language elements that are listed in Figure 148 on page 444.

Running the Job

To run IGYTCARA, the following functions need to be performed using JCL:

- Define a VSAM cluster for use by IGYTCARA
- Load data into temporary transaction files
- Compile IGYTCARA
- Link-edit the object phases produced by the compilation
- Execute IGYTCARA

You will need to insert your own system/installation specific information in the fields that are shown in lowercase letters (volume serial number, catalog name, cluster prefix). You will also need to provide a default model for SAM ESDS files in your VSAM catalog. See your system programmer for assistance. We have used the name IGYTCAR.MASTFILE in these examples, you can use another name if you wish.

1. Use the following JCL to create the required VSAM cluster:

```
// JOB CREATE
// EXEC IDCAMS,SIZE=AUTO
   DELETE your-prefix.IGYTCAR.MASTFILE        -
          PURGE                               -
          CATALOG(your-cat-name)
   DEFINE CLUSTER                             -
         (NAME(your-prefix.IGYTCAR.MASTFILE)  -
          VOLUME(your-volume-serial)          -
          INDEXED                             -
          RECSZ(80 80)                        -
          KEYS(16 0)                          -
          RECORDS(100 20))                    -
          CATALOG(your-cat-name)
/*
/&
```

Note: A delete is issued before the VSAM cluster is created. This eliminates any existing cluster.

2. Use the following JCL to load data into the temporary transaction files.

```
// JOB LOAD
// DLBL INPUT2,'your-prefix.IGYTRANX',0,VSAM,CAT=your-cat-name,        C
               RECSIZE=80,RECORDS=50,DISP=(NEW,KEEP)
// DLBL LOCCODE,'your-prefix.IGYTCODE',0,VSAM,CAT=your-cat-name,       C
               RECSIZE=80,RECORDS=50,DISP=(NEW,KEEP)
// UPSI 1
// EXEC DITTO
$$DITTO SET HEADERPG=NO
$$DITTO CSQ FILEOUT=INPUT2,CISIZE=4000
   ⋮
   IGYTRANX.Z member in the COBOL/VSE sublibrary
   ⋮
/*
$$DITTO CSQ FILEOUT=LOCCODE,CISIZE=4000
   ⋮
   IGYTCODE.Z member in the COBOL/VSE sublibrary
   ⋮
/*
$$DITTO EOJ
/&
```

3. Use the following JCL to compile IGYTCARA, link-edit the object phases, and execute IGYTCARA. Your installation may have standard labels for the compiler work files, and SYSLNK. See your system programmer.

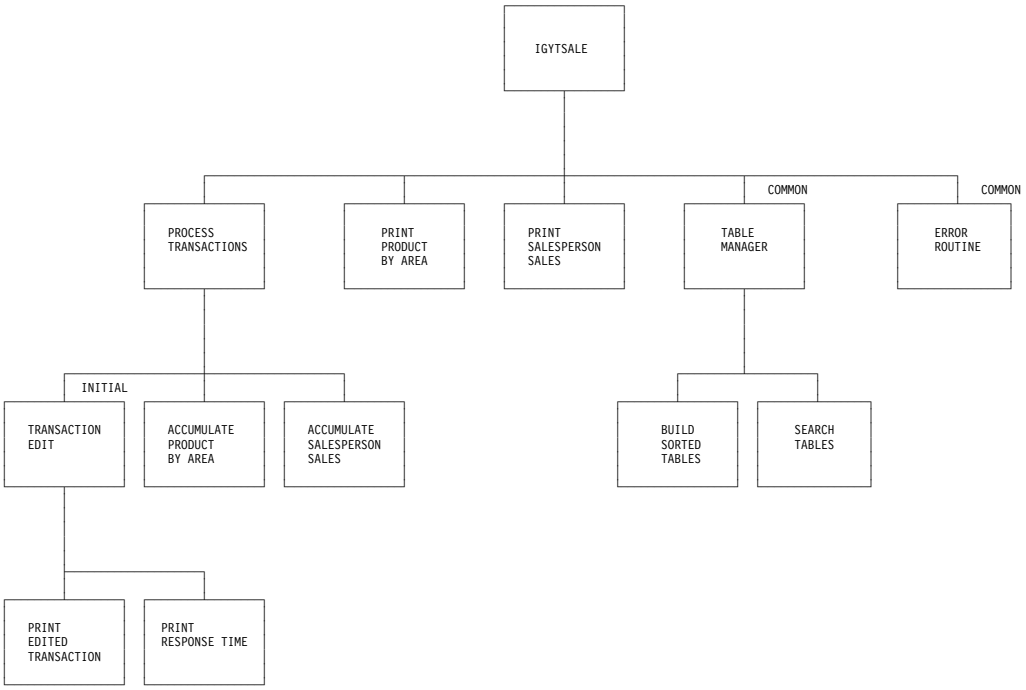
```
// JOB IGYTCARA
// OPTION NODUMP,LINK
// LIBDEF *,SEARCH=(PRD2.PROD,PRD2.SCEEBASE) <- Compiler and Run Time
// DLBL IJSYS01,'your-prefix.IJSYS01',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYS02,'your-prefix.IJSYS02',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYS03,'your-prefix.IJSYS03',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYS04,'your-prefix.IJSYS04',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYS06,'your-prefix.IJSYS06',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYS07,'your-prefix.IJSYS07',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYSLN,'your-prefix.IJSYSLN',0,VSAM,RECSIZE=322,              C
               RECORDS=(400,600),CAT=your-cat-name
// EXEC IGYCRCTL,SIZE=IGYCRCTL
   ⋮
   IGYTCARA.C member in the distribution library
   ⋮
/*
// EXEC LNKEDT
// ASSGN SYS012,SYSIPT
// ASSGN SYS014,SYSLST
// DLBL LOCCODE,'your-prefix.IGYTCODE',0,VSAM,CAT=your-cat-name,       C
               DISP=(OLD,DELETE)
// DLBL UPDTRAN,'your-prefix.IGYTRANX',0,VSAM,CAT=your-cat-name,       C
               DISP=(OLD,DELETE)
// DLBL COMMUTR,'your-prefix.IGYTCAR.MASTFILE',0,VSAM,CAT=your-cat-name
// EXEC ,SIZE=256K
/*
/&
```

Overview of IGYTSALE

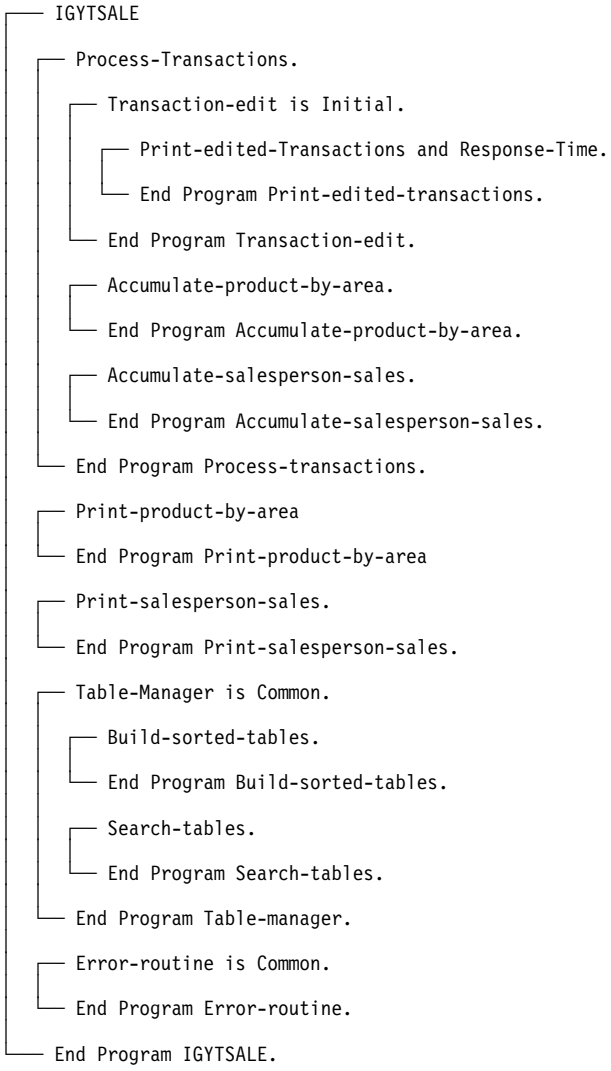
A sporting goods distributor wants to track product sales and sales commissions. This nested program application needs to perform the following tasks:

1. Keep a record of the product line, customers, and number of salespeople. This data is stored in a file called IGYTABLE.
2. Maintain a file which records valid transactions and transaction errors. All invalid transactions are flagged and the results are printed in a report. Transactions to be processed are in a file called IGYTRANA.
3. Process transactions and report sales by location.
4. Record an individual's sales performance and commission and print the results in a report.

Program Chart for IGYTSALE



Nested Program Map for IGYTSALE



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.

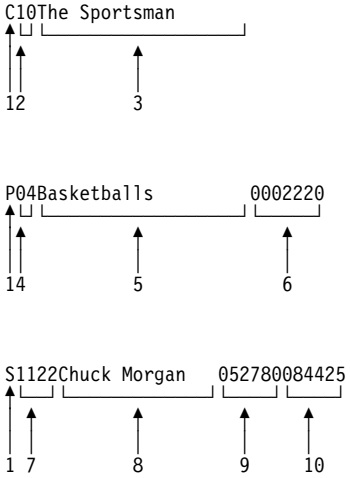


Figure 141. Format of Input File IGYTABLE

Where:

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. Figure 142 on page 437 shows a sample section of IGYTABLE.

```
S1111Edyth Phillips 062484042327
S1122Chuck Morgan   052780084425
S1133Art Tung       022882061728
S1144Billy Jim Bob  010272121150
S1155Chris Preston  122083053377
S1166Willie Al 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
   ⋮
```

Figure 142. Sample Input from IGYTABLE

In addition, the distributor collected information about sales transactions. Each transaction represents an individual salesperson's sales to a particular customer. The customer may purchase from 1 to 8 items during each transaction. The transaction information is coded and put in an input file, called IGYTRANA.

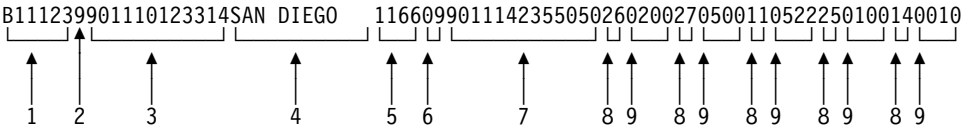


Figure 143. Format of Input File IGYTRANA

Where:

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 1 to 8 times depending on the number of different items ordered (field 2). Figure 144 on page 438 shows a sample section of IGYTRANA.

```
A000011900227010101CNTRL VALLEY1144209002282230152599999
A000041900310100530CNTRL VALLEY114410900403052110150099
A000051900418222409CNTRL VALLEY1144129004192351180599000
A000061900523151010CNTRL VALLEY114420900623010915250004
      4990324591515SAN DIEGO    11615           260200132200110522045100
B111144901111003301SAN DIEGO   116615901114021510260200132200110522041100
A000071901115003205CNTRL VALLEY113320901117221445120023
C0012549001181015275F BAY AREA 113315900120050312160200112200250522145111
B111164901201132013SF BAY AREA 113315901203132215060200102200110522045102
B111173901201070833SAN Diego   116566901203180955330200132200120522041100
B111184901221191544SAN DIEGO   116614901223235501160200142200130522040300
B111194901210211544SAN DIEGO   112212901214015502060200152200160522050500
B111204901212000816SAN DIEGO   112204901213153310150200052200160522040100
B111214901201131544SAN DIEGO   113302901203032507120200112200140522250100
B111224901112073312SAN DIEGO   112210901113083312100200162200260522250100
B111239901110123314SAN DIEGO   116609901114235505260200270500110522250100140010
B111242901313510000SAN DIEGO   116611            1 0200042200120a22141100
B111254901215012510SAN DIEGO   116615901216022510110200162200130522141111
B111261901111000034SAN DIEGO   113316901113003030260022
B111271901110154100SAN DIEGO   112212901113170000122000
   ⋮
```

Figure 144. Sample Input from IGYJTRANA

Reports Produced by IGYTSALE

The following figures are samples of IGYTSALE output. The program records transaction errors (Figure 145), sales by product and area (Figure 146 on page 440), and individual sales performance and commissions (Figure 147 on page 441). Your actual output may vary slightly in appearance, depending on your system.

```
Day of Report: FRIDAY             C O B O L   S P O R T S        04/21/95      18:40    Page:   1
                                              Invalid Edited Transactions
Sales  Inv.  Sales         Sales        Sales  Cust. Product And Quantity Sold       Ship
Order  Items Time Stamp    Area         Pers   Code                                  Time Stamp
-----  ----- ------------  -----------  -----  ----- --------------------------      ------------
           4 990324591515  SAN DIEGO      116     15 60200132200110522045100                  2 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 901201070833  SAN Diego     1165     66 33020o132200120522041100        901203180955 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 901110123314  SAN DIEGO     1166     09 260200270500110522250100140010  901114235505 Error Descriptions
                                                                                              -Invoiced items is invalid
                                                                                              -Product and quantity not checked
                                                                                              -Date of ship time stamp is invalid
B11124     2 901313510000  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 90111944                    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 901117091530  LOS ANGELES   1155        1132001020102603212500004       901119091730 Error Descriptions
                                                                                              -Customer code is invalid
D00009     9 901201222222  CNTRL COAST   115      19 141  1131221                    901202002424 Error Descriptions
                                                                                              -Invoiced items is invalid
                                                                                              -Salesperson number not numeric
                                                                                              -Product and quantity not checked
There were 00041 records processed in this program
```

Figure 145. Example of IGYTSALE Invalid Edited Transaction Report

Day of Report: FRIDAY C O B O L S P O R T S 04/21/95 18:40 Page: 1

Sales Analysis By Product By Area

Product Codes	CNTRL COAST	CNTRL VALLEY	LOS ANGELES	NORTH COAST	SAN DIEGO	SF BAY AREA	Product Totals
			Areas of Sale				
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	
.							
.							
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 *

Figure 146. Example of IGYTSALE Sales Analysis By Product By Area Report

```
Day of Report: FRIDAY              C O B O L   S P O R T S        04/21/95      18:40    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
    .
    .
    .
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
```

Figure 147. Example of IGYTSALE Sales and Commission Report

Running IGYTSALE

The job control statements described later in this section will perform a combined compile, link-edit, and execute of the IGYTSALE program. If you wish only to compile, or compile and link-edit the program, then you will need to modify these job control statements.

All the files required by the IGYTSALE program, the source programs, and the copy members, are supplied on the product installation tape. The files (IGYTABLE and IGYTRANA), the source program (IGYTSALE) and the copy members (IGYTCRC, IGYTPRC, and IGYTSRC) are members in the sublibrary into which COBOL/VSE was installed.

Note: You should check with your system programmer to get copies of these members.

Compiler Options

These options must be in effect, either by installation default, or in addition to the CBL statement in the source file for IGYTSALE:

LIB
NOCMPR2
NOFLAGMIG
NONUMBER
QUOTE
SEQUENCE

With these options in effect, the program will not cause any diagnostic messages to be issued. The sequence number string provided in the source file will be useful for searching for the elements that are listed in Figure 148 on page 444.

Running the Job

To run IGYTSALE , the following functions need to be performed using JCL:

- Load data into temporary transaction files
- Compile, link-edit and run the program IGYTSALE

You will need to insert your own system/installation specific information in the fields that are shown in lowercase letters (catalog name, file-ID prefix, sublibrary name). You will also need to provide a default model for SAM ESDS files in your VSAM catalog. See your system programmer for assistance.

1. Use the following JCL to load data into the temporary transaction files.

```
// JOB LOAD
// DLBL IGYTRAN,'your-prefix.IGYTRANA',0,VSAM,CAT=your-cat-name,       C
               RECSIZE=80,RECORDS=50,DISP=(NEW,KEEP)
// DLBL IGYTABLE,'your-prefix.IGYTABLE',0,VSAM,CAT=your-cat-name,      C
               RECSIZE=80,RECORDS=50,DISP=(NEW,KEEP)
// UPSI 1
// EXEC DITTO
$$DITTO SET HEADERPG=NO
$$DITTO CSQ FILEOUT=IGYTRAN,CISIZE=4000
   ⋮
   IGYTRANA.Z member in the COBOL/VSE sublibrary
   ⋮
/*
$$DITTO CSQ FILEOUT=IGYTABLE,CISIZE=4000
   ⋮
   IGYTABLE.Z member in the COBOL/VSE sublibrary
   ⋮
/*
$$DITTO EOJ
/&
```

2. Use the following JCL to compile, link-edit and run the IGYTSALE program. Your installation may have standard labels for the compiler work files, and SYSLNK. See your system programmer.

```
// JOB IGYTSALE
// LIBDEF *,SEARCH=(PRD2.PROD,PRD2.SCEEBASE)  <- Compiler and Run Time
// DLBL IJSYS01,'your-prefix.IJSYS01',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYS02,'your-prefix.IJSYS02',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYS03,'your-prefix.IJSYS03',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYS04,'your-prefix.IJSYS04',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYS05,'your-prefix.IJSYS05',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYS06,'your-prefix.IJSYS06',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYS07,'your-prefix.IJSYS07',0,VSAM,RECSIZE=4096,             C
               RECORDS=(50,100),DISP=(NEW,DELETE),CAT=your-cat-name
// DLBL IJSYSLN,'your-prefix.IJSYSLN',0,VSAM,RECSIZE=322,              C
               RECORDS=(400,600),CAT=your-cat-name
// DLBL IGYTRAN,'your-prefix.IGYTRANA',0,VSAM,CAT=your-cat-name,       C
               DISP=(OLD,DELETE)
// DLBL IGYTABL,'your-prefix.IGYTABLE',0,VSAM,CAT=your-cat-name,       C
               DISP=(OLD,DELETE)
// ASSGN SYS014,SYSLST
// EXEC IGYCRCTL,SIZE=IGYCRCTL,GO
   ⋮
   IGYTSALE.C member in the COBOL/VSE sublibrary
   ⋮
/*
/&
```

Language Elements and Concepts that Are Illustrated

Figure 148 lists the language elements and programming concepts that are illustrated in the sample programs. The language element or concept is described, and the sequence string (special character string that appears in the sequence field of the source file that can be used as a search argument for locating the elements in the listing) is shown.

To find the applicable language element for a sample program, locate that program's abbreviation in the sequence string in Figure 148.

Sample Program	
IGYTCARA	IA
IGYTSALE	IS

Figure 148 (Page 1 of 4). Sample Program Language Elements and Concepts

Language Element or Concept	Sequence String
ACCEPT ... FROM DAY-OF-WEEK	IS0900
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 (LE/VSE)	
CEEDATM - format date/time output	IS0875, IS2575
CEEDCOD - feedback code check from service call	IS0905
CEEGMTO - UTC offset from local time	IS0904
CEELOCT - local date and time	IS0850
CEESECS - convert date/time stamp to seconds	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

Figure 148 (Page 2 of 4). Sample Program Language Elements and Concepts

Language Element or Concept	Sequence String
"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
EVALUATE ... ALSO	IS2400
EXIT PROGRAM Need Not Be 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
In-line PERFORM statement	IA4410-4520
I-O-CONTROL Paragraphs (optional)	IS0200
INPUT-OUTPUT SECTION (optional)	IS0200

Figure 148 (Page 3 of 4). Sample Program Language Elements and Concepts

Language Element or Concept	Sequence String
Intrinsic Functions	
CURRENT-DATE	IA9005
MAX	IA9235
MEAN	IA9215
MEDIAN	IA9220
MIN	IA9240
STANDARD-DEVIATION	IA9230
UPPER-CASE	IA9015
VARIANCE	IA9225
WHEN-COMPILED	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
OBJECT-COMPUTER (optional)	IS0200
OCCURS DEPENDING ON	IS0710
ODO uses maximum length for receiving item	IS1550
OPEN INPUT	IS1400
OPEN OUTPUT	IS1500
ORGANIZATION (optional)	IS1100
Page eject	IA7180-7210
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
PROGRAM-ID (30 characters allowed)	IS0120
READ .. INTO ... AT END	IS1550
REDEFINES statement	IA1940, IA2060, IA2890, IA3320

Figure 148 (Page 4 of 4). Sample Program Language Elements and Concepts

Language Element or Concept	Sequence String
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, 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
USAGE BINARY	IS1300
USAGE PACKED-DECIMAL	IS1301
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

Bibliography

IBM COBOL for VSE/ESA

General Information, GC26-8068

Migration Guide, GC26-8070

Installation and Customization Guide, SC26-8071

Programming Guide, SC26-8072

Language Reference, SC26-8073

Reference Summary, SX26-3834

Diagnosis Guide, SC26-8528

Licensed Program Specifications, GC26-8069

IBM VisualAge COBOL Millennium Language Extensions for VSE/ESA

Installation and Customization Guide, SC26-8071

COBOL Millennium Language Extensions Guide, GC26-9266

Fact Sheet, GC26-9321

Licensed Program Specifications, GC26-9417

Language Environment Publications

Fact Sheet, GC33-6679

Concepts Guide, GC33-6680

Installation and Customization Guide, SC33-6682

Programming Guide, SC33-6684

Programming Reference, SC33-6685

Debugging Guide and Run-Time Messages, SC33-6681

Licensed Program Specifications, GC33-6683

Run-Time Migration Guide, SC33-6687

Writing Interlanguage Communication Applications, SC33-6686

C Run-Time Library Reference, SC33-6689

C Run-Time Programming Guide, SC33-6688

Related Publications

Debug Tool/VSE

User's Guide and Reference, SC26-8797

Installation and Customization Guide, SC26-8798

CICS/VSE

Application Programming Guide, SC33-0712

Application Programmer's Reference, SC33-0713

DL/I DOS/VS

Application Programming: CALL and RQDLI Interfaces, SH12-5411

Application Programming: High-Level Programming Interface, SH24-5009

DFSORT/VSE

Application Programming Guide, SC26-7040

Messages, Codes and Diagnosis Guide, SC26-7132

SQL/DS

Application Programming Guide for VSE, SH09-8098

VS COBOL II

Application Programming Guide for VSE, SC26-4697

VSE/ESA VSAM

VSE/ESA Commands and Macros, SC33-6532

VSE/ESA Programmer's Reference, SC33-6535

VSE/ESA Version 1 Release 4

Planning, SC33-6503

Administration, SC33-6505

Guide to System Functions, SC33-6511

System Control Statements, SC33-6513

System Macros User's Guide, SC33-6515

System Macros Reference, SC33-6516

System Utilities, SC33-6517

Messages and Codes Vol.1 & 2, SC33-6507

VSE/ESA Version 2

Planning, SC33-6603

Administration, SC33-6605

V2R3 Enhancements, SC33-6629

Guide to System Functions, SC33-6611

System Control Statements, SC33-6613

System Macros User's Guide, SC33-6615

System Macros Reference, SC33-6616

System Utilities, SC33-6617

Messages and Codes Vol. 1, SC33-6698

Messages and Codes Vol. 2, SC33-6699

Softcopy Publications

These collections contain the COBOL/VSE and LE/VSE-conforming language product publications:

VSE Collection, SK2T-0060

Application Development Collection, SK2T-1237

You can order these publications from Mechanicsburg through your local IBM representative.

Glossary

The terms in this glossary are defined in accordance with their meaning in COBOL. These terms may or may not have the same meaning in other languages.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the following publications:

- *American National Standard Programming Language COBOL, ANSI X3.23-1985* (Copyright 1985 American National Standards Institute, Inc.), which was prepared by Technical Committee X3J4, which had the task of revising American National Standard COBOL, X3.23-1974.
- *American National Dictionary for Information Processing Systems* (Copyright 1982 by the Computer and Business Equipment Manufacturers Association).

American National Standard definitions are preceded by an asterisk (*).

A

*** abbreviated combined relation condition**. The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

abend. Abnormal termination of program.

*** access mode**. The manner in which records are to be operated upon within a file.

*** actual decimal point**. The physical representation, using the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

*** alphabet-name**. A user-defined word, in the SPECIAL-NAMES paragraph of the Environment Division, that assigns a name to a specific character set and/or collating sequence.

*** alphabetic character**. A letter or a space character.

*** alphanumeric character**. Any character in the computer's character set.

alphanumeric-edited character. A character within an alphanumeric character string that contains at least one B, 0 (zero), or / (slash).

*** alphanumeric function**. A function whose value contains a string of one or more characters from the computer's character set.

*** alternate record key**. A key, other than the prime record key, whose contents identify a record within an indexed file.

ANSI (American National Standards Institute). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

*** argument.**. An identifier, a literal, an arithmetic expression, or a function identifier that specifies a value to be used in the evaluation of a function.

*** arithmetic expression**. An identifier of a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

*** arithmetic operation**. The process caused by the execution of an arithmetic statement, or the evaluation of an arithmetic expression, that results in a mathematically correct solution to the arguments presented.

*** arithmetic operator**. A single character, or a fixed two-character combination that belongs to the following set:

Character	Meaning
+	addition
–	subtraction
*	multiplication
/	division
**	exponentiation

*** arithmetic statement**. A statement that causes an arithmetic operation to be executed. The arithmetic statements are the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements.

array. In LE/VSE, an aggregate consisting of data objects, each of which may be uniquely referenced by subscripting. Roughly analogous to a COBOL table.

*** ascending key**. A key upon the values of which data is ordered, starting with the lowest value of the key up to the highest value of the key, in accordance with the rules for comparing data items.

ASCII. American National Standard Code for Information Interchange. The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange between data processing systems, data communication

systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

Note: IBM has defined an extension to ASCII code (characters 128-255).

assignment-name. A name that identifies the organization of a COBOL file and the name by which it is known to the system.

*** assumed decimal point**. A decimal point position that does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning with no physical representation.

*** AT END condition**. A condition caused:

1. During the execution of a READ statement for a sequentially accessed file, when no next logical record exists in the file, or when the number of significant digits in the relative record number is larger than the size of the relative key data item, or when an optional input file is not present.
2. During the execution of a RETURN statement, when no next logical record exists for the associated sort or merge file.
3. During the execution of a SEARCH statement, when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

B

binary item. A numeric data item represented in binary notation (on the base 2 numbering system). Binary items have a decimal equivalent consisting of the decimal digits 0 through 9, plus an operational sign. The leftmost bit of the item is the operational sign.

binary search. A dichotomizing search in which, at each step of the search, the set of data elements is divided by two; some appropriate action is taken in the case of an odd number.

*** block**. A physical unit of data that normally contains one or more logical records. For mass storage files, a block may contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical record(s) that are either contained within the block or that overlap the block. The term is synonymous with physical record.

breakpoint. A place in a computer program, usually specified by an instruction, where its execution may be interrupted by external intervention or by a monitor program.

buffer. A portion of storage used to hold input or output data temporarily.

byte. A string consisting of a certain number of bits, usually eight, treated as a unit, and representing a character.

C

callable services. In LE/VSE, a set of services that can be invoked by a COBOL program using the conventional LE/VSE-defined call interface, and usable by all programs sharing the LE/VSE conventions.

called program. A program that is the object of a CALL statement.

*** calling program**. A program that executes a CALL to another program.

case structure. A program processing logic in which a series of conditions is tested in order to make a choice between a number of resulting actions.

century window. A century window is a 100-year interval within which any 2-digit year is unique. There are several types of century window available to COBOL programmers:

1. For windowed date fields, the YEARWINDOW compiler option
2. For windowing intrinsic functions DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD, and YEAR-TO-YYYY, it is specified by argument-2
3. For LE/VSE callable services, it is specified in CEESCEN

*** character**. The basic indivisible unit of the language.

character position. The amount of physical storage required to store a single standard data format character described as USAGE IS DISPLAY.

character set. All the valid characters for a programming language or a computer system.

*** character string**. A sequence of contiguous characters that form a COBOL word, a literal, a PICTURE character string, or a comment-entry. Must be delimited by separators.

checkpoint. A point at which information about the status of a job and the system can be recorded so that the job step can be later restarted.

*** class condition**. The proposition, for which a truth value can be determined, that the content of an item is wholly alphabetic, is wholly numeric, or consists exclusively of those characters listed in the definition of a class-name.

*** class-name**. A user-defined word defined in the SPECIAL-NAMES paragraph of the Environment Division that assigns a name to the proposition for which a truth value can be defined, that the content of a data item consists exclusively of those characters listed in the definition of the class-name.

*** clause**. An ordered set of consecutive COBOL character strings whose purpose is to specify an attribute of an entry.

CMS (Conversational Monitor System). A virtual machine operating system that provides general interactive, time-sharing, problem solving, and program development capabilities, and that operates only under the control of the VM/SP control program.

*** COBOL character set**. The complete COBOL character set consists of the characters listed below:

Character	Meaning
0,1...,9	digit
A,B,...,Z	uppercase letter
a,b,...,z	lowercase letter
b	space
+	plus sign
–	minus sign (hyphen)
*	asterisk
/	slant (virgule, slash)
=	equal sign
$	default 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."

*** collating sequence**. The sequence in which the characters that are acceptable to a computer are ordered for purposes of sorting, merging, comparing, and for processing indexed files sequentially.

*** column**. A character position within a print line. The columns are numbered from 1, by 1, starting at the leftmost character position of the print line and extending to the rightmost position of the print line.

*** combined condition**. A condition that is the result of connecting two or more conditions with the AND or the OR logical operator.

*** comment-entry**. An entry in the Identification Division that may be any combination of characters from the computer's character set.

*** comment line**. A source program line represented by an asterisk (*) in the indicator area of the line and any characters from the computer's character set in area A and area B of that line. The comment line serves only for documentation in a program. A special form of comment line represented by a slant (/) in the indicator area of the line and any characters from the computer's character set in area A and area B of that line causes page ejection prior to printing the comment.

*** common program**. A program which, despite being directly contained within another program, may be called from any program directly or indirectly contained in that other program.

compatible 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, the other, YYXX.

  – One has date format YYYYXXXX, the other, YYYYXX.

  A windowed date field can be subordinate to an expanded date group data item. 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 may be windowed or expanded. For example, a windowed date field with date format YYXXX is compatible with:

- Another windowed date field with date format YYXXX
- An expanded date field with date format YYYYXXX

*** compile**. (1) To translate a program expressed in a high-level language into a program expressed in an intermediate language, assembly language, or a computer language. (2) To prepare a machine language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one computer instruction for each symbolic statement, or both, as well as performing the function of an assembler.

*** compile time**. The time at which a COBOL source program is translated, by a COBOL compiler, to a COBOL object program.

compiler. A program that translates a program written in a higher level language into a machine language object program.

compiler directing statement. A statement, beginning with a compiler directing verb, that causes the compiler to take a specific action during compilation. The SAA* compiler directing statements are COPY, EJECT, SKIP1/2/3, TITLE, and USE. The non-SAA compiler-directing statements are: REPLACE, BASIS, INSERT, and DELETE.

*** complex condition**. A condition in which one or more logical operators act upon one or more conditions. (See also “negated simple condition,” and “combined condition,” “negated combined condition.”)

*** computer-name**. A system-name that identifies the computer upon which the program is to be compiled or run.

*** condition**. A status of a program at run time for which a truth value can be determined. Where the term ‘condition’ (condition-1, condition-2,...) appears in these language specifications in or in reference to ‘condition’ (condition-1, condition-2,...) of a general format, it is a conditional expression consisting of either a simple condition optionally parenthesized, or a combined condition consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses, for which a truth value can be determined.

*** conditional expression**. A simple condition or a complex condition specified in an EVALUATE, IF, PERFORM, or SEARCH statement. (See also “simple condition” and “complex condition.”)

*** conditional phrase**. A conditional phrase specifies the action to be taken upon determination of the truth value of a condition resulting from the execution of a conditional statement.

*** conditional statement**. A statement specifying that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value.

*** conditional variable**. A data item one or more values of which has a condition-name assigned to it.

*** condition-name**. A user-defined word that assigns a name to a subset of values that a conditional variable may assume; or a user-defined word assigned to a status of an implementor defined switch or device. When ‘condition-name’ is used in the general formats, it represents a unique data item reference consisting of a syntactically correct combination of a ‘condition-name’, together with qualifiers and subscripts, as required for uniqueness of reference.

*** condition-name condition**. The proposition, for which a truth value can be determined, that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

*** Configuration Section**. A section of the Environment Division that describes overall specifications of source and object programs.

CONSOLE. A COBOL environment-name associated with the operator console.

*** contiguous items**. Items that are described by consecutive entries in the Data Division, and that bear a definite hierarchic relationship to each other.

*** 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 by
| the CURRENCY SIGN clause in the SPECIAL-NAMES
| paragraph of the Environment Division. If the CUR-
| RENCY 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 cur-
| rency 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 contains a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

Data Division. One of the four main components of a COBOL program. The Data Division describes the data to be processed by the object program: 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 in the Format 2 ACCEPT statement.
- The result of certain arithmetic operations.

The term date field refers to both “expanded date field” and “windowed date field.” See also “non-date.”

date format. The date pattern of a date field, specified either:

- Explicitly, by the DATE FORMAT clause or DATEVAL intrinsic function

or

- Implicitly, by statements and intrinsic functions that return date fields.

DBCS (Double-Byte Character Set). See “Double-Byte Character Set (DBCS).”

*** debugging line**. A debugging line is any line with a ‘D’ in the indicator area of the line.

*** debugging section**. A section that contains a USE FOR DEBUGGING statement.

*** declarative sentence**. A compiler directing sentence consisting of a single USE statement terminated by the separator period.

*** declaratives**. A set of one or more special purpose sections, written at the beginning of the Procedure Division, the first of which is preceded by the keyword DECLARATIVES and the last of which is followed by the keywords END DECLARATIVES. A declarative contains a section header, followed by a USE compiler directing sentence, followed by a set of zero, one, or more associated paragraphs.

*** de-edit**. The logical removal of all editing characters from a numeric edited data item in order to determine that item's unedited numeric value.

*** delimited scope statement**. Any statement that includes its explicit scope terminator.

*** delimiter**. A character or a sequence of contiguous characters that identify the end of a string of characters and separate that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

*** descending key**. A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

digit. Any of the numerals from 0 through 9. In COBOL, the term is not used in reference to any other symbol.

*** digit position**. The amount of physical storage required to store a single digit. This amount may vary depending on the usage specified in the data description entry that defines the data item.

*** direct access**. The facility to obtain data from storage devices or to enter data into a storage device in such a way that the process depends only on the location of that data and not on a reference to data previously accessed.

*** division**. A collection of zero, one or more sections or paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. Each division consists of the division header and the related division body. There are four (4) divisions in a COBOL program: Identification, Environment, Data, and Procedure.

*** division header**. A combination of words followed by a separator period that indicates the beginning of a division. The division headers in a COBOL program are:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```

Double-Byte Character Set (DBCS). A set of characters in which each character is represented by two bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require Double-Byte Character Sets. Since each character requires two bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software which 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 Storage Area (DSA). Dynamically acquired storage containing a register save area and an area available for dynamic storage allocation (such as program variables). DSAs are generally allocated within STACK segments managed by LE/VSE.

E

*** EBCDIC (Extended Binary-Coded Decimal Interchange Code)**. A coded character set consisting of 8-bit coded characters.

EBCDIC character. Any one of the symbols included in the 8-bit EBCDIC (Extended Binary-Coded-Decimal Interchange Code) set.

edited data item. A data item that has been modified by suppressing zeros and/or inserting editing characters.

*** editing character**. A single character or a fixed two-character combination belonging to the following set:

Character	Meaning
b	space
0	zero
+	plus
–	minus
CR	credit
DB	debit
Z	zero suppress
*	check protect
$	default currency sign
,	comma (decimal point)
.	period (decimal point)
/	slant (virgule, slash)

element (text element). One logical unit of a string of text, such as the description of a single data item or verb, preceded by a unique code identifying the element type.

*** elementary item**. A data item that is described as not being further logically subdivided.

enclave. In LE/VSE, an independent collection of routines, one of which is designated as the *main* program. An enclave is roughly analogous to a COBOL program which contains called programs.

*** end of Procedure Division**. The physical position of a COBOL source program after which no further procedures appear.

*** end program header**. A combination of words, followed by a separator period, that indicates the end of a COBOL source program. The end program header is:

```
END PROGRAM program-name.
```

*** entry**. Any descriptive set of consecutive clauses terminated by a separator period and written in the Identification Division, Environment Division, or Data Division of a COBOL program.

*** environment clause**. A clause that appears as part of an Environment Division entry.

Environment Division. One of the four main component parts of a COBOL program. The Environment Division describes the computers upon which the source program is compiled and those on which the object program is executed, and provides a linkage between the logical concept of files and their records, and the physical aspects of the devices on which files are stored.

environment-name. A name, specified by IBM, that identifies system logical units, printer and card punch control characters, report codes, and/or program switches. Valid environment-names for SAA COBOL are SYSIN, SYSOUT, CONSOLE, C01, CSP, and UPSI-0 through UPSI-7. When an environment-name is associated with a mnemonic-name in the Environment Division, the mnemonic-name may then be substituted in any format in which such substitution is valid.

execution time. See "run time."

execution-time environment. See "run-time environment."

expanded date field. A date field containing an expanded (4-digit) year. See also "date field" and "expanded year."

expanded year. Four digits representing a year, including the century (for example, 1998). Appears in expanded date fields. Compare with "windowed year."

*** explicit scope terminator**. A reserved word which terminates the scope of a particular Procedure Division statement.

exponent. A number, indicating the power to which another number (the base) is to be raised. Positive exponents denote multiplication, negative exponents denote division, fractional exponents denote a root of a quantity. In COBOL, an exponential expression is indicated with the symbol '**' followed by the exponent.

*** expression**. An arithmetic or conditional expression.

*** extend mode**. The state of a file after execution of an OPEN statement, with the EXTEND phrase specified for that file, and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

*** external data**. The data described in a program as external data items and external file connectors.

*** external data item**. A data item which is described as part of an external record in one or more programs of a run unit and which itself may be referenced from any program in which it is described.

*** external data record**. A logical record which is described in one or more programs of a run unit and whose constituent data items may be referenced from any program in which they are described.

external decimal item. A format for representing numbers in which the digit is contained in bits 4 through 7 and the sign is contained in bits 0 through 3 of the rightmost byte. Bits 0 through 3 of all other bytes contain 1's (hex F). For example, the decimal value of +123 is represented as 1111 0001 1111 0010 1111 0011. (Also know as "zoned decimal item.")

*** external file connector**. A file connector which is accessible to one or more object programs in the run unit.

*** external switch**. A hardware or software device, defined and named by the implementor, which is used to indicate that one of two alternate states exists.

F

*** figurative constant**. A compiler-generated value referenced through the use of certain reserved words.

*** file**. A collection of logical records.

*** file attribute conflict condition**. An unsuccessful attempt has been made to execute an input-output operation on a file and the file attributes, as specified for that file in the program, do not match the fixed attributes for that file.

*** file clause**. A clause that appears as part of any of the following Data Division entries: file description entry (FD entry) and sort-merge file description entry (SD entry).

*** file connector**. A storage area which contains information about a file and is used as the linkage between a file-name and a physical file and between a file-name and its associated record area.

File-Control. The name of an Environment Division paragraph in which the data files for a given source program are declared.

*** file control entry**. A SELECT clause and all its subordinate clauses which declare the relevant physical attributes of a file.

*** file description entry**. An entry in the File Section of the Data Division that contains 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.

*** fixed file attributes**. Information about a file which is established when a file is created and cannot subsequently be changed during the existence of the file. These attributes include the organization of the file (sequential, relative, or indexed), the prime record key, the alternate record keys, the code set, the minimum and maximum record size, the record type (fixed or variable), the collating sequence of the keys for indexed files, the blocking factor, the padding character, and the record delimiter.

*** fixed length record**. A record associated with a file whose file description or sort-merge description entry requires that all records contain the same number of character positions.

fixed-point number. A numeric data item defined with a PICTURE clause that specifies the location of an optional sign, the number of digits it contains, and the location of an optional decimal point. The format may be either binary, packed decimal, or external decimal.

floating-point number. A numeric data item containing a fraction and an exponent. Its value is obtained by multiplying the fraction by the base of the numeric data item raised to the power specified by the exponent.

*** format**. A specific arrangement of a set of data.

*** function**. A temporary data item whose value is determined at the time the function is referenced during the execution of a statement.

*** function identifier**. A syntactically correct combination of character strings and separators that references a function. The data item represented by a function is uniquely identified by a function-name with its arguments, if any. A function identifier may include a reference modifier. A function identifier that references an alphanumeric function may be specified anywhere in the general formats that an identifier may be specified, subject to certain restrictions. A function identifier that references an integer or numeric function may be referenced anywhere in the general formats that an arithmetic expression may be specified.

function-name. A word that names the mechanism whose invocation, along with required arguments, determines the value of a function.

G

*** global name**. A name which is declared in only one program but which may be referenced from that program and from any program contained within that program. Condition-names, data-names, file-names, record-names, report-names, and some special registers may be global names.

*** group item**. A data item that contains subordinate data items.

H

header label. (1) A file label that precedes the data records on a unit of recording media. (2) Synonym for beginning-of-file label.

*** high order end**. The leftmost character of a string of characters.

I

IBM COBOL extension. Certain COBOL syntax and semantics supported by IBM compilers in addition to those described in ANSI Standard.

Identification Division. One of the four main component parts of a COBOL program. The Identification Division identifies the source program and the object program. The Identification Division may include the following documentation: author name, installation, or date.

*** identifier**. A syntactically correct combination of character strings and separators that names a data item. When referencing a data item which is not a function, an identifier consists of a data-name, together with its qualifiers, subscripts, and reference modifier, as required for uniqueness of reference. When referencing a data item which is a function, a function identifier is used.

*** imperative statement**. A statement that either begins with an imperative verb and specifies an unconditional action to be taken or is a conditional statement that is delimited by its explicit scope terminator (delimited scope statement). An imperative statement may consist of a sequence of imperative statements.

*** implicit scope terminator**. A separator period which terminates the scope of any preceding unterminated statement, or a phrase of a statement which by its occurrence indicates the end of the scope of any statement contained within the preceding phrase.

*** index**. A computer storage area or register, the content of which represents the identification of a particular element in a table.

*** index data item**. A data item in which the values associated with an index-name can be stored in a form specified by the implementor.

indexed data-name. An identifier that contains 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.

*** 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.

*** 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 and that provides information required for transmission and handling of data during execution of the object program.

*** Input-Output statement**. A statement that causes files to be processed by performing operations upon individual records or upon the file as a unit. The input-output statements are: ACCEPT (with the identifier phrase), CLOSE, DELETE, DISABLE, DISPLAY, ENABLE, OPEN, PURGE, READ, RECEIVE, REWRITE, SEND, SET (with the TO ON or TO OFF phrase), START, and WRITE.

*** input procedure**. A set of statements, 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.

*** 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.

intermediate result. An intermediate field containing the results of a succession of arithmetic operations.

*** internal data**. The data described in a program excluding all external data items and external file connectors. Items described in the Linkage Section of a program are treated as internal data.

*** internal data item**. A data item which is described in one program in a run unit. An internal data item may have a global name.

internal decimal item. A format in which each byte in a field except the rightmost byte represents two numeric digits. The rightmost byte contains one digit and the sign. For example, the decimal value +123 is represented as 0001 0010 0011 1111. (Also known as packed decimal.)

*** internal file connector**. A file connector which is accessible to only one object program in the run unit.

*** intra-record data structure**. The entire collection of groups and elementary data items from a logical record which is defined by a contiguous subset of the data description entries which describe that record. These data description entries include all entries whose level-number is greater than the level-number of the first data description entry describing the intra-record data structure.

*** invalid key condition**. A condition, at object time, caused when a specific value of the key associated with an indexed or relative file is determined to be invalid.

*** I-O-CONTROL**. The name of an Environment Division paragraph in which object program requirements for rerun points, sharing of same areas by several data files, and multiple file storage on a single input-output device are specified.

*** I-O-CONTROL entry**. An entry in the I-O-CONTROL paragraph of the Environment Division which contains clauses which provide information required for the transmission and handling of data on named files during the execution of a program.

*** I-O-Mode**. The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phase for that file.

*** I-O status**. A conceptual entity which contains the two-character value indicating the resulting status of an input-output operation. This value is made available to the program through the use of the FILE STATUS clause in the file control entry for the file.

iteration structure. A program processing logic in which a series of statements is repeated while a condition is true or until a condition is true.

K

K. When referring to storage capacity, two to the tenth power; 1024 in decimal notation.

*** key**. A data item that identifies the location of a record, or a set of data items which serve to identify the ordering of data.

*** key of reference**. The key, either prime or alternate, currently being used to access records within an indexed file.

*** 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.

LE-conforming. A characteristic of compiler products (C for VSE/ESA, COBOL for VSE/ESA, PL/I for VSE/ESA) that produce object code conforming to the Language Environment format.

*** letter**. A character belonging to one of the following two sets:

1. Uppercase letters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
2. Lowercase letters: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z.

*** level indicator**. Two alphabetic characters that identify a specific type of file or a position in a hierarchy. The level indicators in the Data Division are: CD, FD, and SD.

*** level-number**. A user-defined word, expressed as a two digit number, which indicates the hierarchical position of a data item or the special properties of a data description entry. Level-numbers in the range from 1 through 49 indicate the position of a data item in the hierarchic structure of a logical record. Level-numbers in the range 1 through 9 may be written either as a single digit or as a zero followed by a significant digit. Level-numbers 66, 77 and 88 identify special properties of a data description entry.

*** library-name**. A user-defined word that names a COBOL library that is to be used by the compiler for a given source program compilation.

*** library text**. A sequence of text words, comment lines, the separator space, or the separator pseudo-text delimiter in a COBOL library.

*** LINAGE-COUNTER**. A special register whose value points to the current position within the page body.

Linkage Section. The section in the Data Division of the called program that describes data items available from the calling program. These data items may be referred to by both the calling and called program.

literal. A character string whose value is specified either by the ordered set of characters comprising the string, or by the use of a figurative constant.

*** logical operator**. One of the reserved words AND, OR, or NOT. In the formation of a condition, either AND, or OR, or both can be used as logical connectives. NOT can be used for logical negation.

*** logical record**. The most inclusive data item. The level-number for a record is 01. A record may be either an elementary item or a group of items. The term is synonymous with record.

*** low order end**. The rightmost character of a string of characters.

M

main program. In a hierarchy of programs and subroutines, the first program to receive control when the programs are run.

*** mass storage**. A storage medium in which data may be organized and maintained in both a sequential and nonsequential manner.

*** mass storage device**. A device having a large storage capacity; for example, magnetic disk, magnetic drum.

*** mass storage file**. A collection of records that is assigned to a mass storage medium.

*** megabyte (M)**. One megabyte equals 1,048,576 bytes.

*** merge file**. A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

*** mnemonic-name**. A user-defined word that is associated in the Environment Division with a specified implementor-name.

multitasking. Mode of operation that provides for the concurrent, or interleaved, execution of two or more tasks. In LE/370, synonymous with *multithreading*.

N

name. A word containing not more than 30 characters that defines a COBOL operand.

*** native character set**. The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

*** native collating sequence**. The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

*** negated combined condition**. The 'NOT' logical operator immediately followed by a parenthesized combined condition.

*** negated simple condition**. The 'NOT' logical operator immediately followed by a simple condition.

nested program. A program that is directly contained within another program.

*** next executable sentence**. The next sentence to which control will be transferred after execution of the current statement is complete.

*** next executable statement**. The next statement to which control will be transferred after execution of the current statement is complete.

*** next record**. The record that logically follows the current record of a file.

*** noncontiguous items**. Elementary data items in the Working-Storage and Linkage Sections that bear no hierarchic relationship to other data items.

non-date. Any of the following:

- A data item whose data description entry does not include the DATE FORMAT clause
- A literal
- A reference-modified date field
- The result of certain arithmetic operations that may include date field operands; for example, the difference between two compatible dates.

The value of a non-date may or may not represent a date.

*** nonnumeric item**. A data item whose description permits its content to contain any combination of characters taken from the computer's character set. Certain categories of nonnumeric items may be formed from more restricted character sets.

*** nonnumeric literal**. A literal bounded by quotation marks. The string of characters may include any character in the computer's character set.

null. Figurative constant used to assign the value of an invalid address to pointer data items. NULLS can be used wherever NULL can be used.

*** numeric character**. A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

numeric-edited item. A numeric item that is in such a form that it may be used in printed output. It may consist of external decimal digits from 0 through 9, the decimal point, commas, the dollar sign, editing sign control symbols, plus other editing symbols.

*** numeric function**. A function whose class and category are numeric but which for some possible evaluation does not satisfy the requirements of integer functions.

*** numeric item**. A data item whose description restricts its content to a value represented by characters chosen from the digits from '0' through '9'; if signed, the item may also contain a '+', '–', or other representation of an operational sign.

*** numeric literal**. A literal containing one or more numeric characters that also contain either a decimal point, or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

O

object code. Output from a compiler or assembler which is itself executable machine code or is suitable for processing to produce executable machine code.

*** OBJECT-COMPUTER**. The name of an Environment Division paragraph in which the computer environment, within which the object program is executed, is described.

*** object computer entry**. An entry in the OBJECT-COMPUTER paragraph of the Environment Division which contains clauses which describe the computer environment in which the object program is to be executed.

object deck. A portion of an object program suitable as input to a linkage editor. Synonymous with *object module* and *text deck*.

object module. Synonym for *object deck* or *text deck*.

*** object of entry**. A set of operands and reserved words, within a Data Division entry of a COBOL program, that immediately follows the subject of the entry.

*** object program**. A set or group of executable machine language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program. Where there is no danger of ambiguity, the word 'program' alone may be used in place of the phrase 'object program.'

*** object time**. The time at which an object program is executed. The term is synonymous with execution time.

*** obsolete element**. A COBOL language element in Standard COBOL that is to be deleted from the next revision of Standard COBOL.

*** open mode**. The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O or EXTEND.

*** operand**. Whereas the general definition of operand is "that component which is operated upon," for the purposes of this document, any lowercase word (or words) that appears in a statement or entry format may be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

*** operational sign**. An algebraic sign, associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

*** optional file**. A file which is declared as being not necessarily present each time the object program is executed. The object program causes an interrogation for the presence or absence of the file.

*** optional word**. A reserved word that is included in a specific format only to improve the readability of the language and whose presence is optional to the user when the format in which the word appears is used in a source program.

OS/2® (Operating System/2). A multi-tasking operating system for the IBM Personal Computer family that allows you to run both DOS mode and OS/2 mode programs.

*** output file**. A file that is opened in either the OUTPUT mode or EXTEND mode.

*** output mode**. The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file.

*** output procedure**. A set of statements to which control is given during execution of a SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function reaches a point at which it can select the next record in merged order when requested.

overflow condition. A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

P

packed decimal item. See "internal decimal item."

*** padding character**. An alphanumeric character used to fill the unused character positions in a physical record.

page. A vertical division of output data representing a physical separation of such data, the separation being based on internal logical requirements and/or external characteristics of the output medium.

*** page body**. That part of the logical page in which lines can be written and/or spaced.

*** paragraph**. In the Procedure Division, a paragraph-name followed by a separator period and by zero, one, or more sentences. In the Identification and Environ-

ment Divisions, a paragraph header followed by zero, one, or more entries.

*** paragraph header**. A reserved word, followed by the separator period, that indicates the beginning of a paragraph in the Identification and Environment Divisions. The permissible paragraph headers in the Identification Division are:

```
PROGRAM-ID.
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.
```

The permissible paragraph headers in the Environment Division are:

```
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
FILE-CONTROL.
I-O-CONTROL.
```

*** paragraph-name**. A user-defined word that identifies and begins a paragraph in the Procedure Division.

parameter. Parameters are used to pass data values between calling and called programs.

password. A unique string of characters that a program, computer operator, or user must supply to meet security requirements before gaining access to data.

*** phrase**. A phrase is an ordered set of one or more consecutive COBOL character strings that form a portion of a COBOL procedural statement or of a COBOL clause.

*** physical record**. See “block.”

pointer data item. A data item in which address values can be stored. Data items are explicitly defined as pointers with the USAGE IS POINTER clause. ADDRESS OF special registers are implicitly defined as pointer data items. Pointer data items can be compared for equality or moved to other pointer data items.

portability. The ability to transfer an application program from one application platform to another with relatively few changes to the source program.

*** prime record key**. A key whose contents uniquely identify a record within an indexed file.

*** priority-number**. A user-defined word which classifies sections in the Procedure Division for purposes of segmentation. Segment-numbers may contain only the characters '0','1', ... , '9'. A segment-number may be expressed either as a one or two digit number.

*** procedure**. A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the Procedure Division.

*** procedure branching statement**. A statement that causes the explicit transfer of control to a statement other than the next executable statement in the sequence in which the statements are written in the source program. The procedure branching statements are: ALTER, CALL, EXIT, EXIT PROGRAM, GO TO, MERGE, (with the OUTPUT PROCEDURE phrase), PERFORM and SORT (with the INPUT PROCEDURE or OUTPUT PROCEDURE phrase).

Procedure Division. One of the four main component parts of a COBOL program. The Procedure Division contains instructions for solving a problem. The Procedure Division may contain imperative statements, conditional statements, compiler directing statements, paragraphs, procedures, and sections.

*** procedure-name**. A user-defined word that is used to name a paragraph or section in the Procedure Division. It consists of a paragraph-name (which may be qualified), or a section-name.

procedure-pointer data item. A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point.

*** program identification entry**. An entry in the PROGRAM-ID paragraph of the Identification Division which contains clauses that specify the program-name and assign selected program attributes to the program.

*** program-name**. In the Identification Division and the end program header, a user-defined word that identifies a COBOL source program.

*** pseudo-text**. A sequence of text words, comment lines, or the separator space in a source program or COBOL library bounded by, but not including, pseudo-text delimiters.

*** pseudo-text delimiter**. Two contiguous equal sign characters (==) used to delimit pseudo-text.

*** punctuation character**. A character that belongs to the following set:

Character	Meaning
,	comma
;	semicolon
:	colon
.	period (full stop)
"	quotation mark
(	left parenthesis
)	right parenthesis
b	space

= equal sign

Q

*** qualified data-name**. An identifier that contains a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

*** qualifier**.

1. A data-name or a name associated with a level indicator which is used in a reference either together with another data-name which is the name of an item that is subordinate to the qualifier or together with a condition-name.
2. A section-name that is used in a reference together with a paragraph-name specified in that section.
3. A library-name that is used in a reference together with a text-name associated with that library.

R

*** random access**. An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

*** record**. See “logical record.”

*** record area**. A storage area allocated for the purpose of processing the record described in a record description entry in the File Section of the Data Division. In the File Section, the current number of character positions in the record area is determined by the explicit or implicit RECORD clause.

*** record description**. See “record description entry.”

*** record description entry**. The total set of data description entries associated with a particular record. The term is synonymous with record description.

recording mode. The format of the logical records in a file. Recording mode can be F (fixed-length), V (variable-length), S (spanned), or U (undefined).

record key. A key whose contents identify a record within an indexed file. Within an indexed file in SAA COBOL, a record key is the prime record key.

*** 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.

reel. A discrete portion of a storage medium, the dimensions of which are determined by each implementor, that contains part of a file, all of a file, or any number of files. The term is synonymous with unit and volume.

reentrant. The attribute of a program or routine that allows more than one user to share a single copy of a phase.

*** reference format**. A format that provides a standard method for describing COBOL source programs.

reference modification. A method of defining a new alphanumeric data item by specifying the leftmost character and length relative to the leftmost character of another alphanumeric data item.

*** reference modifier**. A syntactically correct combination of character strings and separators that defines a unique data item. It includes a delimiting left parenthesis separator, the leftmost character position, a colon separator, optionally a length, and a delimiting right parenthesis separator.

*** relation**. See “relational operator.” or “relation condition”

*** relational operator**. A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meanings are:

Operator	**Meaning**
IS GREATER THAN	Greater than
IS >	Greater than
IS NOT GREATER THAN	Not greater than
IS NOT >	Not greater than
IS LESS THAN	Less than
IS <	Less than
IS NOT LESS THAN	Not less than
IS NOT <	Not less than
IS EQUAL TO	Equal to
IS =	Equal to
IS NOT EQUAL TO	Not equal to
IS NOT =	Not equal to
IS GREATER THAN OR EQUAL TO	Greater than or equal to
IS >=	Greater than or equal to
IS LESS THAN OR EQUAL TO	Less than or equal to
IS <=	Less than or equal to

*** relation character**. A character that belongs to the following set:

Character	Meaning
>	greater than
<	less than
=	equal to

*** relation condition**. The proposition, for which a truth value can be determined, that the value of an arithmetic expression, data item, 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.”)

*** relative file**. A file with relative organization.

*** relative key**. A key whose contents identify a logical record in a relative file.

*** relative organization**. The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the record’s logical ordinal position in the file.

*** relative record number**. The ordinal number of a record in a file whose organization is relative. This number is treated as a numeric literal which is an integer.

*** reserved word**. A COBOL word specified in the list of words that may be used in a COBOL source program, but that must not appear in the program as user-defined words or system-names.

*** resource**. A facility or service, controlled by the operating system, that can be used by an executing program.

*** resultant identifier**. A user-defined data item that is to contain the result of an arithmetic operation.

routine. A set of statements in a COBOL program that causes the computer to perform an operation or series of related operations. In LE/VSE, refers to either a procedure, function, or subroutine.

*** routine-name**. A user-defined word that identifies a procedure written in a language other than COBOL.

*** run time**. The time at which an object program is executed. The term is synonymous with object time.

run-time environment. The environment in which a COBOL program executes.

*** run unit**. One or more object programs which interact with one another and which function, at object time, as an entity to provide problem solutions.

S

SAM (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.

SBCS (Single Byte Character Set). See "Single Byte Character Set (SBCS)".

scope terminator. A COBOL reserved word that marks the end of certain Procedure Division statements. It may be either explicit (END-ADD, for example) or implicit (separator period).

*** section**. A set of zero, one or more paragraphs or entities, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

*** section header**. A combination of words followed by a separator period that indicates the beginning of a section in the Environment, Data, and Procedure Divisions. In the Environment and Data Divisions, a section header contains 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.
LINKAGE SECTION.
```

In the Procedure Division, a section header contains a section-name, followed by the reserved word SECTION, followed by a separator period.

*** section-name**. A user-defined word that names a section in the Procedure Division.

selection structure. A program processing logic in which one or another series of statements is executed, depending on whether a condition is true or false.

*** sentence**. A sequence of one or more statements, the last of which is terminated by a separator period.

*** separately-compiled program**. A program which, together with its contained programs, is compiled separately from all other programs.

*** separator**. A character or two contiguous characters used to delimit character strings.

*** separator comma**. A comma (,) followed by a space used to delimit character strings.

*** separator period**. A period (.) followed by a space used to delimit character strings.

*** separator semicolon**. A semicolon (;) followed by a space used to delimit character strings.

sequence structure. A program processing logic in which a series of statements is executed in sequential order.

*** sequential access**. An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

*** sequential file**. A file with sequential organization.

*** sequential organization**. The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

serial search. A search in which the members of a set are consecutively examined, beginning with the first member and ending with the last.

*** 77-level-description-entry**. A data description entry that describes a noncontiguous data item with the level-number 77.

*** sign condition**. The proposition, for which a truth value can be determined, that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

*** simple condition**. Any single condition chosen from the set:

Relation condition
Class condition
Condition-name condition
Switch-status condition
Sign condition

Single Byte Character Set (SBCS). A set of characters in which each character is represented by a single byte. See also "EBCDIC (Extended Binary-Coded Decimal Interchange Code)."

slack bytes. Bytes inserted between data items or records to ensure correct alignment of some numeric items. Slack bytes contain no meaningful data. In some cases, they are inserted by the compiler; in others, it is the responsibility of the programmer to insert them. The SYNCHRONIZED clause instructs the compiler to insert slack bytes when they are needed for proper alignment. Slack bytes between records are inserted by the programmer.

*** 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 contains the level indicator SD, followed by a file-name, and then followed by a set of file clauses as required.

*** SOURCE-COMPUTER**. The name of an Environment Division paragraph in which the computer environment, within which the source program is compiled, is described.

*** source computer entry**. An entry in the SOURCE-COMPUTER paragraph of the Environment Division which contains clauses which describe the computer environment in which the source program is to be compiled.

*** source item**. An identifier designated by a SOURCE clause that provides the value of a printable item.

source program. Although it is recognized that a source program may be represented by other forms and symbols, in this document it always refers to a syntactically correct set of COBOL statements. A COBOL source program commences with the Identification Division or a COPY statement. A COBOL source program is terminated by the end program header, if specified, or by the absence of additional source program lines.

*** special character**. A character that belongs to the following set:

Character	Meaning
+	plus sign
–	minus sign (hyphen)
*	asterisk
/	slant (virgule, slash)
=	equal sign
$	default currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point, full stop)
"	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol
:	colon

*** special-character word**. A reserved word that is an arithmetic operator or a relation character.

SPECIAL-NAMES. The name of an Environment Division paragraph in which environment-names are related to user-specified mnemonic-names.

*** special names entry**. An entry in the SPECIAL-NAMES paragraph of the Environment Division which provides means for specifying the currency sign values and currency symbols; choosing the decimal point; specifying symbolic characters; relating implementor-names to user-specified mnemonic-names; relating alphabet-names to character sets or collating sequences; and relating class-names to sets of characters.

*** special registers**. Certain compiler generated storage areas whose primary use is to store information produced in conjunction with the use of a specific COBOL feature.

*** standard data format**. The concept used in describing the characteristics of data in a COBOL Data Division under which the characteristics or properties of the data are expressed in a form oriented to the appearance of the data on a printed page of infinite length and breadth, rather than a form oriented to the manner in which the data is stored internally in the computer, or on a particular external medium.

*** statement**. A syntactically valid combination of words, literals, and separators,beginning with a verb, written in a COBOL source program.

structured programming. A technique for organizing and coding a computer program in which the program includes 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.

*** subject of entry**. An operand or reserved word that appears immediately following the level indicator or the level-number in a Data Division entry.

*** subprogram**. See "called program."

*** subscript**. An occurrence number represented by either an integer, a data-name optionally followed by an integer with the operator + or -, or an index-name optionally followed by an integer with the operator + or -, that identifies a particular element in a table. A subscript may be the word ALL when the subscripted identifier is used as a function argument for a function allowing a variable number of arguments.

*** subscripted data-name**. An identifier that conatins a data-name followed by one or more subscripts enclosed in parentheses.

switch-status condition. The proposition, for which a truth value can be determined, that an 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 in pseudo-text which is:

- A separator, except for: space; a pseudo-text delimiter; and the opening and closing delimiters for nonnumeric literals. The right parenthesis and left parenthesis characters, regardless of context within the library, source program, or pseudo-text, are always considered text words.
- A literal including, in the case of 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 which is neither a separator nor a literal.

top-down design. The design of a computer program using a hierarchic structure in which related functions are performed at each level of the structure.

top-down development. See "structured programming."

trailer-label. (1) A file that follows the data records on a unit of recording medium. (2) Synonym for end-of-file label.

*** truth value**. The representation of the result of the evaluation of a condition in terms of one of two values: true or false.

U

*** unary operator**. A plus (+) or a minus (-) sign, that precedes a variable or a left parenthesis in an arithmetic expression and that has the effect of multiplying the expression by +1 or -1, respectively.

unit. A module of direct access, the dimensions of which are determined by IBM.

*** unsuccessful execution**. The attempted execution of a statement that does not result in the execution of all the operations specified by that statement. The unsuccessful execution of a statement does not affect any data referenced by that statement, but may affect status indicators.

UPSI switch. A program switch that performs the functions of a hardware switch. Eight are provided: UPSI-0 through UPSI-7.

*** user-defined word**. A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

V

*** variable**. A data item whose value may be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

*** variable length record**. A record associated with a file whose file description or sort-merge description entry permits records to contain a varying number of character positions.

*** variable occurrence data item**. A variable occurrence data item is a table element which is repeated a variable number of times. Such an item must contain an OCCURS DEPENDING ON clause in its data description entry, or be subordinate to such an item.

*** 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 executed automatically when the end of a unit or reel has been reached before end-of-file has been reached.

VSAM (Virtual Storage Access Method). A high-performance mass storage access method. Three types of data organization are available: entry sequenced files (ESDS), key sequenced data sets (KSDS), and relative record data sets (RRDS). Their COBOL equivalents are, respectively: sequential, indexed, and relative organizations.

VSE/ESA (Virtual Storage Extended/Enterprise Systems Architecture). An IBM operating system that manages multiple address spaces (partitions), up to a maximum combined virtual storage size of 256 million bytes. Address spaces of up to 2 GB are now supported (approx. 2048 megabytes).

W

windowed date field. A date field containing a windowed (2-digit) year. See also “date field” and “windowed year.”

windowed year. Two digits representing a year within a century window (for example, 98). Appears in windowed date fields. See also “century window.”

Compare with “expanded year.”

*** word**. A character string of not more than 30 characters which forms a user-defined word, a system-name, a reserved word, or a function-name.

*** Working-Storage Section**. The section of the Data Division that describes working storage data items, contains either noncontiguous items or working storage records or both.

Z

zoned decimal item. See “external decimal item.”

Index

Special Characters

Numerics

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

We'd Like to Hear from You

IBM COBOL for VSE/ESA
Programming Guide
Release 1

Publication No. SC26-8072-02

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773.
- Electronic mail—Use one of the following network IDs:
  - IBMLink: HLASMPUB at STLVM27
  - Internet: COMMENTS@VNET.IBM.COM

  Be sure to include the following with your comments:
  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

IBM COBOL for VSE/ESA
Programming Guide
Release 1

Publication No. SC26-8072-02

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name

Address

Company or Organization

Phone No.

Readers' Comments
SC26-8072-02

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department HHX/H1
555 Bailey Avenue
SAN JOSE, CA 95141-1099

Fold and Tape **Please do not staple** Fold and Tape

SC26-8072-02

Cut or Fold
Along Line

File Number: S370-40
Program Number: 5686-068

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

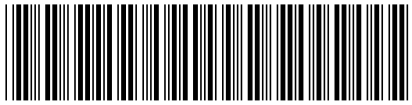
COBOL for VSE/ESA

SC26-8528	Diagnosis Guide
GC26-8068	General Information
GC26-8069	Licensed Program Specifications
SC26-8073	Language Reference
GC26-8070	Migration Guide
SC26-8072	Programming Guide
SC26-8071	Installation and Customization Guide
SX26-3834	Reference Summary

VisualAge COBOL Millennium Language Extensions for VSE/ESA

SC26-8071	Installation and Customization Guide
GC26-9266	COBOL Millennium Language Extensions Guide
GC26-9321	Fact Sheet
GC26-9417	Licensed Program Specifications

SC26-8072-02



Spine information:

IBM COBOL for VSE/ESA Programming Guide *Release 1*