FUJITSU Software
NetCOBOL V11.0

User's Guide

Windows(64)

B1WD-3306-01ENZ0(00)
March 2014

Preface

NetCOBOL allows you to create, execute, and debug COBOL programs with Windows. This manual describes, in detail, the functions and operations of NetCOBOL and how to use NetCOBOL with other languages, databases and products.

It does not detail COBOL syntax. Refer to the NetCOBOL Language Reference for details on COBOL syntax.

Refer to "NetCOBOL Messages" for details on messages generated by the compile commands, the COBOL compiler, the COBOL runtime system and the COBOL Error Report.

Audience

This manual is for users who develop COBOL programs using NetCOBOL. It assumes users possess a basic knowledge of COBOL and are familiar with the appropriate Windows platform.

The manual provides an introduction to OO programming concepts so that those who have no prior OO programming experience can learn OO COBOL.

How this Manual is Organized

This manual consists of the following chapters and appendixes:

Chapter	Contents
Chapter 1 Overview of NetCOBOL	Description of the operating environment and functions of NetCOBOL.
Chapter 2 Creating and Editing a Program	Introduction to the options and tools for writing a COBOL program.
Chapter 3 Compiling Programs	How to compile COBOL programs: required resources, supporting tools, command formats and messages.
Chapter 4 Linking Programs	How to link COBOL programs: required resources, supporting tools, command formats and messages.
Chapter 5 Executing Programs	How to execute a COBOL program and configure its run time environment.
Chapter 6 File Processing	Descriptions of file structures and how to use them.
Chapter 7 Printing	Details of the techniques provided for printing documents and forms.
Chapter 8 Using Screens	Methods of how to display data to, and accept data from, the screen.
Chapter 9 Calling Subprograms (Inter-Program Communication)	How to call subprograms from a COBOL program, including C, C++ and VB.
Chapter 10 Using ACCEPT and DISPLAY Statements	How to use ACCEPT and DISPLAY statements for: simplified file input-output, obtaining command line arguments, and environment variable operations.
Chapter 11 Using SORT/MERGE Statements (Sort-Merge Function)	How to use the Sort and Merge functions.
Chapter 12 System Program Functions	Description of specialist functions useful to those creating system programs.
Chapter 13 Introduction to Object-Oriented Programming	Overview of the goals and concepts behind the OO extensions to COBOL.
Chapter 14 Basic Features of OO COBOL	Descriptions of the basic OO COBOL features and how to use them.
Chapter 15 Developing OO COBOL Applications	Techniques and tools for developing OO COBOL applications.
Chapter 16 Advanced Features of OO COBOL	Details of the more advanced features of OO COBOL

Chapter	Contents
Chapter 17 Database (SQL)	How to code and configure the environment for remote ODBC access.
Chapter 18 Server Type Applications	Explains the background processing function and the Web linkage function. The Web linkage function is unique to the server products.
Chapter 19 Multithread Programs	Description of the multi-thread program. The multi-thread function is unique to the server products.
Chapter 20 Unicode	How to create a COBOL application operating in Unicode.
Chapter 21 Operation of Comma Separated Value data	How to operate the Comma Separated Value data.
Chapter 22 Remote Development Support Function	Describes the remote distributed development support function
Appendix A Compiler Options	Details of the NetCOBOL compiler options.
Appendix B I-O Status List	The values and meanings of the I-O status codes returned from the execution of input-output statements.
Appendix C Global Optimization	Optimizations performed by the NetCOBOL compiler.
Appendix D Intrinsic Function List	List of all the COBOL intrinsic functions supported by NetCOBOL.
Appendix E Writing Special Literals	Details of literals for Windows.
Appendix F OSIV-series Function Comparison	Compares the functions available with the GS- series and NetCOBOL.
Appendix G Special Windows Subroutines	Description of subroutines for receiving Windows handles.
Appendix H Incompatible Syntax Between Standard and OO COBOL	Lists of earlier standard COBOL features that cannot be used with OO COBOL.
Appendix I Other Commands	Description of the DUMPBIN command used to extract information from object files.
Appendix J Character Code-types	Gives an overview of the character code-types.
Appendix K Security	Gives an overview of security issues you should be aware of.
Appendix L COBOL coding techniques	Describes COBOL program coding techniques.

Conventions Used in this Manual

This manual uses the following typographic conventions.

Example of Convention	Description
setup	Characters you enter appear in bold.
Program-name	Underlined text indicates a placeholder for information you supply.
ENTER	Small capital letters are used for the name of keys and key sequences such as ENTER and CTRL+R. A plus sign (+) indicates a combination of keys.
…	Ellipses indicate the item immediately preceding can be specified repeatedly.
Edit, Literal	Names of pull down menus and options appear with the initial letter capitalized.
[def]	Indicates that the enclosed item may be omitted.

Example of Convention	Description
{ ABC } or {ABC|DEF} DEF	Indicates that one of the enclosed items (delimited by |) is to be selected. If items are omitted the underlined item is assumed.
CHECK WITH PASCAL LINKAGE ALL PARAGRAPH-ID COBOL ALL	Commands, statements, clauses, and options you enter or select appear in uppercase. Program section names, and some proper names also appear in uppercase. Defaults are underlined.

`PROCEDURE DIVISION`
`     :`
`  ADD 1 TO POW-FONTSIZE OF LABEL1.`
`  IF POW-FONTSIZE OF LABEL1 > 70 THEN`
`    MOVE 1 TOW POW-FONTSIZE OF LABEL1.`
`  END-IF.`
	This font is used for examples of program code.

The *sheet* acts as an application creation window.	Italics are occasionally used for emphasis.
"COBOL Language Reference" Refer to "Chapter 5 Compile Options".	References to other publications or sections within publications are in quotation marks.

We have tried to make all of our examples position-sensitive. However, given the restrictions of the size of the page, in some examples we have not been able to accomplish this. You should be aware that NetCOBOL is a position-sensitive language, unless you specify that your source is free format by using the SRF compiler option.

The term *national language* or *national* in this manual indicates double byte character languages, such as Japanese, Korean, or Chinese. Functions that are only available in the national language version of this system are indicated by [*XXXXXX*].

Product Names

Product Name	Abbreviation
Microsoft® Windows Server® 2012 R2 Datacenter Microsoft® Windows Server® 2012 R2 Standard Microsoft® Windows Server® 2012 R2 Essentials Microsoft® Windows Server® 2012 R2 Foundation	Windows Server 2012 R2
Microsoft® Windows Server® 2012 Datacenter Microsoft® Windows Server® 2012 Standard Microsoft® Windows Server® 2012 Essentials Microsoft® Windows Server® 2012 Foundation	Windows Server 2012
Microsoft® Windows Server® 2008 R2 Foundation Microsoft® Windows Server® 2008 R2 Standard Microsoft® Windows Server® 2008 R2 Enterprise Microsoft® Windows Server® 2008 R2 Datacenter	Windows Server 2008 R2
Windows® 8.1	Windows 8.1(x64)

Product Name	Abbreviation
Windows® 8.1 Pro Windows® 8.1 Enterprise	
Windows® 8 Windows® 8 Pro Windows® 8 Enterprise	Windows 8(x64)
Windows® 7 Home Premium Windows® 7 Professional Windows® 7 Enterprise Windows® 7 Ultimate	Windows 7(x64)
Microsoft(R) Visual C++(R) development system	Visual C++
Microsoft(R)Visual Basic(R) programming system	Visual Basic
Oracle Solaris	Solaris

- In this manual, when all the following products are indicates, it is written as "Windows" or "Windows(x64)".
    - Windows Server 2012 R2
    - Windows Server 2012
    - Windows Server 2008 R2
    - Windows 8.1(x64)
    - Windows 8(x64)
    - Windows 7(x64)

Product Differences

The following products are not supported in the US English-language version or other English-language versions of this product, but may be mentioned in this manual:

- BS*NET
- IDCM
- MeFt/NET
- MeFt/NET-SV
- PowerAIM
- RDB/7000 Server for Windows NT
- MeFt/Web
- Print Walker/OVL option
- Systemwalker/List Works
- PowerRDBconnector

Note

NetCOBOL Studio is included in NetCOBOL product that develops 32-bit COBOL application on Windows.

Trademarks

- NetCOBOL is a trademark or registered trademark of Fujitsu Limited or its subsidiaries in the United States or other countries or in both.
- Microsoft, MS, MS-DOS, ActiveX, FoxPro, the Fox head design, Internet Explorer, Microsoft Developer Studio, Microsoft Press, Microsoft QuickBasic, PivotTable, Rushmore, Visual Basic, Visual C++, Visual FoxPro, Visual InterDev, Visual J++, Visual SourceSafe, Windows, Windows NT, Windows Vista, and Windows Server are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.
- TrueType is a registered trademark of Apple Computer, Inc.
- MicroKernel Database Architecture, MicroKernel Database Engine, Navigational Client/Server and Scalable SQL are trademarks and Btrieve is a registered trademark of Pervasive Software Inc.
- HP, HP-UX, and SoftBench are trademarks of the Hewlett-Packard Company.
- Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. Oracle Solaris might be described as Solaris, Solaris Operating System, or Solaris OS.
- Other product names are trademarks or registered trademarks of each company. Trademark indications are omitted for some system and product names described in this manual.
- The permission of the Microsoft Corporation has been obtained for the use of the screen shots.

Acknowledgment

The language specifications of COBOL are based on the original specifications developed by the work of the Conference on Data Systems Languages (CODASYL). The specifications described in this manual are also derived from the original. The following passages are quoted at the request of CODASYL.

"COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations. No warranty, expressed or implied, is made by the COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by the committee, in connection therewith.

"The authors of the following copyrighted material have authorized the use of this material in part in the COBOL specifications. Such authorization extends to the use of the original specifications in other COBOL specifications:

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

Export Regulation

Exportation/release of this document may require necessary procedures in accordance with the regulations of the Foreign Exchange and Foreign Trade Control Law of Japan and/or US export control laws.

March 2014

Contents

Chapter 1 Overview of NetCOBOL

This chapter explains the functions of NetCOBOL and its operating environment. If you are unfamiliar with NetCOBOL, you should read this chapter before using the product.

1.1 NetCOBOL Functions

1.1.1 COBOL Functions

NetCOBOL implements the following COBOL functions:

- Nucleus (The base elements of the COBOL language)
- Sequential file
- Relative file
- Indexed file
- Inter-program communication
- Sort-merge
- Source text manipulation
- Presentation file
- Intrinsic function
- Floating-point number
- Screen handling
- Command line argument handling
- Environment variable operation
- Database (SQL)
- System program description (SD)
- Object oriented COBOL programming

For information about how to write COBOL statements to use these functions, refer to the NetCOBOL Language Reference.

NetCOBOL provides the following functions that can be used in server-side applications.

- Multithread
- Developing and operating a batch-type application

1.1.2 Programs and Utilities Provided by NetCOBOL

NetCOBOL provides the programs and the utilities listed below.

Table 1.1 NetCOBOL programs and utilities

Name	Purpose
NetCOBOL compiler	Compiles a program using NetCOBOL.
NetCOBOL runtime system	Executes a NetCOBOL application.
NetCOBOL FILE UTILITY	Processes NetCOBOL data files.
Execution environment setting tool	Editing an initialization file for execution.
COBOL Error Report	Debugging COBOL applications.

Name	Purpose
NetCOBOL Studio	Supporting COBOL application development

NetCOBOL Compiler

The NetCOBOL compiler compiles a COBOL source program to create an object program. The compiler provides the following service functions:

- Outputs compiler listings
- Checks standards and specifications
- Provides global optimization

You can specify these functions in accordance with the compiler options.

NetCOBOL Runtime System

When you execute an application program created with NetCOBOL, the COBOL runtime system is called.

NetCOBOL FILE UTILITY

The NetCOBOL FILE UTILITY responds to utility commands without referencing the COBOL application.

Execution Environment Setting Tool

Use this tool to edit an initialization file for execution of a COBOL application.

COBOL Error Report

Use this function to debug a COBOL application if an application error occurs or a U-level execution message is issued. This function provides the following information:

- Type of error
- Location of error occurrence
- Calling path to the location where an error has occurred
- Type of system and its version information
- Command line
- Environment variable in the process
- COBOL execution environment information
- List of processes in the system
- List of modules in the processes
- Thread information in the processes
- Stack information for each thread

NetCOBOL Studio

NetCOBOL Studio is an Integrated Development Environment to develop the COBOL program.

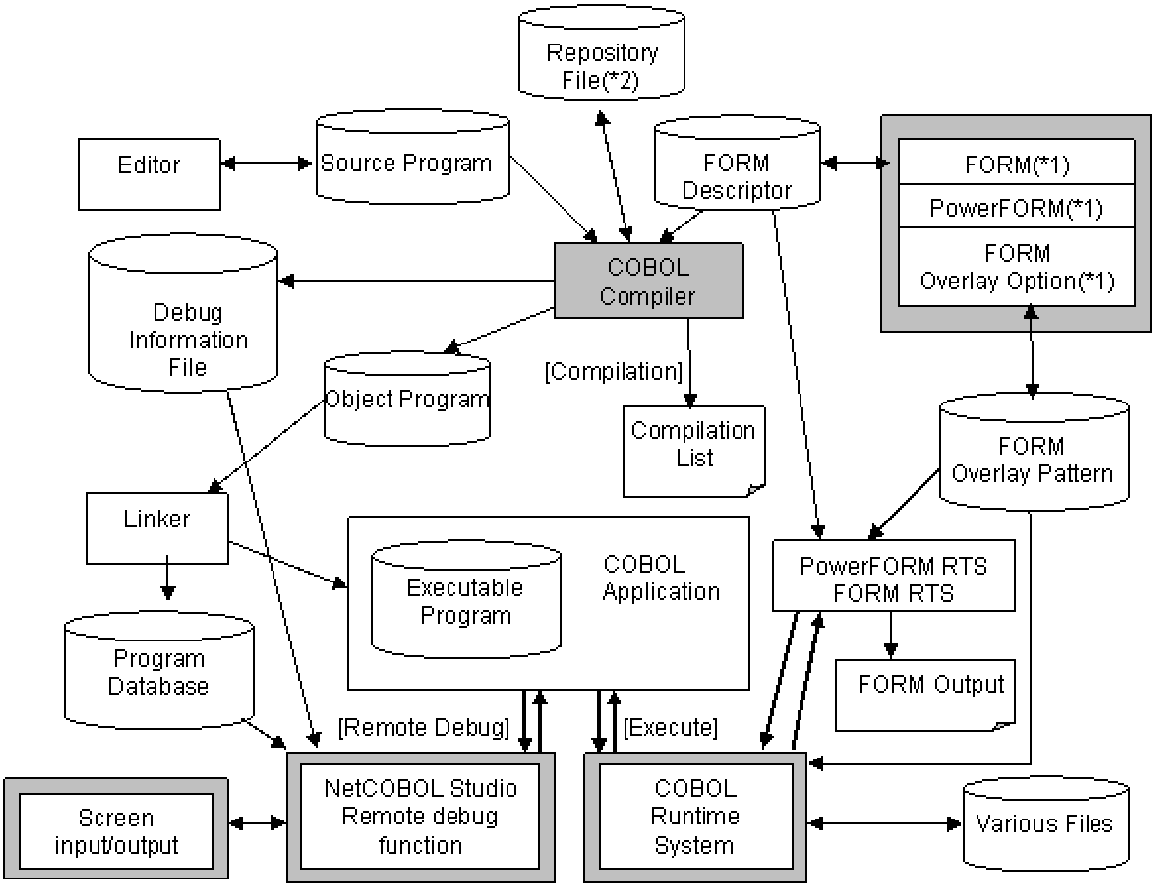
The edit of COBOL program, compilation, debugging and execution of COBOL program can be done efficiently.

Please refer to "NetCOBOL Studio User's Guide" for more information.

1.2 NetCOBOL Development Environment

The following figure provides an overview of the NetCOBOL development environment.

Figure 1.1 The NetCOBOL development environment



*1 : FORM, PowerFORM and FORM Overlay Option are products that operate on 32 bits Windows. It is not possible to install it in 64-bit environment.

*2 : Necessary for Developing an Object Oriented Program

1.2.1 Setting Environment Variables

Some of the NetCOBOL functions require the setting of the environment variables listed in "Table 1.2 Setting environment variables".

- Use the management tool on the control panel to set the environment variables.
- Use the SET command to set the environment variables.

For details on how to set the environment variables with the SET command, see "5.3.2.2 Setting from the SET Command". For details on how to set the environment variable from the control panel, refer to the Windows on-line help.

Table 1.2 Setting environment variables

Setup Timing	Environment Variable	Details	Conditions	
Common	PATH	NetCOBOL Compiler install folder	When using NetCOBOL	Required
		NetCOBOL Runtime system install folder		
		FORM RTS install folder	When using form descriptors	Required

Setup Timing	Environment Variable	Details	Conditions	
At compilation	SMED_SUFFIX	Extension of form descriptors (any character string)	When changing the extension SMD of form descriptors	Optional
			When changing the extension PMD of form descriptors	Optional
	FORMLIB	Folder name of the file storing form descriptors	When using form descriptors	Optional
	COB_COBCOPY	Library file folder	When using the library file	Optional
	COB_LIBRARY-NAME (*1)	Library file folder with the IN/OF specification	When using the library file with the IN/OF specification	Optional
	COB_LIBSUFFIX	Extension of library file (any character string)	When changing the extension of the library file	Optional
	COB_REPIN	Input folder of repository file	When using the repository file	Optional
	COB_OPTIONS	Compilation command option	When using the compilation command option	Optional
At linking	LIB	NetCOBOL Runtime system install folder.	When linking	Required
		Folder name of the file to be combined when linking		Required
	TMP	Working folder name to be used at linking	When linking	Required
At execution	BSORT_TMPDIR	Working folder name	When using the sort/merge function	Optional
			When using the NetCOBOL FILE UTILITY	Optional
	MEFTDIR	Printer information file	When using form descriptors	Optional
	TEMP	Working folder name	When using the sort/merge function	Required
			When using the NetCOBOL FILE UTILITY	Required
At remote development	COB_RDENV_X64	Batch file that sets user peculiarity at remote build (*2)	When setting a peculiar environment to the user at remote build	Optional

*1 : Use the library name specified in the COPY statement for the environment variable name. "XMDLIB" cannot be used for the library name. The environment variable name duplicated with the environment variable name to be supported in NetCOBOL cannot be used.

Only one folder can be specified for environment variable COB_LIBRARY-NAME.

Example of a COPY statement: COPY TEST OF TESTLIB1

```
COB_TESTLIB1=C:\COPYLIB
```

*2 : For details, refer to "22.2 Remote Development Support Function"

Environment variables for setting directories

Multiple directories can be specified for the following environment variables. If multiple directories are specified, the folder is searched in the following order.

- Environment variable FORMLIB
- Environment variable COB_COBCOPY
- Environment variable COB_REPIN

```
COB_COBCOPY=C:\COPY1;"C:\Program Files\NetCOBOL"
```

Note

If the folder name includes a space, the folder name must be specified within double quotation marks (").

Environment variables for setting the extension

Multiple extensions can be specified for the following environment variables. If multiple extensions are specified, the file is searched in the order listed below. If "None" is specified, the file without an extension is searched.

- Environment variable SMED_SUFFIX
- Environment variable COB_LIBSUFFIX

```
SMED_SUFFIX=SMD,None
```

- When environment variable SMED_SUFFIX is not specified, the retrieval is done in order of (1) extension (PMD), and (2) extension (SMD).

Note

When the encoding of a national data item is UTF-32, then a form descriptor is required for UTF-32. In this case, the file is searched after changing PMD to PMU and SMD to SMU. Other extensions are ignored. In regards to the creation of a form descriptor for UTF-32, refer to "I.4 CNVMED2UTF32 Command".

- When environment variable COB_LIBSUFFIX is not specified, the retrieval is done in order of (1) extension CBL, (2) extension COB, and (3) extension COBOL

Environment variable for setting compiler options

Specify the compile command options in environment variable COB_OPTIONS. For details on the compile command options, see "3.5.2 Command Options". For details on the order of compiler option specification, see "A.2 Compiler Option Specification Formats".

```
COB_OPTIONS=-WC,"OPTIMIZE,SRF(FREE)"
```

The following options cannot be specified in the COB_OPTIONS environment variable. If specified, they are considered invalid, and the compile process continues without them.

```
-I, -P, -R, -m, -dd, -do, -dp, -dr, -ds
```

1.2.2 Related Products

NetCOBOL supports the following products.

Name of Product	Function
FORM	Design windows or forms output by programs
FORM RTS	Process output of forms created by FORM

Name of Product	Function
PowerFORM	Design form layouts
PowerFORM RTS	Process output of forms
J Adapter Class Generator	A framework that enables COBOL to use Java classes
PowerBSORT	High performance SORT/MERGE tool

A brief overview of each product is given below.

Note

For United States users:

FORM is not available in the US market. They are provided in some other English-speaking markets for backward compatibility, so COBOL functions relating to these products are documented in this manual.

US users creating new NetCOBOL applications can gain the latest functionality from PowerFORM.

FORM

FORM designs forms to be displayed and printed by COBOL programs. Use FORM interactively to design the layout of and forms.

PowerFORM

PowerFORM is used in form to design the form which a COBOL program prints. The user can design the layout of a form using PowerFORM.

In PowerFORM, based on the style guide of each OS of Windows, a wizard function is equipped and the ease of use to a user is considered. Furthermore, the function in which the Windows function was employed efficiently to the utmost is offered by supporting full color bit map data.

FORM RTS

FORM RTS (sometimes referred to as MeFt) is used implicitly when a program that reads and writes FORM forms is executed. FORM RTS edits the format after receiving a form I/O request from the program.

PowerFORM RTS

PowerFORM RTS is implicitly used, when running the COBOL program which outputs a form. PowerFORM RTS performs format edit to the output demand of the form which a COBOL program publishes.

J Adapter Class Generator

The J adapter class generator is a tool that generates COBOL classes (adapter classes) that invoke Java classes. The Java class library can be used from COBOL by using the generated adapter classes.

PowerBSORT

The PowerBSORT is a high performance SORT/MERGE program oriented for business. A large amount of data can be sorted in a short time by an advanced sorting technology. It is a product only for the server that is designed to correspond to the various types of batch processing.

1.2.3 Resource List

The following table shows a resource list for this product.

Table 1.3 Resource List

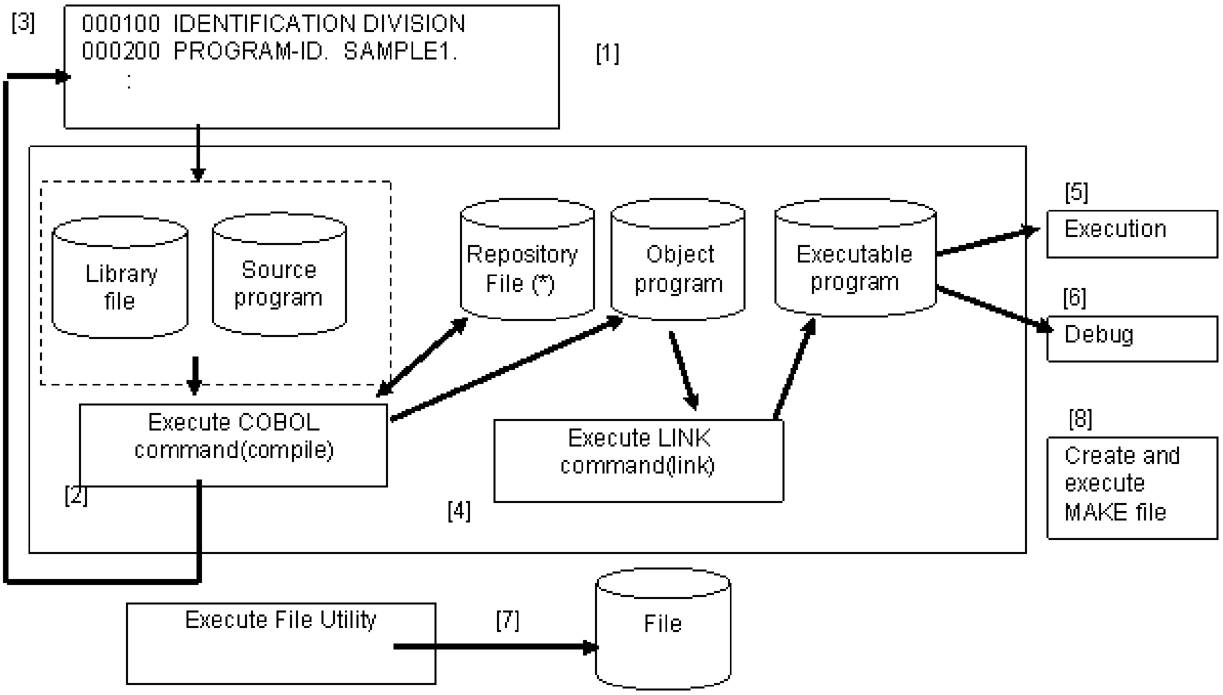
Resource name	Contents	Main Component	File name Rule to be named	Recom. Ext.
COBOL source	COBOL source program.	Compiler	Optional	cob cobol
COBOL library	COBOL library text.	Compiler	Optional	cbl
Form descriptor	Form definition information.	PowerFORM PowerFORM RTS	Optional	smd pmd
Form overlay	Form overlay pattern information.	PowerFORM PowerFORM RTS	Optional	ovd
Source analysis information file	Source analysis information.	Compiler	source-name.sai	sai
Repository	Related class information.	Compiler	Class name.rep	rep
Object	Object programs.	Compiler	Source name.obj	obj
Executable file	Executable program.	Compiler	Optional	exe
Compiler list	Compiler list information.	Compiler	Optional	lst
Text	Such things as COBOL line sequential files.	Runtime system	Optional	txt
Sequential	COBOL sequential files.	Runtime system	Optional	seq
Relativity	COBOL relative files.	Runtime system	Optional	rel
Index	COBOL indexed files.	Runtime system	Optional	idx
Initialized file for execution	Environment variable definitions set when COBOL is executed.	Runtime system	COBOL85.CBR (Optional)	CBR
Printing information	Printing format definitions such as printer types.	Runtime system	Optional	-
Font table	Font number definition used with printing files.	Runtime system	Optional	-
FCB	Definition of attributes such as number of lines per page, line spacing, and lines to be started.	Runtime system	Optional (4 characters)	-
Class information	Specifies such things as the number of object instances acquired when executed.	Runtime system	Optional	-
Trace information (the latest)	Execution path information output with trace function.	Runtime system	Execution format name.trc	trc
Trace information	Trace information of a generation ago.	Runtime system	Execution format name.tro	tro

Resource name	Contents	Main Component	File name Rule to be named	Recom. Ext.
(generation ago)				
COUNT information	Statistics information used with COUNT function.	Runtime system	Optional	-
Printer information	Printer information when forms are printed.	PowerFORM RTS	Optional	-
Debugging information	Debug information for the remote debug function.	Debugger	Source name.svd	svd

1.3 Developing a Program

An overview of the standard procedure for developing a program using NetCOBOL is described below.

Figure 1.2 Developing a program


[3]
000100 IDENTIFICATION DIVISION
000200 PROGRAM-ID. SAMPLE1.
:
[1]
Library file
Source program
Repository File (*)
Object program
Executable program
[5]
Execution
[6]
Debug
[8]
Create and execute MAKE file
Execute COBOL command(compile)
[2]
Execute LINK command(link)
[4]
Execute File Utility
[7]
File

* : Necessary for Developing an Object Oriented Program

Description of the above chart

1. Create the COBOL source program using an editor.
2. Execute COBOL to compile the program. Refer to "Chapter 3 Compiling Programs".
3. If a compile error occurs, debug the COBOL statement using the error jump function. See "Chapter 3 Compiling Programs".
4. Execute LINK to link the program. See "Chapter 4 Linking Programs".
5. Execute the COBOL application. See "Chapter 5 Executing Programs".
6. If the program does not operate as expected, debug the program using the debugging function of NetCOBOL Studio. Refer to the NetCOBOL Studio User's Guide.
7. To process a COBOL file, execute the file utility. See "6.8 COBOL File Utility".

8. For information about MAKE files, refer to section "15.9 MAKE file".

1.4 Remote Development

A COBOL application can be efficiently developed by remotely developing in NetCOBOL Studio for the "Client side" and NetCOBOL for the "Server side".

Client side

It is necessary to install the NetCOBOL product including NetCOBOL Studio. A COBOL application is developed using NetCOBOL Studio on the client side. NetCOBOL Studio connects with the server side, and does the development of the compilation etc. on the server side. NetCOBOL Studio displays the results.

Server side

It is necessary to install the NetCOBOL product.

Advantage of remote development

Many COBOL applications are operated on an expensive server machine. The following problems occur when these COBOL applications are developed on the same system.

- It is necessary to do development using the command line when the GUI environment is not prepared.
- It is necessary for two or more developers to share the server machine.

The Windows system is widely used as a personal terminal. Developers can occupy the GUI environment by using a Windows system.

In remote development, the above issues are solved by doing the development with the Windows system.

- It is possible to develop efficiently by using the GUI environment.
- The stress of the server machine is decreased by editing the source etc. on the client side.

Chapter 2 Creating and Editing a Program

This chapter explains how to create and edit a program, how to create library text and the program format, and describes compiler-directing statements.

2.1 Creating a Program

You create COBOL source programs and library text with editors. This section explains how to create the COBOL source program and library text.

2.1.1 Creating and Editing a COBOL Source Program

Simply, you create a COBOL source program by:

- Activating the Editor
- Creating and editing text
- Storing the program

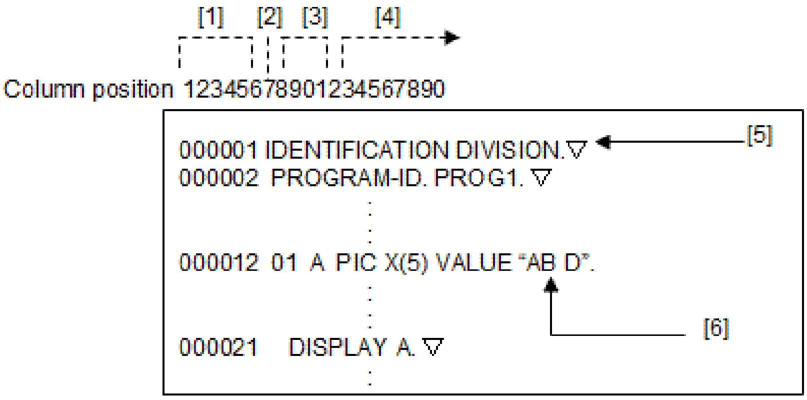
Activating the Editor

To display the Editor window, select Editor from the Tools menu.

Creating and Editing the Program

You format COBOL source programs using the COBOL reference format. Enter the line number, COBOL statement and procedure-name in the positions stipulated in the reference format.

Enter the line number and COBOL statements in the edit screen, as shown in the following example.



```
000001 IDENTIFICATION DIVISION.▽
000002  PROGRAM-ID. PROG1. ▽
            :
000012  01  A  PIC X(5) VALUE "AB D".
            :
000021    DISPLAY A. ▽
            :
```

- Sequence number area (columns 1 to 6)

  Write line numbers in the sequence number area. Line numbers can be omitted.

- Indicator area (column 7)

  Use the indicator area when continuing a line or to change a line to a comment or debug line. In all other cases, be sure to enter a space.

- Area A (columns 8 to 11)

  Normally, COBOL divisions, sections, paragraphs and end program headers are described in this area. Data items whose level-number is 77 or 01 are also described in this area.

- Area B (column 12 and on)

  Normally, COBOL statements and data items whose level-number is not 77 or 01 are described in this area.

- Line feed character (end of line)

  Enter a line feed character at the end of each line.

- TAB character

  A TAB character can be specified as the nonnumeric literal in a COBOL source program. The TAB character takes up 1 byte within the non-numeric literal.

Storing the Program and Quitting the Editor

After creating and editing the source program, store the program in a file, then Exit Editor from the File menu. Normally, you affix the extension COB or CBL to the name of the file.

Affixing the extension COB, CBL or COBOL facilitates specification of the file name when compiling the program.

2.1.2 Creating Library Text

You create library text, fetched by the COBOL source program with COPY statements, in the same manner as you create COBOL source programs. However, the reference format (fixed, variable or free) for the library text does not have to be the same as the format of the COBOL source program that fetches the library text.

Compiler options to set up the reference format for the COBOL source program will also be applied to any copied library text. However, when a single program fetches several library texts, the reference formats for all the library texts must be identical.

Normally, append the extension CBL, COB or COBOL to the name of the file in which the library text is to be stored.

The reference format for the library text is specified by a compiler option.

The extension can be set to an optional character string by specifying environment variable COB_LIBSUFFIX. See "1.2.1 Setting Environment Variables". Form can use any method of reference format.

2.1.3 Program Format

Each line of a NetCOBOL source program is delimited with a line feed character. When creating a NetCOBOL source program, the format of a line must comply with the rules specified in the reference format. There are three types of reference formats: Fixed, variable, and free. When using the fixed or free format, the user needs to specify it in the compiler option SRF. See "A.2.48 SRF(reference format type)".

Each format is described as given below. A line feed character in each line is not assumed a part of the line.

Fixed Format

In a fixed format, each line in the COBOL source program has a fixed length of 80 bytes.

(Column position)

1 6	7	8 11	12 72	73 80
Sequence number area	*	Area A	Area B	**

* Indicator area

** Program identification number area

Variable Format

In a variable format, each line in the COBOL source program can be up to 251 bytes long.

(Column position)

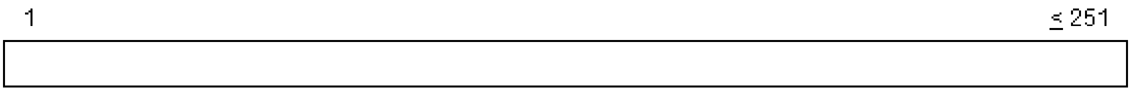
1 6	7	8 11	12 ≤ 251
Sequence number area	*	Area A	Area B

* Indicator area

Free Format

In a free format, you do not need to distinguish among the sequence number area, the indicator area, Area A, Area B, and the program identification number area. Each line can be up to 251 bytes long.

(Column position)



Refer to the sample program described as "Free format".

2.1.4 Compiler Directing Statements

A compiler directing statement indicates the compiler options. Normally, compiler options are specified to the compiler command, but the options can also be defined within the source program.

The description format for the compiler directing statement is shown below.

```
@OPTIONS [compiler-option [,compiler-option]...]
```

- Enter "@OPTIONS" starting at column 8.
- Enter at least one space between "@OPTIONS" and the compiler options.
- Each compiler option must be delimited by a comma.
  Spaces cannot be used as the separator between compiler options. Any options specified following a space are ignored.
- A compiler directing statement indicates the starting position of each compilation unit. The compiler options specified in the compiler directing statement apply only to the corresponding compilation unit.

Example

```
000100 @OPTIONS MAIN,APOST
000200 IDENTIFICATION DIVISION.
      *     :
```

Note

- Do not use a tab character as a separator in the @OPTIONS statement.
- The compiler option values that can be specified in the compiler directing statement are limited. For details, refer to "A.2 Compiler Option Specification Formats".

Chapter 3 Compiling Programs

This chapter describes followings:

- The resources and procedures required for compiling
- How to correct compile errors
- Compiling with compile commands

3.1 Compiling Sample Programs

We shall go through the steps for compiling the Sample 1 program. Each sample program is contained in SAMPLES folder under the NetCOBOL installation folder.

This section assumes the sample program is stored in:

```
C:\COBOL\SAMPLES\COBOL\SAMPLE01\SAMPLE1.COB
```

1. Command prompt is displayed.

   [START] menu -> [All Programs] -> [*NetCOBOL Product Name*] -> [NetCOBOL Command Prompt]

2. Compile the sample program using COBOL command.
3. Compilation of the sample program is now complete. Verify that the object program (SAMPLE1.OBJ) was created in the same folder (in this example C:\COBOL\SAMPLES\COBOL\SAMPLE01) in which the sample program was stored.

If compilation does not complete normally, the program may not have been installed properly. Verify that the program was installed properly.

3.2 Resources Necessary for Compilation

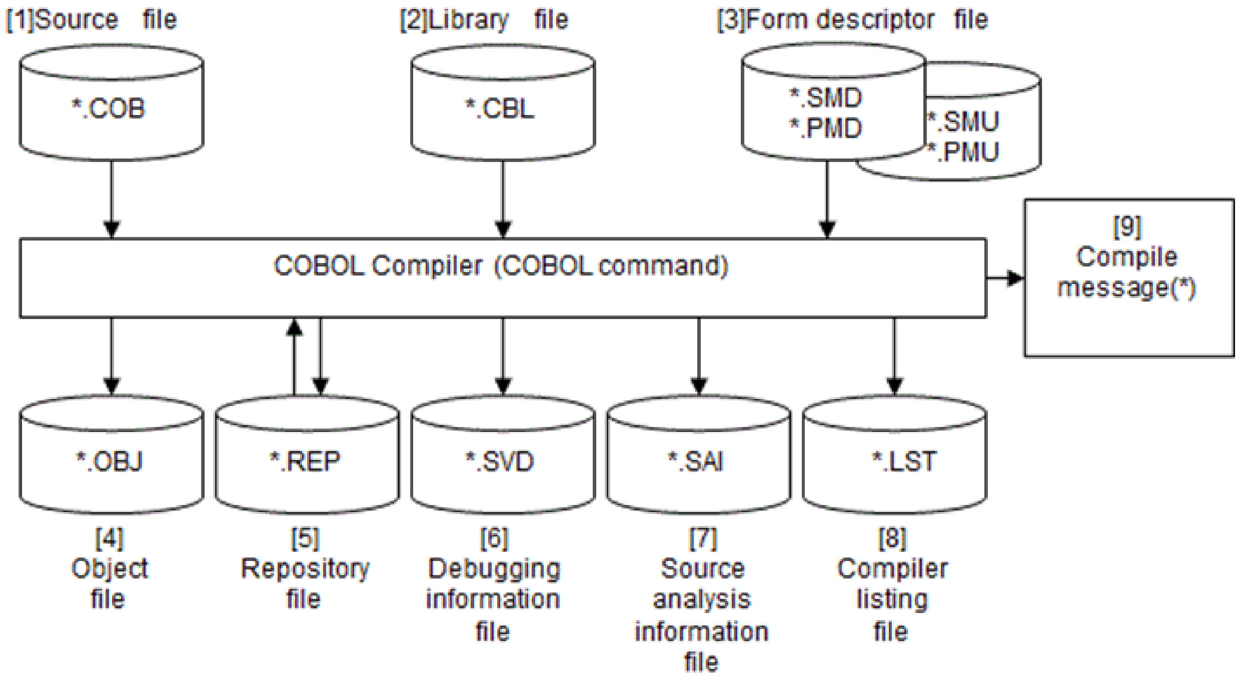
This section explains the files used by the NetCOBOL compiler and the information that it produces.

3.2.1 Files Used by the NetCOBOL Compiler

The NetCOBOL compiler uses or creates the following files:

- Source file (*. COB)
- Library file (*.CBL)
- Form Descriptor File (*. SMD/*. PMD/*.SMU/*.PMU)
- Object file (*. OBJ)
- Repository file (* REP)
- Debugging information file (*. SVD)
- Source analysis information file (*. SAI)
- Compiler listing file (*. LST)

Figure 3.1 The following figure shows the relationships of the files used by the COBOL compiler.



* : A compile message is displayed in the builder window when the compilation is terminated.

Table 3.1 The following table lists the files used by the NetCOBOL compiler.

	File Contents	File Name Format	I/O	Condition to Use or Create	Related Compiler Option	Related environment variable
1	Source file	Optional(In general, use the extension COB, CBL or COBOL) (*1)	I	Required	-	-
2	Library file	library-file-name.CBL(*2)	I	When compiling a source program that uses the COPY statement þ (*3)	-I	COB_COBCOPY COB_LIBSUFFIX COB_library-name
3	Form descriptors	Form-descriptor-name .SMD (*4) .PMD (*4)(*5) .SMU (*5)(*10) .PMU (*5)(*10)	I	When compiling a source program that uses form descriptors þ (*6)	-m	FORMLIB SMED_SUFFIX
4	Object file	source-file-name.OBJ	O	Created when compilation has completed successfully	OBJECT -do	-
5	Repository file (*7)	class-name.REP	I	When compiling a source program that has a REPOSITORY paragraph	-R -dr	COB_REPIN

	File Contents	File Name Format	I/O	Condition to Use or Create	Related Compiler Option	Related environment variable
6	Debugging information file	source-file-name.SVD	O	When compiler option TEST is specified	TEST -Dt	-
7	Source analysis information file(*8)	source-file-name.SAI	O	When compiler option SAI is specified	SAI -ds	-
8	Compiler listing file(*9)	source-file-name. LST	O	When compiler listings is output	-P -dp	-
9	Compile message	-		Displayed when compilation has completed	-	-

- (*1) Do not use the following extensions for the COBOL source file names.
  - OBJ
  - SVD
  - REP
  - SAI
  - LST
- (*2) Each extension can be changed to an optional character string by using compiler option LIBEXT or environment variable COB_LIBSUFFIX. If neither is specified, files are searched in the order of extension (1) CBL, (2) COB, and (3) COBOL. See "1.2.1 Setting Environment Variables".
- (*3) The search order of library files depends on the format of the COPY statement. This is described below.
  - COPY statement without specifying the library name
    1. The folder specified in compiler option -I is searched. If multiple folders are specified, files are searched in the specified order. See "3.5.2.10 -I(Specify the library file folder)".
    2. The folder specified in environment variable COB_COBCOPY is searched. If multiple folders are specified, files are searched in the specified order.
    3. The current folder is searched. For details on the current folder, see "A.2 Compiler Option Specification Formats".
  - COPY statement specifying the library name
    1. The folder related to the library name is searched. If the folder is not related to the library name, an error occurs.
    2. There is no specification in the Library Name dialog box, the folder specified in environment variable COB_LIBRARY-NAME is searched.
- (*4) Extension can be changed to a character string by using the environment variable SMED_SUFFIX. See"1.2.1 Setting Environment Variables".
- (*5) The extensions PMD, PMU, and SMU can be used only for the form descriptor.
- (*6) For details on the search order of the form descriptor, see "3.5.2.12 -m(Specify the FORM descriptor file folder)".
- (*7) For details on the repository file, see "15.6.1 Repository File".
- (*8) The source analysis information file is information to analyze the source.
- (*9) For the details on the compiler listing file, see "3.2.2 Information Provided by the NetCOBOL Compiler".
- (*10) The extensions SMU and PMU are used for the UTF-32 form descriptor. See "Using the encoding form UTF-32".

3.2.2 Information Provided by the NetCOBOL Compiler

The NetCOBOL compiler reports program compilation results as diagnostic messages. The diagnostic messages appear in the window when compilation terminates.

To store the messages in a file, specify compiler option -P before the compilation.

Specifying the compiler option MESSAGE creates lists indicating the compiler options and provides information on compiled programs. These lists are called compiler listings.

Specifying the compiler option MAP directs the compiler to provide data map listings, program control information listings and section size listings, and specifying the option LIST causes object program listings to be output. See the LIST and MAP options "Appendix A Compiler Options" and the section "Debugging Using Compiler Listings and Debugging Tools" of the "NetCOBOL Debugging Guide".

The formats of the compiler listings when compiler options -P and MESSAGE specified are shown below. See the compiler options "3.5.2.14 -P(Compile listing file name)" in this chapter and "A.2.26 MESSAGE(whether the optional information list and statistical information list should be output for separately compiled programs)".

3.2.2.1 Option Information Listing

The numbers in parentheses () correspond to the notes that follow these examples.

```
NetCOBOL Vxx.x.x                     THU AUG 15 13:35:50 2009 0001
         (5)                         (1)                      (2)
** OPTION SPECIFIED **
MAIN MESSAGE                                                   (3)
** OPTIONS ESTABLISHED **
ALPHAL(ALL)         NOEQUALS             NONAME    ...         (4)
                    THREAD(SINGLE)                             (4)
BINARY(WORD,MLBON)  FLAG(I)              NCW(STD)  ...         (4)
    :
```

(1) Indicates the compilation date and time.

(2) Indicates the page number.

(3) Indicates the compiler options specified by the user.

(4) Indicates a list of compiler options established by the NetCOBOL compiler.

(5) Indicates the version of this product.

3.2.2.2 Diagnostic Message Listing

```
NetCOBOL Vxx.x.x SOURCE NAME         THU AUG 15 13:35:50 2009 0002
                 (5)
** DIAGNOSTIC MESSAGE ** (SOURCE NAME)
                           (5)
JMN25031-S  4 USER WORD 'A' IS UNDEFINED.                      (6)
```

(5) Indicates the program, class or method name.

(6) Indicates the diagnostic message output by the NetCOBOL compiler. For details on the format of the diagnostic message, see the topic Compile Time Messages in NetCOBOL Messages.

3.2.2.3 Compile Unit Statistical Information Listing

```
NetCOBOL Vxx.x.x TEST01             THU AUG 15 13:35:50 2009 0003
** STATISTICS  **
  FILE NAME        = TEST.COB
  SOURCE NAME      = TEST01
  DATE AND TIME = THU AUG 15 13:35:50 2009
    SOURCE STATEMENT                  =     4 RECORDS          (7)
    PROGRAM SIZE (CODE)               =     0 BYTES            (8)
    CONTROL LEVEL                     =  0101 LEVEL            (9)
```

```
   CPU TIME                                 =  0.28 SECONDS        (10)
 HIGHEST SEVERITY CODE = S                                         (11)
```

(7) Indicates the number of records in the source program input by the NetCOBOL compiler. When library text has been fetched, the number of fetched records is included.

(8) Indicates the size of the object program. When the compilation operation was completed successfully and the object program has been output, the DATA size is also output.

(9) Indicates the level of the compiler used.

(10) Indicates the time required for compilation.

(11) Indicates the highest severity code among output diagnostic message codes.

When compiler option XREF is specified, the compiler provides a cross-reference list. See "A.2.57 XREF(whether a cross-reference list should be output)".

3.2.2.4 Cross Reference List

```
NetCOBOL Vxx.x.x  ARITH     THU AUG 15 13:35:50 2009 0001

DEFINITION      NAME    REFERENCE

  (12)        (13)      (14)
   1-1          A       8S 12R 15R 18R 24R
     2          ARITH
   1-2          B       10S 12R 15R 18R 21R 24R
     2#         BASE    2D 6D
   1-3          C       12S 13R 15S 16R 18S 19R 24S 25R
   1-4          D
```

Notes on the Cross Reference List:

(12) DEFINITION

Indicates the line number at which the name is defined in the following format:

```
[COPY qualification-value] line number [Mark for name definition in separate compilation]
```

- COPY qualification-value

  An identification number to be added to library text incorporated to the source program. It is allocated to COPY text in ascending order from 1 in increments of 1.

- Line number

  "*" indicates the name was defined implicitly.

- Mark for name definition in separate compilation

  "#" indicates the name is defined in a separate compilation.

(13) NAME

Indicates NAME defined in the source program. It is 30 characters of ANK or National character the size of the NAME area.

(14) REFERENCE

Indicates the line number that explicitly refers to the name and the reference type. The following symbols indicate each reference type:

- A : Parameter of CALL statement, INVOKE statement, and In-line invocation
- D : Reference by identification division, environment division and data division
- P : Reference by PERFORM statement

- R : Reference by procedure division

- S : Setting

3.3 Compiling a COBOL Source Program

The user can compile a COBOL source program using the compiler commands to compile it from a command line. For details, see "3.4 Using Commands to Compile".

Note

- A source program containing several compilation units within one file is compiled by specifying a compiler option NAME. See "A. 2.28 NAME(whether object files should be output for each separately compiled program)".

- It is possible to compile by specifying two or more source files for COBOL command. However, only the same compile option as two or more source files can be specified. Please describe compiler directing statement in COBOL source program when you want to specify a different compile option. See "2.1.4 Compiler Directing Statements".

3.4 Using Commands to Compile

COBOL source programs may be compiled by using the COBOL command.

3.4.1 COBOL Command

With the COBOL command, you can compile from the command prompt.

Specification Example and Output Format

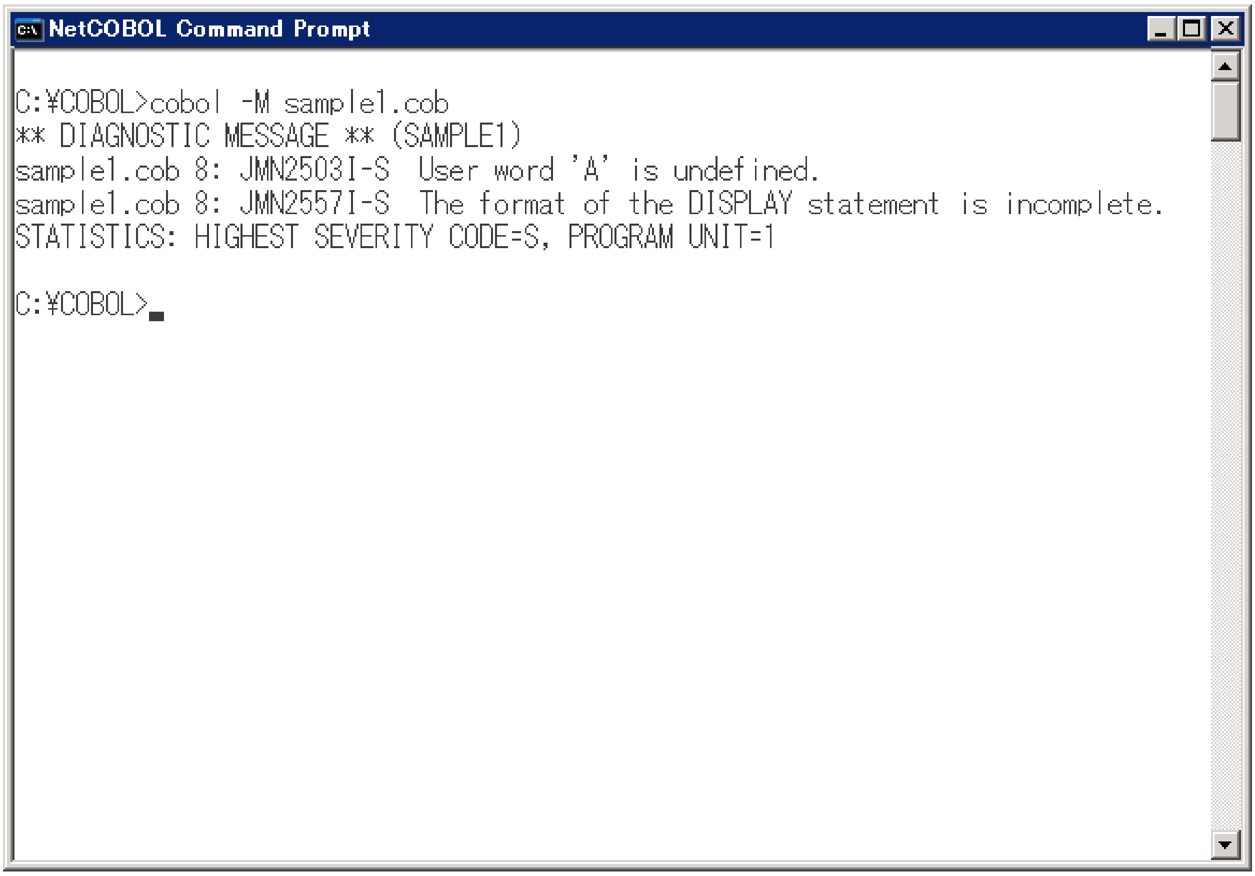
COBOL commands return compilation information such as the results of compilation and diagnostic messages to the command prompt screen. For details on the format of COBOL command and its available options, see "3.5 Command Format".

The following example shows how to compile with a COBOL command, and the output.

General Format

A compile end message is normally written if compiler option MESSAGE has not been specified. If a compile error occurs, the corresponding diagnostic message is generated.

Figure 3.2 The compile end message

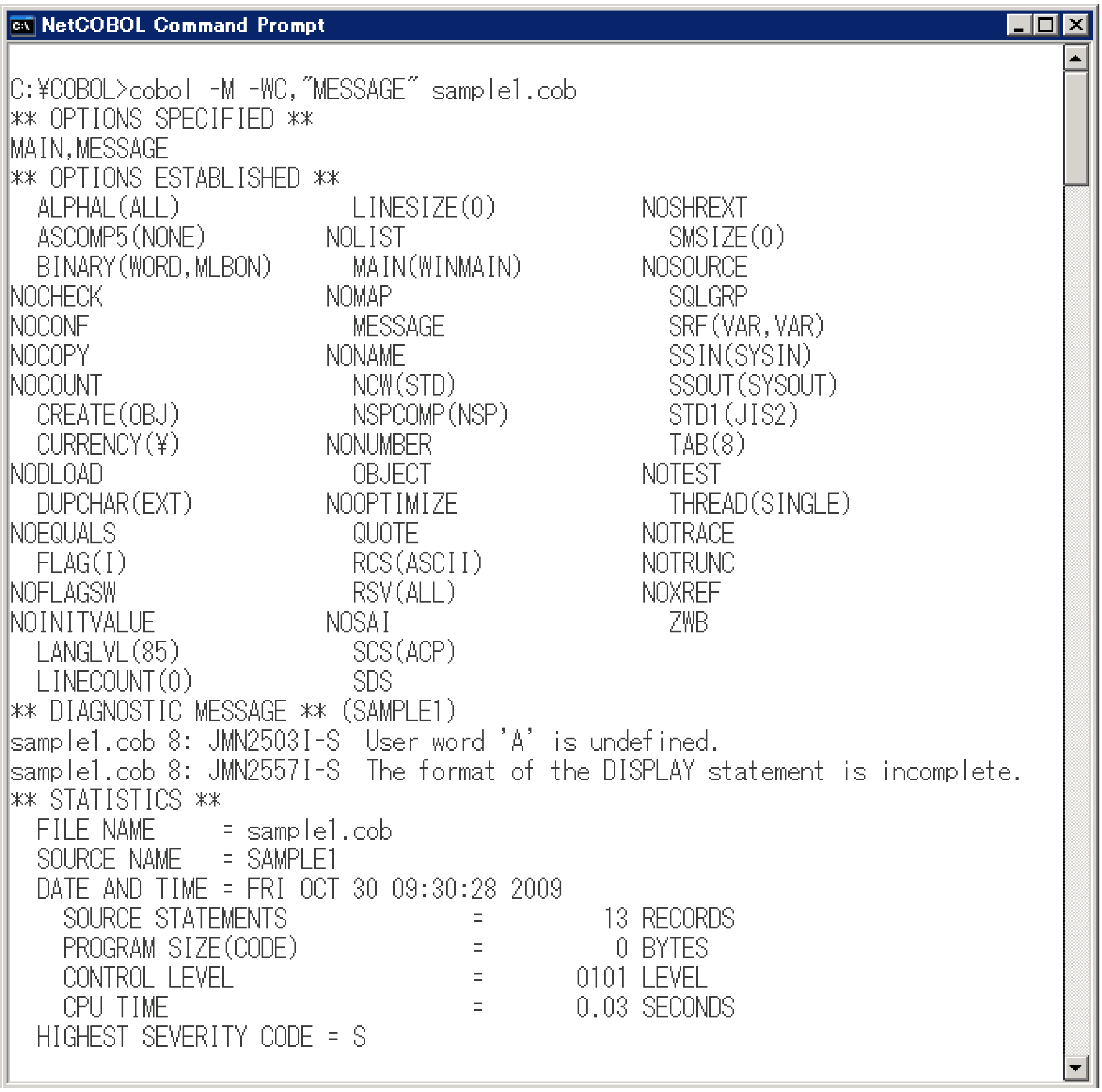


```
C:¥COBOL>cobol -M sample1.cob
** DIAGNOSTIC MESSAGE ** (SAMPLE1)
sample1.cob 8: JMN2503I-S  User word 'A' is undefined.
sample1.cob 8: JMN2557I-S  The format of the DISPLAY statement is incomplete.
STATISTICS: HIGHEST SEVERITY CODE=S, PROGRAM UNIT=1

C:¥COBOL>
```

Output Information

When the compiler option MESSAGE has been specified, the option information listing and compile unit statistical information listing are written. If a compile error occurs, the corresponding diagnostic message is written.

Figure 3.3 Listings when the compiler option MESSAGE is specified



```
C:¥COBOL>cobol -M -WC,"MESSAGE" sample1.cob
** OPTIONS SPECIFIED **
MAIN,MESSAGE
** OPTIONS ESTABLISHED **
  ALPHAL(ALL)             LINESIZE(0)           NOSHREXT
  ASCOMP5(NONE)         NOLIST                    SMSIZE(0)
  BINARY(WORD,MLBON)      MAIN(WINMAIN)         NOSOURCE
NOCHECK                 NOMAP                     SQLGRP
NOCONF                    MESSAGE                 SRF(VAR,VAR)
NOCOPY                  NONAME                    SSIN(SYSIN)
NOCOUNT                   NCW(STD)                SSOUT(SYSOUT)
  CREATE(OBJ)             NSPCOMP(NSP)            STD1(JIS2)
  CURRENCY(¥)           NONUMBER                  TAB(8)
NODLOAD                   OBJECT                NOTEST
  DUPCHAR(EXT)          NOOPTIMIZE                THREAD(SINGLE)
NOEQUALS                  QUOTE                 NOTRACE
  FLAG(I)                 RCS(ASCII)            NOTRUNC
NOFLAGSW                  RSV(ALL)              NOXREF
NOINITVALUE             NOSAI                     ZWB
  LANGLVL(85)             SCS(ACP)
  LINECOUNT(0)            SDS
** DIAGNOSTIC MESSAGE ** (SAMPLE1)
sample1.cob 8: JMN2503I-S  User word 'A' is undefined.
sample1.cob 8: JMN2557I-S  The format of the DISPLAY statement is incomplete.
** STATISTICS **
  FILE NAME     = sample1.cob
  SOURCE NAME   = SAMPLE1
  DATE AND TIME = FRI OCT 30 09:30:28 2009
    SOURCE STATEMENTS              =         13 RECORDS
    PROGRAM SIZE(CODE)             =          0 BYTES
    CONTROL LEVEL                  =       0101 LEVEL
    CPU TIME                       =       0.03 SECONDS
  HIGHEST SEVERITY CODE = S
```

Return Codes for COBOL Commands

The return codes for COBOL commands are set according to the highest severity code when the program is compiled. The relation between the highest severity code and the return code is shown below.

Highest Severity Code	Return Code
I	0
W	
E	1
S	2
U	3

3.5 Command Format

The compile commands compile a COBOL source program and create the object program. For details on files required to execute the compile commands and files to be created, see "3.2 Resources Necessary for Compilation". See "3.4 Using Commands to Compile" for information on how to specify the compile command and how to perform its basic operation.

3.5.1 COBOL Command

Input format

Command	Operand
COBOL	[List of options] File name ...

Descriptions of Operands

One or more spaces are required between the command name and operands. Portions enclosed in [brackets] can be omitted. In the descriptions to follow, an absolute path or relative path name can be specified for the folder and file names.

If the folder and file names contain following characters, the names must be enclosed in double quotation marks ("):

```
space  +  ,  ;  =  [  ]  (  )  `
```

Example

```
Specification Example:
  -I "c:\common\copy LIBRARIES\TEMPLATE"
```

List of options

Specifies command options. See the topic "3.5.2 Command Options".

For details on the priority order of compiler options, see "Compiler option specification methods and their priorities" in "A.2 Compiler Option Specification Formats".

File Name

Specifies the file name of the COBOL source to be compiled. Multiple file names can be specified.

3.5.2 Command Options

The following topics list the Command Options.

Options related to the compiler resources

-dr

Specification of input/output destination folder of the repository file

-I

Specification of library file folder

-m

Specification of form descriptor file folder

-R

Specification of repository file input destination folder

Options related to the COMPILE listing

-dp

Specification of compile list file folder

-P

Specification of various compile lists output and output destination

Options related to the source program compilation

-ds

Specification of source analysis information file folder

Options related to the object program creation

-do

Specification of the object file folder

-M

Specification for the main program compilation

-O

Specification for applying the wide area optimization

Options related to debugging functions during execution

-Dc

Specification for using COUNT function

-dd

Specification for the debugging information file folder

-Dk

Specification for using CHECK function

-Dr

Specification for using TRACE function

-Dt

Specification for using the debugging function and the COBOL Error Report

Others

-v

Specification for output information

-WC

Specification of compiler options

3.5.2.1 -Dc(Specify use of the COUNT function)

```
-Dc
```

When using the COUNT function, specify the -Dc option. For information on the COUNT function, see the topic "Using the COUNT Function" in the "NetCOBOL Debugging Guide".

Note

Specifying the -Dc option causes the process outputting the COUNT information to be incorporated into the object program, which degrades its execution performance. When debugging is finished, recompile without the -Dc option.

Information

The -Dc option is identical to the compiler option COUNT.

See

A.2.8 COUNT(whether the COUNT function should be used)

3.5.2.2 -dd(Specify the debugging information file folder)

```
-dd folder
```

The -dd option is effective when the -Dt option or compiler option TEST is specified.

If the -dd option is not specified, the debugging information file is output according to the output rule of the -Dt option or compiler option TEST.

See

3.5.2.9 -Dt(Create information for the debugging function and the COBOL Error Report)

A.2.53 TEST(whether the debugging function of NetCOBOL Studio and the COBOL Error Report should be used)

3.5.2.3 -Dk(Enable the CHECK function)

```
-Dk
```

When using CHECK function, specify -Dk option. For information on the CHECK function, see "Using the CHECK Function" in the "NetCOBOL Debugging Guide".

Note

The -Dk option causes code for inspecting subscripts, indexes and sub-references to be incorporated into the object program, consequently degrading the execution performance. When you finish debugging, recompile without the -Dk option.

Information

-Dk option is identical to the compiler option CHECK.

See

A.2.5 CHECK(whether the CHECK function should be used)

3.5.2.4 -do(Specify the object file folder)

```
-do folder
```

The -do option is effective when compiler option OBJECT is specified.

If the -do option is not specified, the object file is output according to the output rule of compiler option OBJECT.

See

A.2.33 OBJECT(whether an object program should be output)

3.5.2.5 -dp(Specify the compile list file folder)

```
-dp folder
```

The -dp option is effective when the -P option is specified.

When the -dp option is specified, the name of compile list file is as follows:

- The folder name is specified: "the folder name specified for -dp option" + "the file name specified for -P option"

```
cobol -Pout.lst -dpd:\test cobtest.cob
  ->  d:\test\out.lst
```

- The folder name is not specified: "the folder name specified for the -dp option" + "*source-file-name*.lst"

```
cobol -P -dpd:\test cobtest.cob
  ->  d:\test\cobtest.lst
```

If the -dp option is not specified, the compile list file is output according to the output rule of -P option.

See

3.5.2.14 -P(Compile listing file name)

3.5.2.6 -Dr(Enable the TRACE function)

```
-Dr
```

To use the TRACE function, specify -Dr option. For information on the TRACE function, see the topic "Using the TRACE Function" in the "NetCOBOL Debugging Guide".

Note

The -Dr option causes code for displaying trace information to be incorporated into the object program, consequently degrading the execution performance. When you finish debugging, recompile without the -Dr option.

Information

-Dr option is identical to the compiler option TRACE.

See

A.2.55 TRACE(whether the TRACE function should be used)

3.5.2.7 -dr(Specify the repository file folder)

```
-dr folder
```

The -dr option is effective when the class definitions are compiled.

If the -dr option is not specified, the repository file is output in the same folder as the source program..

The folder specified for the -dr option is used as an input destination folder of an external repository.

3.5.2.8 -ds(Specify the source analysis information file folder)

```
-ds folder
```

The -ds option is effective when compiler option SAI is specified.

If the -ds option is not specified, the source analysis information files are output according to the output rule of compiler option SAI.

See

A.2.41 SAI(whether the source analysis information file should be output)

3.5.2.9 -Dt(Create information for the debugging function and the COBOL Error Report)

```
-Dt
```

When the -Dt option is specified, the debugging information file used by the debugging function of NetCOBOL Studio and the COBOL Error Report is stored in the same folder as the source program. To change the folder, specify it using the -dd option.

For information on the debugging function and the COBOL Error Report, see the "NetCOBOL Studio User's Guide" and "NetCOBOL Debugging Guide".

Information

-Dt option is identical to the compiler option TEST.

See

3.5.2.2 -dd(Specify the debugging information file folder)

A.2.53 TEST(whether the debugging function of NetCOBOL Studio and the COBOL Error Report should be used)

3.5.2.10 -I(Specify the library file folder)

```
-I folder
```

When using COPY text in source programs, use the -I option to specify the folder where the COPY files are stored.

If the COPY files are in a number of folders, specify the -I option for each folder. Folders are searched in the order the -I options are specified.

Note

- If IN or OF is used in the COPY statement, the -I option is ignored.

Information

-I option is identical to the environment variable COB_COBCOPY. If two or more options are specified, folders are searched in the order:

1. Folder specified by the -I option.
2. Folder specified by the COB_COBCOPY environment variable.
3. Current folder.

3.5.2.11 -M(Indicate source file is a main program)

```
-M
```

When compiling a source program that is the main program in the execution unit, use -M option.

Information

-M option is identical to the compiler option MAIN.

See

A.2.24 MAIN(main program/sub-program specification)

3.5.2.12 -m(Specify the FORM descriptor file folder)

```
-m folder
```

When using the COPY text generated by FORM (using COPY descriptor-name IN/OF XMDLIB), use the -m option to specify the folder where the descriptor files are stored.

If the descriptor files are in a number of folders, specify the -m option for each folder. Folders are searched in the order the -m options are specified.

Note

- A peculiar character to Unicode cannot be specified for file name and folder name of the FORM descriptor.
- The length of the absolute path name of the FORM descriptor file must be 256 bytes or less in SHIFT-JIS

Information

-m option is identical to the environment variable FORMLIB.

If two or more options are specified, folders are searched in the order:

1. Folder specified by the -m option.
2. Folder specified by the FORMLIB environment variable.
3. Current folder.

3.5.2.13 -O(Compile for wide area optimization)

```
-O
```

To create an object program optimized for a wide area, specify the -O option. For information on wide area optimization, see "Appendix C Global Optimization".

Information

-O option is identical to the compiler option OPTIMIZE.

See

A.2.34 OPTIMIZE(global optimization handling)

3.5.2.14 -P(Compile listing file name)

```
-P file
```

Specify the name for the compile listing file.

The extensions COB, CBL, and COBOL cannot be used for the file names.

If no file name is specified, the compile listing file is output to the same folder as the source file.

Note

If the -dp option is also specified, the compile listing file name will be coupled with the folder specified in the -dp option. Consequently, an absolute path name cannot be specified in the -P option.

If omitting the file name of the -P option, the source file name cannot be specified just after the -P option. Specify one or more options other than the -P option, between the -P option and the source file name.

```
Incorrect: COBOL -P source-file-name
Correct  : COBOL -P -WC,"OBJECT" source-file-name
```

3.5.2.15 -R(Specify the repository file input folder)

```
-R folder
```

When using external repositories specified in the REPOSITORY paragraph, use the -R option to specify the folder where the repository files are stored.

If the repository files are in a number of folders, specify the -R option for each folder. Folders are searched in the order the -R options are specified.

Information

- -R option is identical to the environment variable COB_REPIN. If two or more options are specified, folders are searched in the order:

    1. Folder specified by the -R option.

    2. Folder specified by the COB_REPIN environment variable.

    3. Folder specified by the -dr option.

    4. Current folder.

3.5.2.16 -v(Specification for output information)

```
-v
```

Specify this option to output the following information to the standard error output:

- COBOL command version information

Information

If -v is specified without any other options - resource file names, object file names, etc., the COBOL command outputs only the version information and terminates normally. In this case, the return code of the COBOL command is 0. Refer to "3.4.1 COBOL Command" for detail.

3.5.2.17 -WC(Specify compiler options)

```
-WC,"compiler option"
```

Use -WC to specify the compiler options as described in "Appendix A Compiler Options".

Multiple compiler options can be specified. Each compiler option is delimited by a comma (,).

If a compiler option is specified several times, the last-specified setting of the option is the one that is used.

For details on the priority order of compiler options, see topic "Compiler Option specification methods and their priorities" in "A.2 Compiler Option Specification Formats".

Chapter 4 Linking Programs

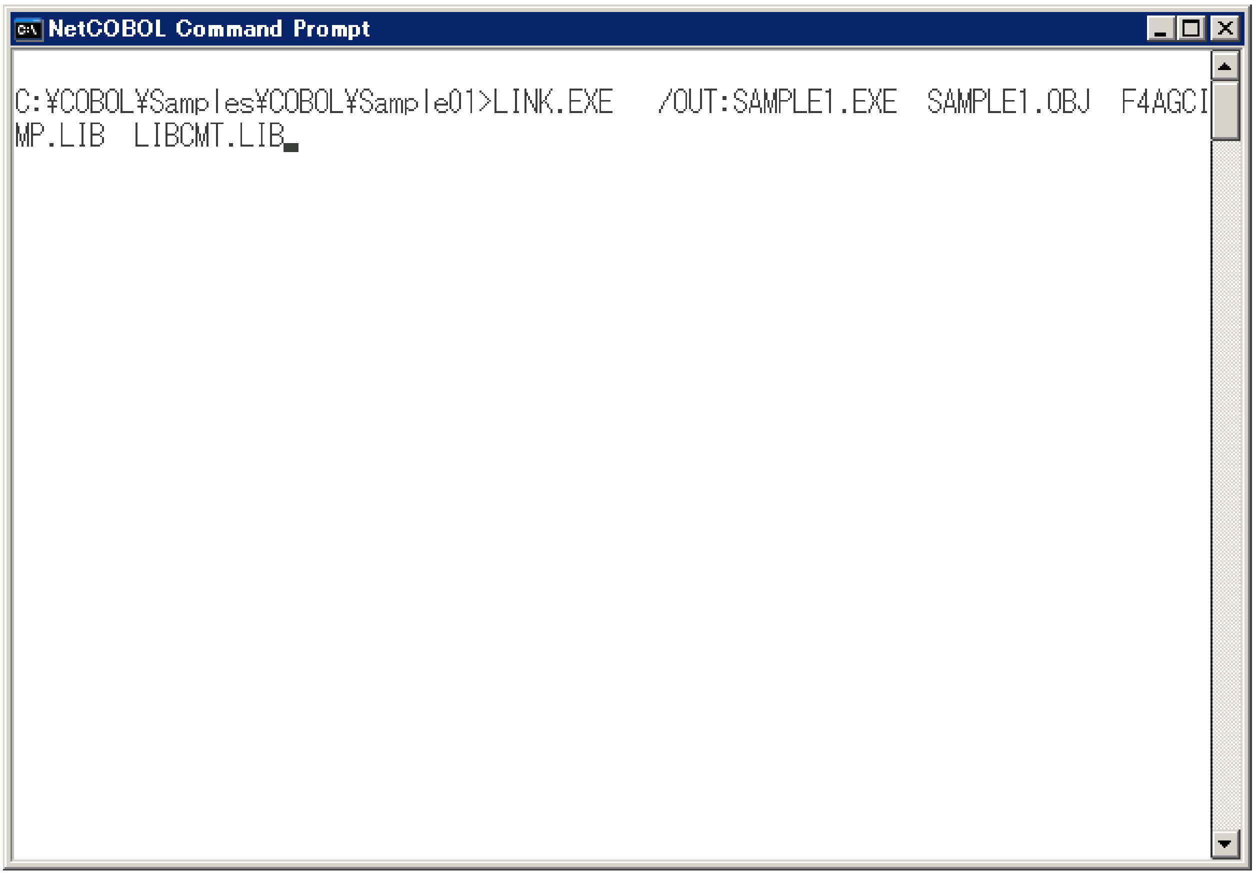
This chapter describes the resources required for linking, program structure, linkage procedures, how to use windows for linking, linking with link commands, and link messages.

4.1 Linking Sample Programs

Compilation creates an object program that must be linked before it can be executed.

This section shows how to use LINK command to link the Sample 1 program. This section assumes the file containing the object program to be:

```
C:\COBOL\SAMPLES\COBOL\SAMPLE01\SAMPLE1.OBJ
```



```
C:¥COBOL¥Samples¥COBOL¥Sample01>LINK.EXE  /OUT:SAMPLE1.EXE  SAMPLE1.OBJ  F4AGCI
MP.LIB  LIBCMT.LIB
```

The sample program is now linked. Make sure that the executable program (SAMPLE1.EXE) was created in the same folder (in this example, C:\COBOL\SAMPLES\COBOL\SAMPLE01) as where the sample program was stored.

4.2 Resources Required for Linking

Object programs compiled from COBOL source programs are linked using LINK command. This section briefly describes the files required for linking.
You must take into account the type of link required and the program structure. Refer to "4.3 Program Structure".

4.2.1 File that LINK command uses

To execute LINK command, the following files are used:

- Object file (*.OBJ)
- Library file (*.LIB)

- Import library file (*.LIB)
- Dynamic link library file (*.DLL)
- Executable file (*.EXE)
- Program Database (*.PDB)

The following table describes the files used by LINK command.

Table 4.1 Files used by LINK command

File Contents	File Name Format	I/O	Condition to Use or Create
Object file	Source-file-name. OBJ	I	Use the object program compiled from a COBOL source program
Standard library (Object code library)	Optional-name. LIB	I	Define when required to create a DLL or executable file
Import library	Optional-name. LIB	I	Specify to create an executable file having the dynamic link structure.
	DLL-name.LIB	O	Created automatically when DLL is created.
Dynamic link library (DLL)	Object-file-name.DLL or Optional-name. DLL	O	Created when linking has completed successfully.
Executable program	Object-file-name.EXE or Optional-name. EXE	O	Created when linking has completed successfully.
Program Database	Object-file-name.PDB or Optional-name. PDB	O	When link option /DEBUG is specified, it is made. COBOL Error Report, Debugging function of NetCOBOL Studio, and Dr. Watson Log and Visual C++ debugging function etc. uses this file.

Note

When linking, link option /DEBUG is recommended to be specified to facilitate the debugging of the application.

4.2.2 Executable Files

When an executable file is created by linking the object program, it becomes a Windows application.

Windows applications run in Windows(x64) system.

4.2.3 DLLs

DLLs are executable modules containing functions called to perform any processing from Windows applications. A DLL is linked with an application at run time, instead of during linking.

A DLL allows multiple applications to use the same library. DLLs are very efficient because multiple applications can share the same DLL.

Note

It is not possible to link by making the object program of 64 bits exist together to the object program of 32 bits. Refer to "4.4.3 Caution".

4.2.4 Import Library

The import library contains information used to set a dynamic link between an active application and a DLL during application execution. This library is required to create an executable file having a dynamic link structure.

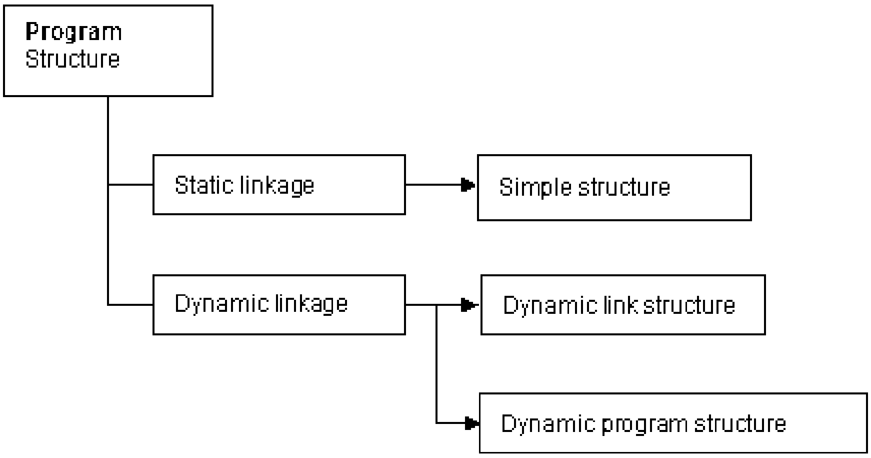
The system copies information indicating where a required library is stored from the import library to the executable program. In other words, the import library provides an interface between the application and the DLL.

4.3 Program Structure

This section explains the structure of the executable program created by linking, shown in the Figure "Program Structure". The following sections explain each structure.

The main program is the one that starts first and subprograms are those either invoked by the main program or invoked by other subprograms.

Figure 4.1 Program Structure



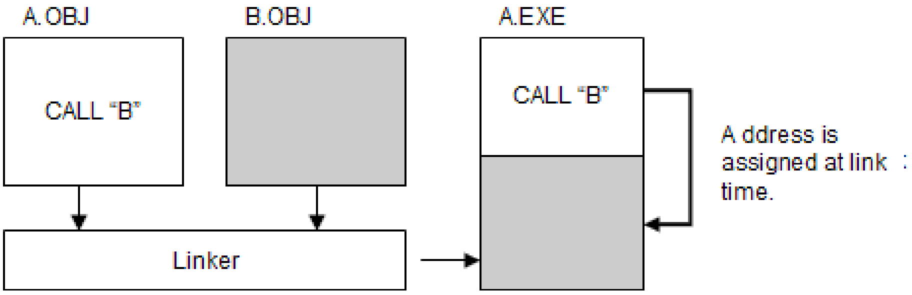
4.3.1 Static Linkage

When programs are statically linked, a group of modules is linked into a single executable file.

Simple Structure

The simple structure one in which all sub-programs are linked with the main program in a single executable file and all programs are loaded into memory at the start of execution. This structure gives highly efficient subprogram invocation.

To create an executable file of simple structure, all the subprograms are required at link time.



4.3.2 Dynamic Linkage

Dynamic linkage is where subprograms are linked into one or more DLL files that are invoked from the EXE file.

Dynamic linkage has the advantages that all executables that use the same DLL files use the same file on disk and the same code in memory. This saves on both disk space and memory space.

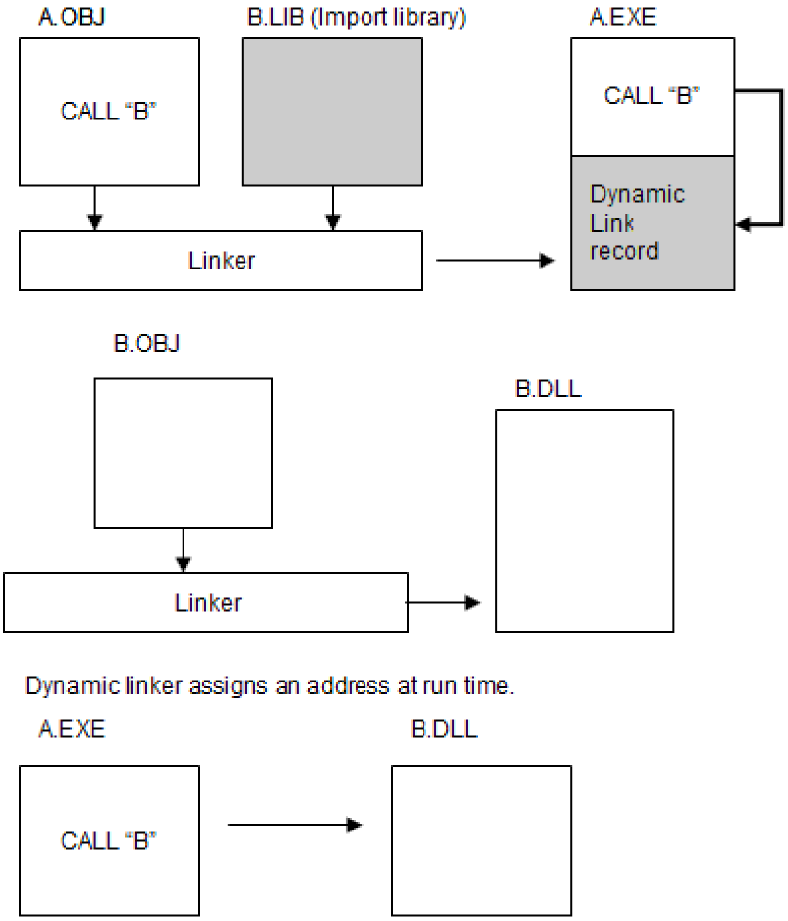
In addition, if you change a program within a DLL you only need to relink the DLL not the main executable.

Dynamic Link Structure

A dynamic link structure is an executable file created by linking an object program with the import libraries of subprograms invoked from that object program.

Unlike the simple structure, the executable program does not contain a subprogram in the dynamic link structure. When the main program is loaded, the subprogram is also loaded into virtual memory. The dynamic linker of the system performs loading by using the subprogram information created in the executable file at link time.

To create an executable file of the dynamic link structure, the import libraries of all the subprograms invoked by the program at link time are needed.



Dynamic Program Structure

Dynamic program structure refers to an executable file made by linking only the object program without using the import library of the subprogram. Unlike the dynamic link structure, the subprogram information is not included in the executable files.

For details of dynamic program structure, refer to "9.1.3 Dynamic Program Structure".

4.3.3 Program Structure and CALL Statement/Compiler Options

The program structure is determined by the format of the CALL statement, and options specified when compiling. Table below lists the relationship among the program structure, CALL statement, and compiler options. For information on the compiler DLOAD option, see "A.2.11 DLOAD(program structure specification)".

Table 4.2 Relationship among program structure, CALL statement, and compiler options

Program Structure	Call Statement	Compiler Option
Simple structure	CALL "program-name"	NODLOAD
Dynamic link structure	CALL "program-name"	NODLOAD
Dynamic program structure	CALL data-name	________
	CALL "program-name" CALL data-name (coexisting)	DLOAD
	CALL "program-name"	DLOAD

4.3.4 Relation between the program structure and CANCEL statement

The CANCEL statement causes any later CALL to reinitialize the called program. When using the CANCEL statement, however, note that the CANCEL statement may not be enabled depending on the program structure.

The relation between the program structure and the CANCEL statement are listed below.

Table 4.3 Relation between the program structure and CANCEL statement

Program structure		CANCEL statement
External program	Simple structure	Disabled
	Dynamic link structure	Disabled
	Dynamic program structure	Enabled
Internal program		Enabled

For details on internal programs, see "9.2.7 Internal Programs".

4.4 Using Link Commands

Object programs can also be linked with commands. This section describes link operation with commands.

4.4.1 LINK Command

The LINK command performs the link operation.

For details on the link command format, refer to "4.5.1 LINK Command Format".

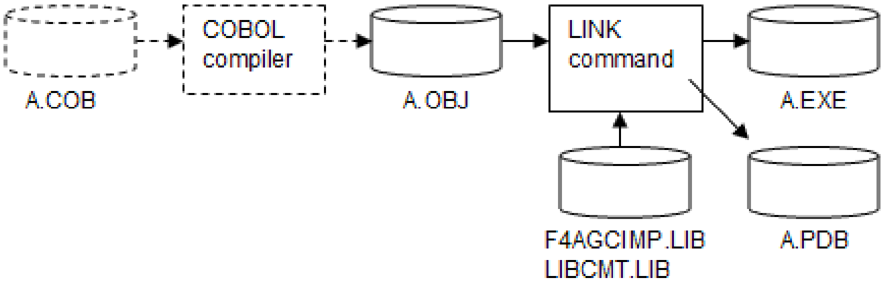
- When executing the LINK command, the following libraries must be specified:
  - F4AGCIMP.LIB
  - LIBCMT.LIB (*)

* : When executable program (EXE) is made, or when C function is called from DLL, LIBCMT.LIB is specified..

4.4.2 Examples of Using the LINK command

Some examples of using the LINK command are shown here.

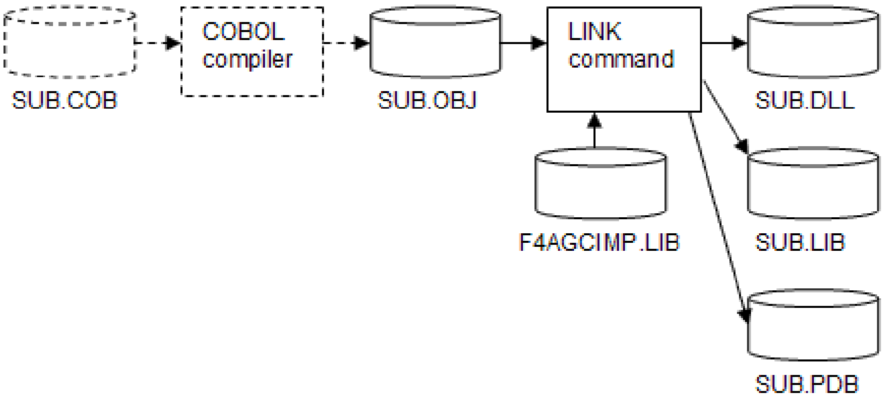
4.4.2.1 When Linking an Object Program



```
LINK A.OBJ F4AGCIMP.LIB LIBCMT.LIB /DEBUG /OUT:A.EXE
```

- /DEBUG : Specifies when creating the debug information.
- /OUT : Specifies the main output file name.

4.4.2.2 When Creating a DLL

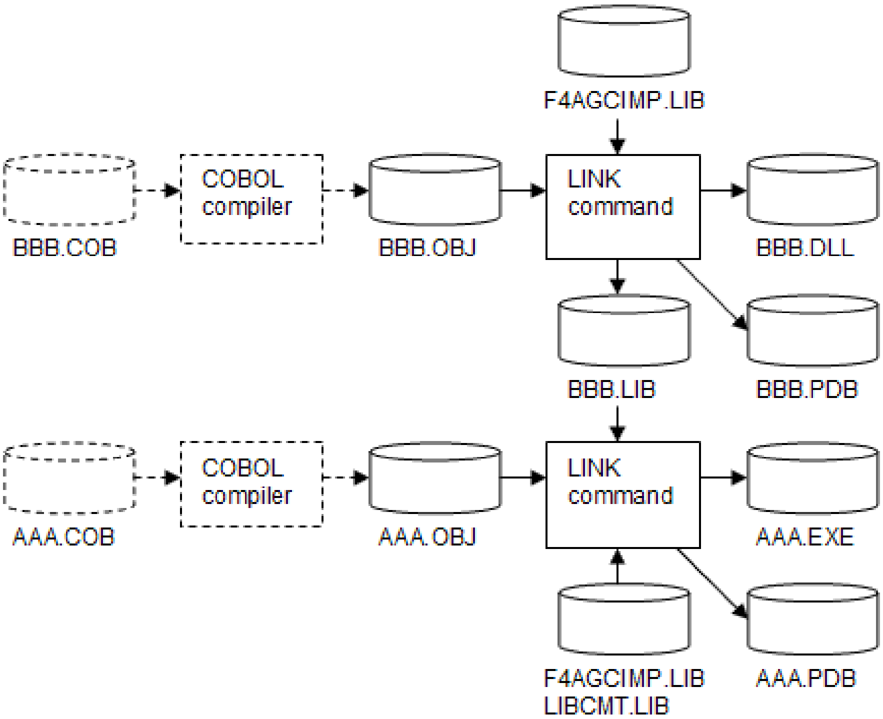


```
LINK SUB.OBJ F4AGCIMP.LIB /DLL /NOENTRY /DEBUG /OUT:SUB.DLL
```

- /DLL : Specifies when creating a dynamic link library (DLL).
- /NOENTRY : Specifies when creating DLL.
- /DEBUG : Specifies when creating the debug information.
- /OUT: Specifies the main output file name.

Only when C function is not called from DLL. When C function is called from DLL, Not /NOENTRY but LIBCMT.LIB is specified.

4.4.2.3 When Creating an Executable Program with a Dynamic Link Structure



To create a DLL and import library :

```
LINK BBB.OBJ F4AGCIMP.LIB /DLL /NOENTRY /DEBUG /OUT:BBB.DLL
```

- /DLL : Specifies when creating a dynamic link library (DLL).
- /NOENTRY : Specifies when creating DLL.
- /DEBUG : Specifies when creating the debug information.
- /OUT: Specifies the main output file name.

Only when C function is not called from DLL. When C function is called from DLL, Not /NOENTRY but LIBCMT.LIB is specified.

To create an executable program :

```
LINK AAA.OBJ F4AGCIMP.LIB LIBCMT.LIB BBB.LIB /DEBUG /OUT:AAA.EXE
```

- /DEBUG : Specifies when creating the debug information.
- /OUT : Specify the main output file name.

4.4.3 Caution

It is not possible to link by making the object program of 64 bits exist together to the object program of 32 bits.

When linking, the following link messages are output when existing together.

Simple structure

Link message LNK1112 is output.

- a.obj : the object program of 64 bits
- b.obj : the object program of 32 bits

Example

```
link /out:a.exe a.obj b.obj f4agcimp.lib libcmt.lib
------------------------------------------------------------------------------
b.obj : fatal error LNK1112: module machine type 'X86' conflicts with target machine type 'x64'
```

Dynamic link structure

Link message LNK2001 is output.

- a.obj : the object program of 64 bits
- b.obj : the object program of 32 bits

Example

```
link /out:a.exe a.obj b.lib f4agcimp.lib libcmt.lib
------------------------------------------------------------------------------
a.obj : error LNK2001: unresolved external symbol B
```

4.5 LINK Command Format

LINK command creates executable programs by linking the object programs.

For the details of the files required to run the LINK command, refer to "4.2 Resources Required for Linking." For the location to specify the LINK command to and how to use it, refer to "4.4 Using Link Commands."

4.5.1 LINK Command Format

This section describes the format of the LINK command.

Input format

Command	Operand
LINK	File name list [Option list]

Descriptions of operands

One or more spaces are required between the command name and each operand. Portions enclosed in [brackets] may be omitted. In the following description, an absolute path name or relative path name may be specified for a folder and file name. To specify a file name including a space, the file name must be specified within double quotation marks (").

File name list

The following shows the format for specification.

```
[Object-file-name]...   [Library-file-name]...
```

- Object-file-name: Specify one or more object file names to link.
- Library-file-name: Specify one or more standard (object code) libraries, import libraries or export files. If a standard object library is specified, only object module(s) necessary for resolving the external reference will be linked.
    - Always specify the following files:
        - COBOL import library
    - Please specify the following files when the executable program is made and when DLL that calls C function is made.:
        - LIBCMT.LIB

Option list

Specify the LINK command options. For the content to specify, refer to Table below.

Table 4.4 LINK Command Option

Content	Format of specification
Specify when creating the debug information. Debug information is created in Program Database (PDB).	/DEBUG (*1)
Specify to create dynamic link library (DLL).	/DLL
Specify to create dynamic link library (DLL). (Only when C function is not called from DLL)	/NOENTRY
Specify to create external reference information.	/EXPORT:External reference name
Specify the main output file name.	/OUT:filename
Specify to change the stack size.	/STACK:Stack size (*2)
Specify to filename.(When the map file is generated)	/MAP:filename

- *1 : For details about debugging method using program database (PDB), refer to "NetCOBOL Studio User's Guide" and "NetCOBOL Debugging Guide".
- *2 : If the STACK option is omitted, the default stack size will be 4MB. Always specify the stack size in bytes. Refer to "5.8 Cautions".

4.5.2 Linking Import Libraries

The LIB command can incorporate multiple import libraries into one. Incorporating the import libraries is especially effective in the following case.

Example 1: Invoking multiple DLLs in the dynamic link structure

[Program A]

```
IDENTIFICATION DIVISION.
PROGRAM-ID. A.
*>  :
PROCEDURE DIVISION.
   CALL "SUB1".
   CALL "SUB2".
   CALL "SUB3".
```

1. Creating SUB1.DLL

```
LINK SUB1.OBJ F4AGCIMP.LIB /DLL /NOENTRY /DEBUG /OUT:SUB1.DLL
```

   - SUB1.OBJ : The object file of the SUB1 program
   - F4AGCIMP.LIB : The import library of the COBOL runtime system

   When linking, SUB1.DLL and the import library, SUB1.LIB are created.

2. Creating SUB2.DLL

```
LINK SUB2.OBJ F4AGCIMP.LIB /DLL /NOENTRY /DEBUG /OUT:SUB2.DLL
```

   - SUB2.OBJ : The object file of the SUB2 program
   - F4AGCIMP.LIB : The import library of the COBOL runtime system

   When linking, SUB2.DLL and the import library, SUB2.LIB are created.

3. Creating SUB3.DLL

```
LINK SUB3.OBJ F4AGCIMP.LIB /DLL /NOENTRY /DEBUG /OUT:SUB3.DLL
```

- SUB3.OBJ : The object file of the SUB3 program
- F4AGCIMP.LIB : The import library of the COBOL runtime system

When linking, SUB3.DLL and the import library, SUB3.LIB are created.

4. Linking import libraries of SUB1.DLL, SUB2.DLL, and SUB3.DLL

```
LIB SUB1.LIB SUB2.LIB SUB3.LIB /OUT:SUB123.LIB
```

- SUB1.LIB : The import library of SUB1.DLL
- SUB2.LIB : The import library of SUB2.DLL
- SUB3.LIB : The import library of SUB3.DLL

When executing the command, SUB1.LIB, SUB2.LIB, and SUB3.LIB are linked to a new import library file, SUB123.LIB.

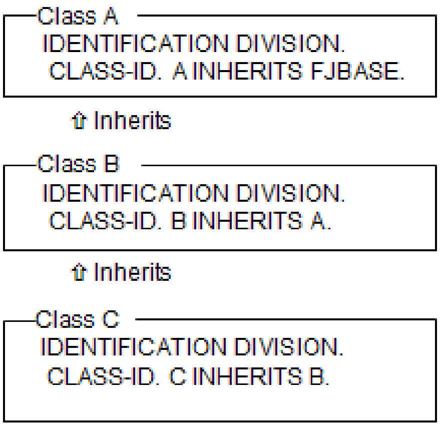
Using the created import library, link program A.

5. Linking program A

```
LINK A.OBJ F4AGCIMP.LIB LIBCMT.LIB SUB123.LIB /DEBUG /OUT:A.EXE
```

- A.OBJ : The object file of program A
- F4AGCIMP.LIB : The import library of the COBOL runtime system
- LIBCMT.LIB : The C runtime library
- SUB123.LIB : The file incorporating SUB1.DLL, SUB2.DLL, and SUB3.DLL

Example 2: When the inheritance of classes has more than one level



Class A

```
IDENTIFICATION DIVISION.
 CLASS-ID.  A INHERITS FJBASE.
```

⇧ Inherits

Class B

```
IDENTIFICATION DIVISION.
 CLASS-ID. B INHERITS A.
```

⇧ Inherits

Class C

```
IDENTIFICATION DIVISION.
 CLASS-ID. C INHERITS B.
```

1. Creating a DLL of class A

```
LINK A.OBJ F4AGCIMP.LIB /DLL /NOENTRY /DEBUG /OUT:A.DLL
```

- A.OBJ : The object file of class A
- F4AGCIMP.LIB : The import library of the COBOL runtime system

When linking, A.DLL and the import library, A.LIB is created.

2. Creating a DLL of class B

```
LINK B.OBJ F4AGCIMP.LIB A.LIB /DLL /NOENTRY /DEBUG /OUT:B.DLL
```

- B.OBJ : The object file of class B
- F4AGCIMP.LIB : The import library of the COBOL runtime system
- A.LIB : The import library of class A

When linking, B.DLL and the import library, B.LIB is created.

When creating a DLL of class C, the import libraries of class A and B are needed. Therefore, build the import libraries of classes A and B.

```
LIB A.LIB B.LIB /OUT:AB.LIB
```

- A.LIB : The import library of class A
- B.LIB : The import library of class B

When executing the command, the import library, AB.LIB linking the import libraries of class A and B is created.

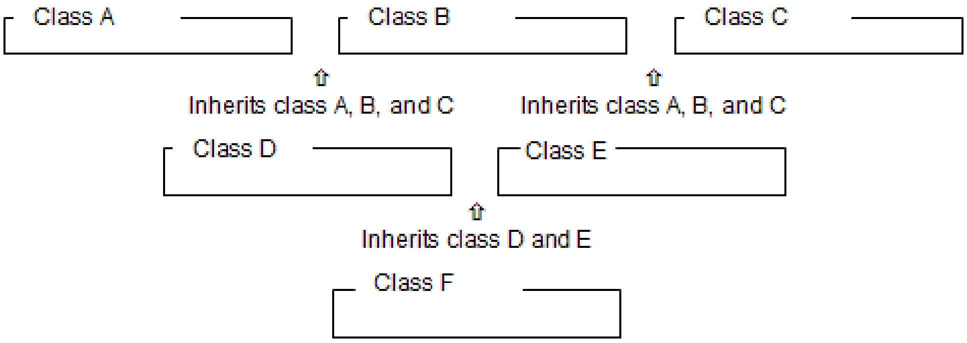
3. Creating a DLL of class C

```
LINK C.OBJ F4AGCIMP.LIB AB.LIB /DLL /NOENTRY /DEBUG /OUT:C.DLL
```

- C.OBJ : The object file of class C
- F4AGCIMP.LIB : The import library of the COBOL runtime system
- AB.LIB : The import library linking the import libraries of class A and B

When linking, C.DLL and the import library, C.LIB is created.

Example 3: When the inheritance of classes is multiplexed



1. Creating a DLL of class A

```
LINK A.OBJ F4AGCIMP.LIB LIBC.LIB /DLL /OUT:A.DLL
```

- A.OBJ : The object file of class A
- F4AGCIMP.LIB : The import library of the COBOL runtime system

When linking A.DLL and the import library, A.LIB is created.

2. Creating a DLL of class B

```
LINK B.OBJ F4AGCIMP.LIB LIBC.LIB /DLL /OUT:B.DLL
```

- B.OBJ : The object file of class B
- F4AGCIMP.LIB : The import library of the COBOL runtime system

When linking, B.DLL and the import library, B.LIB is created.

3. Creating a DLL of class C

```
LINK C.OBJ F4AGCIMP.LIB LIBC.LIB /DLL /OUT:C.DLL
```

- C.OBJ : The object file of class C
- F4AGCIMP.LIB : The import library of the COBOL runtime system

When linking, C.DLL and the import library, C.LIB is created.

4. Creating a DLL of class D

   When creating a DLL of class D, the import libraries of class A, B, and C are needed. Therefore, linked the import libraries of class A, B, and C.

   ```
   LIB A.LIB B.LIB C.LIB /OUT:ABC.LIB
   ```

   When executing the command, the import library, ABC.LIB incorporating the import libraries of class A, B, and C is created.

   ```
   LINK D.OBJ F4AGCIMP.LIB LIBC.LIB ABC.LIB /DLL /OUT:D.DLL
   ```

   - D.OBJ : The object file of class D
   - F4AGCIMP.LIB : The import library of the COBOL runtime system
   - ABC.LIB : The import library incorporating import libraries of class A, B, and C

   When linking, D.DLL and the import library, D.LIB is created.

5. Creating a DLL of class E

   When creating a DLL of class E, the import libraries of class A, B, and C are needed. Therefore, specify the import library used to create class D.

   ```
   LINK E.OBJ F4AGCIMP.LIB LIBC.LIB ABC.LIB /DLL /OUT:E.DLL
   ```

   - E.OBJ : The object file of class E
   - F4AGCIMP.LIB : The import library of the COBOL runtime system
   - ABC.LIB : The import library incorporating the import libraries of class A, B, and C

   When linking, E.DLL and the import library, E.LIB is created.

6. Creating a DLL of class F

   When creating a DLL of class F, the import libraries of class A, B, C, D, and E are needed. Therefore, linked the import libraries of class A, B, C, D, and E.

   ```
   LIB A.LIB B.LIB C.LIB D.LIB E.LIB /OUT:ABCDE.LIB
   ```

   When executing the command, the import library, ABCDE.LIB incorporating the import libraries of class A, B, C, D, and E is created.

   ```
   LINK F.OBJ F4AGCIMP.LIB LIBC.LIB ABCDE.LIB /DLL /OUT:F.DLL
   ```

   - F.OBJ : The object file of class F
   - F4AGICIMP.LIB : The import library of the COBOL runtime system
   - ABCDE.LIB : The import library incorporating the import libraries of class A, B, C, D, and E

Note

When building import libraries, the message LNK4006 might be displayed. The created import library, however, does not have any problem.

4.6 Linker Messages

The following describe linker messages that may be displayed when linking NetCOBOL programs.

LNK1104

Explanation

There is not enough space on the disk or root folder.

Operator response

Delete files to make space.

LNK1123

Explanation

An attempt was made to link Microsoft Windows operating system Version 3.1 objects with the 32 bit linker.

Operator response

Use 32 bit Objects.

LNK1561

Explanation

An attempt was made to create an executable program (EXE) by compiling the main program without the compiler option MAIN.

Operator response

Specify the compiler option MAIN before compiling the main program.

LNK2001

Explanation

- No internal name of the export routine is defined in the EXPORTS definitions in the module definition file.
- No internal name of the export routine is set in the library or object file.

Operator response

Make sure that the export name in the EXPORTS statement of the module definition file is set in the library or object file.

Use the LINK command /DUMP option to confirm that both invoking side and invoked side of the program have the same export name. Refer to "I.1 LINK command /DUMP option" on how to use it.

Export names of the library and the object file are determined depending on the program structure or invoking method. Refer to "9.1 Outline of Calling Relationships" for a specific program structure.

LNK2005

Explanation

Tried to make a single executable file from two or more programs compiled with the MAIN option.

Operator response

Specify only the MAIN compiler option for the main program.

LNK4006

Explanation

- EXPORTS is duplicated in the module definition file.
- The internal name of existent data was found in the library or object file.

Operator response

Make sure that the export name in the EXPORTS statement of the module definition file is duplicated in the library or object file.

Note

When COBOL programs are linked, it is not necessary to file the module definition usually. Please confirm the content of the file when you use the module definition file. When being defined in the file is only EXPORTS, the module definition file is recommended to be deleted.

Please refer to help of the linker of Microsoft company for details of the module definition file.

Chapter 5 Executing Programs

This chapter describes the procedures for executing programs, setting runtime environment information, and operating windows related to executing programs.

5.1 Executing a Sample Program

After linking, try executing the created executable file.

To begin with, we shall go through the steps for executing the Sample 1 program using a command prompt window. Throughout this section, assume the executable file to be:

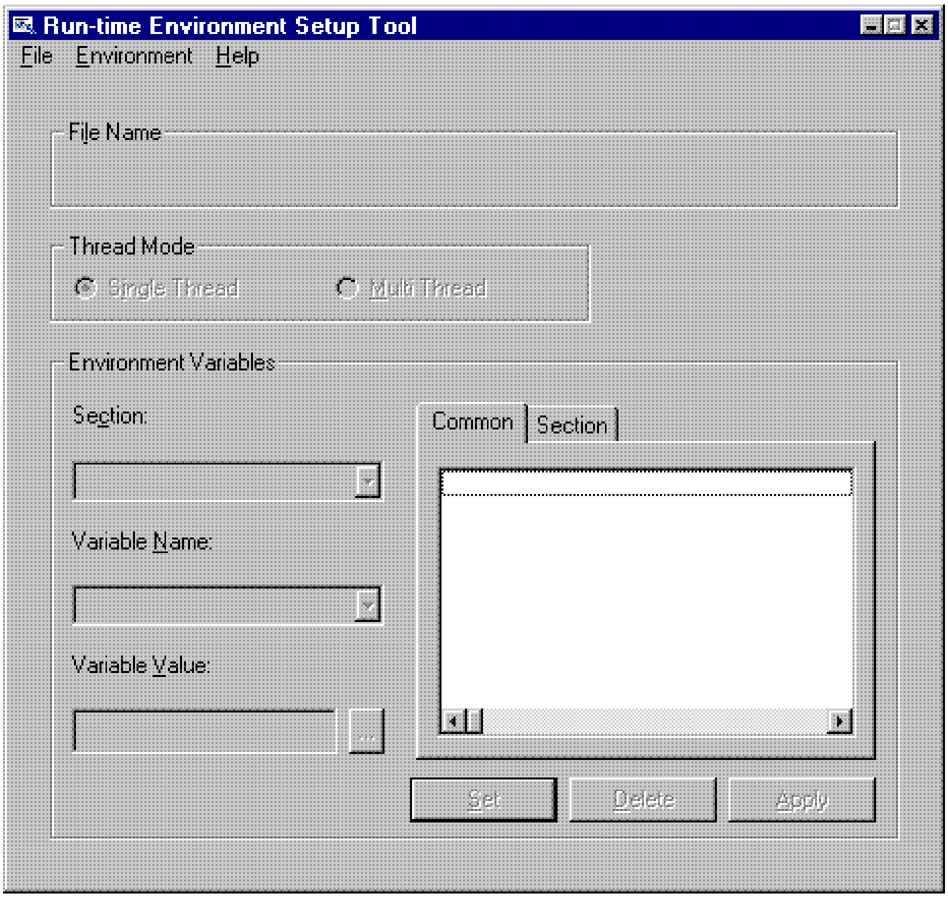
```
C:\COBOL\SAMPLES\COBOL\SAMPLE01\SAMPLE1.EXE.
```

No runtime environment information needs to be set for Sample 1. However, if you want to set the runtime environment information, follow the steps given below before executing the program.

1. Start "Runtime Environment Setup Tool".

   The Runtime Environment Setup Tool dialog appears.

   Figure 5.1 Runtime Environment Setup Tool dialog

2. Select Open from the Files menu.

   The runtime initialization file specification dialog appears.

3. Create the runtime initialization file (COBOL85.CBR) in the folder (C:\COBOL\SAMPLES\COBOL\SAMPLE01) that contains the EXE file.

4. Set the necessary runtime environment information and apply the information.

   The contents of the runtime initialization file (C:\COBOL\SAMPLES\COBOL\SAMPLE01\COBOL85.CBR) change.

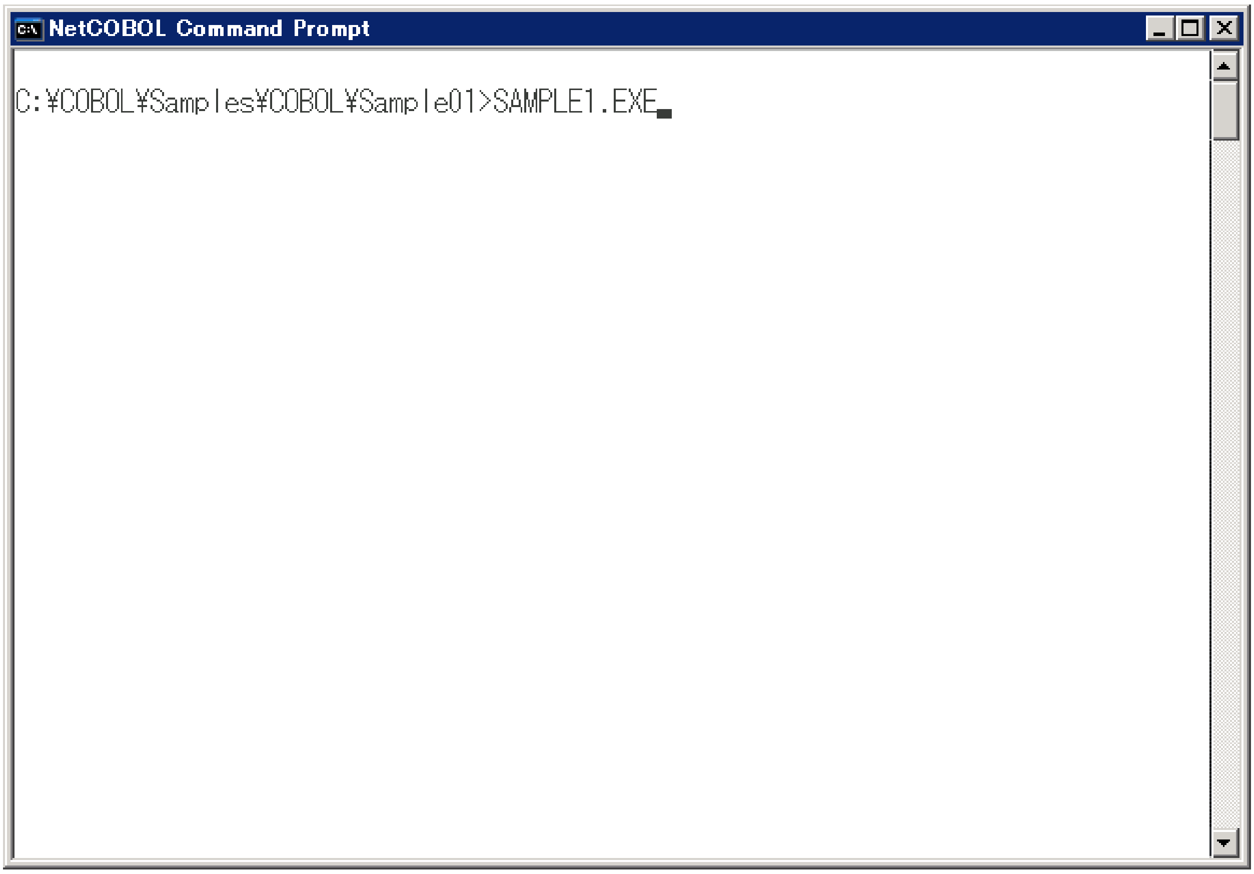
5. Select Close from the Files menu.

6. Select Exit from the Files menu.

   The Runtime Environment Setup Tool terminates.

Now, let's actually execute the executable file.

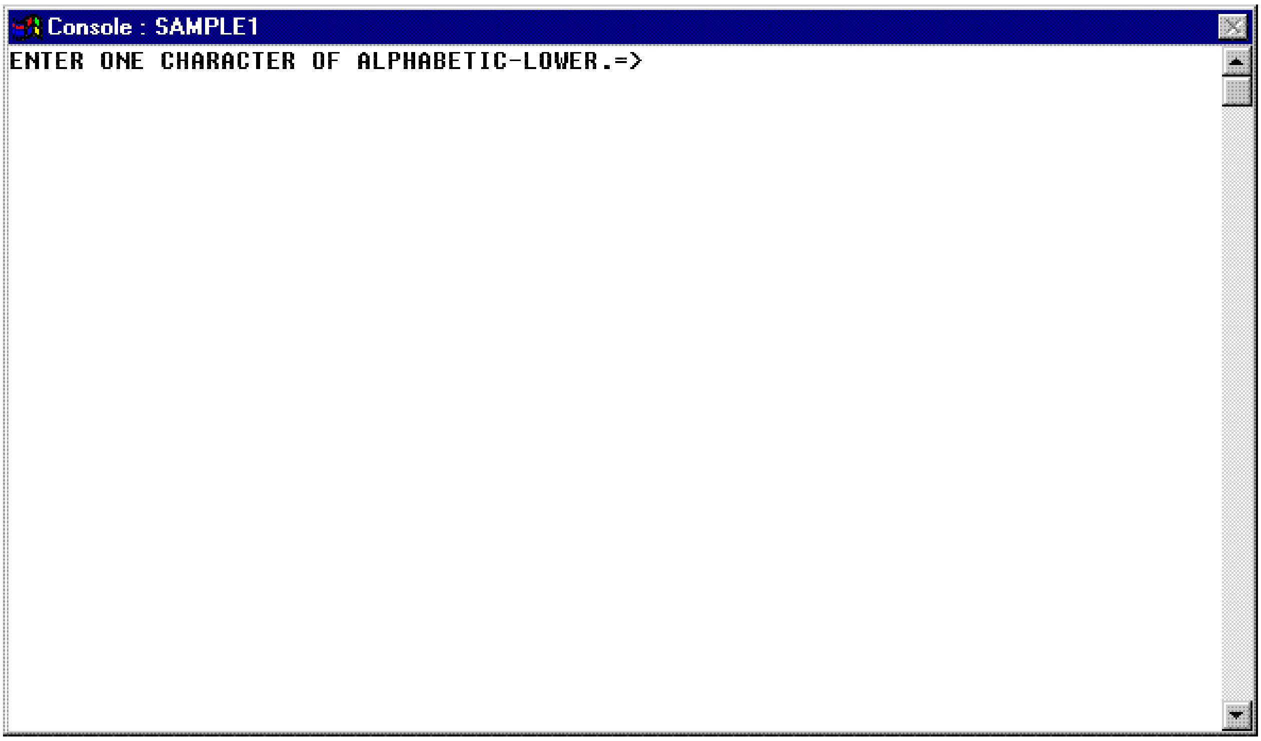
1. Enter the name of the file to execute (SAMPLE1.EXE) in the command prompt window.



```
C:¥COBOL¥Samples¥COBOL¥Sample01>SAMPLE1.EXE
```

2. For Sample 1, the system inputs or outputs data using the COBOL ACCEPT/DISPLAY function. The window given below is displayed. Enter a lowercase letter, then press the ENTER key.

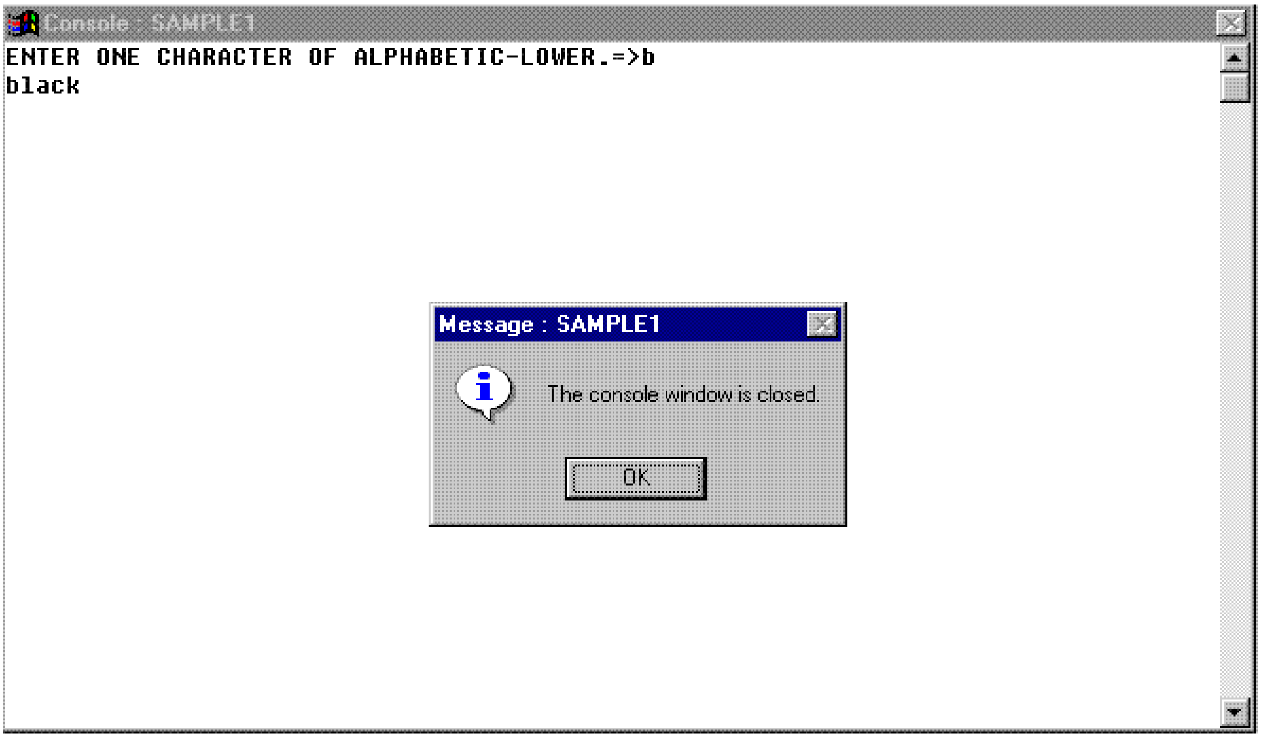
Figure 5.2 The NetCOBOL console window



```
ENTER ONE CHARACTER OF ALPHABETIC-LOWER.=>
```

An English word that begins with the input lowercase letter is displayed.

Figure 5.3 The NetCOBOL console window and message window



```
ENTER ONE CHARACTER OF ALPHABETIC-LOWER.=>b
black
```

The console window is closed.

3. Check the results, then click on the OK button in the message window. The message window and console windows close.

This completes execution of the sample program.

5.2 Execution Procedures

Executable programs compiled and linked from COBOL source programs can be executed in the same manner as normal Windows applications(*).

* : The applications work with Windows(x64)

5.2.1 Setting up the Runtime Environment Information

Before executing COBOL programs, you usually need to set runtime environment information. In COBOL, resources and information allocated for the execution of the COBOL program are called the runtime environment information. The runtime environment information is explained in the "5.3 Setting Runtime Environment Information".

5.2.2 Executing COBOL Programs

The COBOL program can be executed by using execution method of the application used with each window system.

For instance, there are "Execute it from command prompt", "The file is selected and executed by the Explorer", and "The file is executed in the batch file specifying it", etc.

Please refer to the manual or the online help of each window system for details so that these execution methods may depend on each window system.

Moreover, the COBOL program can be executed without doing operator's input waiting when the following environment variable information is set to the initialization file or the environment variable for execution when the COBOL program is executed with the batch file etc

Environment variable information	meaning
@MessOutFile=*filename*	It outputs it to the file for which COBOL execution time message is specified.
@WinCloseMsg=OFF	The message when the window is shut is not displayed.

For details, refer to the section "5.4.1 Environment Variables".

Execution method of COBOL program using batch file

It is necessary to wait for the end of the COBOL program immediately before specifying /WAIT option for START command of OS to take the starting program termination synchronization when the program is started from the batch file.

Note

- When the program is started by the CALL command, the starting program termination synchronization cannot be taken.
- The file of the networked environment (The case where the file of a local environment is used by the UNC specification is included) is accessed or the delay is generated in writing. Therefore, consideration that synchronizes might become necessary as for the end.

Moreover, when terminated state of the COBOL program is examined with ERRORLEVEL of the batch file, /WAIT option is specified for the START command.

As for ERRORLEVEL and the START command, the specification might be different depending on OS. Please refer to help of each OS for details.

The example of the method of executing COBOL program "COBPROG1" and "COBPROG2" with batch file "COBGO.BAT" is shown as follows.

When you use the initialization file for execution

When the initialization file for execution is used, it sets it to "COBOL85.CBR".

COBOL85.CBR

```
[COBPROG1]
@MessOutFile=COBPROG1.MES
@WinCloseMsg=OFF
    :
[COBPROG2]
@MessOutFile=COBPROG2.MES
@WinCloseMsg=OFF
    :
```

COBGO.BAT

```
REM COBOL program execution 1
start /wait COBPROG1
if errorlevel 12 echo An error occurred.  -- True, if errorlevel is 12 or higher.
REM COBOL program execution 2
start /wait COBPROG2
if errorlevel 12 echo An error occurred.  -- True, if errorlevel is 12 or higher.
```

When you specify environment variable information for the environment variable

There is a method to set by the system or the SET command of system control panel when specifying it for the environment variable. Here, the method to set by using the SET command with the batch file is described.

COBGO.BAT

```
REM Environment variable information setting 1
SET @MessOutFile=COBPROG1.MES
SET @WinCloseMsg=OFF
SET @ExitSessionMSG=OFF
REM COBOL program execution 1
start /wait COBPROG1
if errorlevel 12 echo An error occurred.  -- True, if errorlevel is 12 or higher.
REM Environment variable information setting 2
SET @MessOutFile=COBPROG2.MES
REM COBOL program execution 2
start /wait COBPROG2
if errorlevel 12 echo An error occurred.  -- True, if errorlevel is 12 or higher.
```

Please refer to "COBOL Runtime System Messages" in "NetCOBOL Messages" for the return code when the batch file is used.

5.2.2.1 Cautions Running Under Windows Service Facility

Note the points given below when running COBOL programs under the service facility. For cautions for printing under service, refer to "7.1.11 Cautions under Service Range (for printing)". For the service settings, refer to the handbook of each service used.

If the service that calls COBOL programs permits desktop interaction

No special specification is required for the runtime environment information for COBOL programs.

If the service that calls COBOL programs does not permit desktop interaction

Since COBOL programs cannot output windows and message boxes on desktop, operation cannot be guaranteed if waiting for operator input may occur during execution of COBOL programs.

Handle this case by following either of the methods given below.

- Change the service setting so that it permits desktop interaction.
- Set the runtime environment information so that the functions given below that are based on window display will not be used from COBOL programs and waiting for operator input will not occur.
    - Small I-O function using the console window (ACCEPT/DISPLAY)
    - Screen handling functions

Note

To use the DISPLAY statement for debugging COBOL programs that run under service, specify a file as the I-O destination. For the I-O destination in the small I-O function, refer to "10.1.2 Input/Output Destination Types and Specification Methods".

5.3 Setting Runtime Environment Information

This section explains the relationship between the types of runtime environment information and setup procedures, and how to set each item in the runtime environment.

5.3.1 Types of Runtime Environment Information

Information required to execute COBOL applications is called runtime environment information.

There are two types of runtime environment information, environment variable information and entry information. Environment variable information includes items such as the console window size, console font, and file identifier.

Entry information specifies locations of subprograms and entry points within DLL files.

For details of both types of runtime environment information, refer to "5.4 Format of Runtime Environment Information".

5.3.2 How to Set Runtime Environment Information

Runtime environment information can be set as follows:

a. By inserting from the System control panel.

b. By using the SET command

c. By setting-up the runtime initialization file.

d. By setting in the Runtime Environment Setup Tool.

e. By setting from the command line. Runtime option, runtime parameter (GS series format only).

It is recommended that the runtime environment information be set by (a) or (b) and that the entry information is set in the entry information file. See topic "5.4.2 Entry Information for Subprograms" later in this chapter.

Note

The runtime environment information specified in the runtime initialization file and entry information file affects the runtime performance because it is fetched when opening the runtime environment of the COBOL program. Therefore, it is recommended to set the runtime environment information as follows:

- Set the environment variable information in user environment variables with a batch file before starting the program.
- Set only the information required for the program to be executed in the runtime initialization file and entry information file.

5.3.2.1 Control Panel System

If the environment variable information were common to multiple applications, presetting the information here would be convenient.

The control panel system is used to set the environment variable information before the program is executed. For information on how to preset the information, refer to the help menu.

5.3.2.2 Setting from the SET Command

This method is used for setting environment variable information from the command prompt or batch file using the SET command.

The environment variable information that is set from a command prompt, applies to programs started from that command prompt.

Using a batch file enables the performance of all procedures from setup through execution to be performed in one step.

The environment variable information, which is set in a batch file, applied to all programs started from that batch file.

5.3.2.3 How to Set the Runtime Initialization File

This section explains how to create a runtime initialization file before execution of the program and set the runtime environment information.

Note

The COBOL runtime system is accessing the runtime initialization file when the COBOL program is executed. Please do not operate the runtime initialization file until the execution of the COBOL program ends as follows.

- Reference and update by the other programs
- Reference and update by the editor
- Copy

When the above-mentioned operation is done before the program ends, information on the runtime initialization file might not become effective.

5.3.2.3.1 Runtime initialization file

Information for executing programs created in COBOL is saved in the runtime initialization file.

The file, "COBOL85.CBR," in the folder containing the executable program (referred to as an EXE file, henceforth) is normally used as the runtime initialization file. However, the following files can also be employed as runtime initialization files depending on the creation method of the application.

- COBOL85.CBR in the folder containing the dynamic link library (abbreviated to DLL, henceforth)
- Files created with names other than COBOL85.CBR

To use the COBOL85.CBR in the folder containing the DLL as the runtime initialization file, see "5.3.2.3.4 Using the runtime initialization file under DLL". To use files created with names other than COBOL85.CBR as runtime initialization files, see the following:

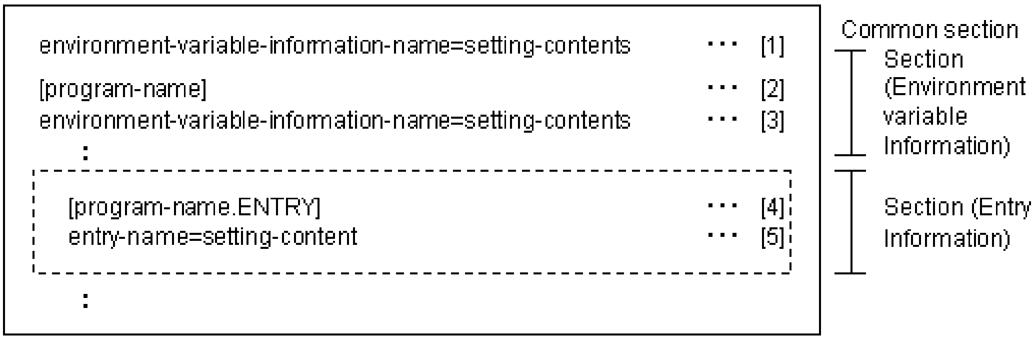
- To use a non-COBOL program (C program or Visual Basic (R) program) as a main program, see "5.6 Specifying the Initialization File Name."
- To specify the runtime initialization file with an environment variable, see "5.4.1.3 @CBR_CBRFILE(Set the runtime initialization file)".
- To specify the file with the command line option, see "5.3.2.5 Setting from the Command Line".

The program may be executed without a runtime initialization file.

5.3.2.3.2 Runtime initialization file contents

The runtime initialization file consists of the common section and several other sections. Write the environment variable information common to each program in the common section. Write the environment variable information and entry information for individual programs in other sections. Write the section name within brackets ([]). The contents up to the next section name are regarded as one section.

The contents of the runtime initialization file are shown below.



```
environment-variable-information-name=setting-contents    ･･･ [1]
[program-name]                                             ･･･ [2]
environment-variable-information-name=setting-contents    ･･･ [3]
    :
  [program-name.ENTRY]                                     ･･･ [4]
  entry-name=setting-content                               ･･･ [5]
    :
```

Common section
Section (Environment variable Information)
Section (Entry Information)

Note

Two or more environment variables or entry information items cannot be written on a single line.

Explanation of the contents

- [1] Write the environment variable information common to each program. For details on the specification format of environment variable, see "5.4.1 Environment Variables". The environment variable information written in this section is effective until the application is terminated.

  The environment variable information written in the common section is enabled in both the multithread mode and single-thread mode. In the single-thread mode, if the same environment variable is specified in the common section and another section having the same name as the name of the program to be executed, the contents in the named section overrides the common section.

- [2] Indicates the beginning of the environment variable information. Specify the name of the COBOL main program for the section name(*1). This section can be written only once for each program name.

- [3] Write the environment variable information for each program. For details on the specification format of the environment variable information, see "5.4.1 Environment Variables". The environment variable information written in this section is enabled until the application is terminated.

  In the multithread mode, the environment variable information written in this section is ignored.

- [4] Indicates the beginning of the entry information for each program. Specify the main COBOL program name (*1) with ".ENTRY" for the section name of the entry information. This section can be written only once for each program name.

- [5] Write the entry information for each program(*2). For details on the specification format of the entry information, see "5.4.2 Entry Information for Subprograms". The entry information written in this section is effective until the runtime environment of COBOL is terminated.

  In the multithread mode, the entry information written in this section is ignored.

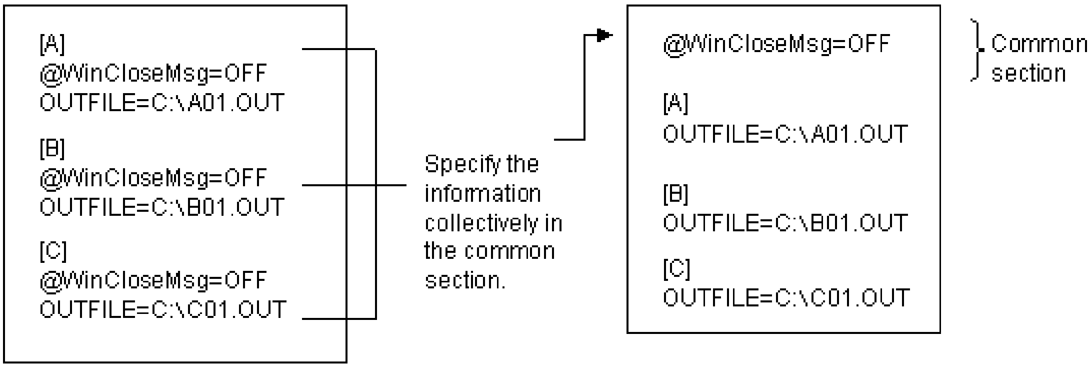
  - *1 : To use JMPCINT2 and JMPCINT3 to call a COBOL program from a non-COBOL program, specify the name of the first COBOL program to be called. For details on the COBOL main program and runtime environments, see "9.1.2 COBOL Inter-Language Environment".

  - *2 : The entry information file can also be used to specify environment information. When using the runtime initialization file, the entry information can be specified only for each program; however, environment information common to all programs can be specified. Therefore, the use of the entry information file to specify both entry and environment information is recommended. For details on the entry information, see "5.4.2 Entry Information for Subprograms".

Note

- Entry information in the runtime initialization file is disabled in the multithread mode. Specify the information in the entry information file.

- Writing common environment variable information, in the common section can save the file size

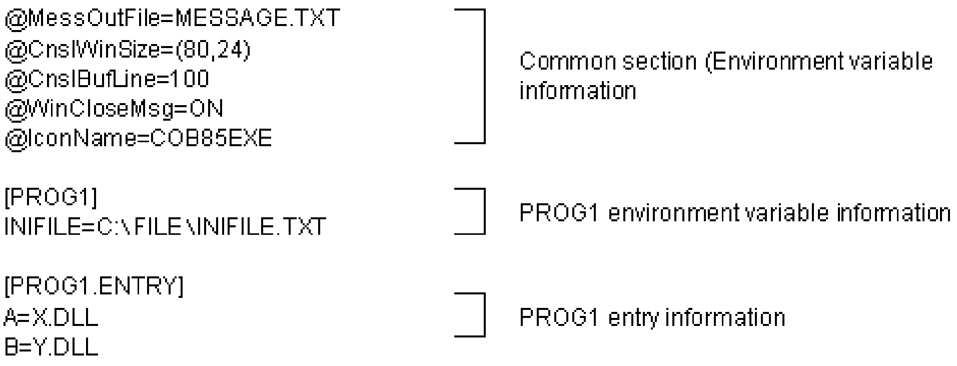


```
[A]
@WinCloseMsg=OFF
OUTFILE=C:\A01.OUT
[B]
@WinCloseMsg=OFF
OUTFILE=C:\B01.OUT
[C]
@WinCloseMsg=OFF
OUTFILE=C:\C01.OUT
```

```
@WinCloseMsg=OFF

[A]
OUTFILE=C:\A01.OUT

[B]
OUTFILE=C:\B01.OUT

[C]
OUTFILE=C:\C01.OUT
```

- Do not specify the same subprogram name or the same environment variable name repeatedly, the operation in these cases is not guaranteed.

Sample coding of runtime initialization file



```
@MessOutFile=MESSAGE.TXT
@CnslWinSize=(80,24)
@CnslBufLine=100
@WinCloseMsg=ON
@IconName=COB85EXE

[PROG1]
INIFILE=C:\FILE\INIFILE.TXT

[PROG1.ENTRY]
A=X.DLL
B=Y.DLL
```

Common section (Environment variable information
PROG1 environment variable information
PROG1 entry information

How to write a comment in the runtime initialization file

When the line starts with a semicolon (;), it is treated as a comment.

If too many comment lines are included, the processing speed may go down due to skipping comment lines.

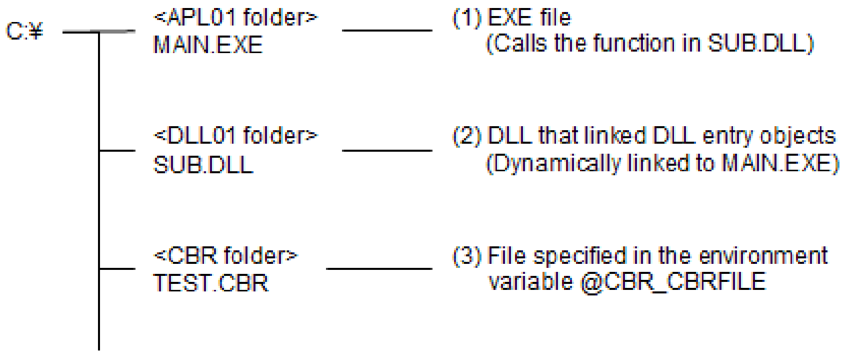
5.3.2.3.3 Searching order of runtime initialization file

The search order of runtime initialization file is listed below.

1. COBOL85.CBR in the folder containing the EXE file
2. COBOL85.CBR in the folder containing the first DLL called
3. Runtime initialization file specified in environment variable @CBR_CBRFILE

Example

Search Order


C:¥
<APL01 folder>
MAIN.EXE
(1) EXE file
(Calls the function in SUB.DLL)
<DLL01 folder>
SUB.DLL
(2) DLL that linked DLL entry objects
(Dynamically linked to MAIN.EXE)
<CBR folder>
TEST.CBR
(3) File specified in the environment
variable @CBR_CBRFILE

In the above example, the search order of searching for runtime initialization is as follows.

1. C:\APL01\COBOL85.CBR
2. When (1) is not provided: C:\DLL01\COBOL85.CBR
3. When (1) and (2) are not provided: C:\CBR\TEST.CBR

If the runtime initialization file is specified with JMPCINTC/JMPCINTB or in the command line, the above files are not searched and the specified file is enabled. See "5.6 Specifying the Initialization File Name" and "5.3.2.5 Setting from the Command Line".

Specifying environment variable information @CBR_CBRINFO=YES will obtain information from the runtime initialization file. A message is generated when opening the runtime environment. See "5.4.1.4 @CBR_CBRINFO(Output of simplified operation status)".

The sample EXE file described here indicates COBOL programs and programs in other languages. The DLL is assumed to be linked to DLL entry objects and is loaded when the COBOL runtime environment is established.

EXE file storage position	DLL storage position	
	With COBOL85.CBR file	Without COBOL85.CBR file
With COBOL85.CBR file	EXE file storage position is valid	EXE file storage position is valid
Without COBOL85.CBR file	DLL storage position is valid	-

5.3.2.3.4 Using the runtime initialization file under DLL

The COBOL85.CBR in the folder containing the EXE file is normally used for the runtime initialization file. Therefore, calling COBOL programs from a non-COBOL program may cause the following problems:

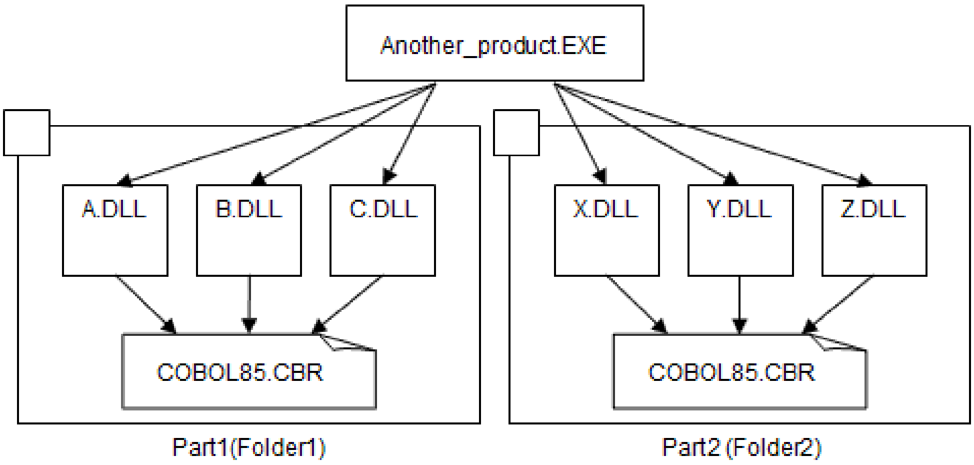
1. The COBOL85.CBR must be copied to the folder containing the EXE file that calls the COBOL application (DLL).
2. All information for each COBOL program must be placed in the same COBOL85.CBR file in the folder containing the EXE file.

   Thus the file size of COBOL85.CBR is increased and performance is degraded.

To solve the above problems, place the COBOL85.CBR files in the folder containing the COBOL application (DLL). Using this method resolves the above problems.

- The storage location of the EXE file is not relevant because the COBOL85.CBR is placed in the folder containing the COBOL application (DLL).

- The COBOL85.CBR containing only relevant information for each application can be used.



In the above case, place the applications (A.DLL, B.DLL, and C.DLL) that use the identical runtime environment (with COBOL85.CBR enabled for each process) in folder 1. Starting part 1 from another product uses the information from COBOL85.CBR in folder 1 and guarantees the runtime environment information until the process is terminated. Likewise, starting part 2 from another product enables the information from COBOL85.CBR in folder 2 for each process of part 2.

As described in the above, when using the COBOL85.CBR in the folder containing the DLL, the DLL entry objects must be static linked when creating the DLL.

Note

- The DLL created by linking DLL entry objects must be loaded on the memory when opening the runtime environment. If the dynamic program structure is used, the COBOL85.CBR in the folder containing the DLL cannot be used because the DLL is loaded on the memory after the runtime environment is opened.
- Do not place the COBOL85.CBR in the folder containing the EXE file because the runtime initialization file is searched from the folder containing the EXE file.
- Store the DLL created by linking the DLL entry objects with the relevant COBOL85.CBR (runtime environment). If the DLL is not stored in a separate folder for each runtime environment, unexpected operation may result during execution.
- The COBOL85.CBR is enabled for all processes in the same runtime environment. To allocate different values to the same environment variable information in each application, start each application as a separate process.

Linking the DLL entry objects

This section explains how to create the DLL by linking DLL entry objects.

To create the DLL, link the following DLL entry objects.

DLL entry object

F4AGCBDM.OBJ : Specified when the COBOL DLL is created.

F4AGMLDM.OBJ : Specified when the COBOL and other languages are linked and DLL is created.

When COBOL DLL is created:

```
LINK COBSUB.OBJ F4AGCBDM.OBJ F4AGCIMP.LIB KERNEL32.LIB /ENTRY:COBDMAIN /DLL /OUT:COBSUB.DLL
```

- COBSUB.OBJ : COBOL program object file
- F4AGCBDM.OBJ : DLL entry object file
- F4AGCIMP.LIB : COBOL runtime system import library
- KERNEL32.LIB : Windows function import library

Note

COBDMAIN must be specified for the LINK option /ENTRY.

When COBOL and other languages are linked and DLL is created:

```
LINK CPROG.OBJ COBSUB.OBJ F4AGMLDM.OBJ F4AGCIMP.LIB KERNEL32.LIB
C runtime library /DLL /OUT:CPROG.DLL
```

- CPROG.OBJ : C program object file
- COBSUB.OBJ : COBOL program object file
- F4AGMLDM.OBJ : DLL entry object file
- F4AGCIMP.LIB : COBOL runtime system import library
- KERNEL32.LIB : Windows function import library
- C runtime library : If the C program is compiled with the /ML option, specify LIBC.LIB. If the C program is compiled with the / MD option, specify MSVCRT.LIB.

5.3.2.4 Setting with the Runtime Environment Setup Tool

This method uses the Runtime Environment Setup tool to set the runtime environment information. Refer to "5.5 Runtime Environment Setup Tool".

5.3.2.5 Setting from the Command Line

This method specifies the content of the runtime environment information as the command argument when the program is started using the command line. Under this method, the global server format runtime parameter (environment variable information @MGPRM), the runtime initialization file name, and the runtime option (environment variable information @GOPT) can be specified. Refer to "5.4.1.57 @MGPRM(Set the GS-series Format Runtime Parameter)" and "5.4.1.53 @GOPT(Set Runtime Options)".

The command line format is:

```
executable-file-name [runtime parameter] [-CBR runtime initialization file-name] [-CBL runtime option]
```

Note

-CBR and -CBL can be in any order.

Specifying the GS-series Format Runtime Parameter

When using a runtime parameter in the Global Server format, the first argument following the command name is assumed to be the runtime parameter in the Global Server format.

Example

```
PROG1.EXE "ABCDE"
```

ABCDE is specified as the GS-series format runtime parameter.

Specifying an Initialization File Name

Specify an initialization file following identifier -CBR or /CBR.

For the format for the runtime initialization file name, refer to "5.6 Specifying the Initialization File Name".

Example

```
PROG1.EXE -CBR ABC.INI
```

ABC.INI is specified as the runtime initialization file name (main program is PROG1.EXE).

Specifying Runtime Options in the Command Line

Runtime options are specified following identifier -CBL or /CBL.

For the format of the runtime options, refer to "5.7 Format of Runtime Options".

Example

```
PROG1.EXE -CBL r20 c20
```

r20 and c20 are specified as runtime options.

5.4 Format of Runtime Environment Information

This section describes the format of runtime environment information.

5.4.1 Environment Variables

The environment variable information is listed below.

Information

The order of priority for duplicated runtime environment information items set in the command line (runtime option, runtime parameter (GS series only)), runtime initialization file, and environment variable is shown below.

1. Command line values (runtime option, runtime parameter (GS series only))
2. Runtime initialization file values
3. Environment variable values

Runtime options

- 5.4.1.53 @GOPT(Set Runtime Options)
- 5.4.1.57 @MGPRM(Set the GS-series Format Runtime Parameter)

Execution

- 5.4.1.3 @CBR_CBRFILE(Set the runtime initialization file)
- 5.4.1.4 @CBR_CBRINFO(Output of simplified operation status)
- 5.4.1.5 @CBR_CODE_SET(Specify the code-set for the output file)
- 5.4.1.7 @CBR_CONVERT_CHARACTER(Specify the code conversion library)
- 5.4.1.20 @CBR_ENTRYFILE(Set the Entry Information File)
- 5.4.1.52 @ExitSessionMSG(Set message if Windows system is closed)
- 5.4.1.54 @IconDLL(Set the DLL name of an Icon Resource)
- 5.4.1.55 @IconName(Set the Identifier of an Icon Resource)

- 5.4.1.64 @ShowIcon(Specifying the suppression of COBOL icon display)

Screen

- 5.4.1.42 @CBR_SCREEN_POSITION(Specify the position of Screen window)
- 5.4.1.40 @CBR_SCR_KEYDEFFILE(Specify key definition file for screen handling)
- 5.4.1.41 @CBR_SCR_HIGHLIGHT(Screen item highlight style)
- 5.4.1.62 @ScrnFont(Set the Font Used for Screen Handling)
- 5.4.1.63 @ScrnSize(Set the Size of the Logical Screen for Screen Handling)

ACCEPT/DISPLAY

- 5.4.1.6 @CBR_CONSOLE(Set the type of the console window)
- 5.4.1.10 @CBR_DISPLAY_CONSOLE_EVENTLOG_LEVEL (Specify the Event Type for DISPLAY UPON CONSOLE output to the Event Log)
- 5.4.1.11 @CBR_DISPLAY_CONSOLE_EVENTLOG_SRCNAME (Specify the Event Source Name for DISPLAY UPON CONSOLE output to the Event Log)
- 5.4.1.12 @CBR_DISPLAY_CONSOLE_OUTPUT (Specify the Event Log for DISPLAY UPON CONSOLE output)
- 5.4.1.13 @CBR_DISPLAY_SYSERR_EVENTLOG_LEVEL (Specify the Event Type for DISPLAY UPON SYSERR output to the Event Log)
- 5.4.1.14 @CBR_DISPLAY_SYSERR_EVENTLOG_SRCNAME (Specify the Event Source Name for DISPLAY UPON SYSERR output to the Event Log)
- 5.4.1.15 @CBR_DISPLAY_SYSERR_OUTPUT (Specify the Event Log for DISPLAY UPON SYSERR output)
- 5.4.1.16 @CBR_DISPLAY_SYSOUT_EVENTLOG_LEVEL (Specify the Event Type for DISPLAY UPON SYSOUT output to the Event Lo)
- 5.4.1.17 @CBR_DISPLAY_SYSOUT_EVENTLOG_SRCNAME (Specify the Event Source Name for DISPLAY UPON SYSOUT output to the Event Log)
- 5.4.1.18 @CBR_DISPLAY_SYSOUT_OUTPUT (Specify the Event Log for DISPLAY UPON SYSOUT output)
- 5.4.1.28 @CBR_JOBDATE(Set the date for this execution)
- 5.4.1.48 @CnslBufLine(Set the Buffer Count for the Console Window)
- 5.4.1.49 @CnslFont(Set the Console Window Font)
- 5.4.1.50 @CnslWinSize(Set the Size of the Console Window)
- 5.4.1.65 @WinCloseMsg(Set a Display Message When the Window Close)

Messages

- 5.4.1.30 @CBR_MESSAGE(Specify the runtime message output destination)
- 5.4.1.31 @CBR_MESS_LEVEL_CONSOLE(Specify the level of importance of runtime messages)
- 5.4.1.32 @CBR_MESS_LEVEL_EVENTLOG(Specify the level of importance of runtime messages)
- 5.4.1.43 @CBR_SYSERR_EXTEND(Specify the SYSERR output information extension)
- 5.4.1.56 @MessOutFile(Set a Message Output File)
- 5.4.1.58 @NoMessage(Set to Suppress Runtime Messages)

Files

- 5.4.1.1 @AllFileExclusive(Set Exclusive Control of Files)
- 5.4.1.47 @CBR_TRAILING_BLANK_RECORD(Specify whether to remove or enable the trailing blank in the record of line sequential file)
- 5.4.1.66 File identifier(Set the File Used by the Program)

- 5.4.1.76 SYSIN Access Name(Set the Input File for the ACCEPT/DISPLAY Function)
- 5.4.1.77 SYSOUT Access Name(Set the Output File for the ACCEPT/DISPLAY Function)
- 5.4.1.26 @CBR_FILE_USE_MESSAGE(Output runtime messages for input/output errors)
- 5.4.1.23 @CBR_FILE_BOM_READ(Specify Unicode BOM treatment)
- 5.4.1.24 @CBR_FILE_DUP_CLOSE(CLOSE statement success/failure)
- 5.4.1.22 @CBR_EXFH_LOAD(Set DLL name of External File Handler)
- 5.4.1.21 @CBR_EXFH_API(Set entry-name of External File Handler)
- 5.4.1.25 @CBR_FILE_SEQUENTIAL_ACCESS (Set high-speed file processing as the runtime default)

Presentation files

- 5.4.1.39 @CBR_PSFILE_xxx(Set the Connected Product Name Used from the Presentation File (by Destination))
- 5.4.1.67 File Identifier(Set the Information File and the Connected Product Name Used from the Presentation File (by File))

Printers

- 5.4.1.19 @CBR_DocumentName_xxxx(Specify I control record document name)
- 5.4.1.34 @CBR_OverlayPrintOffset(I-Control record effects on overlay printing)
- 5.4.1.35 @CBR_PrinterANK_Size(Specification of ANK character size)
- 5.4.1.36 @CBR_PrintFontTable(Specify the font table used for a print file)
- 5.4.1.37 @CBR_PrintInfoFile(Specify a print information file to the file that specified PRINTER in the ASSIGN phrase)
- 5.4.1.38 @CBR_PrintTextPosition(Specify method of calculating character arrangement coordinates)
- 5.4.1.44 @CBR_TextAlign(Specify alignment of print characters with either top or bottom of line)
- 5.4.1.51 @DefaultFCB_Name(Specification of default FCB name)
- 5.4.1.60 @PrinterFontName(Set the Font Used for Print Files)
- 5.4.1.61 @PRN_FormName_xxx(Specify paper name)
- 5.4.1.68 File identifier(Specify the printer information file and various parameters used for the program)
- 5.4.1.70 FCBxxxx(Set FCB Control Statements)
- 5.4.1.71 FOVLDIR(Set the Folder Containing Form Overlay Patterns)
- 5.4.1.72 FOVLTYPE(Set the Format of the Form Overlay Pattern File)
- 5.4.1.73 FOVLNAME(Specifying Form overlay pattern file names)
- 5.4.1.74 OVD_SUFFIX(Set the Extension of the Form Overlay Pattern File)

Debug

- 5.4.1.2 @CBR_ATTACH_TOOL(Specify the attached debugging using NetCOBOL Studio)
- 5.4.1.29 @CBR_JUSTINTIME_DEBUG(Specify inspection using the COBOL Error Report at abnormal termination)
- 5.4.1.33 @CBR_MEMORY_CHECK(Specify the inspection using the memory check function)
- 5.4.1.45 @CBR_TRACE_FILE(Specify the trace information output file)
- 5.4.1.46 @CBR_TRACE_PROCESS_MODE(Unique file name foe each Trace file output)
- 5.4.1.75 SYSCOUNT(Specify the output file for COUNT information)

Database

- 5.4.1.59 @ODBC_Inf(Set the ODBC Information File)

Multithread

- @CBR_SYMFOWARE_THREAD(Specification enabling the multithread operation using linkage with SymfoWARE)
- @CBR_THREAD_MODE(Specifying the thread mode)
- @CBR_THREAD_TIMEOUT(Specifying the wait time of the thread synchronization control subroutine)
- @CBR_SSIN_FILE(Specifying that opens input file of each thread)

OO COBOL

- 16.3.4.1.1 @CBR_ClassInfFile Environment Variable
- 16.3.4.1.2 @CBR_InstanceBlock Environment Variable

Intrinsic Function

- 5.4.1.27 @CBR_FUNCTION_NATIONAL(Specify the conversion mode of NATIONAL function)

CSV data

- 5.4.1.8 @CBR_CSV_OVERFLOW_MESSAGE(Specify to suppress messages at CSV data operation)
- 5.4.1.9 @CBR_CSV_TYPE(Set the variation of generated CSV type)

Note

How to specify the folder name and file name

If the folder name or file name includes a comma (,), the folder name or file name must be specified within double quotation marks (").

5.4.1.1 @AllFileExclusive(Set Exclusive Control of Files)

$$\texttt{@AllFileExclusive=}\quad\left\{\begin{array}{l}\text{YES}\\ \underline{\text{NO}}\end{array}\right\}$$

When set to YES, all files are processed exclusively, regardless of the exclusivity set in the COBOL source program. Files used by the program cannot be accessed from any other program (open error).

When exclusive control is specified, files used by the program cannot be accessed from non-COBOL programs (open error).

Note

File exclusive control can reduce file access time.

5.4.1.2 @CBR_ATTACH_TOOL(Specify the attached debugging using NetCOBOL Studio)

```
@CBR_ATTACH_TOOL=host/STUDIO [ path-list ]
```

Specify this parameter to use NetCOBOL STUDIO as the debugger.

host

Specify the port number and host name of the server remote debugger connection. Use the following format to specify the host..

$$\left\{\begin{array}{l}\text{IP address}\\ \text{Host name}\\ \text{localhost}\end{array}\right\}\quad \text{[: Port number]}$$

Specify "IP address" or "Host name" when connecting to a COBOL application running on a server using the NetCOBOL Studio remote debugger.

Specify "localhost" when using the NetCOBOL Studio debugger on a local PC.

The IP address must be in IPv4 format or IPv6 format.

The Port number must be from 1024 to 65535. If a port number is not specified, it defaults to 59999.

A scoped address can be specified as an IPv6 address. For a scoped address, specify a scope identifier after the address.

If the port number is specified in an IPv6 address, the address part must be enclosed in [ ].

STUDIO

This is a required parameter which specifies the use of NetCOBOL Studio as the debugger.

path-list

Specify the directory that contains the debugging information file. The debugging information is retrieved in the following order, and is used for debugging.

1. The specified order of the additional path list (delimit by semicolon ";" and enter the full path name when you specify two or more folders).
2. The current folder when the COBOL program began to run.
3. The folder that contains the COBOL program.

Note

2. and 3. do not need to be added to PATHLIST.

Example

```
[IPv4 address(192.168.0.1) and port number(2000)]
  @CBR_ATTACH_TOOL=192.168.0.1:2000/STUDIO

[IPv6 address(fe80::1:23:456:789a) and port number(2000)]
  @CBR_ATTACH_TOOL=[fe80::1:23:456:789a]:2000/STUDIO

[IPv6 address(fe80::1:23:456:789a) and scope identifier(%eth0)]
  @CBR_ATTACH_TOOL=fe80::1:23:456:789a%eth0/STUDIO

[IPv6 address(fe80::1:23:456:789a), scope identifier(%5) and port number(2000)]
  @CBR_ATTACH_TOOL=[fe80::1:23:456:789a%5]:2000/STUDIO

[Host name(client-1) and port number(2000)]
  @CBR_ATTACH_TOOL=client-1:2000/STUDIO

[Port number omitted(IPv4)]
  @CBR_ATTACH_TOOL=192.168.0.1/STUDIO

[Port number omitted(IPv6)]
  @CBR_ATTACH_TOOL=fe80::1:23:456:789a/STUDIO
```

5.4.1.3 @CBR_CBRFILE(Set the runtime initialization file)

@CBR_CBRFILE=runtime-initialization-file-name

Specify the runtime initialization file name to be used.

This name becomes valid when the runtime initialization file name is not specified in the argument of JMPCINTC/JMPCINTB or command line and the COBOL85.CBR file does not exist in the folder that contains the EXE file and DLL.

The absolute path or relative path can be specified as the file name. When the relative path is specified, the relative path is from EXE. Refer to "5.3.2.3 How to Set the Runtime Initialization File".

5.4.1.4 @CBR_CBRINFO(Output of simplified operation status)

@CBR_CBRINFO=YES

When the runtime environment is established, the COBOL program runtime information is output as the runtime message (JMP0070I-I). The output information includes the version levels, thread mode, and runtime initialization file names. Refer to "Message Produced by the COBOL Runtime System" in NetCOBOL Messages.

5.4.1.5 @CBR_CODE_SET(Specify the code-set for the output file)

```
                     ┌ ACP   ┐
@CBR_CODE_SET=      {         }
                     └ UTF-8 ┘
```

The character code set of the file that the run-time system input-output processing is specified.

- Input file (note)

- The output file is newly made

Note

The character code set is identified by the presence of BOM.

When the execution character code set is Unicode, the default value becomes UTF-8. The default value becomes ACP for the character code set other than Unicode.

The specification of this environment variable becomes effective for the following files.

- ACCEPT/DISPLAY file

- Message Output file

- TRACE information file

- COUNT information file

5.4.1.6 @CBR_CONSOLE(Set the type of the console window)

```
                     ┌ SYSTEM ┐
@CBR_CONSOLE=       {          }
                     └ COBOL  ┘
```

When the Accept/Display function for a console window is used, this variable specifies whether a message at execution time is written to the system console (SYSTEM) or to the COBOL console window and a message box (COBOL).

If this is omitted, the system console is used when COBOL is the main program and the compiler option MAIN(MAIN) is specified for the compilation. If compiler option MAIN(WINMAIN) is used, or another language is the main program, the COBOL console window is used.

Note

If the system console is used, @CnslBufLine, @CnslFont and @CnslWinSize are ignored.

5.4.1.7 @CBR_CONVERT_CHARACTER(Specify the code conversion library)

@CBR_CONVERT_CHARACTER= $\left\{ \begin{array}{l} \text{ICONV} \\ \text{SYSTEM} \end{array} \right\}$

This variable specifies the code conversion library when the runtime system converts a character-code.

If this variable is omitted, ICONV is assumed.

ICONV

The NetCOBOL runtime system converts the character-code.

SYSTEM

Character-code conversion is similarly carried out to V10 product by system API. The character-code conversion of encoding UTF-32 becomes an error.

5.4.1.8 @CBR_CSV_OVERFLOW_MESSAGE(Specify to suppress messages at CSV data operation)

@CBR_CSV_OVERFLOW_MESSAGE=NO

Specify to suppress messages JMP0262I-W and JMP0263I-W upon execution of STRING statement (format 2) or UNSTRING statement (format 2).

5.4.1.9 @CBR_CSV_TYPE(Set the variation of generated CSV type)

@CBR_CSV_TYPE= $\left\{ \begin{array}{l} \underline{\text{MODE-1}} \\ \text{MODE-2} \\ \text{MODE-3} \\ \text{MODE-4} \end{array} \right\}$

Specify the variation that is generated by the execution of STRING statement (format 2).

This specification applies only to STRING statement without TYPE. Refer to "21.4 Variation of CSV".

5.4.1.10 @CBR_DISPLAY_CONSOLE_EVENTLOG_LEVEL (Specify the Event Type for DISPLAY UPON CONSOLE output to the Event Log)

@CBR_DISPLAY_CONSOLE_EVENTLOG_LEVEL = $\left\{ \begin{array}{l} \text{I} \\ \text{W} \\ \text{E} \end{array} \right\}$

Specify the Event Type to use for the DISPLAY UPON CONSOLE output to the Event Log.

I

Information Event. "Information" is displayed in Event Type.

W

Warning Event. "Warning" is displayed in Event Type.

E

Error Event. "Error" is displayed in Event Type.

Example

Output the event as a Warning in the Event Log.

```
@CBR_DISPLAY_CONSOLE_EVENTLOG_LEVEL=W
```

Note

When @CBR_DISPLAY_CONSOLE_EVENTLOG_LEVEL is not specified or a parameter other than "I/W/E" is specified, the Event Type is "I".

5.4.1.11 @CBR_DISPLAY_CONSOLE_EVENTLOG_SRCNAME (Specify the Event Source Name for DISPLAY UPON CONSOLE output to the Event Log)

@CBR_DISPLAY_CONSOLE_EVENTLOG_SRCNAME = event-source-name

Specify the Event Source Name when DISPLAY UPON CONSOLE is output to the Event Log.

When @CBR_DISPLAY_CONSOLE_OUTPUT is invalid, this environment variable is also invalid.

event-source-name

An event source that is registered on the machine to which the output is directed.

Note

When @CBR_DISPLAY_CONSOLE_EVENTLOG_SRCNAME is not specified, the output destination is "NetCOBOL Application x64".

5.4.1.12 @CBR_DISPLAY_CONSOLE_OUTPUT (Specify the Event Log for DISPLAY UPON CONSOLE output)

@CBR_DISPLAY_CONSOLE_OUTPUT = EVENTLOG[(computer-name)]

Specify the Event Log to be the destination of the DISPLAY UPON CONSOLE output.

Specify the destination computer name (local or networked) of the Event Log. If computer-name is omitted, the output goes to the computer that is running the program.

If the specified computer-name does not exist on the network, or if Windows(x64) is not running on the specified computer, a "failure to output" message and the DISPLAY UPON CONSOLE output will be written to the Event Log of the computer that is running the program.

Note

Install this product in on the computer that outputs the event log.

5.4.1.13 @CBR_DISPLAY_SYSERR_EVENTLOG_LEVEL (Specify the Event Type for DISPLAY UPON SYSERR output to the Event Log)

@CBR_DISPLAY_SYSERR_EVENTLOG_LEVEL = $\left\{\begin{array}{c}\underline{\text{I}}\\ \text{W}\end{array}\right\}$

E

Specify the Event Type to use for the DISPLAY UPON SYSERR output to the Event Log.

I

Information Event. "Information" is displayed in Event Type.

W

Warning Event. "Warning" is displayed in Event Type.

E

Error Event. "Error" is displayed in Event Type.

Example

Output the event as a Warning in the Event Log.

```
@CBR_DISPLAY_SYSERR_EVENTLOG_LEVEL=W
```

Note

When @CBR_DISPLAY_SYSERR_EVENTLOG_LEVEL is not specified or a parameter other than "I/W/E" is specified, the Event Type is "I".

5.4.1.14 @CBR_DISPLAY_SYSERR_EVENTLOG_SRCNAME (Specify the Event Source Name for DISPLAY UPON SYSERR output to the Event Log)

@CBR_DISPLAY_SYSERR_EVENTLOG_SRCNAME = event-source-name

Specify the event source name when the log of DISPLAY UPON SYSERR is output to the Event Log

When @CBR_DISPLAY_SYSERR_OUTPUT is invalid, this environment variable is also invalid.

event-source-name

An event source that is registered on the machine to which the output is directed.

Note

When @CBR_DISPLAY_SYSERR_EVENTLOG_SRCNAME is not specified, the output destination is "NetCOBOL Application x64".

5.4.1.15 @CBR_DISPLAY_SYSERR_OUTPUT (Specify the Event Log for DISPLAY UPON SYSERR output)

@CBR_DISPLAY_SYSERR_OUTPUT = EVENTLOG[(computer-name)]

Specify the Event Log to be the destination of the DISPLAY UPON SYSERR output.

Specify the destination computer name (local or networked) of the Event Log. If computer-name is omitted, the output goes to the computer that is running the program.

If the specified computer-name does not exist on the network, or if Windows(x64) is not running on the specified computer, a "failure to output" message and the DISPLAY UPON SYSERR output will be written to the Event Log of the computer that is running the program.

Note

Install this product on the computer that outputs the Event Log.

5.4.1.16 @CBR_DISPLAY_SYSOUT_EVENTLOG_LEVEL (Specify the Event Type for DISPLAY UPON SYSOUT output to the Event Lo)

@CBR_DISPLAY_SYSOUT_EVENTLOG_LEVEL = $\left\{\begin{array}{c}\underline{\text{I}}\\ \text{W}\\ \text{E}\end{array}\right\}$

Specify the Event Type to use for the DISPLAY UPON SYSOUT output to the Event Log.

I

Information Event. "Information" is displayed in Event Type.

W

Warning Event. "Warning" is displayed in Event Type.

E

Error Event. "Error" is displayed in Event Type.

Example

Output the event as a Warning in the Event Log.

```
@CBR_DISPLAY_SYSOUT_EVENTLOG_LEVEL=W
```

Note

When @CBR_DISPLAY_SYSOUT_EVENTLOG_LEVEL is not specified, or a parameter other than "I/W/E" is specified, the Event Type is "I".

5.4.1.17 @CBR_DISPLAY_SYSOUT_EVENTLOG_SRCNAME (Specify the Event Source Name for DISPLAY UPON SYSOUT output to the Event Log)

@CBR_DISPLAY_SYSOUT_EVENTLOG_SRCNAME = event-source-name

Specify the event source name when the log of DISPLAY UPON SYSOUT is output to the Event Log

When @CBR_DISPLAY_SYSOUT_OUTPUT is invalid, this environment variable is also invalid.

event-source-name

An event source that is registered on the machine to which the output is directed.

Note

When @CBR_DISPLAY_SYSOUT_EVENTLOG_SRCNAME is not specified, the output destination is "NetCOBOL Application x64".

5.4.1.18 @CBR_DISPLAY_SYSOUT_OUTPUT (Specify the Event Log for DISPLAY UPON SYSOUT output)

```
@CBR_DISPLAY_SYSOUT_OUTPUT = EVENTLOG[(computer-name)]
```

Specify the Event Log to be the destination of the DISPLAY UPON SYSOUT output.

Specify the destination computer name (local or networked) of the Event Log. If computer-name is omitted, the output goes to the computer that is running the program.

If the specified computer-name does not exist on the network, or if Windows(x64) is not running on the specified computer, a "failure to output" message and the DISPLAY UPON SYSOUT output will be written to the Event Log of the computer that is running the program.

Note

Install this product on the computer that outputs the Event Log.

5.4.1.19 @CBR_DocumentName_xxxx(Specify I control record document name)

```
@CBR_DocumentName_xxxx=document-name
```

For print files without the FORMAT phrase, set the string "xxxx" to the string (of four characters or less) in the DOC-INFO (document name identification information) field of the I control record. The document name is associated with this environment variable name.

The document name should be a string of alphabetic, numeric or Japanese characters no more than 128 bytes long.

The document name is displayed on the screen of the print manager or the printer which the Windows system offers.

Example

```
Value of the DOC-INFO field of the I CONTROL record:
  "ABCD"
Environment variable name becomes:
  @CBR_DocumentName_ABCD
Set the document name to "Selling slip of NetCOBOL" using the statement:
  @CBR_DocumentName_ABCD=Selling slip of NetCOBOL
```

5.4.1.20 @CBR_ENTRYFILE(Set the Entry Information File)

```
@CBR_ENTRYFILE=entry-information-filename
```

To specify entry information other than the entry description section of the initialization file (COBOL85.CBR) create an entry information file and specify it in this runtime environment variable.

Absolute and relative paths can be specified in the entry information file. If the relative path is specified, it will be from the folder containing the executable file under execution.

For details about entry information, refer to "5.4.2 Entry Information for Subprograms" or "16.2.4.4 Entry Information".

5.4.1.21 @CBR_EXFH_API(Set entry-name of External File Handler)

```
@CBR_EXFH_API = entryname-of-External-File-Handler
```

When an External File Handler is to be used, an entry-name for it must be specified.

For details on the External File Handler, see the topic "6.9.2 External File Handler".

Note

An entry-name must be case sensitive.

5.4.1.22 @CBR_EXFH_LOAD(Set DLL name of External File Handler)

```
@CBR_EXFH_LOAD=DLL-name-of-External-File-Handler
```

When an External File Handler is to be used, a DLL name for it must be specified.

For details on the External File Handler, see the topic "6.9.2 External File Handler".

An absolute path and relative path can be specified for a DLL name. When relative path is specified, it is a relative path from a current folder.

5.4.1.23 @CBR_FILE_BOM_READ(Specify Unicode BOM treatment)

$$\text{@CBR\_FILE\_BOM\_READ=} \quad \left\{ \begin{array}{l} \underline{\text{CHECK}} \\ \text{DATA} \\ \text{AUTO} \end{array} \right\}$$

Specify the treatment of the BOM when referencing a Unicode file.

For details, see "6.3.3 Processing Line Sequential Files".

CHECK

Check for a BOM in the data. If a BOM is included in the data and it matches the BOM specified in the record definition, skip it when executing the READ. If no BOM is included in the data, or if the BOM in the data does not match the BOM specified in the record definition, fail the OPEN.

DATA

If a BOM is included in the data, read it as part of the record data. If no BOM is included in the data, read the record from the beginning of the file.

AUTO

If a BOM is included in the data and it matches the BOM specified in the record definition, skip it when executing the READ. If the BOM in the data does not match the BOM specified in the record definition, fail the OPEN. If no BOM is included in the data, read the record from the beginning of the file.

5.4.1.24 @CBR_FILE_DUP_CLOSE(CLOSE statement success/failure)

$$\text{@CBR\_FILE\_DUP\_CLOSE=} \quad \left\{ \begin{array}{l} \text{YES} \\ \underline{\text{NO}} \end{array} \right\}$$

Specify whether the execution of the CLOSE statement for a file that has never been opened or one that has already been closed will succeed (YES) or fail with a runtime error (NO, the default).

5.4.1.25 @CBR_FILE_SEQUENTIAL_ACCESS (Set high-speed file processing as the runtime default)

```
@CBR_FILE_SEQUENTIAL_ACCESS = BSAM
```

To set high-speed file processing as the runtime default, specify BSAM for @CBR_FILE_SEQUENTIAL_ACCESS. Refer to "6.7.4 High-Speed File Processing".

5.4.1.26 @CBR_FILE_USE_MESSAGE(Output runtime messages for input/output errors)

```
@CBR_FILE_USE_MESSAGE=YES
```

Specify to output runtime messages when a recognized file input/output processing error occurs.

5.4.1.27 @CBR_FUNCTION_NATIONAL(Specify the conversion mode of NATIONAL function)

$$\text{@CBR\_FUNCTION\_NATIONAL=[}\left\{\begin{array}{l}\underline{\text{MODE-1}}\\ \text{MODE-2}\end{array}\right\}\text{][,}\left\{\begin{array}{l}\text{MODE-3}\\ \underline{\text{MODE-4}}\end{array}\right\}\text{]}$$

Specify the conversion mode of the NATIONAL function.

Specify MODE 1 (conversion as V50L10 or before compatible) or MODE 2 (more visual conversion). If this specification is omitted, MODE 1 is assumed as specified.

When the RCS (UTF16) compile option is enabled, specify MODE 3 for conversion in a UNIX-like system or MODE 4 for conversion in a Windows-like system, for the second operand. If the specification is omitted, MODE 4 is assumed specified.

Note

If the specification contains an error, the specification is assumed omitted. Also, if MODE 3 or MODE 4 is specified for a compile option other than RCS (UTF16), the specification is regarded as an error. The operation when MODE 3 or MODE 4 is omitted varies depending on the platform.

The differences between MODE 1 and MODE 2 are listed below.

Table 5.1 When compiler option RCS (SJIS) is enabled:

Specification	Conversion method
MODE1	"-" (0xB0) is converted to "[-]" (0x815C). "[']" (0x60) is converted to "[']" (0x8165).
MODE2	"-" (0xB0) is converted to "[-]" (0x815B). "[']" (0x60) is converted to "[']" (0x814D).

Table 5.2 When compiler option RCS (UTF16) is enabled:

Specification	Conversion method
MODE1	"-" (0xB0) is converted to "[-]" (0x2015). "[']" (0x60) is converted to "[']" (0x2018).
MODE2	"-" (0xB0) is converted to "[-]" (0x30FC). "[']" (0x60) is converted to "[']" (0xFF40).

The differences between MODE 3 and MODE 4 are listed below.

Table 5.3 When compiler option RCS (UTF16) is enabled:

Specification	Conversion method
MODE3	"-" (0x2D) is converted to "[-]" (0x2212). "[~]" (0x7E) is converted to "[-]" (0x301C).
MODE4	"-" (0x2D) is converted to "[-]" (0xFF0D). "[~]" (0x7E) is converted to "[-]" (0xFF5E).

5.4.1.28 @CBR_JOBDATE(Set the date for this execution)

@CBR_JOBDATE=Year.Month.Day

Use this environment variable to specify a date that can be retrieved in your program using an ACCEPT FROM DATE statement or the intrinsic function CURRENT-DATE (necessary if you want to set and retrieve 4 digit years).

Where:

- Year can be two digits from 00 to 99 or four digits from 1900 to 2099
- Month is a two digit month from 01 to 12
- Day is a two digit day from 01 to 31

Years from 1900 to 1999 can be expressed in two or four digits. Years from 2000 must be expressed in four digits. When using the CURRENT-DATE function the runtime system will make 2 digit years into 19xx years.

5.4.1.29 @CBR_JUSTINTIME_DEBUG(Specify inspection using the COBOL Error Report at abnormal termination)

```
                               ┌   ┌ APLERR  ┐                                ┐
@CBR_JUSTINTIME_DEBUG=         │ [ │ CBLERR  │ ]][,SNAP[start-parameter]]      │
                               │   └ ALLERR  ┘                                │
                               └ NO                                           ┘
```

Specify the target error events and the means of examining the errors at abnormal termination. When @CBR_JUSTINTIME_DEBUG is not specified, the COBOL Error Report will be started as the default and it performs as if "@CBR_JUSTINTIME_DEBUG=ALLERR" has been specified.

APLERR

Inspection is performed when an application error occurs.

CBLERR

Inspection is performed when a U-level runtime message is output.

ALLERR

Inspection is performed when an application error occurs and when a U-level runtime message is output.

NO

Inspection is not performed when an application error occurs and when the U-level runtime message is output.

SNAP

When start parameter is specified when the COBOL Error Report is used as a research tool, it is necessary to specify this.

start-parameter

Start parameter of The COBOL Error Report is specified.

Note

This environment variable is invalid during debugging using the debugging function of NetCOBOL Studio.

5.4.1.30 @CBR_MESSAGE(Specify the runtime message output destination)

```
                 ┌ CBLERROUT                  ┐
@CBR_MESSAGE=    │ EVENTLOG[(computer-name)]  │
                 └ ALL[(computer-name)]       ┘
```

Specify the message output destination.

CBLERROUT

Messages are output to the message box, system console (command prompt), or file.

When @MessOutFile is specified, messages are output to the file.

When @CBR_CONSOLE is specified, messages are output to the system console.

When @MessOutFile and @CBR_CONSOLE are specified simultaneously, @MessOutFile becomes valid and @CBR_CONSOLE becomes invalid.

EVENTLOG

Messages are output to the event log.

ALL

Messages are output to both the CBLERROUT output destination and event log.

The default is CBLERROUT output destination.

In computer-name, specify the name of the computer to which the event log is output.

The name of other computers in the network can be specified.

When the computer name is omitted, messages are output to the computer on which the program is running.

In the cases given below, a message that indicates the failure to output to the specified computer is output to the event log of the computer on which the program is running. Then, the output target message is output.

- When the specified computer name does not actually exist in the network.
- When Windows(x64) is not running on the specified computer.

Note

COBOL (or COBOL runtime system) must be installed in the computer to which the event log is output. If the COBOL (or COBOL runtime system) is not installed in the output destination computer, the following message is output to the event log.

```
Explanation of the event ID (nnnn) within the source (COBOL) cannot be found.  The following insert
character string is included.:  $1,$2...$n
```

nnnn: message-number

$1 to $n: Insert character string

5.4.1.31 @CBR_MESS_LEVEL_CONSOLE(Specify the level of importance of runtime messages)

$$\text{@CBR\_MESS\_LEVEL\_CONSOLE=}\quad\left\{\begin{array}{c}\text{NO}\\ \underline{\text{I}}\\ \text{W}\\ \text{E}\\ \text{U}\end{array}\right\}$$

Specify the level of importance at which the runtime system outputs runtime messages. This means you can suppress less important messages. If CBLERROUT, the destination of @CBR_MESSAGE, is not valid, this environment variable has no meaning.

The values of @CBR_MESS_LEVEL_CONSOLE and their meaning are as follows:

NO

Runtime messages are not output.

I

Runtime messages whose code is I or higher are output.

W

Runtime messages whose code is W or higher are output.

E

Runtime messages whose code is E or higher are output.

U

Runtime messages whose code is U are output.

Example

To output messages whose code is I or higher, specify:

```
@CBR_MESS_LEVEL_CONSOLE=I
```

Note

- If there is no @CBR_MESS_LEVEL_CONSOLE specification, or if its value is not one of those listed above, @CBR_MESS_LEVEL_CONSOLE=I is assumed.
- The @CBR_MESS_LEVEL_CONSOLE specification is also valid for the runtime messages output to the file specified in the environment variable @MessOutFile.
- If @CBR_MESS_LEVEL_CONSOLE=NO is specified, runtime messages that do not require user intervention are not output. However, these messages may contain information that is useful in identifying the cause of runtime errors. Take this fact into account before making this specification.
- Runtime messages that require user intervention are output even if @CBR_MESS_LEVEL_CONSOLE=NO is specified.
- If @NoMessage=YES is also specified, the @CBR_MESS_LEVEL_CONSOLE specification takes precedence.

5.4.1.32 @CBR_MESS_LEVEL_EVENTLOG(Specify the level of importance of runtime messages)

$$\text{@CBR\_MESS\_LEVEL\_EVENTLOG} = \left\{ \begin{array}{c} \text{NO} \\ \underline{\text{I}} \\ \text{W} \\ \text{E} \\ \text{U} \end{array} \right\}$$

Specify the level of importance at which the runtime system outputs runtime messages to the event log. This means you can suppress less important messages.

The parameters of @CBR_MESS_LEVEL_EVENTLOG and their meaning are as follows:

NO

Runtime messages are not output.

I

Runtime messages whose code is I or higher are output.

W

Runtime messages whose code is W or higher are output.

E

Runtime messages whose code is E or higher are output.

U

Runtime messages whose code is U are output.

Example

To output messages whose code is I or higher, specify:

```
@CBR_MESS_LEVEL_EVENTLOG=I
```

Note

- If there is no @CBR_MESS_LEVEL_EVENTLOG specification, or if its value is not one of those listed above, @CBR_MESS_LEVEL_EVENTLOG=I is assumed.
- If @CBR_MESS_LEVEL_EVENTLOG=NO is specified, runtime messages that do not require user intervention are not output. However, these messages may contain information that is useful in identifying the cause of runtime errors. Take this fact into account before making this specification.
- If @NoMessage=YES is also specified, the @CBR_MESS_LEVEL_EVENTLOG specification takes precedence.

5.4.1.33 @CBR_MEMORY_CHECK(Specify the inspection using the memory check function)

@CBR_MEMORY_CHECK=MODE1

Use the memory check function to inspect the runtime system area when executing the application. For details on how to use the memory check function, refer to "Using the Memory Check Function." in the "NetCOBOL Debugging guide".

5.4.1.34 @CBR_OverlayPrintOffset(I-Control record effects on overlay printing)

@CBR_OverlayPrintOffset= $\left\{\begin{array}{l}\text{VALID}\\ \underline{\text{INVALID}}\end{array}\right\}$

In the print file without the FORMAT clause, specify VALID or INVALID for Form Overlay. VALID validates and INVALID invalidates the binding margin direction (BIND), binding margin width (WIDTH), and origin of printing (OFFSET) function specified in the I control record. The default is INVALID. This specification is valid within the execution unit. For the specification valid for each file, refer to the topic "7.1.9 Print Information File". For the I control record, refer to the topic "7.1.5 I and S control records".

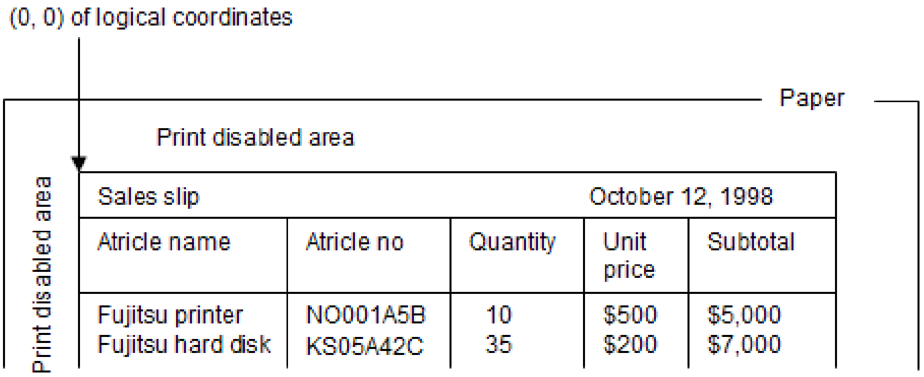
See

Specifying VALID for this runtime environment information enables the I control record function to be validated for both row record and Form Overlay.

The following figures show the differences between the VALID and INVALID (default) print results using the ordinary print results without origin (OFFSET) and print results with one-inch top and left origins (OFFSET) as examples.

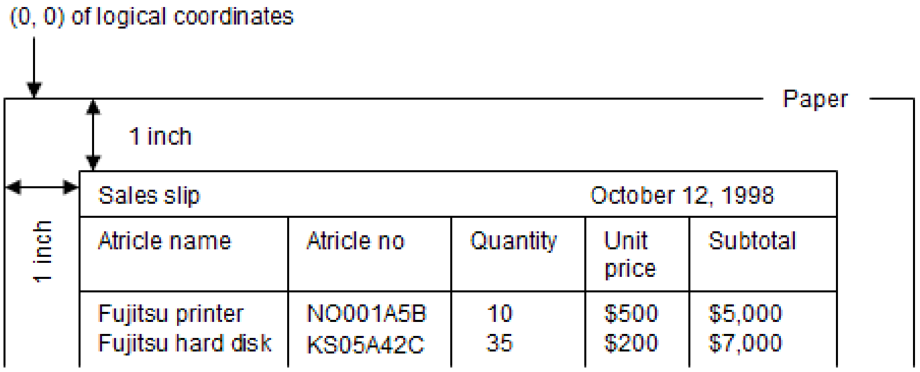
In the following figures, ruled-line indicate overlay and characters indicate line records.

- Print results without specification of the origin of printing (OFFSET)



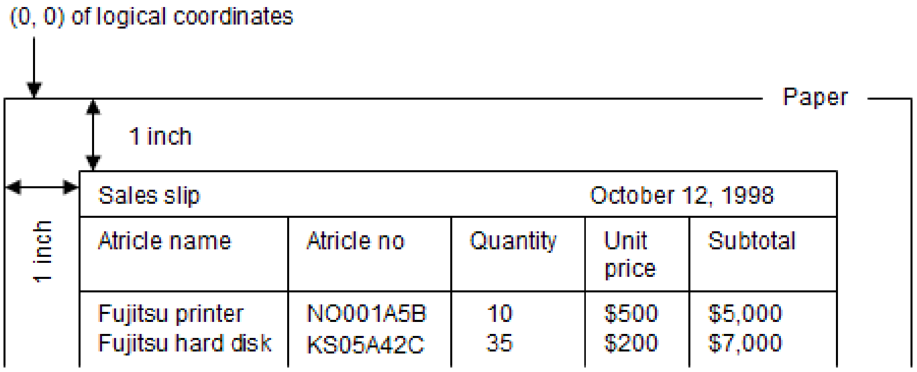
Sales slip				October 12, 1998
Atricle name	Atricle no	Quantity	Unit price	Subtotal
Fujitsu printer Fujitsu hard disk	NO001A5B KS05A42C	10 35	$500 $200	$5,000 $7,000

- Print results with one-inch origin offset on the X and Y coordinates
  - When VALID is specified



Sales slip				October 12, 1998
Atricle name	Atricle no	Quantity	Unit price	Subtotal
Fujitsu printer Fujitsu hard disk	NO001A5B KS05A42C	10 35	$500 $200	$5,000 $7,000

  - When INVALID is specified



Sales slip				October 12, 1998
Atricle name	Atricle no	Quantity	Unit price	Subtotal
Fujitsu printer Fujitsu hard disk	NO001A5B KS05A42C	10 35	$500 $200	$5,000 $7,000

5.4.1.35 @CBR_PrinterANK_Size(Specification of ANK character size)

```
                          { TYPE-M  }
@CBR_PrinterANK_Size=     { TYPE-PC }
                          { TYPE-G  }
```

Specify the ANK character size in the cases where the following are NOT defined in the program:

This specification is valid within the execution unit.

- CHARACTER TYPE phrase

- PRINTING POSITION phrase

Settings are:

TYPE-M

Default ANK character size is brought close to the size in the GS-series. Print size is 9.6 points.

TYPE-PC

Default ANK character size is assumed to be a PC standard size. Print size is 10.5 points.

TYPE-G

Default ANK character size is brought close to the size in the FMG-series. Print size is 10.8 points.

If this environment variable is omitted, the print size is 7.0 points.

Note

If a font (raster font) of a fixed size is selected as the print font, this specification may not be valid. In such a case, printing will be executed in a font closest to the size supported.

As a general example, the device font installed on a serial printer is normally good for printing in the fixed size of 10.5 point only. If such a font is selected, printing will always be made with 10.5 point irrespective of the specification of this runtime environment variable.

5.4.1.36 @CBR_PrintFontTable(Specify the font table used for a print file)

```
@CBR_PrintFontTable=font-table-name
```

Specify the file name of the font table to be used for a print file with the absolute path.

The font table specified in this parameter is enabled for all print files to be operated in the same runtime environment. For details on the specification of the font table for each file, see "5.4.1.68 File identifier(Specify the printer information file and various parameters used for the program)" and "5.4.1.69 File identifier(Specify the printer and various parameters used for the program)".

For details on the font table, see "Print Character Font" and "Print Character Style" in "7.1.2 Print Characters" and "7.1.14 Font table".

5.4.1.37 @CBR_PrintInfoFile(Specify a print information file to the file that specified PRINTER in the ASSIGN phrase)

```
@CBR_PrintInfoFile=Print-information-filename
```

Specifies a Print-information-filename for a file that specified PRINTER in the ASSIGN phrase in a print file without a FORMAT phrase.

For details of print information file, refer to "7.1.9 Print Information File".

Example

```
ASSIGN phrase of the COBOL source program:
  ASSIGN TO PRINTER
Print information file name and path:
  C:\PRINT.INF
Environment variable setting:
  @CBR_PrintInfoFile=C:\PRINT.INF
```

5.4.1.38 @CBR_PrintTextPosition(Specify method of calculating character arrangement coordinates)

```
@CBR_PrintTextPosition=    { TYPE1 }
                           { TYPE2 }
```

Specify the method of calculating coordinates (x,y) for arranging printed characters when the FORMAT clause is omitted from the file control entry.

The effective range of this specification is the execution unit.

You specify whether (TYPE2) or not (TYPE1) to correct the character arrangement coordinates.

TYPE1

> The resolution of the printer to be used for printing is divided by the line spacing (LPI) or character pitch (CPI) specified by the application, then the remainder is discarded. The character pitch is determined based on the resulting value.

TYPE2

> Just as in cases where TYPE1 is specified, the resolution of the printer to be used for printing is divided by the line spacing (LPI) or character pitch (CPI) specified by the application. The character pitch is determined and placed based on the quotient. When data is output to a printer that has an indivisible resolution, the coordinates are corrected within units of one inch.

Note

In the previous version, the default of this specification is TYPE1. But in this version, the default is TYPE2. If problems, such as a gap between forms overlay and characters during overlay print, are caused by this default, explicitly specify TYPE1 as the runtime environment information.

5.4.1.39 @CBR_PSFILE_xxx(Set the Connected Product Name Used from the Presentation File (by Destination))

```
@CBR_PSFILE_xxx=connected-product-name
```

For xxx, specify the destination name described in the SYMBOLIC DESTINATION clause of the presentation file. xxx can be one of the following:

```
PRT
```

For the connected-product-name enter a string indicating the associated product name to be used. The following table shows the strings that can be specified.

Table 5.4 Supported @CBR_PSFILE strings

Specification in SYMBOLIC DESTINATION Clause	Product Used	String Specified
PRT	FORM RTS	omitted

Note

- If a connected product name is specified as a file identifier in the presentation file the specified connected product name is not effective. Refer to "5.4.1.67 File Identifier(Set the Information File and the Connected Product Name Used from the Presentation File (by File))" in this section.

5.4.1.40 @CBR_SCR_KEYDEFFILE(Specify key definition file for screen handling)

```
@CBR_SCR_KEYDEFFILE=key-definition-file
```

When the user defined function keys are used by the screen handling function, specifies the file name in which the function keys are defined.

A relative path name or an absolute path name can be specified for the file name. When the relative path name is specified, the file name becomes a relative path from the current folder.

5.4.1.41 @CBR_SCR_HIGHLIGHT(Screen item highlight style)

```
@CBR_SCR_HIGHLIGHT=BRIGHT
```

When this environment variable is specified, screen items with the HIGHLIGHT clause are displayed in a bright style. When this environment variable is not specified, screen items with the HIGHLIGHT clause are displayed in a bold style.

5.4.1.42 @CBR_SCREEN_POSITION(Specify the position of Screen window)

@CBR_SCREEN_POSITION= $\left\{\begin{array}{l}\underline{\text{CENTER}}\\ \text{SYSTEM}\\ \text{(x, y)}\end{array}\right\}$

It is specified the position of Screen window displayed by Screen feature.

If this environment variable is omitted, "CENTER" is specified implicitly.

CENTER

Screen Window is displayed at center of DISPLAY terminal.

SYSTEM

Screen Window is displayed at system's default position.

(x, y)

Screen Window is displayed at the position specified using x and y. x and y are specified numeric.

5.4.1.43 @CBR_SYSERR_EXTEND(Specify the SYSERR output information extension)

```
@CBR_SYSERR_EXTEND=YES
```

The process ID and thread ID information is appended to the DISPLAY statement output to SYSERR.

5.4.1.44 @CBR_TextAlign(Specify alignment of print characters with either top or bottom of line)

@CBR_TextAlign= $\left\{\begin{array}{l}\text{TOP}\\ \underline{\text{BOTTOM}}\end{array}\right\}$

In the print file without the FORMAT clause, specify the print character position to be aligned on the line. This specification is valid within the execution unit.

Specify whether to align characters to be printed to the top (TOP) on the line with the upper left of the character cell as the base point or bottom (BOTTOM) with the lower left as the base point.

5.4.1.45 @CBR_TRACE_FILE(Specify the trace information output file)

```
@CBR_TRACE_FILE=file-name
```

When the TRACE function is used, specify the name of the file to which the trace information is output. Refer to "Using the TRACE Function" in the "NetCOBOL Debugging Guide".

The absolute path or relative path can be specified as the file name. When the relative path is specified, the relative path is from the current folder.

When an extension is specified with the file name, the extension is ignored and is replaced with TRC or TRO.

When the environment variable information is not specified, the information is stored in the file that has the executable file name appended with the extension TRC or TRO.

5.4.1.46 @CBR_TRACE_PROCESS_MODE(Unique file name foe each Trace file output)

@CBR_TRACE_PROCESS_MODE = MULTI

The runtime will create a trace file name that is ensured to be unique when the same executable is run in different processes at the same time.

The file name generated when this environment variable is specified is as follows.

```
name-processID_date(YYYYMMDD)_time(HHMMSS).TRC or TRO
```

Using environment variable @CBR_TRACE_FILE, the file name when specifying the trace information output place is as follows.

```
DestinationFileName-ProcessID_Date(YYYYMMDD)_Time(HHMMSS).TRC or TRO
```

5.4.1.47 @CBR_TRAILING_BLANK_RECORD(Specify whether to remove or enable the trailing blank in the record of line sequential file)

@CBR_TRAILING_BLANK_RECORD= $\left\{\begin{array}{l}\text{REMOVE}\\ \underline{\text{VALID}}\end{array}\right\}$

Specify whether to remove (REMOVE) or enable (VALID) the trailing blank in the record when executing the WRITE statement of the line sequential file.

If this specification is omitted, "VALID" is assumed as specified.

Note

The following types of space characters are deleted depending on the code system:

- If the code system is Unicode and the character type is alphanumeric characters, en-size space characters are deleted.
- If the code system is Unicode and the character type is national characters, em-size space characters are deleted.
- If the code system is other than Unicode, en-size and em-size space characters are deleted.

5.4.1.48 @CnslBufLine(Set the Buffer Count for the Console Window)

@CnslBufLine= $\left\{\begin{array}{l}\text{buffer-lines}\\ \underline{\text{100}}\end{array}\right\}$

Specify the number of buffers for the console window used by the ACCEPT/DISPLAY function. Specify buffer lines in the range from 1 to 9999.

If (console window columns + 1) * (buffer lines) exceeds 65000, the specified value is decreased. For example, buffer-lines can be up to 802 when the console window columns is 80; and 706 when 91.

Note

This specification is ignored when the system console is used. See "5.4.1.6 @CBR_CONSOLE(Set the type of the console window)".

5.4.1.49 @CnslFont(Set the Console Window Font)

@CnslFont=(font-name, font-size)

Specify the font of the console window used for the ACCEPT/DISPLAY function.

Note

- This specification is ignored when the system console is used. See "5.4.1.6 @CBR_CONSOLE(Set the type of the console window)".
- Specify a monospaced font for the font-name. Proportional fonts do not function correctly.
- When you use a surrogate pair character, specify a font that supports the particular characters that are present in the data stream.
- Neither the input nor the display of control characters is supported.

5.4.1.50 @CnslWinSize(Set the Size of the Console Window)

@CnslWinSize= { (columns,lineage) | (80,24) }

Specify the size of the console window used for the ACCEPT/DISPLAY function. Specify the number of columns and lines in the range from 1 to 999. The minimum and maximum values of the window size are the system values. If a value outside the system value range is specified, it is adjusted to the system minimum or maximum value.

Note

This specification is ignored when the system console is used. See "5.4.1.6 @CBR_CONSOLE(Set the type of the console window)".

5.4.1.51 @DefaultFCB_Name(Specification of default FCB name)

@DefaultFCB_Name=FCBxxxx

Specify the default FCB name. The FCB name specified here must be defined in the FCB control statement, as well as the case to change FCB from I control record dynamically.

The example below demonstrates setting the default FCB name to "6LPI" which has a line interval of 6 lines per inch, 66 lines per page, starts printing at line 1, and is 11 inches long. The default FCB name in this example is "FCB6LPI".

Example

```
@DefaultFCB_Name=FCB6LPI
FCB6LPI=LPI((6,66)),CH1(1),SIZE(110)
FCB8LPI=LPI((8,88)),CH1(1),SIZE(110)
FCBA4LD=LPI((12)),CH1(1),FORM(A4,LAND)
```

5.4.1.52 @ExitSessionMSG(Set message if Windows system is closed)

@ExitSessionMSG= { ON | OFF }

Specifies whether a message is displayed if the Windows system, in which the COBOL application was started, is itself closed.

With @ExitSessionMSG=ON, a warning message is displayed telling the user that a COBOL application is executing.

With @ExitSessionMSG=OFF the COBOL application terminates at the same time as the Windows system, with no warning message.

Note

When you use OFF, the application will be terminated even if the COBOL program is in the middle of an I-O operation using file or other resources. This may cause the corruption of the resources being used, so take care when using this option.

5.4.1.53 @GOPT(Set Runtime Options)

@GOPT=list-of-runtime-options

Specify runtime options as a list. For the format of the runtime options, refer to "5.7 Format of Runtime Options".

Example

```
@GOPT=r20 c20
```

Information

When a program is executed from the command line, the runtime option can be specified by argument.

If the runtime option specified by argument overlaps with @GOPT, the argument is given priority.

5.4.1.54 @IconDLL(Set the DLL name of an Icon Resource)

@IconDLL=DLL-name-of-icon-resource

Specify the DLL name containing an icon resource to be displayed.

To display the icon resource in the specified DLL, it is necessary to specify the icon resource name in the environment variable @IconName.

An absolute or relative path can be used for the DLL name of the icon resource. If a relative path is used, the DLL will be retrieved according to the following retrieval sequence:

1. Folder containing the executable file
2. Current folder
3. Windows system folder
4. Windows folder
5. Folder specified in the environment variable path

If @IconDLL is not specified, the icon resource specified in @IconName is retrieved from the executable files in the main program. See also "5.4.1.55 @IconName(Set the Identifier of an Icon Resource)".

Note

During operation in multithread mode, the icon is not displayed unless compilation is performed with the MAIN compiler option specified.

5.4.1.55 @IconName(Set the Identifier of an Icon Resource)

@IconName=icon-resource-identifier

To change the icon, specify the icon resource identifier name.

The icon resource to be displayed must be linked to an application in advance or a DLL that contains the icon resource must be created.

To display the icon resource in the DLL, the DLL name that contains the icon resource must be specified in the environment variable information @IconDLL. Refer to "5.4.1.54 @IconDLL(Set the DLL name of an Icon Resource)".

Note

During operation in multithread mode, the icon is not displayed unless compilation is performed with the MAIN compiler option specified.

5.4.1.56 @MessOutFile(Set a Message Output File)

@MessOutFile=file-name

The runtime messages and DISPLAY statement with UPON SYSERR specified are output to the specified file. When a file name is specified, the message box is not displayed on the screen.

For details on handling with the Unicode, refer to "20.3.4 Execution".

Please specify @MessOutFile when you output the message at the program execution of operation under the control of various application servers. Refer to "Chapter 18 Server Type Applications".

Note

- Absolute and relative paths can be specified for the file name. Relative paths are relative to the current folder.
- If a file of the specified name already exists, the information will be appended to that file.
- If the file specified in @MessOutFile is the same as the output file of an I-O function, the contents of the file is not guaranteed.
- When it is not possible to output it to the file, the execution time message is output to the message box.
- The maximum size of the file is up to a limitation of the system.

5.4.1.57 @MGPRM(Set the GS-series Format Runtime Parameter)

@MGPRM="string of runtime parameter"

Specify the character string to be passed to the program within double quotation marks ("). The specified string is passed to other programs in the same manner as the program is executed on the system of the GS-Series.

The character string can be up to 100 bytes.

For more information about the GS-Series format runtime parameter, refer to "Appendix F OSIV-series Function Comparison".

Example

```
@MGPRM="ABCDE"
```

Information

When a program is executed from the command line, a GS-series format runtime parameter can be specified by argument.

If the runtime parameter specified by argument overlaps with @MGPRM, the argument is given priority.

5.4.1.58 @NoMessage(Set to Suppress Runtime Messages)

@NoMessage=YES

Suppresses the following runtime messages:

- Runtime messages other than U level
- Runtime messages not requiring operator response
- SYSERR of DISPLAY statement

To suppress the message displayed when the window closes, use environment variable @WinCloseMsg. Refer to "5.4.1.65 @WinCloseMsg(Set a Display Message When the Window Close)".

5.4.1.59 @ODBC_Inf(Set the ODBC Information File)

```
@ODBC_Inf=ODBC-information-file-name
```

Specify the name of the ODBC information file containing information required by the runtime system to use ODBC. Information in this file is mainly used to connect the client and server (with the CONNECT statement).

5.4.1.60 @PrinterFontName(Set the Font Used for Print Files)

```
@PrinterFontName=(Minchou-font-name, Gothic-font-name)
```

Specify the font used for print files.

Note

- Specify the typeface name of the font by alphanumeric characters within 32 bytes.
- Spaces before and after the typeface name of the font are included in the name.
- The typeface name of the font containing comma "," cannot be specified.
- Select the control panel font for the font name specifiable for the typeface name of the font and select the font name from a list of fonts to be displayed.
- The character code set of initialization file (COBOL85.CBR) supports UTF-8. Please make the initialization file with UTF-8, and operate it by the Unicode program when you specify the font name that contains the multi byte character. However, the specification of the font name including the multi byte character is enabled with SJIS in a Japanese environment.

5.4.1.61 @PRN_FormName_xxx(Specify paper name)

```
@PRN_FormName_xxx=paper-name
```

In a print file without the FORMAT clause, a string of three characters or less specified in the SIZE (paper size) field of the I control record is replaced with the xxx part of this environment variable name. A relationship is established with the paper name that corresponds to this environment variable name.

The paper name indicates the paper name displayed in the list box on the right side of each Paper Feed Method in "Paper Feed Method and Paper Allocation" on the Device Options tab of the Properties dialog of each printer.

Note

- For the paper name, the support range differs with the printer or printer driver used. Check the paper size supported by the printer driver in the print manager or printer property or refer to the printer handbook.
- To specify the new form, the system must be restarted after the new form is created. Note that the created new form may not be valid unless the system is restarted.
- The character code set of initialization file (COBOL85.CBR) supports UTF-8. Please make the initialization file with UTF-8, and operate it by the Unicode program when you specify the font name that contains the multi byte character. However, the specification of the font name including the multi byte character is enabled with SJIS in a Japanese environment.

5.4.1.62 @ScrnFont(Set the Font Used for Screen Handling)

```
@ScrnFont=(font-name, font-size)
```

Specify the font used for screen handling.

5.4.1.63 @ScrnSize(Set the Size of the Logical Screen for Screen Handling)

```
@ScrnSize=   { (columns, lineage) }
             { (80,25)            }
```

Specify the logical screen size of the window used by the screen handling function. Specify columns and lineage in the range from 1 to 999. This logical screen size defaults to the physical screen size.

If (columns + 1) * lineage exceeds 16250, an error occurs upon execution of the program.

5.4.1.64 @ShowIcon(Specifying the suppression of COBOL icon display)

```
@ShowIcon=   { YES }
             { NO  }
```

Specify whether to display the COBOL icon on the task bar (YES) or not (NO).

Note

- If the icon display is suppressed, the application name is not output on the system task manager, hence losing the means for forcibly ending the application concerned.
  If the icon display is suppressed, the specification in the environment variable information @ExitSessionMSG becomes invalid. When the Windows system terminates during execution of the COBOL program, a system warning message is output. When the system warning message is output and the program is forcibly terminated, the results of I-O processing performed by the forcibly terminated program are not guaranteed.

- During operation in multithread mode, the icon is not displayed unless compilation is performed with the MAIN compiler option specified.

5.4.1.65 @WinCloseMsg(Set a Display Message When the Window Close)

```
@WinCloseMsg=   { ON  }
                { OFF }
```

Specify whether a confirmation message is displayed (ON) or not (OFF) when the console window used by the ACCEPT/DISPLAY function or the window used by the screen handling function closes.

5.4.1.66 File identifier(Set the File Used by the Program)

```
File-identifier=
filename [, access-type or file-system-type ] [, MOD ] [, CONCAT (filename-1 filename-2 ...) ] [, DUMMY ]
```

Specify the file identifier entered in the ASSIGN clause of the COBOL source program as the file identifier, and the name of the file to be processed as file-name.

This specification relates the logical file name defined in the program with the actual physical file to be processed.

Absolute and relative paths can be specified for the file name. If the relative path is specified, it will be from the current folder.

Specify runtime environment information in uppercase letters even if the file identifier was defined in lowercase letters in the program.

Example

```
Description of the ASSIGN clause in the COBOL source program
  ASSIGN TO OUTFILE
Name of file processed
  F:\WORK.DAT
Runtime environment information
  OUTFILE=F:\WORK.DAT
```

Access type or file system type can be the following strings depending on the type used.

BSAM

Provides fast file processing. Refer to "6.7.4 High-Speed File Processing".

BTRV

Processes the Btrieve file. Refer to the topic "6.9.1 Btrieve Files".

MOD

Adds information to a file. For more information on the MOD option see "6.7.5 Appending to Existing Files".

CONCAT

Connects files. For more information on the CONCAT option see "6.7.6 Concatenating Files".

EXFH

Process the External File Handler. Refer to the topic "6.9.2 External File Handler".

DUMMY

Process the dummy file. For more information on the DUMMY option see "6.7.7 Dummy File".

For combinations that can be specified and cautions, refer to "6.7.8 Caution".

5.4.1.67 File Identifier(Set the Information File and the Connected Product Name Used from the Presentation File (by File))

File-identifier=[information-filename] [,connected-productname]

How to specify the file identifier for using the presentation file is explained below.

Specify the file identifier entered in the ASSIGN clause of the COBOL source program.

Specify runtime environment information in upper-case letters even if the file identifier was entered in lower-case letters in programs.

When using FORM RTS, specify the FORM RTS window information file name or printer information file name used as data file name. When using PowerFORM RTS, specify the PowerFORM RTS printer information file name used as data file name.

Specify a string indicating the connected product name actually used as the connected product name. For details on the character string indicating the connected product, see Table "Supported @CBR_PSFILE strings" If omitted, the value specified in @CBR_PSFILE_xxx is effective. Refer to "5.4.1.39 @CBR_PSFILE_xxx(Set the Connected Product Name Used from the Presentation File (by Destination))". If the connected product name in the runtime environment information is also omitted, the default destination in the presentation file is assumed.

Example

```
Description of the ASSIGN clause in the COBOL source program
  ASSIGN TO GS-DSPFILE
SYMBOLIC DESTINATION clause
  SYMBOLIC DESTINATION IS "DSP"
Name of file data actually used
  F:\WORK.WRC
Connected product actually used
```

```
  FORM RTS
Runtime environment information
  DSPFILE=F:\WORK.WRC,MEFT
```

5.4.1.68 File identifier(Specify the printer information file and various parameters used for the program)

file-identifier=printer-information-file-name[,[,FONT(font-table-name)]]

How to specify the file identifier when using the print file with the FORMAT clause is explained below.

Specify the file identifier written in the ASSIGN clause of the COBOL source program for the file identifier.

Even if the file identifier is written in lower-case letters in the program, specify the file identifier of the environment variable information in upper-case letters.

Specify the printer information file name of FORM RTS or PowerFORM RTS to be actually used for the printer information file.

Specify the font table file name with the absolute path for the font table name. For the font table, define the typeface name of the font and print style information corresponding to the FONT-nnn (font number) specification of the PRINTING MODE clause. The font table specified in this parameter is enabled only for the specified file. For details on the specification of font table (font table enabled for each runtime environment) common to all print files, see "5.4.1.36 @CBR_PrintFontTable(Specify the font table used for a print file)".

For details on the font table, see "Print Character Font" and "Print character style" in "7.1.2 Print Characters" and "7.1.14 Font table".

If the font table name enabled for each runtime environment is different from the font table name enabled for each file, the specification of the font table name enabled for each file takes priority.

Example

Example 1

```
[ASSIGN clause in the COBOL source program]
  ASSIGN TO PRTFILE
[Name of the printer information file to be actually used]
  C:\WORK.PRC
[Environment variable information]
  PRTFILE=C:\WORK.PRC
```

Example 2

```
[ASSIGN clause in the COBOL source program]
  ASSIGN TO PRTFILE
[Name of the printer information file to be actually used]
  C:\WORK.PRC
[Name of the font table to be actually used]
  C:\DEFAULT.FTB
[Environment variable information]
  PRTFILE=C:\WORK.PRC,,FONT(C:\DEFAULT.FTB)
```

5.4.1.69 File identifier(Specify the printer and various parameters used for the program)

file-Identifier = { LPTn: | COMn: | PRTNAME:printer-name } [,[,INF(P-name)] [,FONT(F-name)]]

- P-name : Print information file name

- F-name : Font tabel name

How to specify the file identifier for using the print file without the FORMAT clause is explained below.

Specify the file identifier written in the ASSIGN clause of the COBOL source program for the file identifier. Even if the file identifier is written in lower-case letters in the program, be sure to specify the file identifier of the environment variable information in upper-case letters.

Specify the name (LPTn:) of the local printer port to which the printer that actually outputs forms is connected, communication port name (COMn:), and printer name (PRTNAME: Printer name) for "LPTn:"/"COMn:"/"PRTNAME:printer-name." The specifications of local printer port name (LPTn:), communication port name (COMn:), and printer name (PRTNAME: Printer name) can be omitted. If omitted, the above must be specified with the "PRTOUT" key of the print information file. If different local printer port name (LPTn:), communication port name (COMn:), and printer name (PRTNAME: Printer name) are used for each of the file identifier and print information file, the specification for the print information file takes priority. If both specifications are omitted or the specification is incorrect, a runtime error occurs.

Specify the print information file name with the absolute path for the print information file name. For the print information file, define some information items for controlling the status related to the forms to be output. For details on the print information file, see "7.1.9 Print Information File".

Specify the font table file name with the absolute path for the font table name. For the font table, define the typeface name of the font and print style information corresponding to the FONT-nnn (font number) specification of the PRINTING MODE clause. The font table specified in this parameter is enabled only for the specified file. For details on the specification of the font table (enabled for each runtime environment) common to all print files, see "5.4.1.36 @CBR_PrintFontTable(Specify the font table used for a print file) ".

For details on the font table, see "Print Character Font" and "Print character style" in "7.1.2 Print Characters" and "7.1.14 Font table".

If the font table name enabled for each runtime environment is different from the font table name enabled for each file, the specification of the font table name enabled for each file takes priority.

The specifications of the print information file name and font table name can be in any order.

Example

Example 1

```
[ASSIGN clause in the COBOL source program]
  ASSIGN TO PRTFILE
[Name of the printer information that actually output forms]
  FUJITSU VSP4620
[Name of the print information file to be actually used and the path]
  C:\PRINT.INF
[Font table to be actually used and the path]
  C:\DEFAULT.FTB
[Environment variable information]
  PRTFILE=PRTNAME:FUJITSU VSP4620,,INF(C:\PRINT.INF),FONT(C:\DEFAULT.FTB)
  or
  PRTFILE=PRTNAME:FUJITSU VSP4620,,FONT(C:\DEFAULT.FTB),INF(C:\PRINT.INF)
```

Example 2

```
[ASSIGN clause in the COBOL source program]
  ASSIGN TO PRINTER-1
[Port name of the local printer that actually outputs forms]
  LPT1:
[Name of the print information file to be actually used and the path]
  Not used.
[Font table to be actually used and the path]
  C:\DEFAULT.FTB
[Environment variable information]
  PRINTER-1=LPT1:,,FONT(C:\DEFAULT.FTB)
```

Example 3

```
[ASSIGN clause in the COBOL source program]
  ASSIGN TO PRINTER-2
[Port name of the local printer that actually outputs forms]
  COM1:
[Name of the print information file to be actually used and the path]
  Not used.
[Font table to be actually used and the path]
  Not used.
[Environment variable information]
  PRINTER-2=COM1:
```

Example 4

```
[ASSIGN clause in the COBOL source program]
  ASSIGN TO PRTFILE
[Port name of the local printer that actually outputs forms]
  FUJITSU VSP4620
[Name of the print information file to be actually used and the path]
  C:\PRINT.INF
[Font table to be actually used and the path]
  Not used.
[Environment variable information]
  PRTFILE=,,INF(C:\PRINT.INF)
[Print information file (C:\PRINT.INF) specification]
  PRTOUT=PRTNAME:FUJITSU VSP4620
```

Example 5

```
[ASSIGN clause in the COBOL source program]
  ASSIGN TO PRTFILE
[Port name of the local printer that actually outputs forms]
  FUJITSU VSP4620A
[Name of the print information file to be actually used and the path]
  C:\PRINT.INF
[Font table to be actually used and the path]
  Not used.
[Environment variable information]
  PRTFILE=PRTNAME:FUJITSU VSP4620A,,INF(C:\PRINT.INF)
[Print information file (C:\PRINT.INF) specification]
  PRTOUT=PRTNAME:FUJITSU VSP4620A
```

Note

The character code set of initialization file (COBOL85.CBR) supports UTF-8. Please make the initialization file with UTF-8, and operate it by the Unicode program when you specify the font name that contains the multi byte character. However, the specification of the font name including the multi byte character is enabled with SJIS in a Japanese environment.

5.4.1.70 FCBxxxx(Set FCB Control Statements)

FCBxxxx=FCB-control-statement

The string "xxxx" is the FCB name specified in the I control record. For the format of the FCB control statement refer to "7.1.4 Forms Control Buffers(FCB)".

5.4.1.71 FOVLDIR(Set the Folder Containing Form Overlay Patterns)

FOVLDIR=folder-name

Specify the folder containing form overlay patterns by using an absolute path name. If omitted, no form overlay patterns are printed. Only one folder can be specified in folder name.

5.4.1.72 FOVLTYPE(Set the Format of the Form Overlay Pattern File)

FOVLTYPE=format

Specify the first four characters of the form overlay pattern file name if they are not "KOL5" (default). If there is no format in the name, specify "None".

5.4.1.73 FOVLNAME(Specifying Form overlay pattern file names)

FOVLNAME=Filename

Omit the 4 starting characters of the format part of the overlay pattern file name. Specify the second half of the file name using a maximum of 4 characters.

Example

```
When the form overlay pattern file is "C:\FOVLDATA\KOL5FOVL.OVD"
  FOVLDIR=C:\FOVLDATA
  FOVLTYPE=KOL5
  FOVLNAME=FOVL
  OVD_SUFFIX=OVD
```

5.4.1.74 OVD_SUFFIX(Set the Extension of the Form Overlay Pattern File)

OVD_SUFFIX=extension

Specify an extension string in extension if the default extension of the form overlay pattern file "OVD" is not used. When the file name has no extension, specify "None".

5.4.1.75 SYSCOUNT(Specify the output file for COUNT information)

SYSCOUNT=filename[,MOD]

To use the COUNT function, specify the name of a file that will be the destination of the COUNT information. Refer to "Using the COUNT Function" in the "NetCOBOL Debugging Guide".

Absolute and relative paths can be specified in the file name. If the relative path is specified, it will be from the current folder.

If the file name or folder name contains a comma (,), it must be enclosed in double quotation marks (").

If a character string other than MOD is specified for the second argument, the JMP0726I-W message is output. No COUNT information is output at this time.

If MOD is not specified and an existing file has the name specified for file-name, the file is overwritten. If no existing file has the name specified for file-name, a new file is created.

If MOD is specified and an existing file has the name specified for file-name, information is added to the file. If no existing file has the name specified for file-name, a new file is created.

The upper limit of the COUNT information file size is 1 GB. If information is added to a COUNT information file whose size has reached 1 GB, the JMP0727I-W message is output and the added information is not included in the COUNT information.

5.4.1.76 SYSIN Access Name(Set the Input File for the ACCEPT/DISPLAY Function)

SYSIN-access-name=filename [, DUMMY]

In SYSIN-access-name, specify the environment variable information name specified in the compiler option SSIN. In file-name, specify the data input destination file name for when the ACCEPT statement of the small I-O function is executed. Refer to "A.2.49 SSIN(ACCEPT statement data input destination)".

The absolute path or relative path can be specified as the file name. When the relative path is specified, the relative path is from the current folder. If the same file already exists, the file is overwritten.

Example

```
Compiler option SSIN
  SSIN(INFILE)
Name of input file of ACCEPT statement
  A:\INDATA.TXT
Runtime environment information
  INFILE=A:\INDATA.TXT
```

The specified character string has the following meaning:

DUMMY

Process the dummy file.

Refer to "10.1.6.5 File Input Extension Function for the ACCEPT Statement"..

5.4.1.77 SYSOUT Access Name(Set the Output File for the ACCEPT/DISPLAY Function)

SYSOUT-access-name=filename [, MOD] [, DUMMY]

SYSOUT-access-name is the environment variable name specified in the compiler option SSOUT. Specify the name of the file used as the data output destination when the DISPLAY statement of the ACCEPT/DISPLAY function is run. Refer to "A.2.50 SSOUT(DISPLAY statement data output destination)".

Absolute and relative paths can be specified in the file name. If the same file exists, the file is overwritten.

Example

```
Compiler option SSOUT
  SSOUT(OUTFILE)
Name of output file of DISPLAY statement
  A:\OUTDATA.TXT
Runtime environment information
  OUTFILE=A:\OUTDATA. TXT
```

The specified character string has the following meaning:

MOD

Adds data to an existing file.

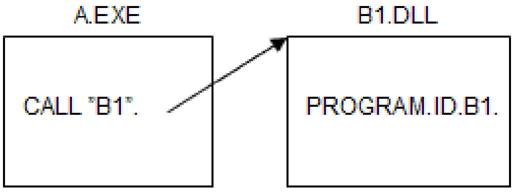
DUMMY

Process the dummy file.

Refer to "10.1.6.4 File Output Extension Function for the DISPLAY Statement".

5.4.2 Entry Information for Subprograms

For programs having a dynamic program structure, the entry information is required to specify the DLL filename that includes the called program. However, the entry information may be omitted, if the DLL file name is "program-name.DLL" as follows:



For details of dynamic program structure, see the topic "4.3.2 Dynamic Linkage".

5.4.2.1 Method of specifying Entry Information

There are two methods to specify the entry information as follows:

- The entry information is specified in an entry information file.
- The entry information is specified in a runtime initialization file.

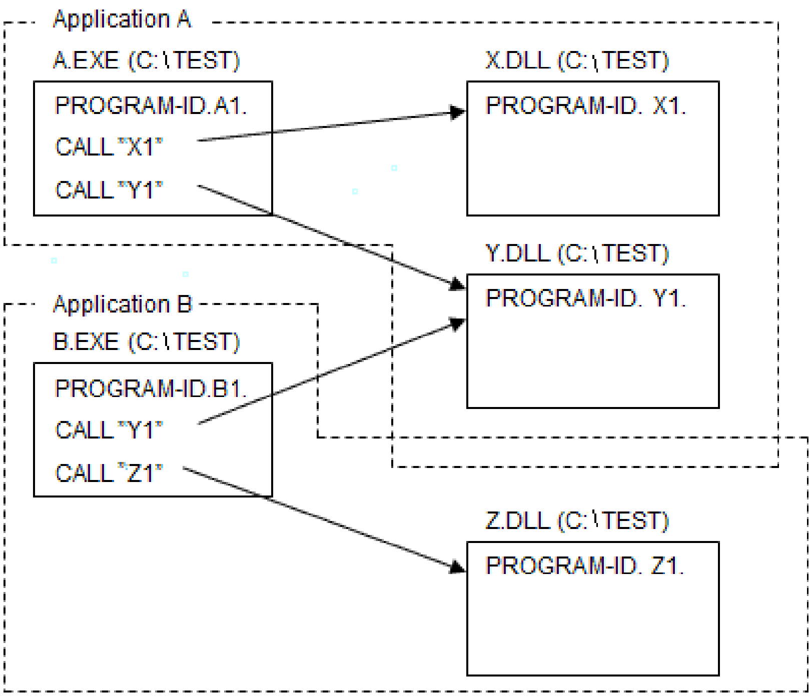
The entry information is recommended to be specified in an entry information file so as to share the entry information of common DLL file where called by any applications.

When the specification of entry information is duplicated in the runtime initialization file and the entry information file, the specification in the runtime initialization file takes precedence.

Note

In the multithread mode, the specification of entry information in the runtime initialization file is invalid. Therefore the entry information must be specified in an entry information file.

An explanation is given in the following examples. Application A and application B both call X.DLL, Y.DLL, and Z.DLL by the dynamic structure. Y.DLL is common DLL in this called from application A and application B.



Method of specification in Entry Information File

When the entry information file is used, it is necessary to specify the entry information file name for environment variable @CBR_ENTRYFILE. Refer to "5.4.1.20 @CBR_ENTRYFILE(Set the Entry Information File)".

The entry information file name : C:\TESTFILE.ENT

```
@CBR_ENTRYFILE=C:\TESTFILE.ENT
```

The content of the entry information file(C:\TESTFILE.ENT) is shown below.

The entry information file is a text file that starts with a section label "[ENTRY]":

```
[ENTRY]                                  ... (1)
Entry information                        ... (2)
```

- (1) The entry information file has only the single entry section. The entry information defines the mappings of subprograms and entry points to DLL files. Subprograms are mapped to DLL file names. Entry points are mapped to subprogram names. The formats for these mappings are described below.

- (2) Entry information is specified. Refer to "5.4.2.2 Entry Information Format - Subprograms" and "5.4.2.3 Entry Information Format - Secondary Entry Points"

The entry information is case sensitive.

A line that begins with a semi-colon (;) is regarded as a comment. If there are many comments, processing speed may be slowed down because the program must actively skip the comments.

The content of the entry information file is as follows for the above example.

The entry information file (C:\TESTFILE.ENT)

```
[ENTRY]
X1=C:\TEST\X.DLL
Y1=C:\TEST\Y.DLL
Z1=C:\TEST\Z.DLL
```

The content specified for the [ENTRY] section is effective for all calling applications. In other words, although only defined once, both applications, A and B, use the same entry information for Y.DLL.

Note

- The upper-case letters and the lower-case letters are significant in entry information.

- Do not specify the same subprogram name more than once. Program operation in that case is not guaranteed.

Method of specification in the initialization file of the runtime

Refer to "5.3.2.3 How to Set the Runtime Initialization File" for the content and the use of the initialization file. Here, only the method of specifying entry information is explained.

The runtime initialization file (COBOL85.CBR)

```
[PROGRAM-NAME]                      ---- (1)
[PROGRAM-NAME.ENTRY]                ---- (2)
ENTRY INFORMATION                   ---- (2)
```

- (1) The program name of the main program(name specified for the PROGRAM-ID paragraph) is specified as a section name.

- (2) The name of an entry point, followed by the literal, ".ENTRY" is specified as a section name. Entry information immediately follows.

The content of the initialization file for the above example is shown below:

The initialization file of the runtime (C:\TEST\COBOL85.CBR)

```
[A1]
   :
[A1.ENTRY]
X1=C:\TEST\X.DLL
Y1=C:\TEST\Y.DLL                         ---- (1)
```

```
    :
[B1]
    :
[B1.ENTRY]
Y1=C:\TEST\Y.DLL                     ---- (1)
Z1=C:\TEST\Z.DLL
```

- (1) It is necessary to specify Y.DLL in two places.

The section is specified in each application, and entry information for the application is specified under the section. In other words it is necessary to define entry information for a DLL used by two or more applications under each application.

5.4.2.2 Entry Information Format - Subprograms

Map subprogram names to the DLL file name containing the subprogram using the entry:

```
Subprogram-name=DLL-filename
```

Subprogram-name is related to the DLL file name that includes this subprogram.

Subprogram-name is the name of a program (name specified for the PROGRAM-ID paragraph) that is called.

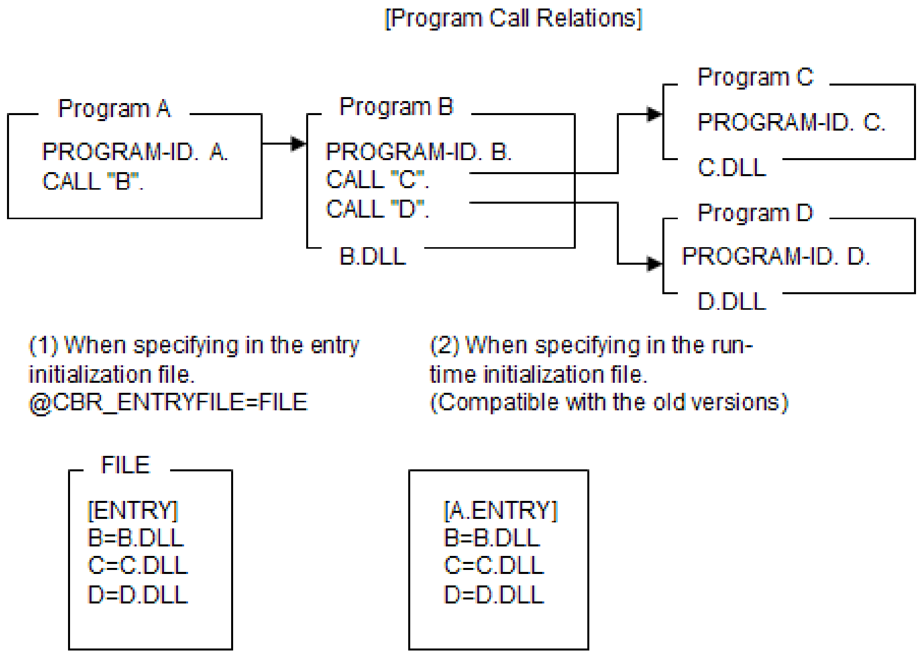
DLL-file-name is the absolute or relative path name of the DLL file that contains the subprogram. If a relative path is used, the following folders are searched in this sequence:

1. Folder containing the executable file
2. Current folder
3. Windows system folder
4. Windows folder
5. Folders specified in the environment variable PATH

The extension of the DLL file must be ".DLL".

Example

Each program contained in a separate DLL



(1) When specifying in the entry initialization file.
@CBR_ENTRYFILE=FILE

```
[ENTRY]
B=B.DLL
C=C.DLL
D=D.DLL
```

(2) When specifying in the run-time initialization file.
(Compatible with the old versions)

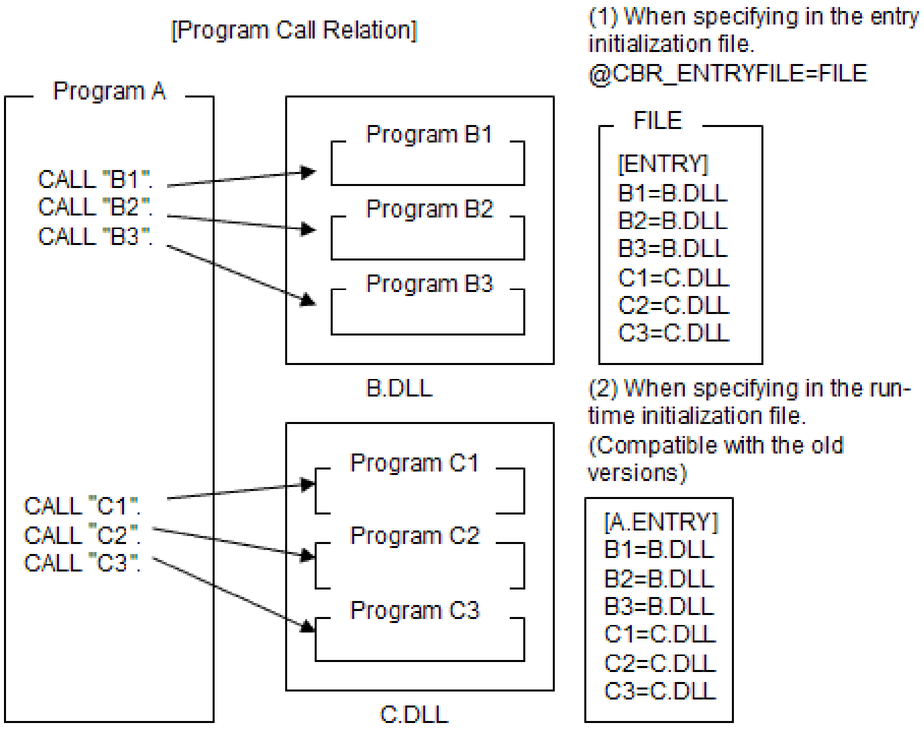
```
[A.ENTRY]
B=B.DLL
C=C.DLL
D=D.DLL
```

Information

In this example, the DLL file names are in the form
"Program name. DLL", so the entry information can be omitted.

Example

DLL's containing multiple subprograms



Information

CANCEL Behavior

The DLL is only deleted from (virtual) memory when a CANCEL statement is executed on all of the called subprograms in the DLL. In the above example, B. DLL is deleted from virtual memory after each of the subprograms B1, B2 and B3, have been canceled. Likewise, C. DLL is deleted from virtual memory when a CANCEL statement is executed for each of the subprograms C1, C2 and C3.

See "19.10.3 Dynamic structure" for the use of CANCEL statement in multithread mode.

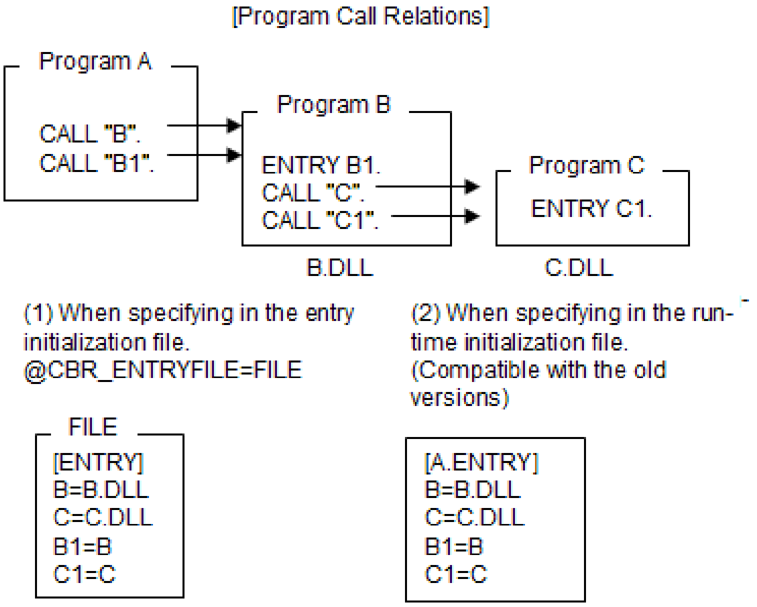
When a secondary entry point name is specified, add the specification of "5.4.2.3 Entry Information Format - Secondary Entry Points" to the subprogram name specification format.

5.4.2.3 Entry Information Format - Secondary Entry Points

```
secondary-entry-point-name=subprogram-name
```

In the secondary-entry-point-name, specify the name described in the ENTRY statement of the calling program. In subprogram-name, specify the program name that has the ENTRY statement.

Example



(1) When specifying in the entry initialization file.
@CBR_ENTRYFILE=FILE

FILE

```
[ENTRY]
B=B.DLL
C=C.DLL
B1=B
C1=C
```

(2) When specifying in the run-time initialization file. (Compatible with the old versions)

```
[A.ENTRY]
B=B.DLL
C=C.DLL
B1=B
C1=C
```

Information

In this example, the DLL file name is in the form "Program name. DLL". Therefore, the entry information "B=B. DLL" and "C=C. DLL" can be omitted.

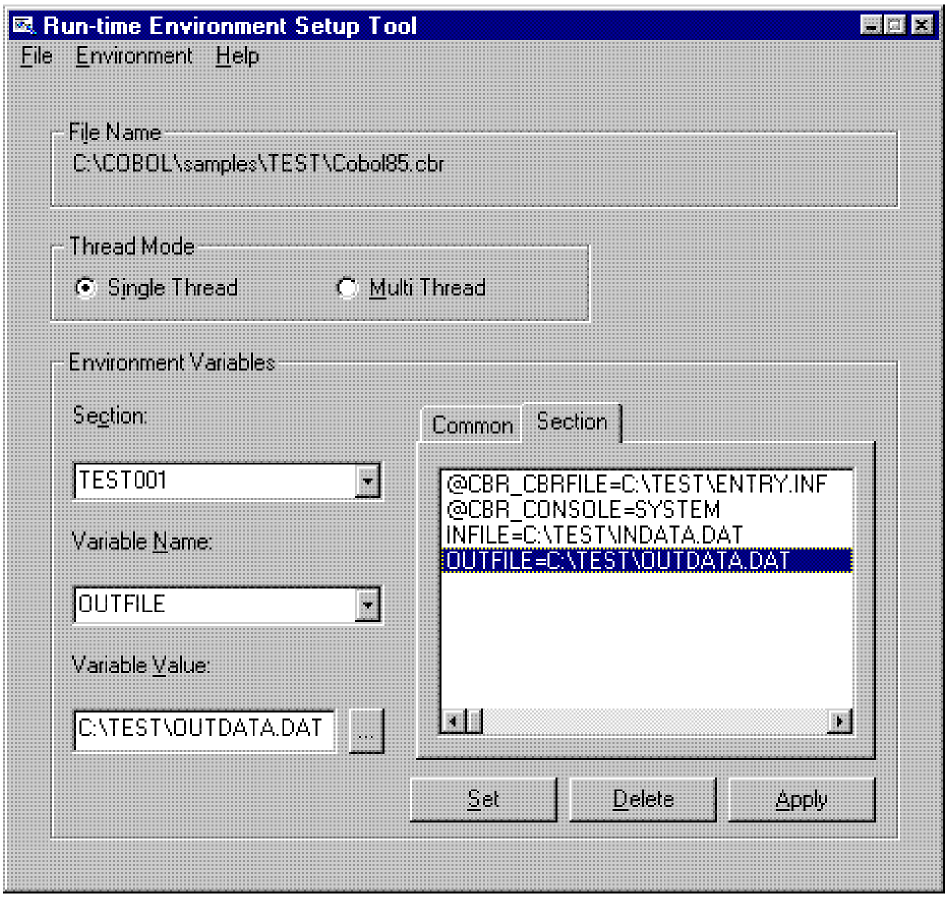
5.5 Runtime Environment Setup Tool

The Runtime Environment Setup tool (COBENVUT.EXE) is used for editing the contents of the runtime initialization file used for execution of the COBOL program. To use the runtime initialization file for execution of the COBOL program, edit the contents of the runtime initialization file using this tool before execution of the program. Note that the specification of the file path name and printer name may differ with the operation environment.

For the storage position of the runtime initialization file, refer to the topic "5.3.2.3 How to Set the Runtime Initialization File".

The Runtime Environment Setup tool is shown below. For details of the specification format of the runtime environment information to be set, refer to the topic "5.3 Setting Runtime Environment Information". For details of how to use the tool, refer to the online help.

The components of the Runtime Environment Setup Tool are:



File Name
C:\COBOL\samples\TEST\Cobol85.cbr

Thread Mode
● Single Thread ○ Multi Thread

Environment Variables

Section:
TEST001

Variable Name:
OUTFILE

Variable Value:
C:\TEST\OUTDATA.DAT

```
@CBR_CBRFILE=C:\TEST\ENTRY.INF
@CBR_CONSOLE=SYSTEM
INFILE=C:\TEST\INDATA.DAT
OUTFILE=C:\TEST\OUTDATA.DAT
```

Menu bar

File:

Opens and closes the runtime initialization file, and exits the tool.

Environment:

Fetches the printer name, sets the character code, and specifies various fonts.

Help:

Shows the help of the Runtime Environment Setup tool.

Edit box

Section:

Enter the edit target section name (main COBOL program). The section names of the currently loaded runtime initialization file are displayed in the list.

Variable Name:

Enter the environment variable name. The environment variable names reserved for the COBOL runtime system are displayed in the list.

Variable Value:

Enter the environment variable information value.

List box

Common tab:

Displays the setting contents of the common section of the currently set runtime initialization file.

Section tab:

Displays the setting contents of the runtime initialization file that corresponds to the section displayed in section-name.

Radio button

Thread mode:

Specify whether the edit target program operates in single thread mode or multithread mode. In the COBOL program that operates in multithread mode, only the common section information of the runtime initialization file is valid.

Button

Set:

Click this button when you want to set the information, which is described in the variable name and variable value, in the selected list box.

To save the setting contents of the runtime initialization file, the Apply button must be clicked.

Delete:

Click this button when you want to delete the selected variable names and variable values from the list box.

Apply:

Click this button when you want to save the contents of the selected list box in the runtime initialization file.

Static Text

File Name:

Displays the name of the runtime initialization file.

5.5.1 Setting Environment Variable Information

Environment variable information is set using the section name, variable name, and variable value edit box of the Runtime Environment Setup tool.

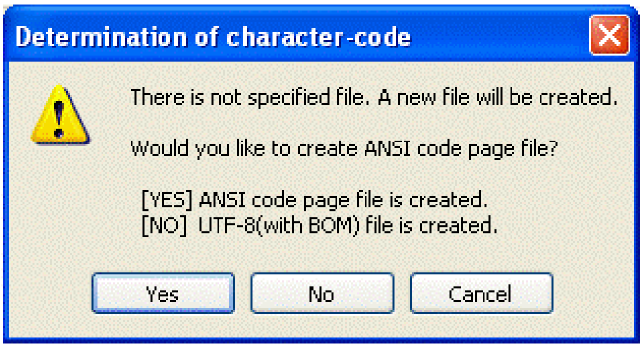
This section explains the operating procedures for setting the environment variable information.

5.5.1.1 Opening the Runtime Initialization File

To open the runtime initialization file to be edited, use the following procedures.

1. Select Open from the Files menu.
2. The Runtime Initialization File Specification dialog appears. To open an existing file, select the file. To create a new runtime initialization file, enter the file name in the edit box.

   The file can be specified in ANSI code page (SHIFT-JIS) and UTF-8 (with BOM). When you create a new file, select a character code using the following dialog. When "Yes" is selected, an ANSI code page file is created. When "No" is selected, a UTF-8 (with BOM) file is created.

   Determination of character-code

   There is not specified file. A new file will be created.

   Would you like to create ANSI code page file?

   [YES] ANSI code page file is created.
   [NO] UTF-8(with BOM) file is created.

   When an existing file is opened, the code of the file (ANSI code page or UTF-8 (with BOM)), is automatically determined.

   When a UTF-8 (with BOM) file is opened, "UTF-8" is displayed in the title of the ODBC information setup tool.
3. Click on the Open button in the Runtime Initialization File Specification dialog.

   The contents of the runtime initialization file are displayed in the list box.

5.5.1.2 Adding Environment Variable Information

To add new environment variable information, use the following procedures.

1. Select the environment variable to add from the Variable Name combo box list.
2. Set information in Variable Value.
3. Click on the Set button. The environment variable information is added to the list box.

5.5.1.3 Changing Environment Variable Information

To change the environment variable information displayed in the Environment Variables Information list box:

1. Select the environment variable information to change from the list box. The selected information is displayed in the Environment Variables Information text box.
2. Change the contents.
3. Click on the Set button. The changed environment variable information is displayed in the Environment Variables Information list box.

5.5.1.4 Deleting Environment Variable Information

To delete environment variable information from the Environment Variables Information list box:

1. Select the environment variable to delete from the list box.
2. Click on the Delete button. The selected environment variables are deleted from the list box.

5.5.1.5 Closing runtime initialization file

To close the runtime initialization file, which is currently open, use the following procedures.

1. Select Close from the Files menu.

Note

Information that has not been applied when the runtime initialization file is closed will not be saved in the runtime initialization file.

5.5.2 Setting Entry Information

For entry information, an entry information file must be created in advance and the entry information file name must be specified in the environment variable @CBR_ENTRYFILE. For the specification format, refer to "5.4.1.20 @CBR_ENTRYFILE(Set the Entry Information File)".

5.5.3 Saving to the Initialization File

When the "Apply" button is clicked, the information in the list box is saved in the initialization file.

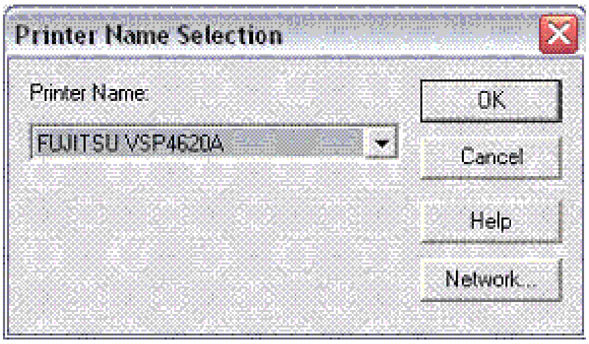
Note

Saving modifies the existing information.

5.5.4 Setting Printers

When using a printer, printer information can be set. Select Printer from the Environment menu of the Runtime Environment Setup Tool, then set required information in the Printer Name Selection dialog box.

Figure 5.4 The Printer Name Selection dialog box



Printer Name: FUJITSU VSP4620A

5.5.5 Exiting the Runtime Environment Setup Window

Select Exit from the Files menu of the Runtime Environment Setup tool. The Runtime Environment Setup tool terminates.

Note

Information that has not been applied when the runtime initialization file is closed will not be saved in the runtime initialization file.

5.6 Specifying the Initialization File Name

This section describes how to specify the runtime initialization file name when the main program of an application is COBOL or another language such as C or Microsoft Visual Basic. The main program of an application is the program to which control is first passed from the system.

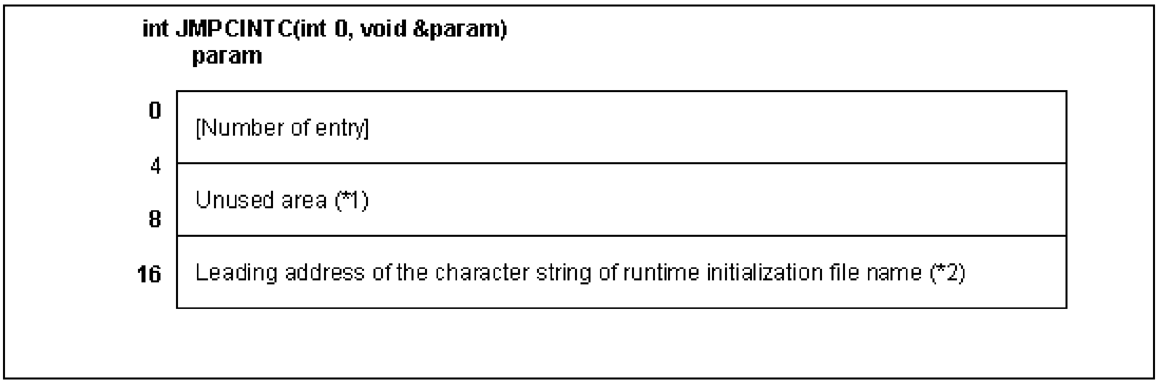
Specifying with a COBOL Main Program

Specify the runtime initialization file name using the runtime parameter -CBR or /CBR.

Specifying with a C Main Program

Call JMPCINTC immediately after calling JMPCINT2.

Information



```
int JMPCINTC(int 0, void &param)
    param
```

0	[Number of entry]
4 8	Unused area (*1)
16	Leading address of the character string of runtime initialization file name (*2)

*1 : Please initialize it by 0.

*2 : The first address of the character string that ends with NULL (terminating symbol). NULL indicates the runtime initialization file that contains the path name (optional).}
The maximum length of the character string that can be specified is up to 255 characters (including the terminating symbol).

Return value of JMPCINTC	Meaning
0	Normal end
-1	JMPCINT2 not called/specifying error of function code/shortage of operation area
1	Multiple invocation (*3)

*3 : Means that JMPCINTC has already been called. The runtime initialization file name specified with the initial call to JMPCINTC is the one that is used.

Specifying with a Visual Basic Main Program

Specify the following in the Imports statement and the Declare phrase and then call JMPCINTB immediately after calling JMPCINT2.

Description in the Declare phrase:

```
Imports System.Runtime.InteropServices
         :
Private Declare Sub JMPCINT2 LIB "F4AGPRCT.DLL" ()
Private Declare Sub JMPCINT3 LIB "F4AGPRCT.DLL" ()
Private Declare Function JMPCINTB LIB "F4AGPRCT.DLL"_  (*1)
  (ByVal A As Integer, <MarshalAs(UnmanagedType.LPStr)> ByVal D As String) As lnteger
```

*1 : This line is line and the next should be code as a single line.

Example of calling JMPCINTB (*2):

```
Static DATA As String
Dim ANS As Integer

DATA = "C:\TEST.CBR" & Chr(0)
ANS = JMPCINTB(0, DATA)
```

*2 : The meanings of each argument/return value of JMPCINTB are as follows:

Argument of JMPCINTB	Meaning
First argument	Function code (set 0 for changing CBR name)
Second argument	Add character string (Chr(0) to the end of the runtime initialization file name. The maximum length of the string is 127 characters (including the end code).

Return value of JMPCINTB	Meaning
0	Normal end
-1	JMPCINT2 not called/specifying error of function code/shortage of operation area
1	Multiple invocation (*3)

*3 : Means that JMPCINTC has already been called. The runtime initialization file name specified with the initial call to JMPCINTC is the one that is used.

5.7 Format of Runtime Options

Runtime options specify information or operations to COBOL programs at runtime.

Define runtime options depending on the functions used in the COBOL program or the options specified when compiling the COBOL source program. The runtime options can be specified with the following methods:

- Specification method with the command line

  See "5.3.2.5 Setting from the Command Line."

- Specification method with the environment variable @GOPT

  See "5.4.1.53 @GOPT(Set Runtime Options)".

Types and the specification formats of the runtime option are listed in Table below "Runtime options". To specify multiple runtime options, use a comma (,) to delimit each option.

Table 5.5 Runtime options

Function	Format
Set the trace data limit, and suppress the TRACE function	[r count | nor]
Set the number of CHECK messages, and suppress the CHECK function	[c count | { noc | nocb | noci | nocn | nocp }]
Set external switch values	[s value]
Set the memory capacity to be used by PowerBSORT	[smsize value k]

Details of each option is described as follows.

[r count | nor] (Set the trace data limit, and suppress the TRACE function)

Set this option to change the amount of trace information produced by the TRACE function. Specify the amount of trace information in the range from 1 to 999999.

When the TRACE function is suppressed, nor is specified.

These options are effective in programs defined with the -Dr option or the compiler option TRACE during compilation.

Refer to "3.5.2.6 -Dr(Enable the TRACE function)" and "A.2.55 TRACE(whether the TRACE function should be used)".

[c count | { noc | nocb | noci | nocn | nocp }] (Set the number of CHECK messages, and suppress the CHECK function)

Set this option to change the process when an error is detected by the CHECK function. Specify the process count in the range from 9 to 999999.

A value of 0 assumes no limit. A value of noc suppresses the CHECK function.

The following CHECK functions can be suppressed. More than one of the following can be specified.

- noc : All the CHECK function
- nocb : CHECK(BOUND)
- noci : CHECK(ICONF)
- nocn : CHECK(NUMERIC)
- nocp : CHECK(PRM)

These options are effective only in the program defined with the -Dk option or the compiler option CHECK(ALL) during compilation.

Refer to "3.5.2.3 -Dk(Enable the CHECK function)" and "A.2.5 CHECK(whether the CHECK function should be used)".

[s value] (Set external switch values)

Set this option to set values for the external switches, SWITCH-0 to SWITCH-7, specified in the SPECIAL-NAMES paragraph in the COBOL program. Enter eight consecutive switch values from SWITCH-0 sequentially.

The values can be 0 or 1. If omitted, "S00000000" is assumed.

SWITCH-8 is equivalent to SWITCH-0. When SWITCH-8 is used, therefore, the switch values correspond to SWITCH-8, SWITCH-1 to SWITCH-7 from the left.

[smsize value k] (Set memory capacity used by PowerBSORT)

Set this option to restrict the capacity of the memory space used by PowerBSORT that is called from the SORT or MERGE statements. Specify a number in kilobytes. The specified value is set for memory_size of the BSRTPRIM structure of PowerBSORT. For details about which values are valid, refer to the "PowerBSORT Online Manual".

When the smsize option is specified at the same time as the SORT-CORE-SIZE special register and/or the SMSIZE() compiler option, the order of precedence is first the SORT-CORE-SIZE, then the smsize runtime option, and last the SMSIZE() compiler option.

5.8 Cautions

This section explains the cautions for running programs.

5.8.1 Stack Overflow

Stack overflow occurs when the amount of stack used by the COBOL program exceeds that assigned at link time.

The default stack size is 4Mb.

First, check the program logic to ensure that there are no problems, such as an unintentional recursive structure. If there are no logic problems, define a larger stack size at link time.

Refer to "4.5.1 LINK Command Format".

How to Estimate the Required Stack Size

You can estimate the stack size using the following formulae:

```
Total stack size = max (program space)
```

Where:

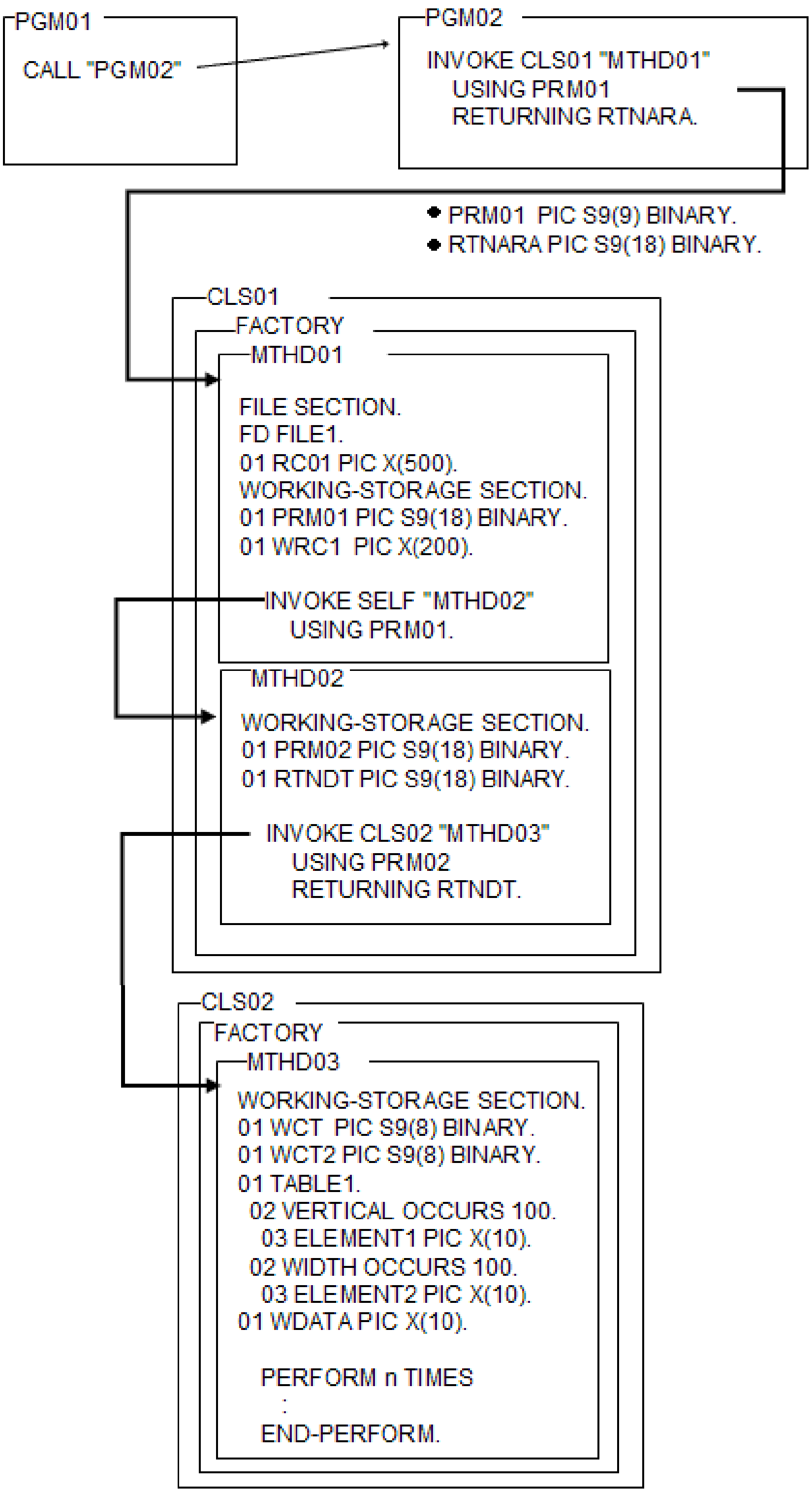
"program space" means: The sum of all stack sizes of programs or methods in the active call chain. For example, if program A1 calls programs B1 and B2, and program B2 calls program C1, then the two call chains will happen at different points in the execution. One call chain will consist of programs A1 and B1, the other will consist of A1, B2, and C1.

The size of each stack can be found from the section size listing. For details on section size listings, refer to "Listings Relating to Data Areas" in "NetCOBOL Debugging Guide".

"max" means: The maximum value over all possible call chains.

Example

Non-overflowing case



```
CALL "PGM02"
```

```
INVOKE CLS01 "MTHD01"
   USING PRM01
   RETURNING RTNARA.
```

- PRM01 PIC S9(9) BINARY.
- RTNARA PIC S9(18) BINARY.

```
FILE SECTION.
FD FILE1.
01 RC01 PIC X(500).
WORKING-STORAGE SECTION.
01 PRM01 PIC S9(18) BINARY.
01 WRC1  PIC X(200).

   INVOKE SELF "MTHD02"
      USING PRM01.
```

```
WORKING-STORAGE SECTION.
01 PRM02 PIC S9(18) BINARY.
01 RTNDT PIC S9(18) BINARY.

   INVOKE CLS02 "MTHD03"
      USING PRM02
      RETURNING RTNDT.
```

```
WORKING-STORAGE SECTION.
01 WCT  PIC S9(8) BINARY.
01 WCT2 PIC S9(8) BINARY.
01 TABLE1.
  02 VERTICAL OCCURS 100.
    03 ELEMENT1 PIC X(10).
  02 WIDTH OCCURS 100.
    03 ELEMENT2 PIC X(10).
01 WDATA PIC X(10).

   PERFORM n TIMES
      :
   END-PERFORM.
```

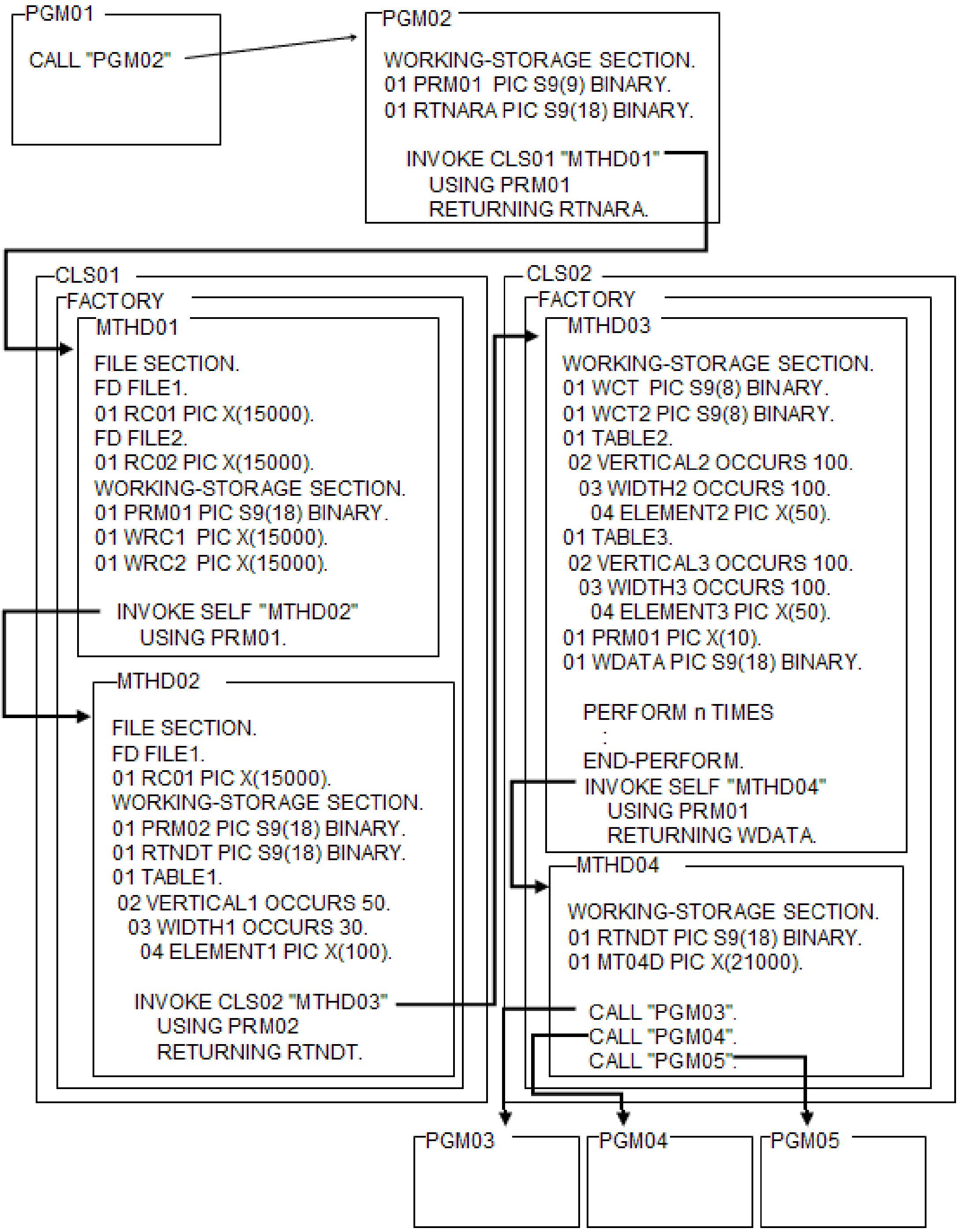
Calculating the stack size from the programs in Figure above.

```
PGM (Program definition)
       PGM01:                              176 bytes
       PGM02:                              200 bytes
CLS (Class definition)
       CLS01
         METHOD (1):                     1,208 bytes
         METHOD (2):                       376 bytes
       CLS02
```

```
        METHOD (3):                          2,328 bytes

               Total:                        4,288 bytes
```

Example

Overflowing Case



```
CALL "PGM02"
```

```
WORKING-STORAGE SECTION.
01 PRM01  PIC S9(9) BINARY.
01 RTNARA PIC S9(18) BINARY.

  INVOKE CLS01 "MTHD01"
    USING PRM01
    RETURNING RTNARA.
```

```
FILE SECTION.
FD FILE1.
01 RC01 PIC X(15000).
FD FILE2.
01 RC02 PIC X(15000).
WORKING-STORAGE SECTION.
01 PRM01 PIC S9(18) BINARY.
01 WRC1  PIC X(15000).
01 WRC2  PIC X(15000).

  INVOKE SELF "MTHD02"
    USING PRM01.
```

```
FILE SECTION.
FD FILE1.
01 RC01 PIC X(15000).
WORKING-STORAGE SECTION.
01 PRM02 PIC S9(18) BINARY.
01 RTNDT PIC S9(18) BINARY.
01 TABLE1.
 02 VERTICAL1 OCCURS 50.
  03 WIDTH1 OCCURS 30.
    04 ELEMENT1 PIC X(100).

  INVOKE CLS02 "MTHD03"
    USING PRM02
    RETURNING RTNDT.
```

```
WORKING-STORAGE SECTION.
01 WCT  PIC S9(8) BINARY.
01 WCT2 PIC S9(8) BINARY.
01 TABLE2.
 02 VERTICAL2 OCCURS 100.
  03 WIDTH2 OCCURS 100.
    04 ELEMENT2 PIC X(50).
01 TABLE3.
 02 VERTICAL3 OCCURS 100.
  03 WIDTH3 OCCURS 100.
    04 ELEMENT3 PIC X(50).
01 PRM01 PIC X(10).
01 WDATA PIC S9(18) BINARY.

  PERFORM n TIMES
     :
  END-PERFORM.
  INVOKE SELF "MTHD04"
    USING PRM01
    RETURNING WDATA.
```

```
WORKING-STORAGE SECTION.
01 RTNDT PIC S9(18) BINARY.
01 MT04D PIC X(21000).

  CALL "PGM03".
  CALL "PGM04".
  CALL "PGM05".
```

Calculating the stack size from the programs in Figure above.

```
PGM (Program definition)
       PGM01:                                        176 bytes
       PGM02:                                       200 bytes
       Maximum stack size between
          PGM03, PGM04 and PGM05:                   240 bytes
CLS (Class definition)
       CLS01
         METHOD (1):                             60,656 bytes
         METHOD (2):                            165,520 bytes
       CLS02
         METHOD (3):                          1,000,400 bytes
         METHOD (4):                             21,296 bytes

               Total:                         1,248,488 bytes
```

How to Avoid Stack Overflow

To avoid an overflow, specify the /STACK option to expand the stack size at link time.

Format

```
/STACK: reserve [, commit]
```

reserve:

Specify the total size of stack to be reserved in virtual memory. The default value is 4M bytes. Space is assigned in multiples of 4-bytes.

commit:

Specify an initial value of stack reserved in virtual memory.

Example

```
/STACK:8192000,1024000
          [1]     [2]
```

[1] Reserve 8Mbytes of stack in virtual memory

[2] Assign an initial space of 1Mbyte for the stack.

5.8.2 Virtual Memory Shortages

Virtual memory shortages when COBOL programs are executing may be caused by one of the problems listed below. In each case, the process is to review the environment and program structure at the time of the memory shortage, and apply the appropriate remedy.

Environmental Problems

- Too little installed memory.

  Extend the memory.

- Too little virtual memory.

  Increase the space available for virtual memory.

- Other applications executing at the same time are consuming the available memory.

  Stop one or more of the other applications.

Program Structural Problems

- Too many files open at the same time in the execution unit.

  Reduce number of files open at the same time.

- Too many data items or file declarations use the EXTERNAL phrase in the execution unit.

  Reduce the use of EXTERNAL data.

- Too many objects (instances) are used simultaneously in the execution unit.

  Reduce the number of simultaneously active objects.

Others

- The executed application destroys the memory area.

  Use the debugging function of NetCOBOL Studio, CHECK function, or memory check function to investigate the cause and correct the program. For the debugging function of NetCOBOL Studio, refer to the "NetCOBOL Studio User's Guide". For the CHECK Function and the Memory Check Function, refer to the "NetCOBOL Debugging Guide".

5.8.3 Call of the application that operates on 32-bit Windows

The application that operates on 32-bits Windows cannot be called from the application made with this product by the dynamic structure.

Similarly, the application made with this product cannot be called from the application that operates on 32-bit application.

The runtime error message of JMP0015I-U (detail code 0xc1) is output when calling it.

Chapter 6 File Processing

This chapter explains how to read and write data files, how to use the COBOL File Utility and other file systems.

6.1 File Organization Types

This section explains file organization types and characteristics, and details how to design records and process files.

6.1.1 File Organization Types and Characteristics

Tables below list the types of file that can be processed using the sequential, relative, and indexed file functions, and the characteristics of those types.

Table 6.1 File types

File Function	File Types
Sequential file functions	Record sequential files Line sequential files Print files
Relative file functions	Relative files
Indexed file functions	Indexed files

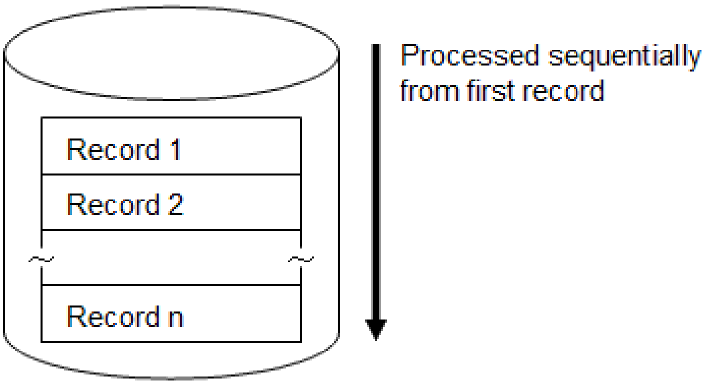
Table 6.2 File organization types and characteristics

File Types	Record- Sequential File	Line-Sequential File	Print File	Relative File	Indexed File
Record processing	Record storage sequence			Relative record number	Record key value
Usable data medium	Hard disk (*) Floppy disk	Hard disk (*) Floppy disk	Printer	Hard disk (*) Floppy disk	Hard disk (*) Floppy disk
Usage example	Saving data Work file	Text file	Printing data	Work file	Master file

* : A virtual device (ex. NUL) cannot be specified.

File organization is determined when a file is generated, and cannot be changed. Before generating a file, therefore, fully understand the characteristics of the file and be careful to select the file organization that most agrees with its use. Each file organization is explained below.

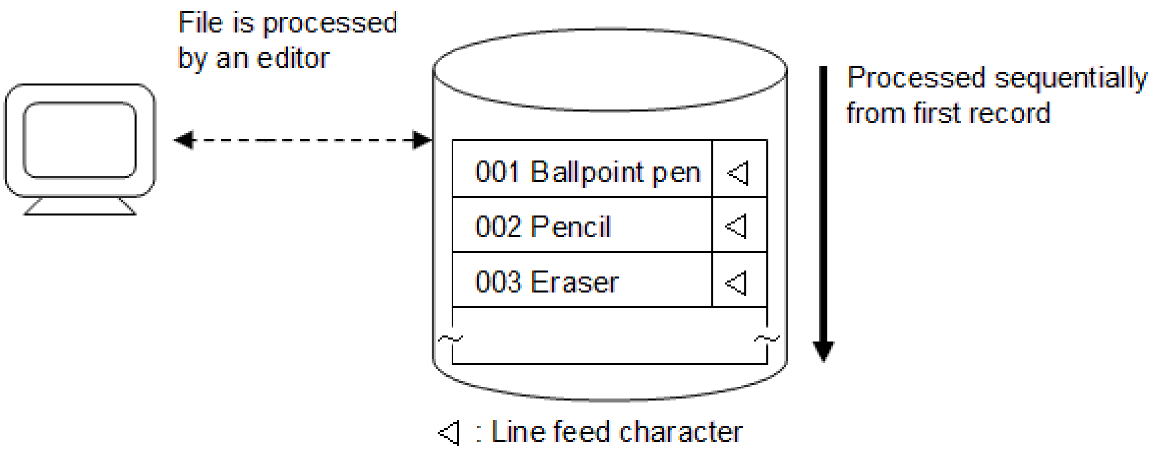
Record Sequential Files



In a record sequential file, records are read and updated in order, starting with the first record in the file.

A record sequential file is the easiest to create and maintain. Record sequential files are generally used to store data sequentially and maintain a large amount of data.

Line Sequential Files


File is processed by an editor
Processed sequentially from first record
001 Ballpoint pen ◁
002 Pencil ◁
003 Eraser ◁
◁ : Line feed character

In a line sequential file, records are also read from the first record in the physical order they are written.

Line feed characters are used as record delimiters in a line sequential file.

For example, line sequential files can be used to handle text files created by text editors.

The line feed character is two bytes in the size. The content of the line feed character is shown by the hexadecimal number mark.

0x0D	0x0A

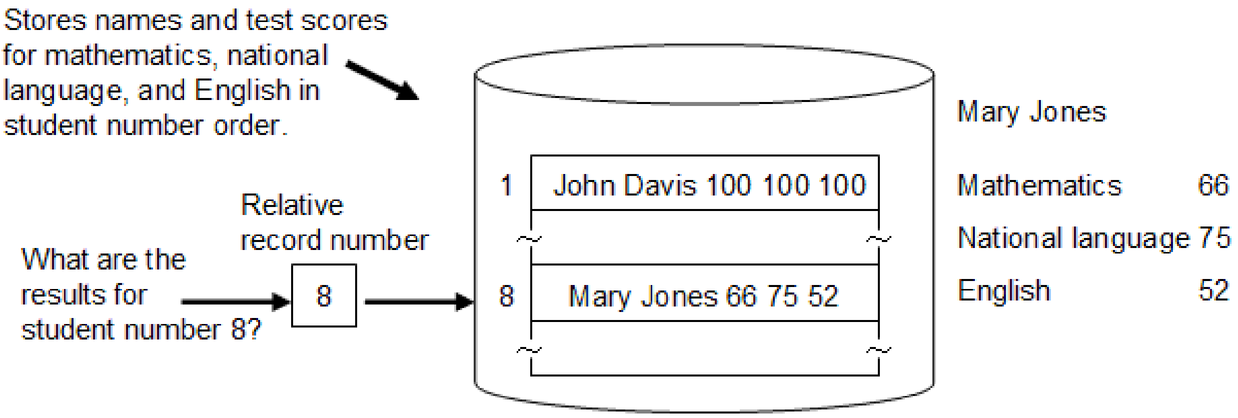
Note

- For more information regarding the line feed character, refer to "Treatment of Control characters".
- When a WRITE statement with an ADVANCING clause is executed, non-line feed control characters are written. For more details, refer to "Operation with ADVANCING clause".
- For details of CSV format data operation, refer to "Chapter 21 Operation of Comma Separated Value data"

Print Files

A print file is used for printing with the sequential file function; there is no special file name for a print file. For details of a print file, refer to "Chapter 7 Printing".

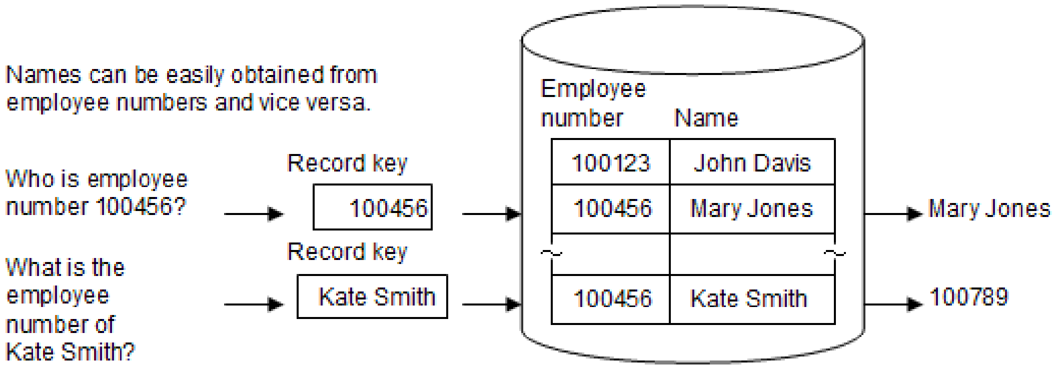
Relative Files


Stores names and test scores for mathematics, national language, and English in student number order.
What are the results for student number 8?
Relative record number
8
1
John Davis 100 100 100
8
Mary Jones 66 75 52
Mary Jones
Mathematics 66
National language 75
English 52

Records can be read and updated by specifying their relative record numbers (the first record is addressed by relative record number 1) in a relative file.

For example, a relative file can be used as a work file accessed by using relative record numbers as key values.

Indexed Files


Names can be easily obtained from employee numbers and vice versa.
Who is employee number 100456?
Record key
100456
What is the employee number of Kate Smith?
Record key
Kate Smith
Employee number
Name
100123
John Davis
100456
Mary Jones
100456
Kate Smith
Mary Jones
100789

Records can be read and updated by specifying the values of items (record keys) in an indexed file. Use an indexed file as a master file when retrieving information associated with the values of items in a record.

6.1.2 Designing Records

This section explains the types and characteristics of record formats and the record keys of indexed files.

6.1.2.1 Record Formats

There are two types of record formats, fixed length and variable length.

Fixed length record

In a fixed length record, all records in a file contain the same number of character positions.

Variable length record

In a variable length record, records contain a varying number of character positions. The record size is determined when a record is placed in a file. Because each record can be written with an appropriate length for its data, variable length record format is useful if you want to minimize file size.

6.1.2.2 Record Keys of Indexed Files

When designing a record in an indexed file, a record key must be determined. Multiple record keys can be specified for items in the record.

There are primary record keys (primary keys)) and alternate record keys (alternate keys). Records in the file are stored in ascending order of primary record keys.

Specifying a primary or alternate key, or both, determines which record is processed in the file. In addition, records can be processed in ascending order, starting from any record.

Note

- To process the same file with multiple record descriptions, primary keys must be at the same position and of the same size in each record description.
- In the variable length record format, the record key must be set at a fixed position.

6.1.3 Processing Files

There are six types of file processing:

- File creation: Writes records to files.
- File extension: Writes records after the last record in a file.
- Record insertion: Writes records at specified positions in files.
- Record reference: Reads records from files.
- Record updating: Rewrites records in files.
- Record deletion: Deletes records from files.

File processing depends on the file access mode. There are three types of access modes:

- Sequential access: Allows serial records to be processed in a fixed order.
- Random access: Allows arbitrary records to be processed.
- Dynamic access: Allows records to be processed in sequential and random access modes.

The following table indicates the processing available with each file type.

Table 6.3 File organization types and processing

File Types	Access Mode	Processing					
		Creation	Extension	Insertion	Reference	Updating	Deletion
Record sequential file	Sequential	Yes	Yes	No	Yes	Yes	No
Line sequential file	Sequential	Yes	Yes	No	Yes	No	No
Print file	Sequential	Yes	Yes	No	No	No	No
Relative file	Sequential	Yes	Yes	No	Yes	Yes	Yes

File Types	Access Mode	Processing					
		Creation	Extension	Insertion	Reference	Updating	Deletion
Indexed file	Random	Yes	No	Yes	Yes	Yes	Yes
	Dynamic	Yes	No	Yes	Yes	Yes	Yes

Yes: Can be processed.

No: Cannot be processed.

Disable access by locking records in use. This is called exclusive control of files. For details of exclusive control of files, see "6.7.2 Exclusive Control of Files".

6.2 Using Record Sequential Files

This section explains how to define and process record sequential files, and how to define the records. The code below illustrates the features described in the following topics.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. program-name.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
 FILE-CONTROL.
      SELECT file-name
        ASSIGN TO file-reference-identifier
                  ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
 FILE SECTION.
 FD file-name
    [RECORD record-size].
 01 record-name.
     record-description-entry.
PROCEDURE DIVISION.
    OPEN open-mode file-name.
    [READ file-name.]
    [REWRITE record-name.]
    [WRITE record-name.]
    CLOSE file-name.
END PROGRAM program-name.
```

6.2.1 Defining Record Sequential Files

This section explains the file definitions required to use record sequential files in a COBOL program.

File Name and File-Reference-Identifier

Determine a file name in conformance to the rules of COBOL user-defined words, then write it in the SELECT clause.

Determine a file-reference-identifier, then write it in the ASSIGN clause. For the file-reference-identifier, specify a file-identifier, file-identifier-literal, data-name, or the character string DISK.

Use the file-reference-identifier to associate the file name specified in the SELECT clause with the file of the actual input-output medium.

How you associate the file name in the SELECT clause to the file of the actual input-output medium depends on what is specified for the file-reference-identifier.

The recommended method to determine what to specify for the file-reference-identifier is:

- When the file name of the input-output medium is determined at COBOL program generation and is not subsequently changed, specify a file-identifier-literal or character string DISK.

- When the file name of the input-output medium is undetermined at COBOL program generation, or to determine the file name at every program execution, specify a file identifier.

- To determine the file name in the program, specify a data name.

- If the file is temporary and is not needed after the program terminates, specify character string DISK.

Note

When the character string DISK is specified, the file is not deleted after the program terminates.

The file name of an actual input-output medium can include the following characters:

```
Blank, "+", ",", ";", "=", "[", "]", "(", ")", "'"
```

Any file name that includes a comma (",") must be enclosed in double quotes (").

Information

Files can be processed at high speed in record sequential or line sequential files. For the specification method, refer to "6.7.4 High-Speed File Processing".

The following table lists examples of SELECT and ASSIGN clauses.

Table 6.4 Description examples of SELECT and ASSIGN clauses

Type of File-Reference-Name	File Description Example	Remarks
File-identifier	SELECT file-1 ASSIGN TO INFILE	Must be related to the actual input-output medium at program execution.
Data-name	SELECT file-2 ASSIGN TO data name	The data name must be defined in the WORKING-STORAGE SECTION in the DATA DIVISION
File-identifier literal	SELECT file-3 ASSIGN TO "c.dat"	-
DISK	SELECT data1 ASSIGN TO DISK	The file name cannot be specified with an absolute path name

File Organization

Specify SEQUENTIAL in the ORGANIZATION clause. If the ORGANIZATION clause is omitted, SEQUENTIAL is used as the default.

6.2.2 Defining Record Sequential File Records

This section explains record formats, lengths, and organization.

Record Formats and Lengths

Record sequential files are either fixed or variable length.

The record length in fixed length record format is the value specified in the RECORD clause or, if the RECORD clause is omitted, it is the length of the longest record defined in the record description entries.

For variable length record format, the length of output records is determined by the value of the data item specified in the DEPENDING ON phrase of the RECORD clause. You can obtain the record length when reading a record by using this data item.

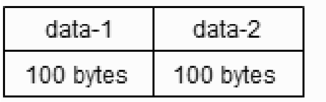
Record Organization

Use record description entries to define the attributes, positions, and sizes of the data in a record.

The following are examples of record definitions:

Example

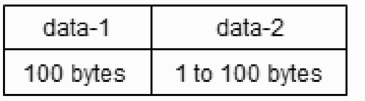
[A fixed length record]



data-1	data-2
100 bytes	100 bytes

```
FD   data-storage-file.
01   data-record.
      02  data-1  PIC X(100).
      02  data-2  PIC X(100).
```

[A variable length record]



data-1	data-2
100 bytes	1 to 100 bytes

```
FD   data-storage-file
      RECORD IS VARYING IN SIZE
      FROM 101 TO 200 CHARACTERS
      DEPENDING ON record-length.
 01   data-record.
       02  data-1  PIC X(100).
       02  data-2.
         03  PIC X
             OCCURS 1 TO 100 TIMES
             DEPENDING ON length-1.
```

Note

Note the following points about the variable length record format using variable length data items containing the OCCURS clause.

- Writing records

  To move data to a record item, you must move data length and then the data respectively, beginning with the first variable length data item defined, in order to avoid unintentional results.

- Reading records

  When you read a record, although length of the record you have read is returned, the length of the variable portion of the record is not returned. For this reason, subtract the fixed part of the length from the whole record find out the length of a single variable length data item. If several variable length data items have been defined, you cannot find out the length using this calculation. Instead, you must determine the length of each variable length data item based on fields in the record.

6.2.3 Processing Record Sequential Files

Use input-output statements to create, extend, reference, and update record sequential files. This section explains the types of input-output statements used in record sequential file processing, and outlines each type of processing.

Types of Input-Output Statements

OPEN, CLOSE, READ, REWRITE, and WRITE statements are used for input and output.

Using Input-Output Statements

The section provides explanation and examples of how to use input-output statements.

OPEN and CLOSE Statements

Execute an OPEN statement once, at the beginning of file processing, and a CLOSE statement at the end of file processing.

The open mode specified in the OPEN statement depends on the kind of file processing. For example, for creation, use the OPEN statement in the OUTPUT mode (OUTPUT specified).

Other Statements

To read records in a file, use a READ statement. To update records in a file, use a REWRITE statement. To write records to a file, use a WRITE statement.

Processing Outline

Creating

To generate a record sequential file, open a file in OUTPUT mode, then write records to the file with WRITE statements.

```
OPEN OUTPUT file-name.
Editing-records
WRITE record-name ... .
CLOSE file-name.
```

Note

If an attempt is made to generate a file that already exists, the file is regenerated and the original contents are lost.

Extending

To extend a record sequential file, open the file in EXTEND mode, then write records to the file with WRITE statements. Records are added after the last record in the file.

```
OPEN EXTEND file-name.
Editing-records
WRITE record-name ... .
CLOSE file-name.
```

Referencing

To refer to records, open the file in INPUT mode, then read records in the file, starting with the first record, with READ statements.

```
OPEN INPUT file-name.
READ file-name ... .
CLOSE file-name.
```

Note

When OPTIONAL is specified in the SELECT clause of the file control entry, the OPEN statement is successful even if the file does not exist, and the AT END condition is satisfied upon execution of the first

For information on the at end condition, see "6.6 Input-Output Error Processing".

Updating

To update records, open the file in I-O mode, read the records from the file by using READ statements, then rewrite them by using REWRITE statements.

```
OPEN I-O file-name.
READ file-name... .
Editing-records
REWRITE record-name ... .
CLOSE file-name.
```

Note

- When a REWRITE statement is executed, the record read by the last READ statement is updated.
- For variable length records, the record length cannot be changed.

6.3 Using Line Sequential Files

This section explains how to define and process line sequential files, and how to define the records. The code below illustrates the features described in the following topics.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. program-name.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
 FILE-CONTROL.
      SELECT file-name
        ASSIGN TO file-reference-identifier
                  ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
 FILE SECTION.
 FD file-name
    [RECORD record-size].
 01 record-name.
     record-description-entry.
PROCEDURE DIVISION.
    OPEN open-mode file-name.
    [READ file-name.]
    [WRITE record-name.]
    CLOSE file-name.
END PROGRAM     program-name.
```

6.3.1 Defining Line Sequential Files

This section explains the file definitions required to use line sequential files in a COBOL program.

File Name and File-Reference-Identifier

As with a record sequential file, specify a file name and file-reference-identifier for a line sequential file. For more information about specifying these items, see "6.2.1 Defining Record Sequential Files".

File Organization

Specify LINE SEQUENTIAL in the ORGANIZATION clause.

6.3.2 Defining Line Sequential File Records

This section explains record formats, lengths, and organization.

Record Formats and Lengths

The explanations of the record formats and lengths of line sequential files are the same as for record sequential files. See "6.2.2 Defining Record Sequential File Records" for more information.

Because a record is delimited by a line feed character in a line sequential file, the end of a record is always a line feed character regardless of the record format. The line feed character, however, is not included in the record length.

Record Organization

Define the attribute, position, and size of record data in the record description entries. You do not have to define line feed characters in the record description entries because they are added when records are written.

The following is an example of record definitions in the variable length records format:

Example

Text character-string ◁

◁ : Line feed character

A text character string can be up to 80 alphanumeric characters.

```
FD text-file
    RECORD IS VARYING IN SIZE FROM 1 TO 80 CHARACTERS
                            DEPENDING ON record-length.
 01 text-record.
   02 text-character-string.
     03 character-data  PIC X OCCURS 1 TO 80 TIMES
                        DEPENDING ON record-length.
*>  :
 WORKING-STORAGE SECTION.
 01 record-length PIC 9(3) BINARY.
```

6.3.3 Processing Line Sequential Files

In a line sequential file processing, creation, extension, and reference can be done with input-output statements. This section explains the types and use of input-output statements in line sequential file processing, and outlines the processing.

Types of Input-Output Statements

OPEN, CLOSE, READ, and WRITE statements are used in line sequential file processing.

Using Input-Output Statements

OPEN and CLOSE Statements

Execute an OPEN statement once at the beginning of file processing, and a CLOSE statement at the end of file processing.

The open mode specified in the OPEN statement depends on the kind of file processing. For example, for creation, use the OPEN statement in the OUTPUT mode (OUTPUT specified).

Other Statements

To read records in a file, use a READ statement. To write records to a file, use a WRITE statement.

Processing Outline

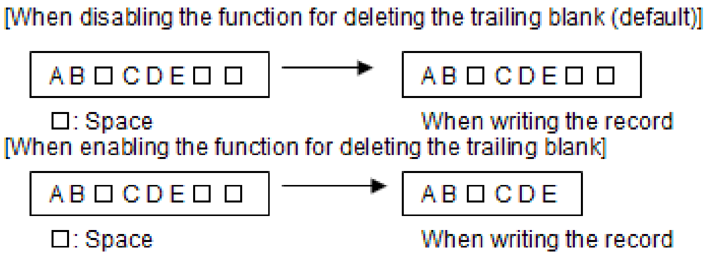
Creating

To create a line sequential file, open a file in OUTPUT mode, then write records to the file with a WRITE statement.

```
OPEN OUTPUT file-name.
Editing-records
WRITE record-name ... .
CLOSE file-name.
```

Note

- If an attempt is made to create a file that already exists, the file is overwritten and the original contents are lost.
- The contents of the record area and line feed character are written at record write.
- To output the record and remove trailing blanks in the record, specify character string "REMOVE" in environment variable @CBR_TRAILING_BLANK_RECORD. For details, see the runtime option "5.4.1.47 @CBR_TRAILING_BLANK_RECORD(Specify whether to remove or enable the trailing blank in the record of line sequential file)".


[When disabling the function for deleting the trailing blank (default)]
AB□CDE□□
AB□CDE□□
□: Space
When writing the record
[When enabling the function for deleting the trailing blank]
AB□CDE□□
AB□CDE
□: Space
When writing the record

Trailing spaces within the record are deleted and the record output to the file.

Extending

To extend a line sequential file, open the file in EXTEND mode, then write records sequentially to the file with a WRITE statement. Records are added after the last record in the file.

```
OPEN EXTEND file-name.
Editing-records
WRITE record-name ... .
CLOSE file-name.
```

Note

To output the record and remove trailing blanks in the record, specify character string "REMOVE" in environment variable @CBR_TRAILING_BLANK_RECORD. See the environment variable "5.4.1.47 @CBR_TRAILING_BLANK_RECORD(Specify whether to remove or enable the trailing blank in the record of line sequential file)".

Referencing

To refer to records, open the file in INPUT mode, then read records in the file from the first record by using a READ statement.

```
OPEN INPUT file-name.
READ file-name ... .
CLOSE file-name.
```

Note

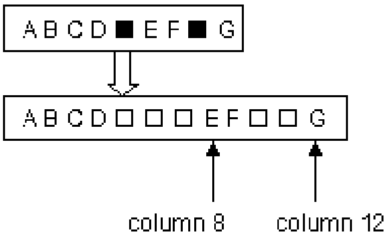
- When OPTIONAL is specified in the SELECT clause in the file control entry, the OPEN statement is successful even if the file does not exist, and the AT END condition is satisfied upon execution of the first READ statement. For information on the at end condition, see "6.6 Input-Output Error Processing".

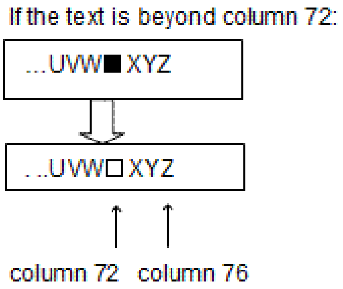
- If the size of a read record is greater than the record length, data input stops at the end of the record area when a READ statement is executed and the continuation of the data from the same record is placed in the record area when the next READ statement is executed. If the size of a read record is smaller than the defined record length, spaces are written to the remaining portion of the record area.

Treatment of Tab characters in a record

If there are tab characters in an input record, spaces are inserted to align the characters that follow to preset tab positions. The tab positions are 1, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, and 72. If a tab is present at a character position beyond 72, it is read as a single space.

For example, using "n" to indicate a tab character and "" to indicate a space character, the following input record is expanded as shown:





Treatment of Control characters

When a record is read, any control characters it may contain are handled as follows:

- 0x0C (page break): Handled as a record delimiter.
- 0x0D (line feed): Handled as a record delimiter.
- 0x1A (end-of-data marker): Handled as end-of-file.

Operation with ADVANCING clause

When a WRITE statement with an ADVANCING clause is executed, non-line feed control characters are written.

ADVANCING clause		Data output to the file
None		{ record-data } 0x0D 0x0A
BEFORE	n LINES (*1)	{ record-data }0x0D $\underbrace{\text{0x0A} \cdots \text{0x0A}}_{n}$
	PAGE	{ record-data }0x0D 0x0C

ADVANCING clause		Data output to the file
AFTER	n LINES (*1)	$\underbrace{\texttt{0x0A} \cdots \texttt{0x0A}}_{n}$ { record-data } 0x0D
	PAGE (*2)	0x0C { record-data } 0x0D

(*1) When zero is specified in the line count, 0x0A is not output - 0x0D is added after the record-data instead.

(*2) When a WRITE statement is executed immediately after OPEN OUTPUT, 0x0C is not output.

Note

If two or more adjoined control characters are at the end of a record that is output to a WRITE statement with an ADVANCING clause, the record may not be read by the intended record delimiter.

- The following adjoined control characters are handled as one record delimiter:

  0x0D 0x0A
  0x0D 0x0C
  0x0A 0x0D
  0x0C 0x0D

- If two or more control characters (0x0A or 0x0C) are adjoined, they are handled as one record delimiter.
- If control character 0x0A or 0x0C is in the record header, it is skipped.

Refer to "Treatment of Control characters" for information regarding single control characters.

Treatment of Unicode BOM

When a line sequential file in Unicode application is referenced, you can specify how the BOM (Byte Order Mark) that has been added to the head of the file is to be treated.

Usage

"CHECK", "DATA", or "AUTO" is set to the environment variable @CBR_FILE_BOM_READ.

$$\text{@CBR\_FILE\_BOM\_READ=} \quad \left\{ \begin{array}{l} \underline{\text{CHECK}} \\ \text{DATA} \\ \text{AUTO} \end{array} \right\}$$

CHECK

Check for a BOM in the data. If a BOM is included in the data and it matches the BOM specified in the record definition, skip it when executing the READ. If no BOM is included in the data, or if the BOM in the data does not match the BOM specified in the record definition, fail the OPEN.

DATA

If a BOM is included in the data, read it as part of the record data. If no BOM is included in the data, read the record from the beginning of the file.

AUTO

If a BOM is included in the data and it matches the BOM specified in the record definition, skip it when executing the READ. If the BOM in the data does not match the BOM specified in the record definition, fail the OPEN. If no BOM is included in the data, read the record from the beginning of the file.

6.4 Using Relative Files

This section explains how to define and process relative files, and how to define the records. The code below illustrates the features described in the following topics.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. program-name.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
 FILE-CONTROL.
  SELECT file-name
         ASSIGN TO file-reference-identifier
                   ORGANIZATION IS RELATIVE
                   [ACCESS MODE IS access-mode]
                   [RELATIVE KEY IS wk-relative-record-number].
DATA DIVISION.
 FILE SECTION.
 FD file-name
    [RECORD record-size].
01 record-name.
     record-description-entry.
WORKING-STORAGE SECTION.
[01 wk-relative-record-number  PIC 9(5) BINARY.]
PROCEDURE DIVISION.
    OPEN open-mode file-name.
    [MOVE relative-record-number TO wk-relative-record-number.]
    [READ file-name.]
    [REWRITE record-name.]
    [DELETE file-name.]
    [START file-name.]
    [WRITE record-name.]
    CLOSE file-name.
END PROGRAM program-name.
```

6.4.1 Defining Relative Files

This section explains file definitions required to use relative files in a COBOL program.

File Name and File-Reference-Identifier

As with a record sequential file, specify a file name and file-reference-identifier for a relative file. For more information about how to specify these items, see "6.2.1 Defining Record Sequential Files".

File Organization

Specify RELATIVE in the ORGANIZATION clause.

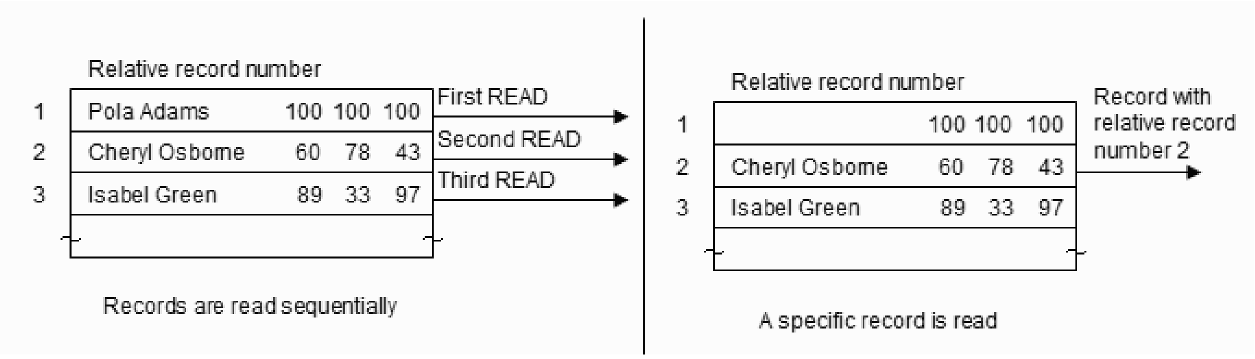
Access Mode

Specify one of the following access modes in the ACCESS MODE clause:

- Sequential access (SEQUENTIAL): Enables records to be processed in ascending order of the relative record numbers from the beginning of the file or a record with a specific relative record number.
- Random access (RANDOM): Enables a record with a specific relative record number to be processed.
- Dynamic access (DYNAMIC): Enables records to be processed in sequential and random access modes.

Referencing records in sequential and random access modes is illustrated below.

Figure 6.1 Sequential and random access modes



Relative Record Numbers

Specify the name of the data item in which the relative record number is stored, in the RELATIVE KEY clause. When using sequential access, this clause can be omitted.

In dynamic access you set this item to the relative record number to be read or written. In sequential access the RELATIVE KEY data item is set to the relative record number read and is ignored on a write.

This data name must be defined as an unsigned numeric data item in the WORKING-STORAGE section.

6.4.2 Defining Relative File Records

This section explains record formats, lengths, and organization.

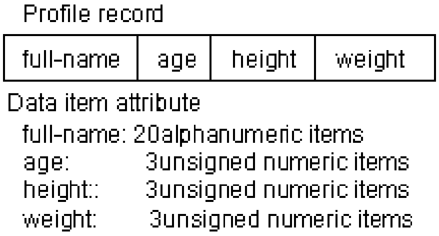
Record Formats and Lengths

The explanations of the record formats and lengths of relative files are the same as for record sequential files. See "6.2.2 Defining Record Sequential File Records" for more information.

Record Organization

Define the attribute, position, and size of record data in the record description entries. You do not have to define the area to set the relative record number.

Figure 6.2 Relative record definition



Profile record

full-name	age	height	weight

Data item attribute

full-name: 20alphanumeric items
age: 3unsigned numeric items
height:: 3unsigned numeric items
weight: 3unsigned numeric items

Record definitions in Data Division :

```
FD   profile.
01   profile-record.
     02  full-name   PIC X(20).
     02  age         PIC 9(3).
     02  height      PIC 9(3).
     02  weight      PIC 9(3).
```

6.4.3 Processing Relative Files

In relative file processing, creation, extension, insertion, reference, updating, and deletion is done with input-output statements. This section explains the types of input-output statements used in relative file processing, and outlines each type of processing.

Types of Input-Output Statements

OPEN, CLOSE, DELETE, READ, START, REWRITE, and WRITE statements are used for input and output in relative file processing.

Using Input-Output Statements

OPEN and CLOSE Statements

Execute an OPEN statement only once at the beginning of file processing and a CLOSE statement only once at the end of file processing.

The open mode specified in the OPEN statement depends on the kind of file processing. For example, for creation, use the OPEN statement in the OUTPUT mode (OUTPUT specified).

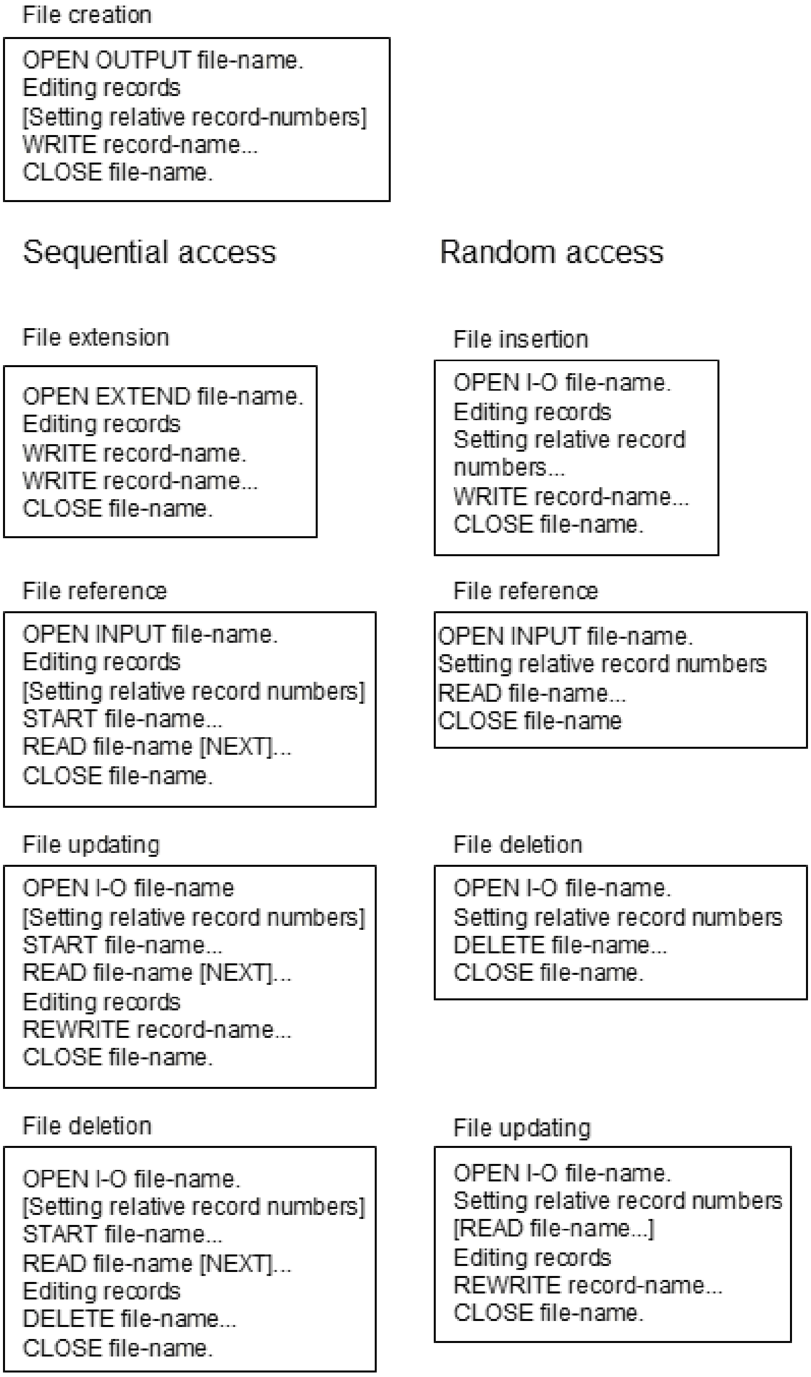
Other Statements

To delete records from a file, use a DELETE statement. To read records in a file, use a READ statement. To indicate a record for which processing is to be started, use a START statement.

To update records in a file, use a REWRITE statement. To write records to a file, use a WRITE statement.

Processing Outline

Execution Order of Input-Output Statements



File creation

OPEN OUTPUT file-name.
Editing records
[Setting relative record-numbers]
WRITE record-name...
CLOSE file-name.

Sequential access

File extension

OPEN EXTEND file-name.
Editing records
WRITE record-name.
WRITE record-name...
CLOSE file-name.

File reference

OPEN INPUT file-name.
Editing records
[Setting relative record numbers]
START file-name...
READ file-name [NEXT]...
CLOSE file-name.

File updating

OPEN I-O file-name
[Setting relative record numbers]
START file-name...
READ file-name [NEXT]...
Editing records
REWRITE record-name...
CLOSE file-name.

File deletion

OPEN I-O file-name.
[Setting relative record numbers]
START file-name...
READ file-name [NEXT]...
Editing records
DELETE file-name...
CLOSE file-name.

Random access

File insertion

OPEN I-O file-name.
Editing records
Setting relative record numbers...
WRITE record-name...
CLOSE file-name.

File reference

OPEN INPUT file-name.
Setting relative record numbers
READ file-name...
CLOSE file-name

File deletion

OPEN I-O file-name.
Setting relative record numbers
DELETE file-name...
CLOSE file-name.

File updating

OPEN I-O file-name.
Setting relative record numbers
[READ file-name...]
Editing records
REWRITE record-name...
CLOSE file-name.

Enter the relative record number for the data-name specified in the RELATIVE KEY clause, for example, MOVE 1 to data-name.

The execution orders of input-output statements for creation, extension, insertion, reference, updating, and deletion are as follows:

Creating (Sequential, Random, and Dynamic)

To create a relative file, open a file in OUTPUT mode, then write records to the file with a WRITE statement. Records are written with the record length specified in the WRITE statement.

In sequential access, relative record numbers 1, 2, 3, ... are set in the order the records are written.

In random or dynamic access, records are written at the positions specified by the relative record numbers.

Note

If an attempt is made to create a file that already exists, the file is recreated and the original file is lost.

Extending (Sequential)

To extend a relative file, open a file in EXTEND mode, then write records sequentially with a WRITE statement. When writing to files in this mode, records are added to the end of the last record in the file.

The relative record number of the record to be written is incremented by one from the maximum relative record number in the file.

The file can be extended only in sequential access mode.

Referencing (Sequential, Random, and Dynamic)

To refer to records, open the file in INPUT mode, then read records with a READ statement.

In sequential access, specify the start position of the record to be read with a START statement, then read records sequentially from the specified record in order of the relative record numbers.

In random access, the record with the relative record number specified at execution of the READ statement is read.

Note

- When OPTIONAL is specified in the SELECT clause in the file control entry, the OPEN statement is successful even if the file does not exist. The at end condition is satisfied with the execution of the first READ statement. For information on the at end condition, see "6.6 Input-Output Error Processing".

- If the record with the specified relative record number is not found in random or dynamic access, the invalid key condition is satisfied. For information on the invalid key condition, see "6.6 Input-Output Error Processing".

Updating (Sequential, Random, and Dynamic)

To update records, open the file in I-O mode.

In sequential access, read records with a READ statement, then update them with a REWRITE statement. Executing the REWRITE statement updates the record read by the last READ statement.

In random access, specify the relative record number of the record to be updated, then execute the REWRITE statement.

Note

If the record with the specified relative record number is not found in random or dynamic access, the invalid key condition is satisfied. For information on the invalid key condition, see "6.6 Input-Output Error Processing".

Deleting (Sequential, Random, and Dynamic)

To delete records, open the file in I-O mode.

In sequential access, read records with a READ statement, then delete them with a DELETE statement. Executing the DELETE statement deletes the records read by the last READ statement.

In random access, specify the relative record number of the record to be deleted, then execute the DELETE statement.

Note

If the record with the specified relative record number is not found in random or dynamic access, the invalid key condition is satisfied. For information on the invalid key condition, see "6.6 Input-Output Error Processing".

Inserting (Random and Dynamic)

To insert records, open the file in I-O mode.

Specify the relative record number of the insertion position, then execute a WRITE statement. The record is inserted at the position of the specified relative record number.

Note

If the record with the specified relative record number already exists, the invalid key condition is satisfied. For information about the invalid key condition, see "6.6 Input-Output Error Processing".

Information

Relative record number is set to the data name specified for RELATIVE KEY clause.

```
MOVE 1 TO data-name.
```

6.5 Using Indexed Files

This section explains how to define and process files, and define records for indexed files.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. program-name.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
 FILE-CONTROL.
  SELECT file-name
         ASSIGN TO file-reference-identifier
                   ORGANIZATION IS INDEXED
                   [ACCESS MODE IS access-mode]
                   RECORD KEY IS prime-key-name-1
                       [prime-key-name-n]... [WITH DUPLICATES]
                   [[ALTERNATE RECORD KEY IS alternate-key-name-1
                       [alternate-key-name-n]... [WITH DUPLICATES]]...].
DATA DIVISION.
 FILE SECTION.
 FD file-name
    [RECORD record-size].
 01 record-name.
   02 prime-key-name-1 ... .
  [02 prime-key-name-n ... .]
  [02 alternate-key-name-1 ... .]
  [02 alternate-key-name-n ... .]
  [02 data-other-than-keys ... .]
PROCEDURE DIVISION.
    OPEN open-mode file-name.
   [MOVE prime-key-value TO prime-key-name-n.]
   [MOVE alternate-key-value TO alternate-key-name-n]
   [READ file-name.]
   [REWRITE record-name.]
   [DELETE file-name.]
   [START file-name.]
   [WRITE record-name.]
    CLOSE file-name.
END PROGRAM program-name.
```

6.5.1 Defining Indexed Files

This section explains the file definitions required to use indexed files in a COBOL program.

File Name and File-Reference-Identifier

As with a record sequential file, specify a file name and file-reference-identifier for an indexed file. For more information about how to specify these items, see "6.2.1 Defining Record Sequential Files"

File Organization

Specify INDEXED in the ORGANIZATION clause.

Access Mode

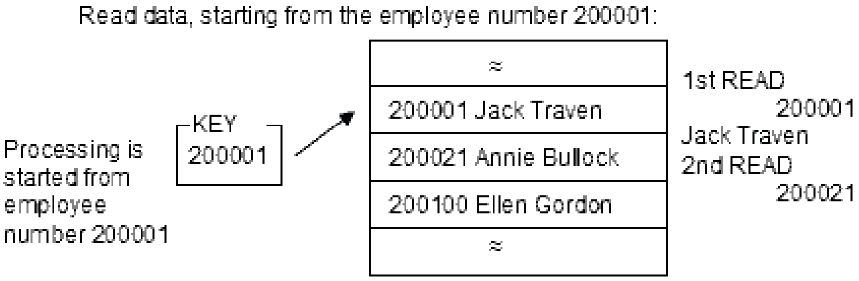
Specify one of the following access modes in the ACCESS MODE clause:

- Sequential access (SEQUENTIAL): Enables records to be processed in ascending key order from the beginning of the file or a record with a specific key.
- Random access (RANDOM): Enables a record with a specific key to be processed.
- Dynamic access (DYNAMIC): Enables records to be processed in sequential and random access modes.

Referencing records in sequential and random access modes is illustrated in the following examples.

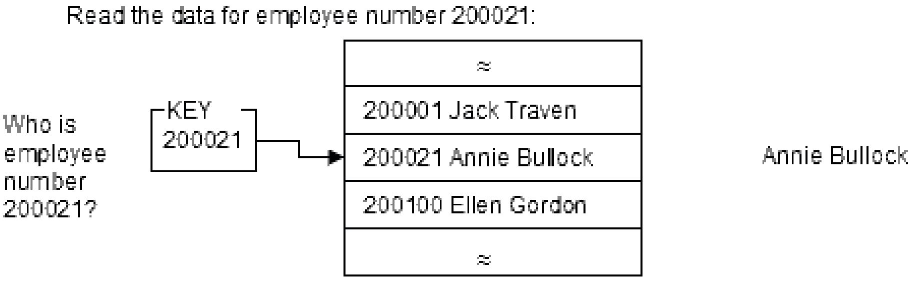
- Sequential Access

Figure 6.3 Sequential access of an indexed file



- Random Access

Figure 6.4 Random access of an indexed file



Primary and Alternate Keys

Keys are classified into primary record keys (primary keys) and alternate record keys (alternate keys). The number of keys, positions in records, and sizes are determined during file creation, and cannot be changed once determined.

Records in the file are in logical ascending order of the primary keys, and specific records can be selected with the primary key values. When defining an indexed file, specify the name of a data item as the primary key in the RECORD KEY clause.

As with the primary key, an alternate key is information used to select specific records in the file. Specify the name of a data item to use as the alternate key in the ALTERNATE RECORD KEY clause as required.

Multiple data items can be specified in the RECORD KEY and ALTERNATE RECORD KEY clauses. When multiple data items are specified in the RECORD KEY clause, the primary key consists of these data items. Data items specified in the RECORD KEY clause need not be contiguous.

Multiple records can have the same key value (duplicate key value) by specifying DUPLICATES in the RECORD KEY and ALTERNATE RECORD KEY clauses. An error occurs if the key value is duplicated when DUPLICATES is not specified.

6.5.2 Defining Indexed File Records

This section explains record formats, lengths, and organization.

Record Formats and Lengths

The explanations of the record formats and lengths of indexed files are the same as for record sequential files. See "6.2.2 Defining Record Sequential File Records".

Record Organization

Define the attributes, positions, and sizes of keys and data other than keys in records in the record description entries.

To process an existing file, the number of items, item positions, sizes, and specification order for the primary or alternate keys must be the same as those defined at file creation.

When writing two or more record description entries for a file, write the primary key data item in only one of these record description entries. The character position and size defining the primary key are applied to other record description entries.

In the variable length records format, the key must be set at the same fixed part (position from the beginning of the record is always the same).

The following is an example of record definitions in the variable length records format:

Primary key	Alternate key	
Employee number	Full-name	Section-name
6-digit number	20 characters	Up to 32 characters

```
*>     :
      RECORD KEY IS  employee-number
      ALTERNATE RECORD KEY IS  full-name.
*>     :
DATA DIVISION.
FILE SECTION.
 FD  employee-file
     RECORD IS VARYING IN SIZE FROM 27 TO 58 CHARACTERS
                               DEPENDING ON record-length.
 01  embployee-record.
    02  employee-number  PIC 9(6).
    02  full-name        PIC X(20).
    02  secton-name.
        03  PIC X OCCURS 1 TO 32 TIMES
                  DEPENDING ON section-length.
*>     :
WORKING-STORAGE SECTION.
 01  record-length   PIC 9(3) BINARY.
 01  section-length  PIC 9(3) BINARY.
```

6.5.3 Processing Indexed Files

Use input-output statements to create, extend, insert, reference, update, and delete indexed files. Some processing may not be used, depending on the access mode.

This section explains the types and input-output statements used in indexed file processing, and outlines each type of processing.

Types of Input-Output Statements

OPEN, CLOSE, DELETE, READ, START, REWRITE, and WRITE statements are used to process indexed files.

Using Input-Output Statements

OPEN and CLOSE Statements

Execute an OPEN statement only once at the beginning of file processing and a CLOSE statement only once at the end of file processing.

The open mode specified in the OPEN statement depends on the kind of file processing. For example, for creation, use the OPEN statement in the OUTPUT mode (OUTPUT specified).

Other Statements

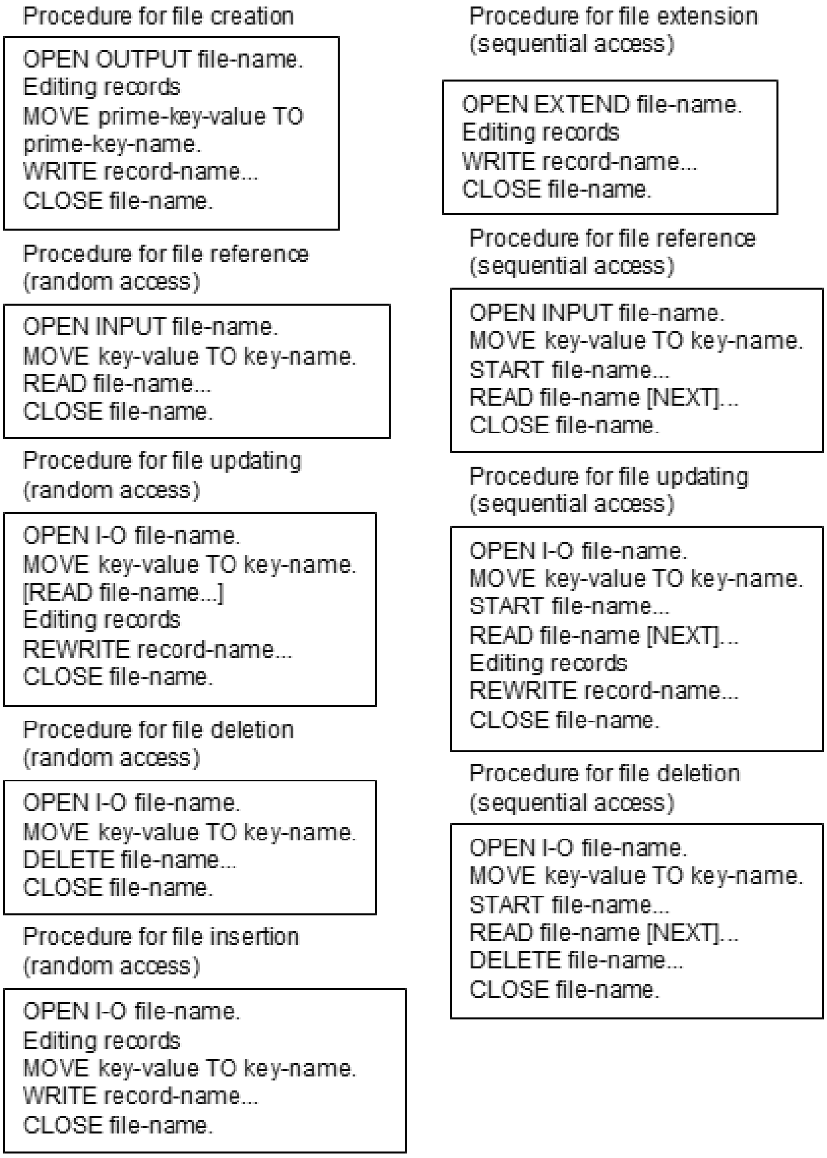
To delete records from a file, use a DELETE statement. To read records in a file, use a READ statement. To indicate a record for which processing is to be started, use a START statement.

To update records in a file, use a REWRITE statement. To write records to a file, use a WRITE statement.

Processing Outline

Execution Order of Input-Output Statements

The execution order of input-output statements for creating, extending, referencing, updating, deleting, and inserting are as follows:



Procedure for file creation

```
OPEN OUTPUT file-name.
Editing records
MOVE prime-key-value TO
prime-key-name.
WRITE record-name...
CLOSE file-name.
```

Procedure for file reference
(random access)

```
OPEN INPUT file-name.
MOVE key-value TO key-name.
READ file-name...
CLOSE file-name.
```

Procedure for file updating
(random access)

```
OPEN I-O file-name.
MOVE key-value TO key-name.
[READ file-name...]
Editing records
REWRITE record-name...
CLOSE file-name.
```

Procedure for file deletion
(random access)

```
OPEN I-O file-name.
MOVE key-value TO key-name.
DELETE file-name...
CLOSE file-name.
```

Procedure for file insertion
(random access)

```
OPEN I-O file-name.
Editing records
MOVE key-value TO key-name.
WRITE record-name...
CLOSE file-name.
```

Procedure for file extension
(sequential access)

```
OPEN EXTEND file-name.
Editing records
WRITE record-name...
CLOSE file-name.
```

Procedure for file reference
(sequential access)

```
OPEN INPUT file-name.
MOVE key-value TO key-name.
START file-name...
READ file-name [NEXT]...
CLOSE file-name.
```

Procedure for file updating
(sequential access)

```
OPEN I-O file-name.
MOVE key-value TO key-name.
START file-name...
READ file-name [NEXT]...
Editing records
REWRITE record-name...
CLOSE file-name.
```

Procedure for file deletion
(sequential access)

```
OPEN I-O file-name.
MOVE key-value TO key-name.
START file-name...
READ file-name [NEXT]...
DELETE file-name...
CLOSE file-name.
```

Creating (Sequential, Random, and Dynamic)

To create an indexed file, open a file in the OUTPUT mode, and write records to the file with a WRITE statement.

Note

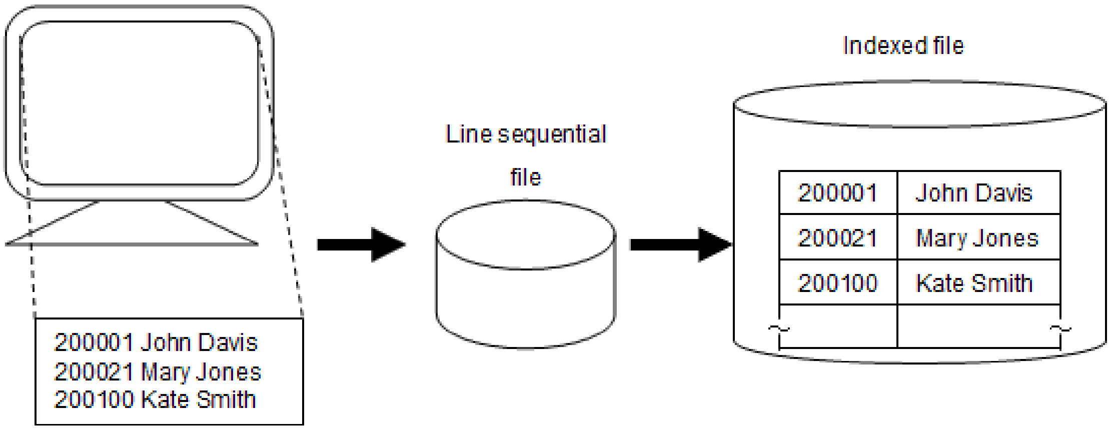
- If an attempt is made to create a file that already exists, the file is overwritten and the original file is lost.
- Before writing a record, the primary key value must be set. In sequential access, records must be written so that the primary keys are in ascending order.

Information

The following tasks are useful for collecting data to create index files:

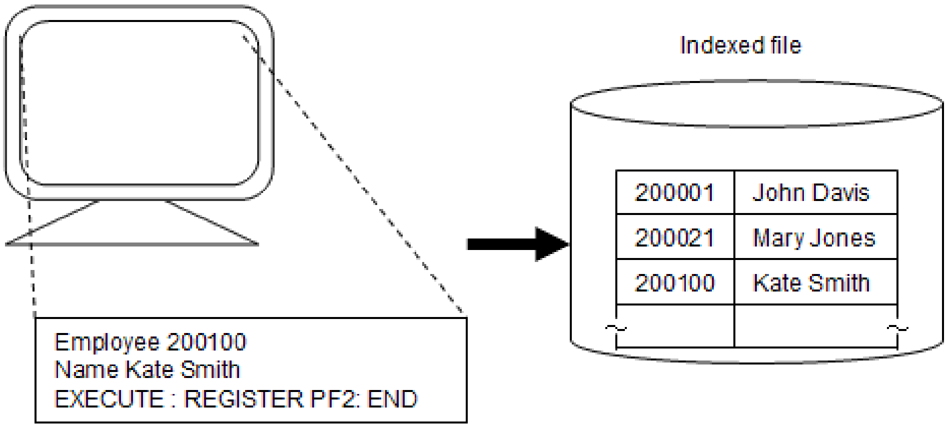
- Create data with the editor, read the data with the line sequential file function, and then write it to an indexed file, as shown in the following example. (Create the records in key order or sort the file.)

Figure 6.5 Writing data to an indexed file



- For screen handling/screen operation functions, enter data from the screen and then write it to an indexed file, as shown in the following example.

Figure 6.6 Writing data from the screen to an indexed file



Extending (Sequential)

To extend an indexed file, open the file in EXTEND mode, then write records sequentially. At this time, records are added to the end of the last record in the file.

The file can be extended only in sequential access mode.

Note

To insert the records to be written, the primary key values must be in ascending order.

When DUPLICATES is not specified in the RECORD KEY clause in the file control entry, the primary key value first processed must be greater than the maximum primary key value in the file.

When DUPLICATES is specified, the primary key value first processed must be equal to or greater than the maximum primary key value in the file.

Referencing (Sequential, Random, and Dynamic)

To refer to records, open the file in INPUT mode, then read records with a READ statement.

In sequential access, specify the start position of the record to be read with a START statement. Then, start reading records from the specified position in ascending order of the primary or alternate key values.

In random access, the records to be read are determined by the primary or alternate key values.

Note

- When OPTIONAL is specified in the SELECT clause in the file control entry, the OPEN statement is successful even if the file does not exist. The at end condition is satisfied with the execution of the first READ statement. For information about the at end condition, see "6.6 Input-Output Error Processing".
- If the record with the specified key value is not found in RANDOM or DYNAMIC access, the invalid key condition is satisfied. For information on the invalid key condition, see "6.6 Input-Output Error Processing".
- A START or READ statement can be executed by specifying multiple keys.

Updating (Sequential, Random, and Dynamic)

To update records, open the file in I-O mode, then rewrite records in the file.

In sequential access, records read with the last READ statement are updated.

In random access, the primary key records with the specified key values are updated.

Note

- If the record with the specified key value is not found in RANDOM or DYNAMIC access, the invalid key condition is satisfied. For information on the invalid key condition, see "6.6 Input-Output Error Processing".
- The contents of primary keys cannot be changed. The contents of alternate keys, however, can be changed.

Deleting (Sequential, Random, and Dynamic)

To delete records, open the file in I-O mode, then delete records from the file.

In sequential access, records read with the last READ statement are deleted.

In random access, the primary key records with the specified key values are deleted.

Note

If the record with the specified key value is not found in random or dynamic access, the invalid key condition is satisfied. For information about the invalid key condition, see "6.6 Input-Output Error Processing".

Inserting (Random and Dynamic)

To insert records, open the file in I-O mode, then insert records in the file. The record insertion position is determined by the primary key value.

Note

When DUPLICATES is not specified in the RECORD KEY or ALTERNATE RECORD KEY clause in the file control entry, and the record with the specified key value already exists, the invalid key condition is satisfied. For information about the invalid key condition, see "6.6 Input-Output Error Processing".

6.6 Input-Output Error Processing

This section explains input-output error detection methods and execution results if input-output errors occur.

The four input-output error detection methods are:

- AT END specification (detecting occurrence of the at end condition)
- INVALID KEY specification (detecting occurrence of the invalid key condition)
- FILE STATUS clause (detecting input-output errors by checking I-O status)
- Error procedures (detecting input-output errors)

6.6.1 AT END Specification

The AT END condition occurs when all records in a file are read sequentially and no next logical record exists in the file. To detect occurrence of the at end condition, specify AT END in a READ statement. With AT END specified, processing to be done at occurrence of the AT END condition can be written.

A coding example of a READ statement with AT END specified is:

```
     READ sequential-file AT END
          GO TO at-end-processing
     END-READ.
```

With the execution of the next READ statement after the last record in the file is read, the at end condition occurs and the GO TO statement is executed.

6.6.2 INVALID KEY Specification

In input-output processing with an indexed or relative file, an input-output error occurs if no record with a specified key or relative record number exists in the file. This is called an invalid key condition.

To detect occurrence of the invalid key condition, specify INVALID KEY in the READ, WRITE, REWRITE, START, and DELETE statements. With INVALID KEY specified, processing to be done upon the occurrence of an invalid key condition can be written.

A coding example of a READ statement with INVALID KEY specification is:

```
     MOVE "Z" TO primary-key.
     READ indexed-file INVALID KEY
                  GO TO invalid-key-processing
     END-READ.
```

If no record with primary key value "Z" is present in the file, the invalid key condition occurs and the GO TO statement is executed.

6.6.3 FILE STATUS Clause

When a FILE STATUS clause is written in the file control entry, the I-O status is posted to the data-name specified in the FILE STATUS clause upon execution of the input-output statement. By writing a statement (IF or EVALUATE statement) to check the contents (I-O status value) of this data name, input-output errors can be detected.

If the I-O status value not is checked after the input-output statement, program execution continues even if an input-output error occurs. Subsequent program operation is unpredictable.

For I-O status values to be posted, refer to "Appendix B I-O Status List".

A coding example of a FILE STATUS clause is:

```
   SELECT file-1
          FILE STATUS IS input-output-status
*>        :
```

```
WORKING-STORAGE SECTION.
 01 input-output-status PIC X(2).
PROCEDURE DIVISION.
    OPEN INPUT file-1.
    IF input-output-status NOT = "00"
       THEN GO TO open-error-processing.
```

If a file could not be opened, a value other than "00" is set for the input-output value. The IF statement checks this value, and the GO TO statement is executed.

6.6.4 Error Procedures

You specify error procedures by writing a USE AFTER ERROR/EXCEPTION statement in Declaratives in the PROCEDURE DIVISION.

Specifying error procedures executes the statements in the error procedures if an input-output error occurs. After executing error processing, control is passed to the statement immediately after the input-output statement at which an input-output error occurred. Thus, a statement indicating control of processing of the file where an input-output error occurred must be written immediately after the input-output statement.

Control is not passed to the error procedures in the following cases:

- AT END is specified in the READ statement with which the at end condition occurred
- INVALID KEY is specified in the input-output statement with which the invalid key condition occurred
- An input-output statement is executed before the file is opened (the open mode is specified)

Example of a branch after error procedures to elsewhere in the PROCEDURE DIVISION with the GO TO statement in the following cases:

- An input-output statement is executed for the file in which an input-output error occurred
- Error procedures are re-executed before they are terminated

```
PROCEDURE DIVISION.
 DECLARATIVES.
  error-procedures SECTION.
     USE AFTER ERROR PROCEDURE ON file-1.
        MOVE error-occurrence TO file-status.       *> (1)
 END DECLARATIVES.
*>       :
     OPEN INPUT file-1.
     IF file-status = error-occurrence THEN         *> (2)
        GO TO open-error-processing.
*>       :
```

If a file could not be opened, error procedures (MOVE statement of (1)) is executed, and control is passed to the statement (IF statement of (2)) immediately after the OPEN statement.

6.6.5 Input-Output Error Execution Results

The execution result when an input-output error occurs depends on whether the AT END specification, INVALID KEY specification, FILE STATUS clause, and error procedures are written. The following table lists execution results when input-output errors occur.

Table 6.5 Execution results when input-output errors occur

Error type		With error procedures		Without error procedures	
		With the FILE STATUS clause	Without the FILE STATUS clause	With the FILE STATUS clause	Without the FILE STATUS clause
AT END condition or invalid key condition	Detected by using an executed input-output	The statement written for the AT END or INVALID KEY specification is executed.			

Error type		With error procedures		Without error procedures	
		With the FILE STATUS clause	Without the FILE STATUS clause	With the FILE STATUS clause	Without the FILE STATUS clause
	statement with AT END or INVALID KEY specified				
	Detected by using an executed input-output statement without AT END or INVALID KEY specified	Error procedures are executed immediately after the input-output statement in which an error occurred.		The statement immediately following an input-output statement (in which an error occurred) is executed.	A U-level message is output and the program terminates abnormally.
Other input-output errors		An error procedure is executed, then execution continues with the statement immediately following the I-O statement in which the error occurred.		An I-level message is output then the statement immediately following an input-output statement (in which an error occurred) is executed.	A U-level message is output and the program terminates abnormally.

Information

- When a valid error procedure is specified, if the environment variable @CBR_FILE_USE_MESSAGE=YES is set, an I-level message is output.

```
@CBR_FILE_USE_MESSAGE=YES
```

- A U-level message is output when an environment variable error occurs and information is provided when the application is executed using the COBOL Error Report. Do not specify an error procedure or the FILE STATUS phrase to output the diagnosis report with the COBOL Error Report. Correct the program temporarily via debugging. Refer to "Using the COBOL Error Report" in the "NetCOBOL Debugging Guide" for debugging the program using the COBOL Error Report.

6.7 File Processing

This section explains file assignment, file processing results, exclusive control of files, and methods for improving file processing.

6.7.1 Assigning Files

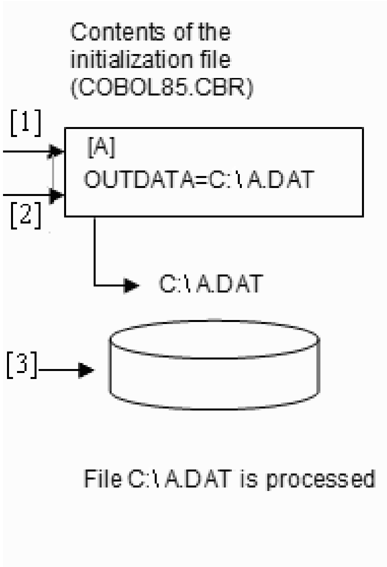
The method of file input-output processing at program execution is determined by the contents of the ASSIGN clause in the file control entry. The relation between the contents of the ASSIGN clause and files is explained below.

Using a File Identifier in the ASSIGN Clause

Map the file identifier to the real file name by setting the file identifier as a runtime environment variable. For details about how to set runtime environment variables, refer to "5.3 Setting Runtime Environment Information".

The following is an example of setting runtime environment variable information by using an initialization file.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. A.             *>---[1]
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT  file-1
       ASSIGN TO OUTDATA.   *>---[2]
DATA DIVISION.
FILE SECTION.
  FD file-1.
  01 record-1  PIC X(80).
PROCEDURE DIVISION.
    OPEN OUTPUT  file-1.    *>--+
    WRITE  record-1.        *>   |[3]
    CLOSE  file-1.          *>--+
    EXIT PROGRAM.
END PROGRAM A.
```


Contents of the initialization file (COBOL85.CBR)
[1]
[A]
OUTDATA=C:\A.DAT
[2]
C:\A.DAT
[3]
File C:\A.DAT is processed

Note

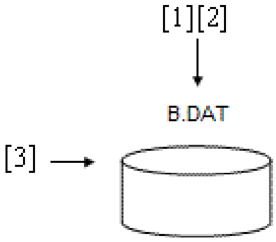
- When a lowercase file identifier is specified in the ASSIGN clause:
    - A compiler error occurs if a file is compiled with compiler option NOALPHAL specified.
    - When setting runtime environment variable information, use uppercase letters.
- If runtime environment variable information is left blank, a file assignment error occurs.

Using a Data Name in the ASSIGN Clause

Input-output processing uses the file name specified in the data item. For example:

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. B.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT  file-1
       ASSIGN TO data-name-1.     *>---[1]
DATA DIVISION.
FILE SECTION.
```

```
  FD file-1.
  01 record-1  PIC X(80).
WORKING-STORAGE SECTION.
  01 data-name-1 PIC X(30).
PROCEDURE DIVISION.
    MOVE "B.DAT" TO data-name-1.  *>---[2]
    OPEN OUTPUT  file-1.          *>--+
    WRITE  record-1.              *>  |[3]
    CLOSE  file-1.                *>--+
    EXIT PROGRAM.
END PROGRAM B.
```



**When program B is executed, file B.DAT is processed.

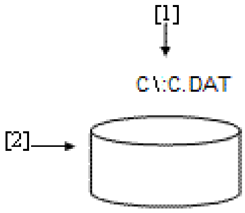
Note

- When the file name specified in the program is a relative path name, the current folder is prefixed to the relative path name.
- If the data name is left blank, a file assignment error occurs.

Using a File-Identifier Literal in the ASSIGN Clause

Input-output processing uses the file whose name is written as the file-identifier literal. For example:

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. C.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT  file-1
       ASSIGN TO "C:\C.DAT".      *>---[1]
DATA DIVISION.
FILE SECTION.
  FD file-1.
  01 record-1  PIC X(80).
PROCEDURE DIVISION.
    OPEN OUTPUT  file-1.          *>--+
    WRITE  record-1.              *>  |[2]
    CLOSE  file-1.                *>--+
    EXIT PROGRAM.
END PROGRAM C.
```



**When program C is executed, file C\:C.DAT is processed.

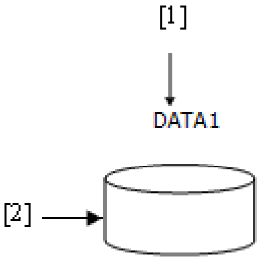
Note

When the file name written in the program is a relative path name, the file having the current folder name prefixed is eligible for input-output processing.

Using "DISK" in the ASSIGN Clause

Input-output processing uses the file name specified in the SELECT clause. For example:

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. D.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT DATA1
       ASSIGN TO DISK.            *>---[1]
DATA DIVISION.
FILE SECTION.
  FD DATA1.
  01 record-1  PIC X(80).
PROCEDURE DIVISION.
    OPEN OUTPUT DATA1.            *>--+
    WRITE  record-1.              *>  |[2]
    CLOSE DATA1.                  *>--+
    EXIT PROGRAM.
END PROGRAM D.
```



**When program D is executed, file DATA1 is processed.

Note

Files under the current folder are eligible for input-output processing.

6.7.2 Exclusive Control of Files

Disable access by other programs during file processing by setting the exclusive mode for files or locking records in use. This is called exclusive control of files.

Exclusive control of files is ensured when accessing the files only when using COBOL programs, the COBOL file access routines, and the COBOL file utility. When different languages and tools access those files, exclusive control of files is not ensured.

This section explains the relationship between file processing and exclusive control of files.

6.7.2.1 Setting Files in Exclusive Mode

When a file is opened in exclusive mode, other users cannot access the file.

A file is opened in exclusive mode in the following cases:

- An OPEN statement is executed for a file with EXCLUSIVE specified in the LOCK MODE clause in the file control entry.
- An OPEN statement in other than INPUT mode is executed for a file without the LOCK MODE clause specified in the file control entry.
- An OPEN statement with the WITH LOCK phrase is executed.
- An OPEN statement with the OUTPUT mode is executed.

Table below shows the file sharing modes for the possible combinations of LOCK MODE, OPEN with or without the WITH LOCK phrase, and the OPEN statement modes.

Table 6.6 File Sharing Modes

	LOCK MODE Not Specified		LOCK MODE EXCLUSIVE		LOCK MODE AUTOMATIC OR MANUAL	
OPEN statement MODE	OPEN ... WITH LOCK	NO "WITH LOCK" phrase	OPEN ... WITH LOCK	NO "WITH LOCK" phrase	OPEN ... WITH LOCK	NO "WITH LOCK" phrase
INPUT	Exclusive	Shared	Exclusive	Exclusive	Exclusive	Shared
I-O	Exclusive	Exclusive	Exclusive	Exclusive	Exclusive	Shared
EXTEND	Exclusive	Exclusive	Exclusive	Exclusive	Exclusive	Shared
OUTPUT	Exclusive	Exclusive	Exclusive	Exclusive	Exclusive	Exclusive

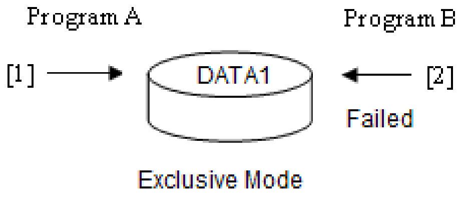
Figure below shows an example of exclusive control of files:

[Program A]

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. A.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT file-1
     ASSIGN TO "DATA1"
     LOCK MODE IS EXCLUSIVE.
 DATA DIVISION.
  FILE SECTION.
  FD  file-1.
  01  record-1  PIC X(80).
 PROCEDURE DIVISION.
     OPEN I-O file-1.      *>---[1]
```

[Program B]

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. B.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT file-2
     ASSIGN TO data-name
     FILE STATUS IS FS.
 DATA DIVISION.
 FILE SECTION.
  FD  file-2.
  01  record-1  PIC X(80).
 WORKING-STORAGE SECTION.
  01 data-name PIC X(12).
  01 FS        PIC X(2).
 PROCEDURE DIVISION.
     MOVE "DATA1" TO data-name.
     OPEN I-O file-2.              *>---[2]
     IF FS = "93" THEN             *>---[3]
       MOVE "DATA2" TO data-name
       OPEN I-O file-2
     END-IF.
```



- [1] File DATA1 is opened in exclusive mode.
- [2] If an attempt is made to execute an OPEN statement for file DATA1 that is already in use in exclusive mode by program A, the attempt fails.
- [3] I-O status value "93" (error caused by exclusive control of files) is set for the data name specified in the FILE STATUS clause.

6.7.2.2 Locking Records

If records are locked, other users cannot lock them. To lock an individual record or set of records, open the file containing the record in share mode.

A file is opened in share mode in the following cases:

- An OPEN statement without WITH LOCK specified in other than OUTPUT mode is executed for a file with AUTOMATIC or MANUAL specified in the LOCK MODE clause of the file control entry.
- An OPEN statement with the INPUT mode is executed to a file not specified with LOCK MODE clause.

Files opened in share mode can be used by other users. If a file is already in use in exclusive mode by another user, however, an OPEN statement in share mode fails.

Records in a file opened in share mode are locked with the execution of a READ statement with exclusive control specified.

Records are locked in the following cases:

- A file with AUTOMATIC specified in the LOCK MODE clause in the file control entry is opened in I-O mode, and a READ statement is executed that does not have a WITH NO LOCK phrase.
- A file with MANUAL specified in the LOCK MODE clause in the file control entry is opened in I-O mode, and a READ statement containing the WITH LOCK phrase is executed.

Table below summarizes the above combinations.

Table 6.7 Record lock status

LOCK MODE clause description	AUTOMATIC			MANUAL		
READ WITH LOCK phrase	No phrase	WITH LOCK	WITH NO LOCK	No phrase	WITH LOCK	WITH NO LOCK
Record locked	Yes	Yes	No	No	Yes	No

Records are released from lock in the following cases:

- For a file with AUTOMATIC specified in the LOCK MODE clause
  - A READ, REWRITE, WRITE, DELETE, or START statement is executed
  - An UNLOCK statement is executed
  - A CLOSE statement is executed
- For a file with MANUAL specified in the LOCK MODE clause
  - An UNLOCK statement is executed
  - A CLOSE statement is executed

The following is an example of locking records.

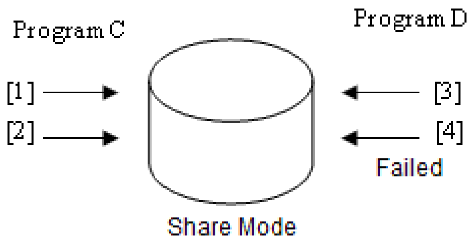
Example

[Program C]

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. C.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT file-1
     ASSIGN TO SAMPLE
     LOCK MODE IS AUTOMATIC.
 DATA DIVISION.
  FILE SECTION.
  FD  file-1.
  01  record-1  PIC X(80).
 PROCEDURE DIVISION.
     OPEN I-O file-1.      *>---[1]
     READ file-1.          *>---[2]
```

[Program D]

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. D.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT file-2
     ASSIGN TO SAMPLE
     LOCK MODE IS AUTOMATIC
     FILE STATUS IS FS.
 DATA DIVISION.
 FILE SECTION.
  FD  file-2.
  01  record-1  PIC X(80).
 WORKING-STORAGE SECTION.
  01 FS        PIC X(2).
 PROCEDURE DIVISION.
     OPEN I-O file-2.             *>---[3]
     READ file-2.                 *>---[4]
     IF FS = "99" THEN            *>---[5]
       GO TO exclusive-error
     END-IF.
```



- [1] The file is opened in share mode.
- [2] The first record in the file is locked upon execution of a READ statement.
- [3] The file is opened in share mode.
- [4] A READ statement for the locked record fails.
- [5] I-O status value "99" (error caused by locking records) is set for the data name specified in the FILE STATUS clause.

6.7.3 File Processing Results

File processing generates new files or updates files. This section explains file status when file processing is done.

File creation

Generates a new file upon execution of an OPEN statement. If a file with the same name already exists, the file is overwritten and the original contents are lost.

File extension

Extends an existing file upon execution of a WRITE statement. If an attempt is made to extend a file not existing at program execution (optional file), a new file is generated upon execution of an OPEN statement.

Record reference

Does not change the contents of the file. With an optional file, if the file does not exist, the AT END condition occurs with the first READ statement.

Record updating, deletion, and insertion

Changes the contents of an existing file with the execution of a REWRITE, DELETE, or WRITE statement. With an optional file, a new file is generated with the execution of an OPEN statement. Since no data exists in this file, however, the at end condition occurs with the first READ statement.

Note

- When a program is terminated without executing a CLOSE statement, files are closed unconditionally.
  An unconditional close is performed when the following statements are executed.
  - STOP RUN statement
  - EXIT PROGRAM statement in main program
  - CANCEL statement of an external program
  - JMPCINT3 call

- If the unconditional close fails, a message is output, and the files become unusable, and they remain open.

- A file can be assigned with the file identifier as a runtime environment variable name. If so, an input-output statement is executed for the file assigned with the file identifier, even if the file assignment destination is changed with an environment variable function while the file is open.
  When an OPEN statement is executed after the file assigned with the file identifier is closed with a CLOSE statement, subsequent input-output statements are executed for the file changed with the environment variable function.
  Therefore, use the OPEN statement to start the sequence of file processing and the CLOSE statement to terminate the sequence.

- If the area is insufficient for file creation, extension, record updating, or insertion, subsequent operation for the file is undefined. If an attempt is made to write records to the file when the area is insufficient, how they are stored in the file is also undefined.

- When an indexed file is opened in OUTPUT, I-O, or EXTEND mode, it may become unusable if the program terminates abnormally before it is closed. Therefore, make backup copies before executing programs that may terminate abnormally.

  Files that became unusable can be recovered with the Recovery command of the COBOL File Utility. A function having the same feature as the above command can be called from an application. For more information, see "6.8 COBOL File Utility" and "6.10 Recovering Indexed Files".

6.7.4 High-Speed File Processing

Record sequential files and line sequential files can be accessed faster by specifying an available range.

High speed file processing can be used in the following cases:

- The file is opened in exclusive mode, and is written to as an output file.

- The input restricted file is read.

This section explains methods of specification for high-speed file processing and notes on high-speed file processing.

Specification Methods

- When defining a file-identifier as a file-reference-identifier in the program, specify ",BSAM" following the file-name in the environment variable. Refer to "5.4.1.66 File identifier(Set the File Used by the Program)" for details on setting environment variables information.

```
file-identifier=[path-name]file-name,BSAM
```

- When defining a data name as a file-reference-identifier in the program, specify ",BSAM" following the file-name. When the drive name is omitted, it is considered the current drive.

```
MOVE "[path-name]file-name,BSAM" TO data-name.
```

- When defining a file-identifier literal as a file-reference-identifier in the program, specify ",BSAM" following the file-name in the file-identifier literal.

```
ASSIGN TO "[path-name]file-name,BSAM"
```

Note

- Records cannot be updated (the REWRITE statement cannot be executed). If a record is updated, an error occurs during execution (Record sequential files only).

- If files are shared and you want to permit file sharing among different processes, all files must be in shared mode and opened with INPUT specified. Operation is not guaranteed if a file is opened without specifying INPUT. A single process may not allocate the file multiple times. Operation is not guaranteed if a file is shared within the same process.
  Files open in exclusive mode except when the open mode is INPUT. Therefore, when accessing it from other programs, it is likely to become an open error. Refer to "Notes on shared files" for detail.

- High-speed file processing cannot be used if DISK is specified as the file-reference-identifier.

- If a record read from a line sequential file includes a tab, the tab code is not replaced by a blank. And, if it includes control character 0x0C (page feed), 0x0D (return) or 0x1A (data end symbol), it is not handled as a record delimiter. For more information, see "6.3.3 Processing Line Sequential Files".
- The ADVANCING clause is ignored when a record is written. If the ADVANCING clause is specified, it produces the same as a WRITE statement without an ADVANCING clause.

Package specification

This section describes how to specify high-speed file processing.

Usage

To enable high-speed file processing, specify "BSAM" for environment variable @CBR_FILE_SEQUENTIAL_ACCESS.

```
@CBR_FILE_SEQUENTIAL_ACCESS = BSAM
```

High speed processing can be used for the following:

File Organization	- Record sequential file - Line sequential file
ASSIGN clause	- File identifier - File identifier literal - Data name - DISK

High speed processing cannot be used for the following:

Function	File Function Name
Appending to Existing Files (*)	MOD
Concatenating Files (*)	CONCAT
Dummy File	DUMMY
Other File System	BTRV
	EXFH

* : This function can be used for the file when you make high-speed processing of that file is enabled at the same time. Refer to "6.7.8 Caution" for more information.

Note

The limitations of "High-speed processing of the file" apply when this environment variable is specified. In particular, , the operation of the application might change if the file has been opened in shared mode. Do not specify this environment variable when there is a file that corresponds to these limitations. Refer to "Notes on shared files" for more information.

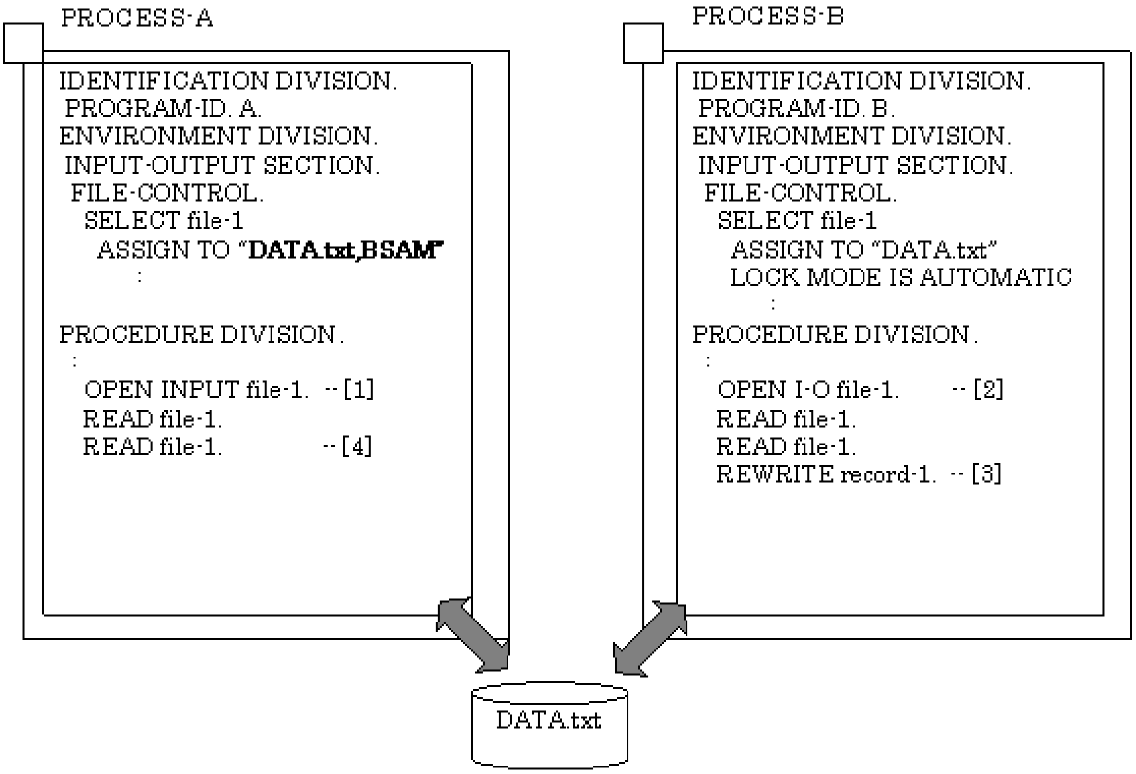
Notes on shared files

Examples of issues that may be encountered when a file is shared are shown below.

Example

Example 1

This function cannot be used for a runtime system which refreshes the record using other processes.



```
IDENTIFICATION DIVISION.
 PROGRAM-ID. A.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
      ASSIGN TO "DATA.txt,BSAM"
         :

PROCEDURE DIVISION.
  :
    OPEN INPUT file-1.  -- [1]
    READ file-1.
    READ file-1.             -- [4]
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. B.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
      ASSIGN TO "DATA.txt"
      LOCK MODE IS AUTOMATIC
         :
PROCEDURE DIVISION.
  :
    OPEN I-O file-1.        -- [2]
    READ file-1.
    READ file-1.
    REWRITE record-1.  -- [3]
```

[1] PROCESS-A : The file is opened in shared mode with INPUT specified. (High-speed File Processing)

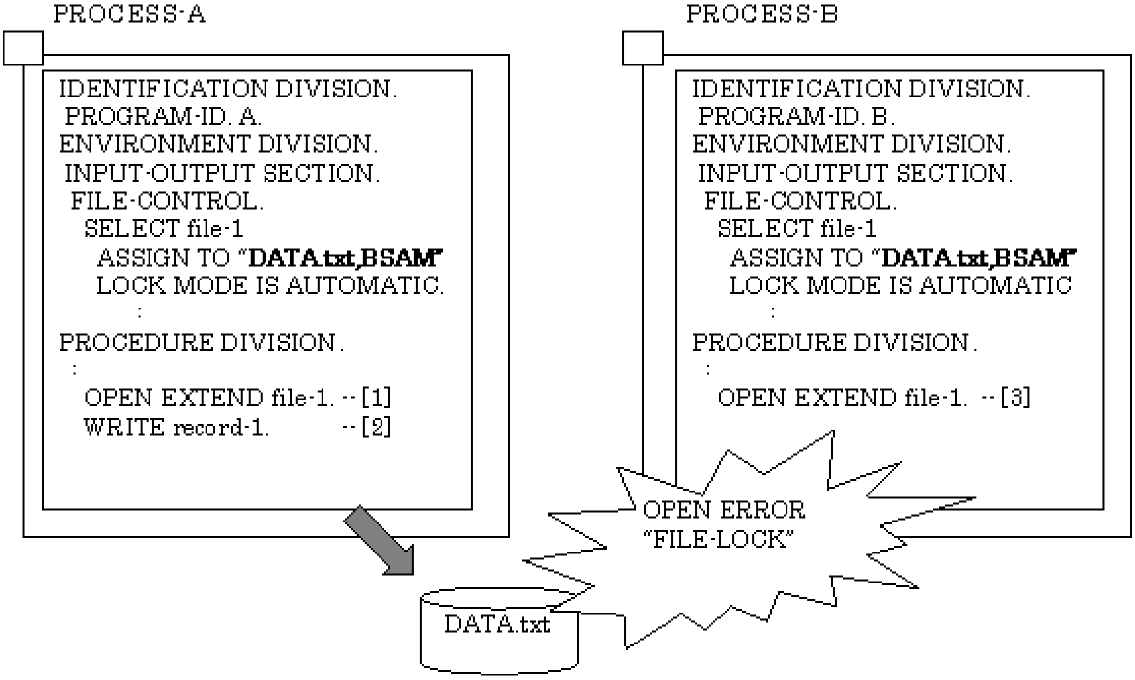
[2] PROCESS-B : The file is opened in shared mode with I-O specified.

[3] PROCESS-B : The second record is refreshed.

[4] PROCESS-A : The second record is read. At this time, stale data might be read.

Example 2

An error occurs on the second OPEN when writing to a shared file.



```
IDENTIFICATION DIVISION.
 PROGRAM-ID. A.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
      ASSIGN TO "DATA.txt,BSAM"
      LOCK MODE IS AUTOMATIC.
         :
PROCEDURE DIVISION.
  :
    OPEN EXTEND file-1. -- [1]
    WRITE record-1.         -- [2]
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. B.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
      ASSIGN TO "DATA.txt,BSAM"
      LOCK MODE IS AUTOMATIC
         :
PROCEDURE DIVISION.
  :
    OPEN EXTEND file-1.  -- [3]
```

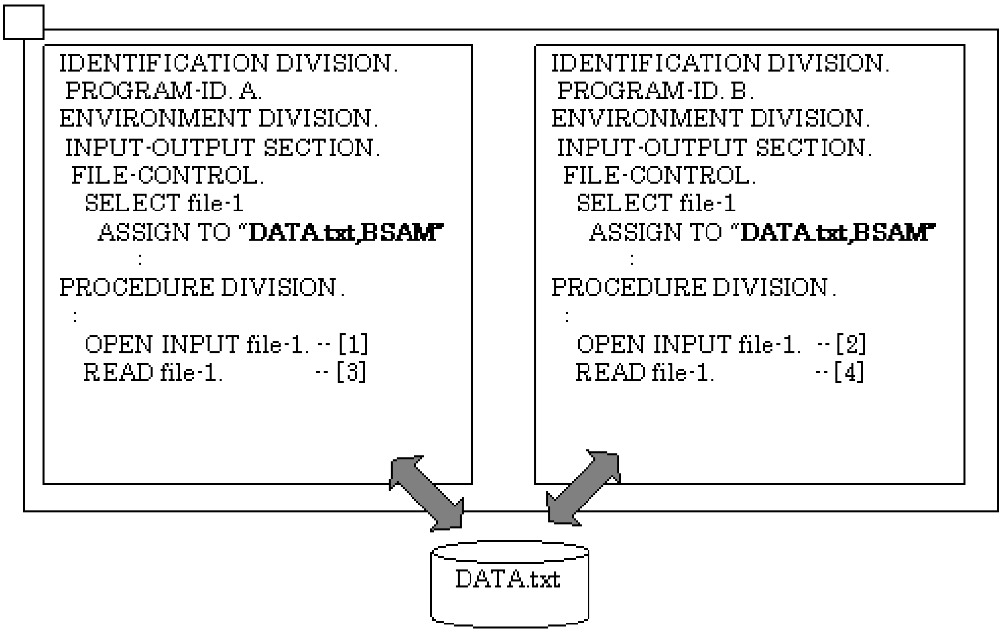
[1] PROCESS-A : The file is opened in shared mode with EXTEND specified.

[2] PROCESS-A : The record is written.

[3] PROCESS-B : The file is opened in shared mode with EXTEND specified. At this time, an error occurs in the OPEN statement.

Example 3

When the file is shared in the same process, this function cannot be used.



```
IDENTIFICATION DIVISION.
 PROGRAM-ID. A.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
      ASSIGN TO "DATA.txt,BSAM"
            :
PROCEDURE DIVISION.
  :
    OPEN INPUT file-1. -- [1]
    READ file-1.       -- [3]
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. B.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
      ASSIGN TO "DATA.txt,BSAM"
            :
PROCEDURE DIVISION.
  :
    OPEN INPUT file-1.  -- [2]
    READ file-1.        -- [4]
```

[1] PROGRAM-A : The file is opened in shared mode with INPUT specified.

[2] PROGRAM-B : The file is opened in shared mode with INPUT specified.

[3] PROGRAM-A : The first record is read.

[4] PROGRAM-B : The first record is read. At this time, the second and subsequent records might be read or the at-end-condition might be generated.

6.7.5 Appending to Existing Files

To indicate that you want to append data to an existing file, specify ",MOD" after the access type or file system type in the file identifier.

```
File identifier=file-name,,MOD
```

(This example shows a file identifier with no special access or file system type.)

In your program execute an OPEN OUTPUT statement and records written will be appended to the existing file.

Note

This is only supported with COBOL record sequential files.

6.7.6 Concatenating Files

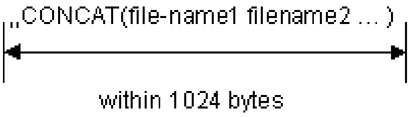
Multiple files can be concatenated (logically) for referencing and updating records.

To do this, specify ",,CONCAT" in the file identifier after the access type or file system type. With CONCAT you specify the list of files that are to be concatenated as the data is read, in the order you want the data to be concatenated, as in:

```
File identifier=,,CONCAT(file-name1 file-name2 ...)
```

Note

- File names are delimited by spaces.
- If a file name contains a space, that file name must be enclosed in double quotation marks (").
- This is only supported with COBOL record sequential files.
- Using either the OUTPUT or EXTEND specifications with the OPEN statement of a file whose file identifier contains the CONCAT option causes an error at runtime.
- If the same file is specified more than once, the files are treated as separate files.
- The maximum length that can be specified for File identifier is 1024 bytes. Therefore, the combined file name character string cannot exceed 1024 bytes.



When the OPEN statement is executed, an error is generated when the combined file name character string specified exceeds 1024 bytes.

6.7.7 Dummy File

When a dummy file is specified, a physical file is not actually created. A dummy file is useful when an output file is not needed, or when no input file is available during development. For example, a dummy file can be created in place of a physical log file when an error occurs. And when a program is being developed and no input file is available, a dummy file can be used for testing purposes.

Usage

",DUMMY" is specified for the file identifier following the file name of the allocated file. The file name can be omitted.

```
File identifier=[file-name],DUMMY
```

Note

- Place a comma (,) in front of the character string "DUMMY", otherwise the character string "DUMMY" will be interpreted as a file name.
- Dummy file operation is the same regardless of whether the file name is specified. Even if the specified file exists, no operation is done to the file.
- For more information about specifying file identifiers see "5.4.1.66 File identifier(Set the File Used by the Program)".

Functional range

Table below shows the range where the dummy file becomes effective.

Table 6.8 Functional Range of Dummy File

File organization	Record sequential file Line sequential file Relative file Index file

Open mode	OUTPUT EXTEND I-O INPUT		
I-O statement	OPEN statement	The execution of the input/output statement succeeds.	
	CLOSE statement		
	WRITE statement		
	START statement		
	UNLOCK statement		
	READ statement	Sequential access	AT END condition is occurred.
		Random access	INVALID KEY condition is occurred.
	REWRITE statement DELETE statement	Sequential access	Because the READ statement ahead unsuccessfully becomes it, it becomes the execution order mistake.
		Random access	INVALID KEY condition is occurred.

6.7.8 Caution

Byte number of character string that can be specified for file-identifier

Please specify the character string of 1024 bytes or less for file-identifier.

When the character string exceeds 1024 bytes, the character string to 1024 bytes is processed considering that it is effective.

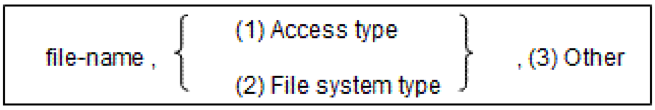
However, it becomes an error at OPEN statement execution time when the character string exceeds 1024 bytes during the specification of the concatenating files function.

Combination of specifiable at the same time file function

The file function can be classified into 3 types.

Types	Function Name	Function
(1) Access type	BSAM	High-speed file processing
	DUMMY	Dummy file
(2) File system type	BTRV	Btrieve file
	EXFH	External File Handler
(3) Other	MOD	Appending to Existing Files
	CONCAT	Concatenating Files

Specify the file function using the following format. When the specification order is different, the file function is ignored.



```
            { (1) Access type      }
file-name , {                      } , (3) Other
            { (2) File system type }
```

The combination and the operation that can be specified at the same time are as follows.

Example

- High-speed file processing (BSAM) in (1) and (3) can be specified at the same time. Both are valid.
  - **Example 1** High-speed file processing (BSAM) and (3)

```
file-name, BSAM,MOD
,BSAM, CONCAT(file-name1  file-name2  ...)
```

- Dummy File (DUMMY) in (1) and other functions can be specified at the same time. However, only the Dummy File becomes effective; other functions are invalid.
  - **Example 2** High-speed file processing (BSAM) and DUMMY

```
file-name,BSAM,DUMMY
```

  - **Example 3** (2) and DUMMY

```
           ┌ BTRV                              ┐
file-name, ┤                                   ├ ,DUMMY
           └ EXFH[,INF(information-file)]      ┘
```

  - **Example 4** (3) and DUMMY

```
file-name,,MOD,DUMMY
,,CONCAT(file-name1  file-name2  ...),DUMMY
```

NOTE: DUMMY can be specified right after the first comma.

  - **Example 5** High-speed file processing (BSAM), (3), and DUMMY

```
file-name,BSAM,MOD,DUMMY
,BSAM,CONCAT(file-name1  file-name2  ...),DUMMY
```

- DUMMY in (1) and (3) becomes an error when the OPEN statement is executed, except when the specification of ASSIGN clause is file identifier name.
- When an invalid combination is specified: When (2) is specified first, the function specified later becomes invalid and processing continues. An error occurs when the OPEN statement is executed excluding the invalid function.

6.8 COBOL File Utility

This section explains the COBOL File Utility.

The COBOL File Utility allows you to use utility commands to process COBOL file systems, without using COBOL applications. In this section, files (record sequential, line sequential, relative, and indexed files) handled by COBOL file systems are simply called COBOL files.

With the COBOL File Utility, you can create COBOL files based on data generated with text editors, and manipulate COBOL files and records (such as display, edit, and sort).

Other operations for COBOL files include copying, moving, and deleting files, converting file organization, and reorganizing, recovering, and displaying attributes of indexed files. These utilities can be easily operated by selecting menus in the COBOL Utility window or by using the command and function within a COBOL program.

6.8.1 Using the COBOL File Utility

This section explains how to use the COBOL File Utility.

1. Setting Up Environments
   - Specify the name of the path containing the COBOL runtime system in the environment variable, PATH.

- Specify the environment variable TEMP with the full path name of a temporary file storage folder. When the Recovery command is executed with these utilities, a temporary work file (UTYnnnn.TMP (nnnn indicates alphanumeric characters)) of the same size as the file to be processed is generated under the specified folder.
- You may specify a full path name to a folder for a sort work file in environment variable BSORT_TMPDIR. When the Sort command is executed with these utilities, a temporary file is generated under the specified folder. Refer to "11.2.4 Program Execution" for the work file used when environment variable BSORT_TMPDIR is not specified.

2. Activating the COBOL File Utility

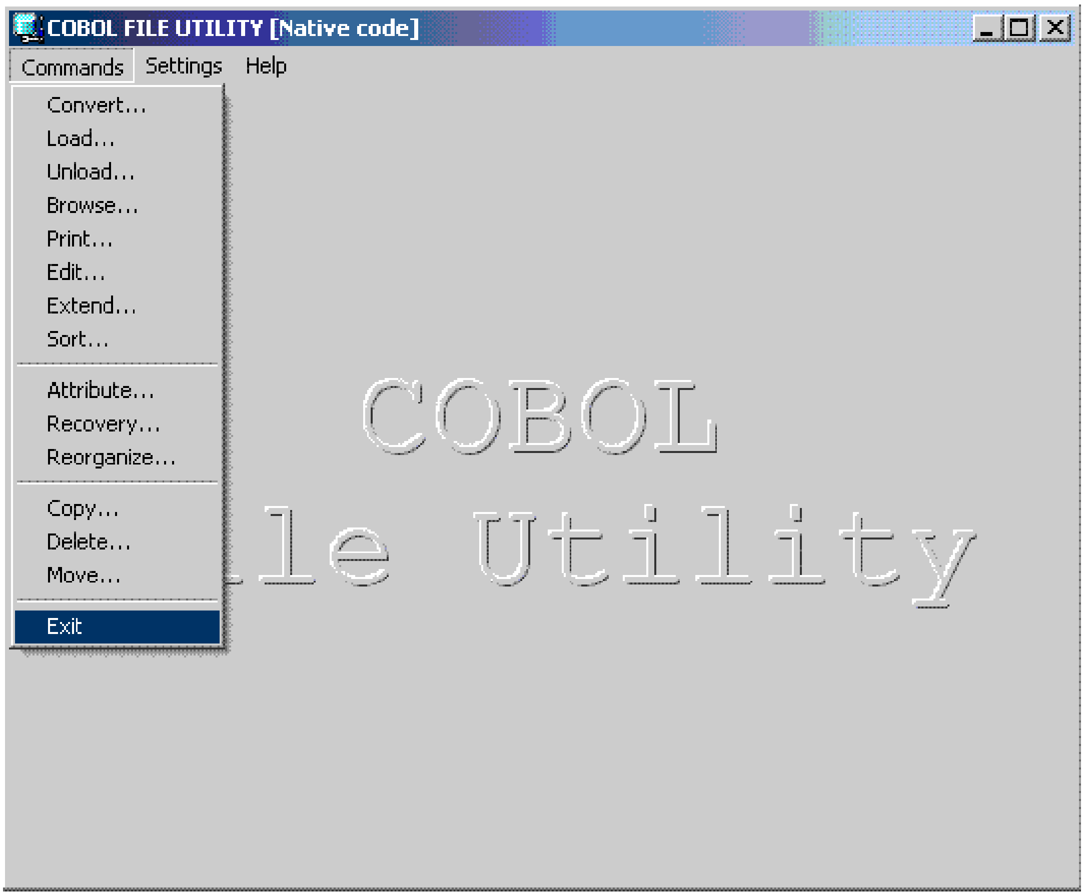
   Start the COBOL File Utility window as follows:

   - Start "COBOL File Utility" from START menu

3. Selecting Commands

   Select the command to be executed from the Command pull down menu on the menu bar. Selecting the command displays a dialog box used to execute each command. For details of each function, see "6.8.2 COBOL File Utility Functions".

   Figure 6.7 The COBOL File Utility window



   The COBOL File Utility contains the following commands:

   Convert

   Create a new COBOL record sequential file in variable length format with data generated by text editors as input. You can also convert the COBOL file back to a text file.

   Load

   Convert COBOL file organization to another kind of file organization using a record sequential file in variable length record format as input.

   In addition, all variable length records of a sequential file can be added to an arbitrary COBOL file.

Unload

Convert COBOL file organization to variable length records in a sequential file using an arbitrary COBOL file as input.

Browse

Browse the content of records in COBOL files.

Print

Print the records in COBOL files.

Edit

Edit the contents of records from COBOL files.

Extend

Add variable length records in a sequential file to existing arbitrary COBOL files.

Sort

Sort records of an arbitrary COBOL file and place the results in a variable length record sequential file.

Attribute

Display attribute information (record length etc.) of an index file.

Recovery

Recover corrupt index files.

Reorganize

Reduce file size by deleting empty blocks in index files.

Copy

Copy a file.

Delete

Delete a file.

Move

Move a file to another location.

4. Quitting the COBOL File Utility

   Select Exit from the Command pull down menu on the menu bar.

Note

- As with file processing using COBOL programs, file processing using the COBOL File Utility depends on file organization. For example, since record and line sequential files are processed in sequential access, records are processed only in a fixed order or cannot be inserted or deleted. See "6.1.1 File Organization Types and Characteristics".

- If the window of this utility is used and the specified file already exists in the output file, the utility inquires whether to overwrite the file. However, if the command or function is used, an error occurs without inquiring whether to overwrite the file.

- This utility is not effective in high-speed file processing. When the file name is specified with ",BSAM", it is assumed to be an error.

- In this utility, when the file is accessed from a COBOL program or another utility process, an error occurs. Therefore, this utility cannot be executed multiple times simultaneously for the same file - especially when trying to recover corrupt index files. Execute the next command after confirming that execution of the previous command was completed.

6.8.2 COBOL File Utility Functions

The COBOL File Utility provides the functions below.

- Creating Files

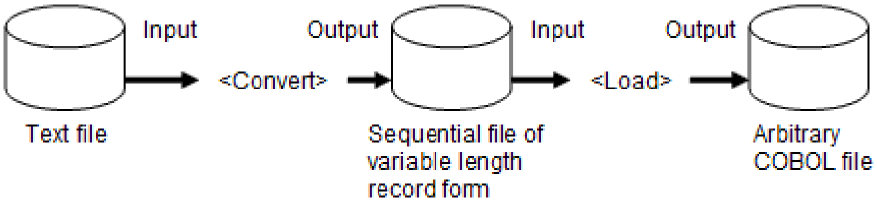
- Extending Files
- Browsing Records
- Editing Records
- Sorting Records
- Manipulating Files (Copy, Delete, Move)
- Printing Files
- Converting File Types
- Manipulating Indexed Files (Display of attribute files, Recovery, Reorganize)

In quotation marks (") are commands (< > indicates a command name and [ ] indicates its use is optional) to use the functions. For more information about how to operate the dialog box, refer to the online help.

6.8.2.1 Creating Files

Create a new COBOL file with data generated by text editors as input using the "Convert [+ Load]" commands.

Operation procedure



Note

If the <Load (extend) > command causes an error with the backup check box off, the content of the existing file is lost.

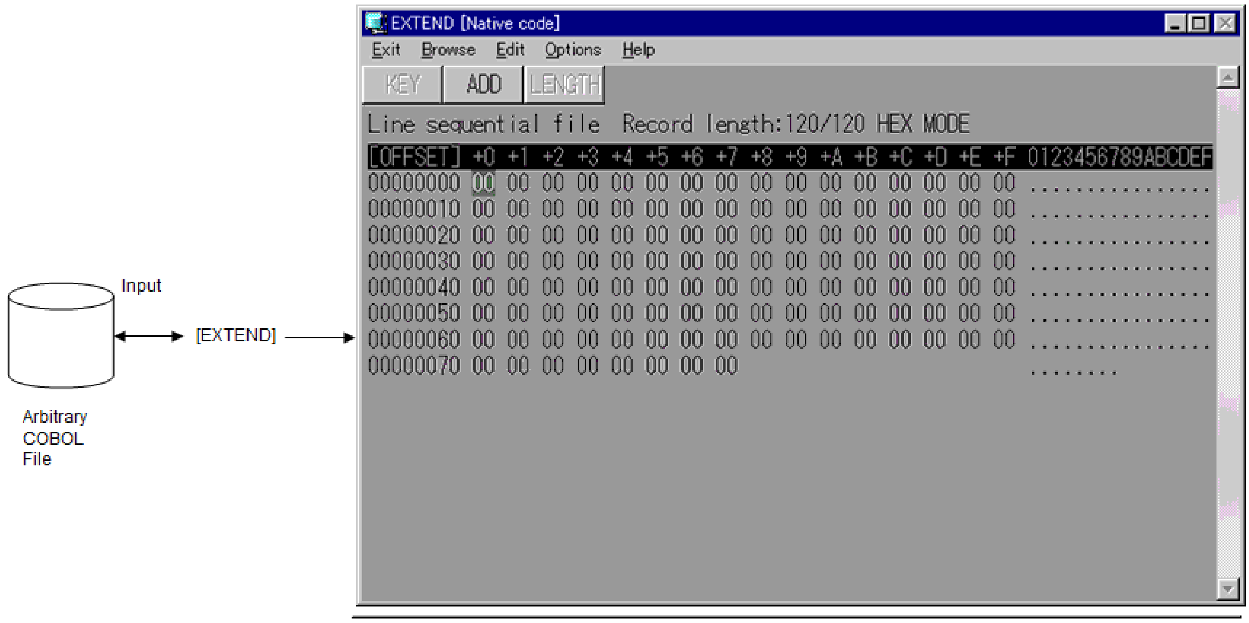
6.8.2.2 Extending Files

Records can be added to an existing file. Records of a file can also be added to another file.

Adding Records

Add records to existing COBOL files with the "Extend" command.

Figure 6.8 Adding records with the Extend command



```
Line sequential file  Record length:120/120 HEX MODE
[OFFSET] +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F 0123456789ABCDEF
00000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000070 00 00 00 00 00 00 00 00                         ........
```

The Extend window contains the following elements:

Menu Bar

Exit

Quit (exit) the Extend window.

Browse

Browse the content of records in COBOL files.

Edit

Edit a record.

Options

Determine the method of data input (hexadecimal or character) when adding new records.

Help

Access the online help.

Button Bar

KEY

Select to change the read sequence. If reading sequence is changed, the sequence of the records to be displayed on the window becomes dependent on the sequence of selected record keys.

ADD

Select to add the record being displayed on the window to the file.

LENGTH

Select to change the length of the record being displayed on the window.

Record Processing Window

File type

Displays the type of file being extended e.g. "Line sequential file", "Indexed file".

Record length

Displays the length of the displayed record/maximum record length.

EDIT MODE

Displays whether the data entry mode is HEX (enter characters values using 2 hex digits) or CHAR (enter values using the keyboard characters).

"OFFSET" column and row headers

You can calculate the offset of any byte within the record by adding the hexadecimal number (+0 to +F) at the head of the column containing the byte to the hexadecimal number (00000000, 00000010, ...) at the beginning of the row.

Two character strings under "+0 +1 ... +E +F"

The data record is displayed by a hexadecimal every one byte.

Character string under "0123456789ABCDEF"

One-byte characters display the record data.

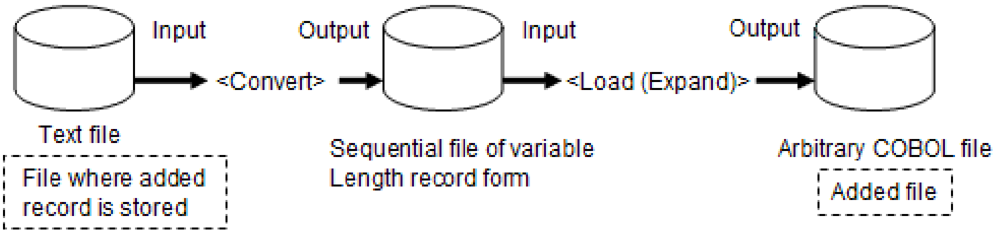
Note

Characters that cannot be displayed, such as national characters, are displayed with periods (.). This is the same for browsing, editing, and printing records. Refer to the online help for additional details.

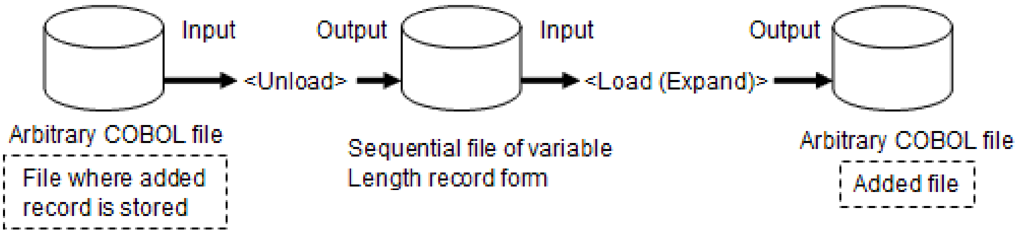
Adding records from another file

Use the "[Convert +] Load (extend) or [Unload +] Load (extend)" commands to add the contents of other files to existing COBOL files.

Figure 6.9 Two methods of adding records from another file



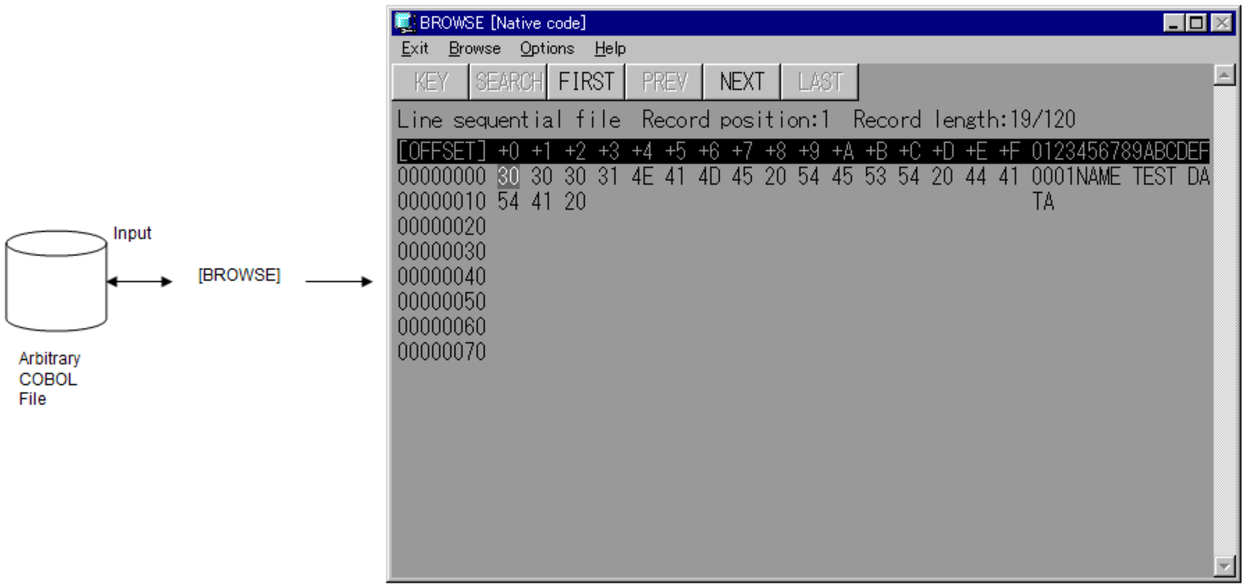
or



6.8.2.3 Browsing Records

Browse the contents of records in COBOL files with the "Browse" command.

Figure 6.10 Browsing files with the Browse command



```
Line sequential file  Record position:1  Record length:19/120
[OFFSET] +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F 0123456789ABCDEF
00000000 30 30 30 31 4E 41 4D 45 20 54 45 53 54 20 44 41 0001NAME TEST DA
00000010 54 41 20                                        TA
00000020
00000030
00000040
00000050
00000060
00000070
```

The content of the file is displayed in characters and hexadecimal digits in units of records.

The Browse window contains the following elements:

Menu Bar

Exit

Quit (exit) the Browse window.

Browse

Browse the contents of records in COBOL files.

Options

Determine the method of data input (hexadecimal or character) when adding new records.

Help

Access the online help.

Button Bar

KEY

Select to change the read sequence. If read sequence is changed, the sequence of the records to be displayed on the window becomes dependent on the sequence of selected record keys.

SEARCH

Locate a record by relative record number or indexed key value.

FIRST

Display the (logical) first record in the file.

PREV

Display the (logical) previous record in a file.

NEXT

Display the (logical) next record in a file.

LAST

Display the (logical) last record in the file.

Record Processing Window

File type

Displays the type of file being extended e.g. "Line sequential file", "Indexed file".

Record length

Displays the length of the displayed record/maximum record length.

EDIT MODE

Displays whether the data entry mode is HEX (enter characters values using 2 hex digits) or CHAR (enter values using the keyboard characters).

"OFFSET" column and row headers

You can calculate the offset of any byte within the record by adding the hexadecimal number (+0 to +F) at the head of the column containing the byte to the hexadecimal number (00000000, 00000010, ...) at the beginning of the row.

Two character strings under "+0 +1 ... +E +F"

The data record is displayed by a hexadecimal every one byte.

Character string under "0123456789ABCDEF"

One-byte characters display the record data.

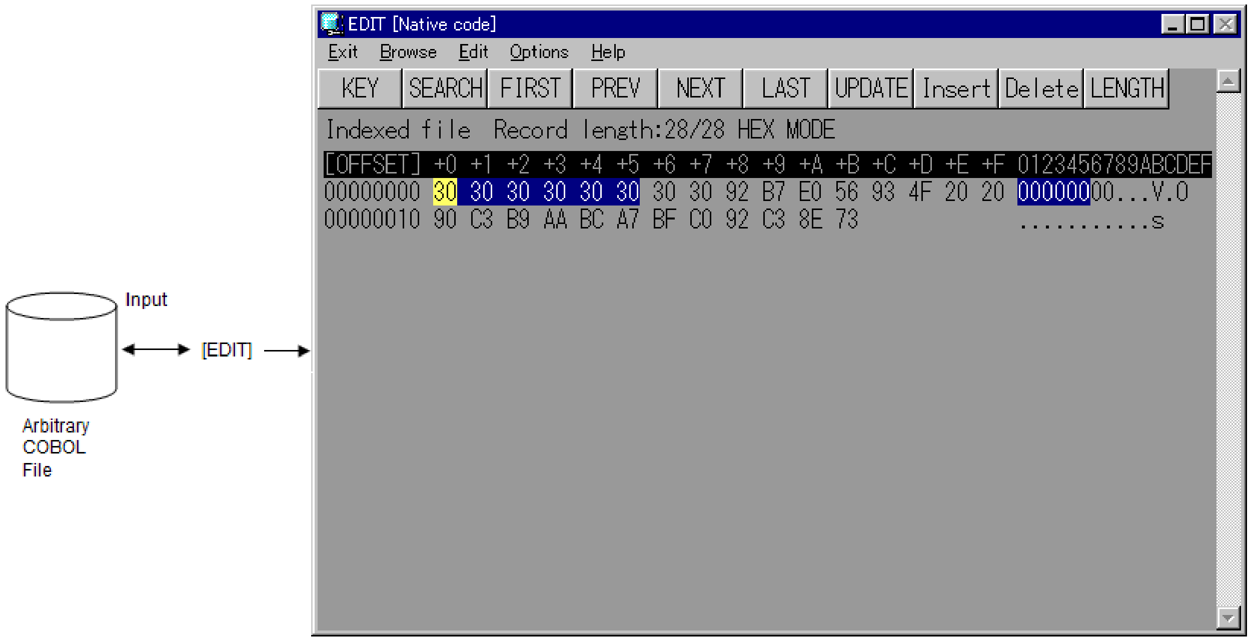
Note

Characters that cannot be displayed, such as national characters, are displayed with periods (.). Refer to the online help for additional details.

6.8.2.4 Editing Records

Update, insert, and delete records within COBOL files using the "Edit" command.

Figure 6.11 Updating, inserting or deleting records with the Edit command



```
Indexed file  Record length:28/28 HEX MODE
[OFFSET] +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F 0123456789ABCDEF
00000000 30 30 30 30 30 30 30 30 92 B7 E0 56 93 4F 20 20 00000000...V.O
00000010 90 C3 B9 AA BC A7 BF C0 92 C3 8E 73             ...........s
```

Edit the content of the file in units of records.

Edit (insert, delete, or update) the records displayed in characters and hexadecimal digits.

The Edit window contains the following elements:

Menu Bar

Exit

Quit (exit) the Edit window.

Browse

Browse the content of records in COBOL files.

Edit

Edit a record.

Options

Determine the method of data input (hexadecimal or character) when adding new records.

Help

Access the online help.

Button Bar

KEY

Select to change the read sequence. If read sequence is changed, the sequence of the records to be displayed on the window becomes dependent on the sequence of selected record keys.

SEARCH

Locate a record by relative record number or indexed key value.

FIRST

Display the (logical) first record in the file.

PREV

Display the (logical) previous record in a file.

NEXT

Display the (logical) next record in a file.

LAST

Display the (logical) last record in the file.

UPDATE

Update the displayed record in the file.

Insert

Insert a new record in the file.

Delete

Delete the displayed record from the file.

LENGTH

Change the length of the displayed record.

Record Processing Window

File type

Displays the type of file being extended e.g. "Line sequential file", "Indexed file".

Record length

Displays the length of the displayed record/maximum record length.

EDIT MODE

Displays whether the data entry mode is HEX (enter characters values using 2 hex digits) or CHAR (enter values using the keyboard characters).

"OFFSET" column and row headers

You can calculate the offset of any byte within the record by adding the hexadecimal number (+0 to +F) at the head of the column containing the byte to the hexadecimal number (00000000, 00000010, ...) at the beginning of the row.

Two character strings under "+0 +1 ... +E +F"

The data record is displayed by a hexadecimal every one byte.

Character string under "0123456789ABCDEF"

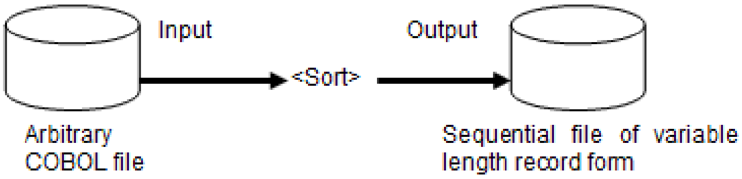
One-byte characters display the record data.

Note

If the maximum record length of an indexed file is 16 kilobytes or more, the file cannot be opened with the Edit command.

6.8.2.5 Sorting Records

Sort records in COBOL files with the "Sort" command.

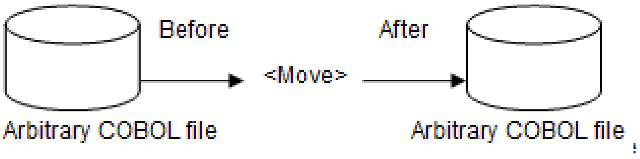


6.8.2.6 Manipulating Files

Copy, delete, and move COBOL files with the "<Move>, <Copy>, and <Delete>" commands.

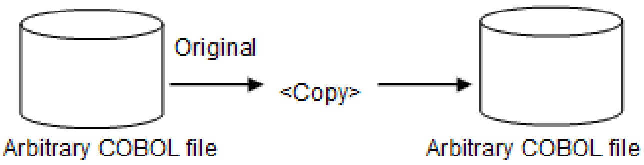
Moving Files

Moves COBOL files with the "Move" command.



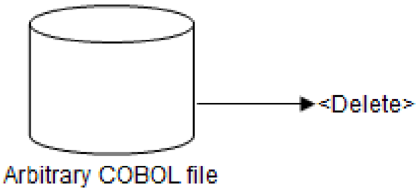
Copying Files

Copies COBOL files with the "Copy" command.



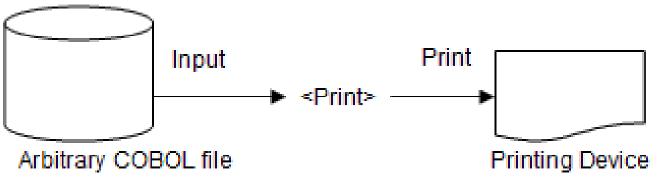
Deleting Files

Deletes COBOL files with the "Delete" command.



6.8.2.7 Printing Files

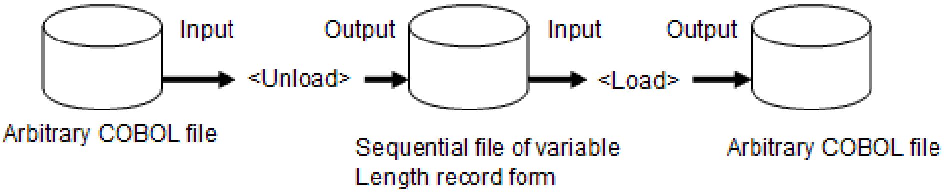
Print the contents of COBOL files with the "Print" command.



6.8.2.8 Converting File Types

Convert COBOL file to another kind of file organization using the "Load" or "Unload" commands. Use the "Load" command to input a record sequential file in variable length record format and convert it to a COBOL file. Use the "Unload" command to input a COBOL file and convert it to a record sequential file in variable length record format.

Figure 6.12 Converting files using the Load and Unload commands

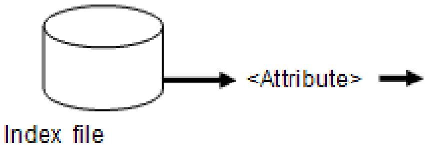


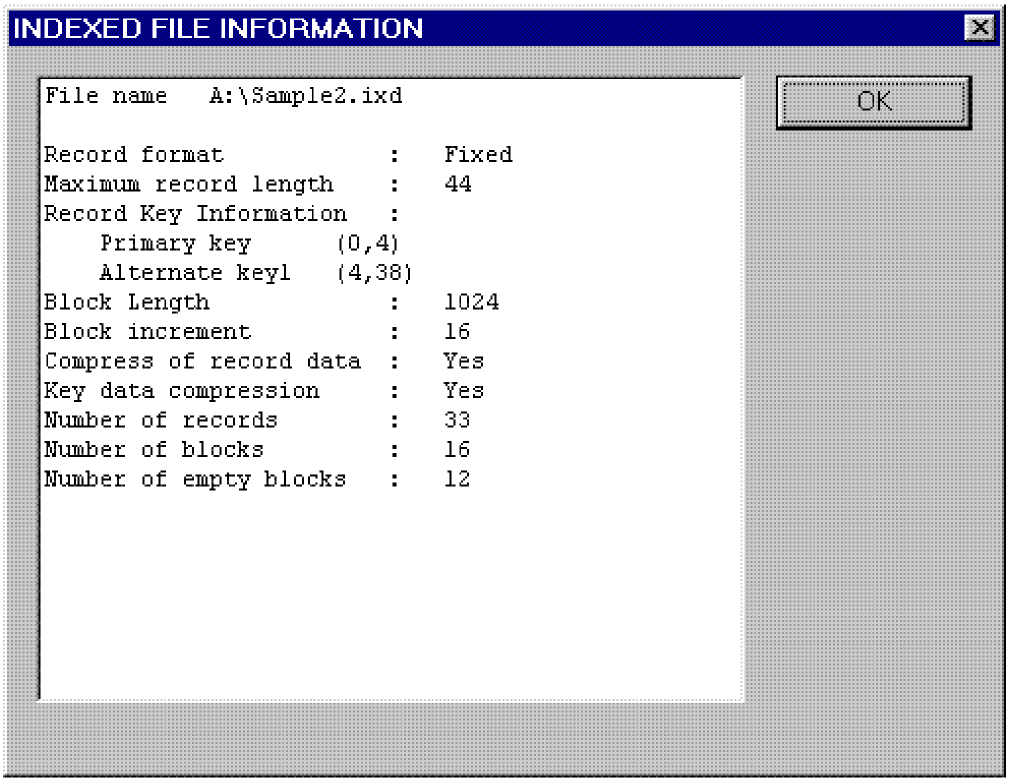
6.8.2.9 Manipulating Indexed Files

You can execute the following operations for indexed files:

Display of attribute information

Display attribute information with the "Attribute" command





```
File name   A:\Sample2.ixd

Record format             :   Fixed
Maximum record length     :   44
Record Key Information    :
    Primary key      (0,4)
    Alternate key1   (4,38)
Block Length              :   1024
Block increment           :   16
Compress of record data   :   Yes
Key data compression      :   Yes
Number of records         :   33
Number of blocks          :   16
Number of empty blocks    :   12
```

The Indexed File Information window contains the following information:

- File name

  Displays the index file name whose attributes follow.

- Record format

  Displays the record format (fixed length/variable length).

- Maximum record length

  Displays the maximum record length.

- Minimum record length

  If the minimum record length is fixed, the minimum record length is not displayed.

  If the minimum record length is variable, the minimum record length is displayed.

- Record key information

  Displays key information for the index file. The format of the key information is as follows:

```
[D](offset,length[,{N|N32})[/offset,length[,{N|N32}]]...)
```

  - D: Indicates that the file may contain records having duplicate alternate record key values. It is equivalent to WITH DUPLICATES.
  - Offset: The relative position (number of bytes, starting from 0 bytes) from the head of the record to the key.
  - Length: The length of the key, in number of bytes.
  - N: Indicates that the key is UCS-2 little endian.
  - N32: Indicates that the key is UTF-32 little endian.
  - /: Delimiter that indicates that one key is composed of two or more non-continuous data items.

- Block length

  Displays the length of one block in the index file.

- Block increment

  Displays block increments in the index file.

- Compress of record data

  Displays whether or not the stored record data is compressed.

- Key data compression

  Displays whether or not the stored key data is compressed.

- Number of records

  Displays the number of records in the index file.

- Number of blocks

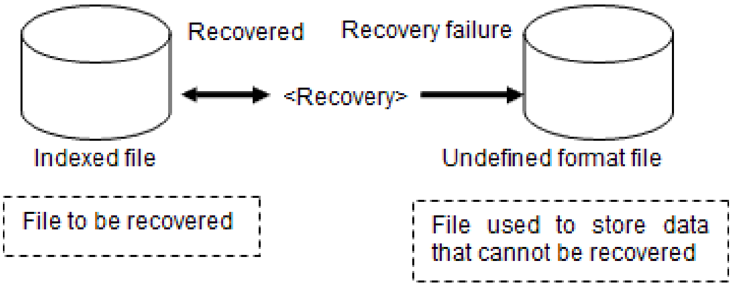
  Displays the number of blocks in the index file.

- Number of empty blocks

  Displays the number of unused blocks in the index file.

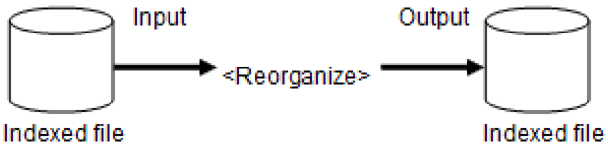
Recovery

Recover corrupt indexed files with the "Recovery" command.



Reorganize

Remove unused space from indexed files to reduce the file size with the "Reorganize" command.



6.8.3 Using COBOL file utility command mode

This section explains how to use the COBOL file utilities in command mode.

Use the commands directly on the MS-DOS command (DOS command) screen.

Note

A peculiar character to Unicode can be specified for input/output file name of each command. However, the message of each command is output by the character within the range of native code. A peculiar character to Unicode is replaced by "?" and output.

6.8.3.1 File conversion command(cobfconv)

The cobfconv command converts a text file into a variable length record sequential file or vice versa.

In the text file, binary data is represented as a character string in hexadecimal.

Specification format

```
cobfconv [-k character-code] -o output-file-name -c mode, format input-file-name
```

Interface

Specify the command parameters as shown below:

character-code

Specify one of the following character codes following the -k option:

```
-k{native | unicode[,{LE | BE}] utf32[,{LE | BE}]}
```

- native: native
- unicode: UCS-2
- utf32: UTF-32
- LE: Little-endian format (default)
- BE: Big-endian format

If the -k option is omitted or the -knative option is specified, the character code of text file and record sequential file is handled as the native code.

If the -kunicode option is specified, the character code of the text file is handled as the UCS-2 little-endian code. The contents of the record sequential file are handled as the UCS-2 code, UTF-8 code, or binary value in hexadecimal conforming to the definition of the record data item. The UCS-2 code of the record sequential file is specified following a comma(,). When it is omitted, LE is used.

If the -kutf32 option is specified, the character code of the text file is handled as UTF-32 little-endian code. The contents of the record sequential file are handled as UCS-2 code, UTF-32 code, UTF-8 code, or binary value in hexadecimal conforming to the definition of the record data item. The UTF-32 code of the record sequential file is specified after a comma(,). When it is omitted, LE is used.

Note

In the record sequential file, endian (UCS-2 or UTF-32) cannot be specified.

output-file-name

Specify the path name of the text file or the record sequential file that is created. If an existing file is specified, an error occurs.

Conversion mode, data format

Specify the conversion mode and record composition of record sequential file (data format) as shown below:

When -k option is omitted or -knative is specified:

$$\text{-c} \left\{\begin{array}{l} \text{b} \\ \text{t} \end{array}\right\}, \left\{\begin{array}{l} \text{allchar} \\ \text{alltxtbin} \\ \text{"} \left\{\begin{array}{l} \text{c length} \\ \text{t length} \\ \text{k length} \end{array}\right\} \text{[; ...]"} \end{array}\right\}$$

When the -kunicode is specified:

```
                    ┌ allucs2                     ┐
                    │ allutf8                     │
      ┌ b ┐         │ allutf32                    │
-c   {     }  ,    {                               }
      └ t ┘         │      ┌ u length ┐           │
                    │   " {  f length  }  [; ...]" │
                    └      └ t length ┘           ┘
                             w length
```

Conversion mode

Specify the conversion mode using one of the following characters:

- b: Convert a text file into a record sequential file.
- t: Convert a record sequential file into a text file.

Data format

Specify the format of records of the record sequential file in the data format. The data format definitions are listed below.

- For the native code, "Data item to be displayed" (*1) is called a character format and other data items are called a hexadecimal format.

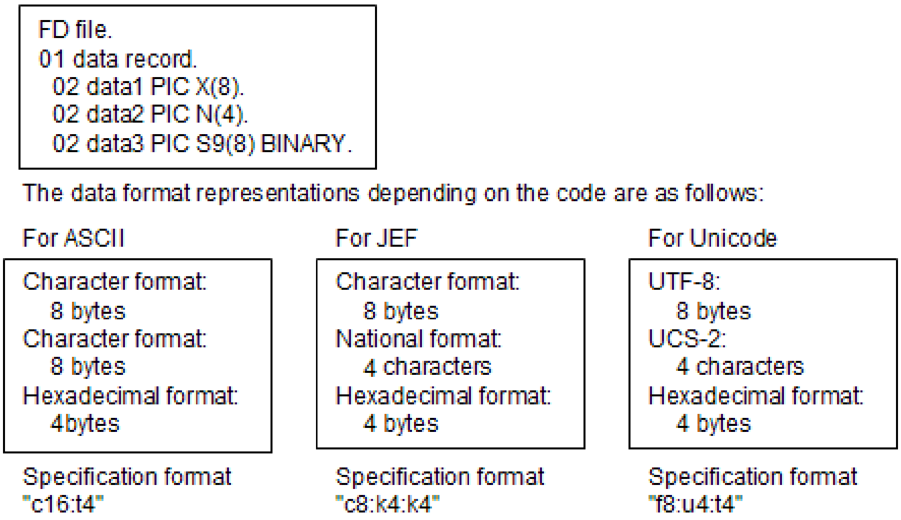
  *1 For details on the "Data item to be displayed," see the COBOL Reference Manual.

- For Unicode, national data items and national edited data items are regarded as UCS-2 format or UTF-32 format, and "Data item to be displayed" without the national data item or national edited data item is regarded as UTF-8 format. Other data items are regarded as hexadecimal format.

Specify the conversion data format using one of the following character strings:

- allchar: All data in the record is regarded as character format.
- alltxtbin: All data in the record is regarded as hexadecimal format.
- "{c length | t length | k length} [;...]": If different types of data formats are included together in a record, specify the length of the data format next to the keyword character (number of characters for national format). Meanings of individual characters are listed below.
  - c: Character format
  - t: Hexadecimal format
- allucs2: All data in the record is regarded as UCS-2 format.
- allutf8: All data in the record is regarded as UTF-8 format.
- allutf32: All data in the record is regarded as UTF-32 format.
- "{u length | f length | t length | w length} [;...]": If different types of data formats are included together in the record, specify the length of the data format next to the keyword character (number of characters for UCS-2 format or UTF-32 format). Meanings of individual keyword characters are listed below.
  - u: UCS-2 format
  - f: UTF-8 format
  - t: Hexadecimal format
  - w: UTF-32 format

Specification example when different data types are included together:



```
FD file.
01 data record.
  02 data1 PIC X(8).
  02 data2 PIC N(4).
  02 data3 PIC S9(8) BINARY.
```

The data format representations depending on the code are as follows:

For ASCII

Character format:
  8 bytes
Character format:
  8 bytes
Hexadecimal format:
  4bytes

Specification format
"c16:t4"

For JEF

Character format:
  8 bytes
National format:
  4 characters
Hexadecimal format:
  4 bytes

Specification format
"c8:k4:k4"

For Unicode

UTF-8:
  8 bytes
UCS-2:
  4 characters
Hexadecimal format:
  4 bytes

Specification format
"f8:u4:t4"

Note

Up to 256 data formats can be specified in a combination of two or more data formats.

input-file-name

Specify the path name of the text file or record sequential file in the conversion origin.

Example

1) Create the text file from the record sequential file.

- Character code : ASCII
- Input file name : infile
- Output file name : outfile.txt
- Conversion mode : Record sequential file to text file
- Data format : All character format

```
cobfconv -ooutfile.txt -ct,allchar infile
```

2) Create the record sequential file from the text file (Unicode code)

- Character code : Unicode)
- Input file name : infile.txt
- Output file name : outfile
- Conversion mode : Text file to record sequential file
- Data format : All UCS-2 format

```
cobfconv -kunicode -ooutfile -cb,"allucs2" infile.txt
```

6.8.3.2 File load command(cobfload)

The cobfload command creates a variable or fixed length record sequential, relative, or indexed file based on a variable length record sequential file. The command also adds record data in a variable length record sequential file to an existing record sequential, relative, or indexed file (this operation is referred to as a file extension). A backup file is created during file extension so that the output file can be restored to the state before command execution if an error occurs.

Specification format

```
cobfload -o output-file-name [-e] -d file-attribute input-file-name
```

Interface

Specify the command parameters as shown below.

character-code

To create an index file in the Unicode format, specify the character code in the following format:

```
-kunicode[,{LE | BE}]
```

- LE: Little-endian format (default value)
- BE: Big-endian format

output-file-name

Specify the path to the file to be created or extended.

If the existing file is specified when creating the file, an error occurs. If the specified file does not exist when extended, an error occurs.

-e

Specify this parameter for file extension.

file-attribute

Specify the attribute of the file to be created or extended, using the following format:

```
        { S }   { f }
-d      { R } , {   } ,record-length,"(offset,length[,N] [,N32] [/ offset,length[,N] [,N32]]...)
        {   }   {   } [,D][;...]"
          I       v
```

File organization

Specify the file organization using one of the following characters:

- S: Record sequential file
- R: Relative file
- I: Indexed file

When extending an index file, the record format, record length, and record key information cannot be specified.

Record format

Specify the record format using one of the following characters:

- f: Fixed length
- v: Variable length

Record length

Specify the record length. When using variable length records, specify the maximum record length.

Record key information

To create an index file, specify record key information as follows.

- offset: Specify the location-in-record of the data item to be used as the record key by the relative number of bytes assuming the top of the record is 0.
- length: Specify the number of bytes for the length of the data item to be used as the record key.
- N: Specify this operand if the data item to be used as the record key is in the UCS-2 little-endian format. When -k unicode,BE is specified, N has no effect.
- N32: Specify this operand if the data item to be used as the record key is in the UTF-32 little-endian format. When -k unicode,BE is specified, N has no effect.
- /: When using two or more discontinuous data items as one record key, specify individual data items by separating their offset and lengths with "/".
- D: Specify this operand to permit duplication of the record key.
- ;: When defining sub record keys, specify the sub record key data items by separating such items with ";".

input-file-name

Specify the path name of the record sequential file in the conversion origin.

Example

1) Create an indexed file from a record sequential file.

- Input file name: infile
- Output file name: ixdfile
- File attribute:
  - File organization: Indexed file
  - Record format: Variable length
  - Record length: 80
  - Record key information:
    - Data item 1 (Location in record: 0/ Length: 5)
    - Data item 2 (Location in record: 10/ Length: 5)
    - Duplicate specification: Yes
    - Alternate record key (Location in record: 5/ Length: 5)

```
cobfload -oixdfile -dI,v,80,"(0,5/10,5),D;(5,5)" infile
```

2) Add the content of a record sequential file to an relative file (Extend).

- Input file name: infile
- Output file name: relfile
- File attribute:
  - File organization: Relative file
  - Record format: Fixed length
  - Record length: 80

```
cobfload -orelfile -e -dR,f,80 infile
```

Note

If the input file contains a record that is longer than the maximum record length of the output file, an error occurs when the command is executed.

6.8.3.3 File unload command(cobfulod)

The cobfulod command creates a variable length sequential file based on a record sequential, relative, or indexed file.

Specification format

```
cobfulod -o output-file-name -i file-attribute input-file-name
```

Interface

Specify the command parameters as shown below.

output-file-name

Specify the path name of the record sequential file to be created. If the existing file is specified, an error occurs.

file-attribute

Specify the file attribute of the conversion origin, using the following format:

$$\text{-i} \quad \begin{Bmatrix} \text{S} \\ \text{R} \\ \text{I} \end{Bmatrix} , \begin{Bmatrix} \text{f} \\ \\ \text{v} \end{Bmatrix} \text{,record-length}$$

file organization

Specify the file organization using one of the following characters:

- S: Record sequential file
- R: Relative file
- I: Indexed file

record format

Specify the record format using one of the following characters: When an indexed file is specified for the file organization, the record format cannot be specified.

- f: Fixed length
- v: Variable length

record length

Specify the record length. When using variable length records, specify the maximum record length. When an indexed file is specified for the file organization, the record length cannot be specified.

input-file-name

Specify the path name of the record sequential file or relative file or indexed file in the conversion origin.

Example

Create a record sequential file from a relative file.

- Input file name: relfile
- Output file name: outfile

- File attribute:
  - File organization: Relative file
  - Record format: Fixed length
  - Record length: 80

```
cobfulod -ooutfile -iR,f,80 relfile
```

6.8.3.4 File display command(cobfbrws)

The cobfbrws command displays the content of a file in both character format and hexadecimal format in units of records. Data other than alphanumeric characters (0x20 to 0x7e) is replaced by periods before display. The display range can be specified in units of records. If no display range is specified, all records in the file are displayed.

Specification format

```
cobfbrws [-k character-code] -i file-attribute [-ps start-location]
         [-pe end-location] [-po display-order] [-pk search-key-number]
         input-file-name
```

Interface

Specify the command parameters as shown below:

character-code

- Specify one of the following character codes following the -k option:

  ```
  -k{native | unicode[,{LE | BE}] | utf32[,{LE | BE}]}
  ```

  - native: native
  - unicode: UCS-2
  - utf32: UTF-32
  - LE: Little-endian format(default)
  - BE: Big-endian format
- If the -k option is omitted or the -knative option is specified, the file character code is handled as the native code.
- If the -k unicode option is specified, the file character code is handled as the UCS-2 code. The UCS-2 code of the record sequential file is specified after a comma(,). When it is omitted, LE is used.
- If the -k utf32 option is specified, the file character code is handled as the UTF-32 code. The UTF-32 code of the record sequential file is specified after a comma(,). When it is omitted, LE is used.

file-attribute

Specify the attribute of the file to be displayed, using the following format:

$$\text{-i} \quad \left\{ \begin{matrix} \text{S} \\ \text{L} \\ \text{R} \\ \text{I} \end{matrix} \right\} , \quad \left\{ \begin{matrix} \text{f} \\ \text{v} \end{matrix} \right\} \text{,record-length}$$

file organization

Specify the file organization, using one of the following characters:

- S: Record sequential file

- L: Line sequential file
- R: Relative file
- I: Indexed file

record format

Specify the record format, using one of the following characters. When an indexed file is specified as the file organization, the record format cannot be specified because the attribute information in the file is used.

- f: Fixed length
- v: Variable length

record length

Specify the record length. When using variable length records, specify the maximum record length. When an indexed file is specified as the file organization, the record length cannot be specified.

start-location

Specify the location of the record to start display, using the following format:

```
        store-order
        r relative-record-number
-ps  {  ic record-key-value        }
        it record-key-value
        in record-key-value
```

The options available depend on the file organization:

- For a record sequential or line sequential file, specify the record store order.
- For a relative file, specify the relative record number following the keyword character "r".
- For an indexed file, the specification method varies depending on whether the record key value is specified in character format or hexadecimal format.
  - When using character format, specify the record key value following the keyword character string "ic".
  - When using hexadecimal format, specify the record key value following the keyword character string "it".
  - When record keys are all national items using the character format, specify the record key value following the keyword character string "in".

If the start location is omitted, the utility displays data starting with the first record of the file.

end-location

Specify the location of the last record to display, using the following format:

```
        store-order
        r relative-record-number
-pe  {  ic record-key-value        }
        it record-key-value
        t output-count
```

Parameters other than the output count are the same as those of the start location.

To specify the end location using the output count, specify the number of output records following the keyword character "t".

If the end location is omitted, the utility displays all data from start location to the last record of the file.

display-order

When there are multiple records between the start and end locations, specify the display order of the records as follows:

```
        ┌ A ┐
-po    {     }
        └ D ┘
```

- A: Records are displayed in ascending order. When using record sequential or line sequential files, records are displayed in ascending order of the record store order. When using relative files, records are displayed in ascending order of the relative record number. Indexed file records are displayed in ascending order of the record key value.
- D: Records are displayed in descending order. In the case of a relative file, records are displayed in descending order of the relative record number. In the case of an indexed file, records are displayed in descending order of the record key value. The descending order cannot be specified for a record sequential or line sequential file.

If this parameter is omitted, the ascending order is used.

search-key-number

When displaying an indexed file, specify the number of the record key used to search for a record. Record key number 0 is assigned to the main record key, 1 is assigned to the first subrecord key, and 2 and subsequent serial numbers are assigned to the second and subsequent subrecord keys.

If the search key number is omitted, the main record key is used.

input-file-name

Specify the folder name of the file to be displayed.

Example

Display the content of record sequential files.

- Input file name: seqfile
- File attribute:
    - File organization: Record sequential file
    - Record format: Variable length
    - Record length: 80
- Start location (store order): 5
- End location (output-count): 10

```
cobfbrws  -iS,v,80  -ps5  -pet10  seqfile
```

Note

- When the character code is Unicode, data other than 1-byte alphanumeric characters that can be represented in the UTF-8 code is replaced by periods before display.
- -psic and -peic are specifiable to record key where the alphanumeric character exists together with the national character. However, when the character code is Unicode big endian form or UTF-32 encoding form, it is excluded.
- -psin and -pein cannot be specified when the character code is UTF-32 encoding form.

6.8.3.5 File sort command(cobfsort)

The cobfsort command sorts the records in a file in ascending or descending order using a specified data item in the records as a key. The sorted records are output to a variable length record sequential file.

Specification format

```
cobfsort [-k character-code] -o output-file-name -s sort-condition -i file-attribute input-file-name
```

Specify the command parameters as shown below:

Interface

Specify the command parameters as shown below:

character-code

- Specify one of the following character codes following the -k option:

```
-k{native | unicode[,{LE | BE}] | utf32[,{LE | BE}]}
```

  - native: native
  - unicode: UCS-2
  - utf32: UTF-32
  - LE: Little-endian format (default)
  - BE: Big-endian format

- If the -k option is omitted or the -ksjis option is specified, the character code of the file to be sorted is handled as the native code.
- If the -kunicode option is specified, the character code to be sorted is handled as the UCS-2 code. The UCS-2 code of the record sequential file is specified after a comma(,). When it is omitted, LE is used.
- If the -kutf32 option is specified, the character code to be sorted is handled as UTF-32 code. The UTF-32 code of the record sequential file is specified after a comma(,). When it is omitted, LE is used.

output-file-name

Specify the path name of the record sequential file to which the sorted records are to be output. If the existing file is specified, an error occurs.

sort-condition

Specify the attribute of the data item to be used as a key and sort order, using the following format:

```
          { 1[B]    }
          { X       }
          { N       }
          { [S]9    }
          { S9L     }
          { S9T     }                 { A }
-S"(      { S9LS    } ,offset,size[,  {   } ][;...]"
          { S9TS    }                 { D }
          { [S]9PD  }
          { [S]9B   }
          { [S]9C5  }
          { C1      }
          { C2      }
```

key item attribute

Specify one of the following values for the key item attribute:

- 1[B] : PIC 1() [BIT]
- X : PIC X()
- N : PIC N()
- [S]9 : PIC [S] 9()
- S9L : PIC S9() LEADING
- S9T : PIC S9() TRAILING
- S9LS : PIC S9() LEADING SEPARATE
- S9TS : PIC S9() TRAILING SEPARATE
- [S]9PD : PIC [S]9() PACKED-DECIMAL
- [S]9B : PIC [S]9() BINARY
- [S]9C5 : PIC [S]9() COMP-5
- C1 : COMP-1
- C2 : COMP-2

Note

When "N" is specified for the key item attribute, the character code of the data item follows the -k option. The key item that has a different character code cannot be specified.

offset

Specify the offset in a record for the key item using relative byte location, where the first byte is byte 0.

size

Specify the size of the key item using the specified number of digits of a numeric data item in a PICTURE clause or the number of characters of an alphanumeric or national data item.

Note

- When "1B" is specified for the key item, the size of the key item is fixed to 1 byte and cannot be specified. Instead of the size, specify a 1-byte mask value in decimal notation.
- When "C1" or "C2" is specified for the key item, the size can be omitted. Only 4 or 8 bytes can be specified.

Sort order

Specify whether records should be sorted in ascending or descending order according to the attribute of the key item.

- A: Ascending order
- D: Descending order

Default records are sorted in ascending order.

Note

Up to 64 sort conditions can be specified.

file-attribute

Specify the attribute of the file to be sorted, using the following format:

```
      ┌ S ┐
      │ L │      ┌ f ┐
-i   {     } ,  {     } ,record-length
      │ R │      └ v ┘
      └ I ┘
```

file organization

Specify the file organization using one of the following characters:

- S: Record sequential file
- L: Line sequential file
- R: Relative file
- I: Indexed file

record format

Specify the record format using one of the characters given below. When an indexed file is specified for the file organization, the record format cannot be specified.

- f: Fixed length
- v: Variable length

record length

Specify the record length. When using variable length records, specify the maximum record length. When an indexed file is specified as the file organization, the record length cannot be specified.

input-file-name

Specify the folder name of the file to be sorted.

Example

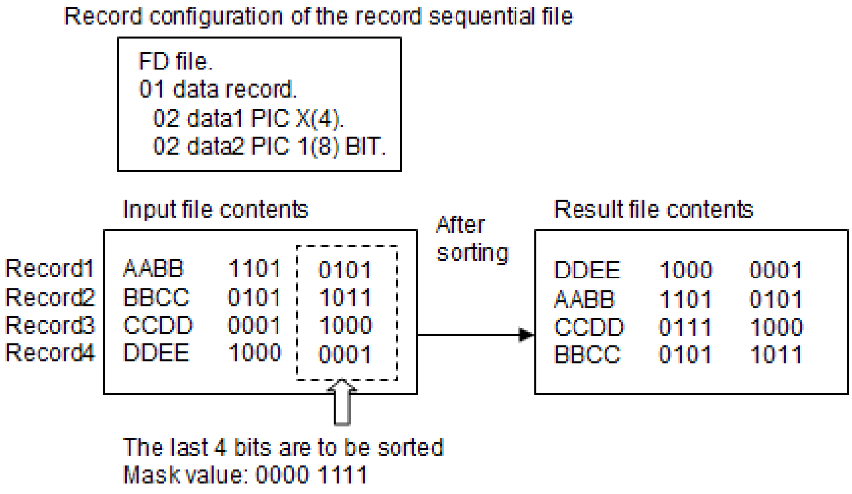
1) Sort a relative file and output the result to a record sequential file.

- Input file name: relfile
- Output file name: outfile
- Sort condition:
  - First key
    - Item attribute: National (N)
    - Offset in record: 0
    - Size: 2
    - Sort order: Ascending order
  - Second key
    - Item attribute: Alphanumeric
    - Offset in record: 10
    - Size: 5
    - Sort order: Descending order
- File attribute
  - File organization: Relative file
  - Record format: Variable length

- Record length: 80

```
cobfsort  -ooutfile  -s"(N,0,2),A;(X,10,5),D"  -iR,v,80  relfile
```

2) Sort a record sequential file and output the result to a record sequential file



Record configuration of the record sequential file

```
FD file.
01 data record.
  02 data1 PIC X(4).
  02 data2 PIC 1(8) BIT.
```

Input file contents

Record1	AABB	1101	0101
Record2	BBCC	0101	1011
Record3	CCDD	0001	1000
Record4	DDEE	1000	0001

After sorting

Result file contents

DDEE	1000	0001
AABB	1101	0101
CCDD	0111	1000
BBCC	0101	1011

The last 4 bits are to be sorted
Mask value: 0000 1111

- Input file name: inlfile
- Output file name: outfile
- Sort condition:
  - First key
    - Item attribute: Boolean (1B)
    - Offset in record: 4
    - Mask value: 15
    - Sort order: Ascending order
- File attribute
  - File organization: Record sequential file
  - Record format: Fixed length
  - Record length: 5

```
cobfsort  -ooutfile  -s"(1B,4,15),A"  -iS,f,5  infile
```

6.8.3.6 File attribute command(cobfattr)

The cobfattr command displays the attribute (record length, record format, key information etc.) of an indexed file.

This command can display attribute information of only an indexed file.

Specification format

```
cobfattr input-file-name
```

Interface

Specify the command parameter as shown below:

input-file-name

Specify the folder name of an indexed file for which attribute information is to be displayed.

Example

Display the attribute of the indexed file

- Input file name : ixdfile

```
cobfattr ixdfile
```

6.8.3.7 File recovery command(cobfrcov)

An indexed file may not be correctly closed at the abnormal end of a process. The cobfrcov command recovers such an indexed file so that it can be accessed normally again. However, if an abnormality is found in data and some records are unrecoverable, the records are output to a variable length sequential file as an unrecoverable data file.

Specification format

```
cobfrcov recovery-file-name unrecoverable-data-file-name
```

Interface

Specify the command parameters as shown below:

recovery-file-name

Specify the folder name of the indexed file to be recovered.

unrecoverable-file-name

Specify the folder name of the file to which unrecoverable record data is to be output. If no unrecoverable data is found, the unrecoverable data file is not created.

Example

Recover the indexed file

- Recovery file name: ixdfile
- Unrecoverable data file: seqfile

```
cobfrcov ixdfile seqfile
```

Note

When "File recovery command" is executed at the same time for the same index file, operation is not guaranteed. Execute the next command after confirming that execution of the previous command was completed.

6.8.3.8 File reorganization command(cobfreog)

The cobfreog command erases the empty blocks in an indexed file, and outputs the reorganizes content to another indexed file. Erasing empty blocks results is a smaller file size.

Specification format

```
cobfreog -ooutput-file-name input-file-name
```

Interface

Specify the command parameters as shown below:

output-file-name

Specify the folder name of the indexed file to be created by reorganization. If an existing file is specified, an error occurs.

input-file-name

Specify the folder name of the indexed file to be reorganized.

Example

Reorganize an indexed file and output the reorganized file to outfile.

- Input file name : ixdfile
- Output file name : outfile

```
cobfreog -ooutfile ixdfile
```

Note

Erasing empty blocks may cause file access performance to deteriorate depending on the indexed file structure.

6.8.4 COBOL File Utility Functions

The COBOL file utility provides the following functions.

Function	Function name
File conversion function	COB_FILE_CONVERT
File load function	COB_FILE_LOAD
File unload function	COB_FILE_UNLOAD
File sort function	COB_FILE_SORT
File reorganization function	COB_FILE_REORGANIZE
File copy function	COB_FILE_COPY
File move function	COB_FILE_MOVE
File deletion function	COB_FILE_DELETE

Library

To call the COBOL file utility functions, add code to the WORKING-STORAGE SECTION:

```
WORKING-STORAGE  SECTION.
    COPY  COBF-INF.
```

This library file is located in the folder where you installed NetCOBOL.

Note

- Do not specify the REPLACING phrase of the COPY statement used to read the library above.
- If the library is modified, operation is not guaranteed.

- Before calling each COBOL file utility function, always initialize COBF-INF in the interface area using LOW-VALUE.

Link time

Please link F4AGFUTC.LIB when it is called from the program compiled without compiler option "DLOAD". This library is stored in the folder where NetCOBOL has been installed.

Execution time

The following ENTRY information is necessary when it is called from the program compiled with compiler option "DLOAD".

About ENTRY information, please refer to "5.4.2 Entry Information for Subprograms".

```
[ENTRY]
COB_FILE_COMVERT=F4AGFUTC.DLL
COB_FILE_LOAD=F4AGFUTC.DLL
COB_FILE_UNLOAD=F4AGFUTC.DLL
COB_FILE_SORT=F4AGFUTC.DLL
COB_FILE_REORGANIZE=F4AGFUTC.DLL
COB_FILE_COPY=F4AGFUTC.DLL
COB_FILE_MOVE=F4AGFUTC.DLL
COB_FILE_DELETE=F4AGFUTC.DLL
```

6.8.4.1 File conversion function

Function

The file conversion function creates a record sequential file in the variable length format based on a text format file. The function can also create a text format file based on a record sequential file in the variable length format.

The binary data is written as a character string of the hexadecimal mark in a text format file.

Specification method

Calling format

```
CALL  "COB_FILE_CONVERT" USING BY REFERENCE COBF-INF
```

Parameters

Specify the appropriate data items in COBF-INF to describe the record sequential file. The data format definitions are listed below.

- When using native code, "Data item to be displayed" is called a character format and other data items are called a hexadecimal format.

  For details on the "Data item to be displayed," see the COBOL Reference Manual.

- When using Unicode code, national data item and national edited data item are called an UCS-2/UTF-32 format, and "Data item to be displayed" without national data items and national edited data items is called an UTF-8 format and other data items are called a hexadecimal format.

- When using Unicode code, national data items and national edited data items are regarded as UCS-2/UTF-32 format, and "Data item to be displayed" without national data items and national edited data items is regarded as UTF-8 format. Other data items are called a hexadecimal format.

The character code of text files are handled as shown below.

- When using native code, they are handled as native character codes.

- When using Unicode code (UCS-2), they are handled as UCS-2 little endian character codes.

- When using Unicode code (UTF-32), they are handled as UTF-32 little endian character codes.

Update the following data items:

COBF-INPUT-FILENAME

Specify the folder name of the text file or record sequential file input to the conversion origin.

COBF-OUTPUT-FILENAME

Specify the folder name of the text file or record sequential file created by the conversion process.

COBF-CONVERT-COND

Specify the conversion mode.

- "B" : Converts a text file to a record sequential file
- "T" : Converts a record sequential file to a text file

COBF-CONVERT-NUM

Specify the number of data format that composes a record of the record sequential file.

Note

Up to 256 can be specified.

COBF-CONVERT-TYPE

Specify the data item of the record sequential file. The following condition names are used for the specification of the data format.

Condition name	Value	Meaning
COBF-CONVERT-TYPE-ALLCHAR	"1"	All data in the record is represented by characters.
COBF-CONVERT-TYPE-ALLTXTBIN	"2"	All data in the record is represented by hexadecimal digits.
COBF-CONVERT-TYPE-CHAR	"3"	Only the specified length is represented by characters.
COBF-CONVERT-TYPE-TXTBIN	"4"	Only the specified length is represented by hexadecimal digits.
COBF-CONVERT-TYPE-KANJI	"5"	Only 2-byte notation kanji format is recognized as the length that can be specified.
COBF-CONVERT-TYPE-ALLUCS2	"6"	All data in the record is represented in the UCS-2 format.
COBF-CONVERT-TYPE-ALLUTF8	"7"	All data in the record is represented in the UTF-8 format.
COBF-CONVERT-TYPE-UCS2	"8"	Only the specified length is represented in the UCS-2 format.
COBF-CONVERT-TYPE-UTF8	"9"	Only the specified length is represented in the UTF-8 format.
COBF-CONVERT-TYPE-ALLUTF32	"A"	All data in the record is represented in the UTF-32 format.
COBF-CONVERT-TYPE-UTF32	"B"	Only the specified length is represented in the UTF-32 format.

COBF-CONVERT-LEN

Specify the length of data to be converted (number of characters for UCS-2/UTF-32 format). When using COBF-CONVERT-TYPE, the following data items must also be updated:

- COBF-CONVERT-TYPE-CHAR
- COBF-CONVERT-TYPE-TXTBIN
- COBF-CONVERT-TYPE-UCS2
- COBF-CONVERT-TYPE-UTF8
- COBF-CONVERT-TYPE-UTF32

However, this specification is disregarded if COBF-CONVERT-TYPE is any of the following,

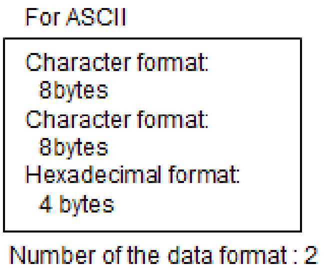
- COBF-CONVERT-TYPE-ALLCHAR
- COBF-CONVERT-TYPE-ALLTXTBIN
- COBF-CONVERT-TYPE-ALLUCS2
- COBF-CONVERT-TYPE-ALLUTF8
- COBF-CONVERT-TYPE-ALLUTF32

In the following example, a record composed of different data types is used:

Record structure of the record sequential file

```
FD file-1.
 01 data-record.
   02 data1  PIC X(8).
   02 data2  PIC N(4).
   02 data3  PIC S9(8) BINARY.
```

The data format representations are as follows:



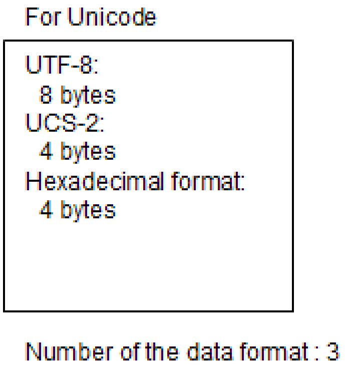
For ASCII

Character format:
 8bytes
Character format:
 8bytes
Hexadecimal format:
 4 bytes

Number of the data format : 2

Data to be set for calling

```
     MOVE 2 TO COBF-CONVERT-NUM
     SET COBF-CONVERT-TYPE-CHAR(1) TO TRUE
     MOVE 16 TO COBF-CONVERT-LEN(1)
     SET COBF-CONVERT-TYPE-TXTBIN(2) TO TRUE
     MOVE  4 TO COBF-CONVERT-LEN(2)
```



For Unicode

UTF-8:
 8 bytes
UCS-2:
 4 bytes
Hexadecimal format:
 4 bytes

Number of the data format : 3

Data to be set for calling

```
     MOVE 3 TO COBF-CONVERT-NUM
     SET COBF-CONVERT-TYPE-UTF8(1) TO TRUE
     MOVE  8 TO COBF-CONVERT-LEN(1)
     SET COBF-CONVERT-TYPE-UCS2(2) TO TRUE
     MOVE  4 TO COBF-CONVERT-LEN(2)
     SET COBF-CONVERT-TYPE-TXTBIN(3) TO TRUE
     MOVE  4 TO COBF-CONVERT-LEN(3)
```

Return code

The return code for this function is received in the special register PROGRAM-STATUS.

- 0 : File conversion was successful.
- -1 : File conversion was unsuccessful.

When return code -1 is returned, details of the error are set in COBF-MESSAGE character string.

Example

1) Create a text file from a record sequential file.

- Input file name: C:\INFILE
- Output file name: C:\OUTFILE.TXT
- Conversion mode: Record sequential file -> text format file
- Conversion type: All character format

```
MOVE LOW-VALUE TO COBF-INF.
MOVE "C:\INFILE" TO COBF-INPUT-FILENAME.
MOVE "C:\OUTFILE.TXT" TO COBF-OUTPUT-FILENAME.
MOVE "T" TO COBF-CONVERT-COND.
MOVE 1 TO COBF-CONVERT-NUM.
SET  COBF-CONVERT-TYPE-ALLCHAR(1) TO TRUE.
CALL "COB_FILE_CONVERT" USING BY REFERENCE COBF-INF.
```

2) Create a record sequential file from a text file.

- Input file name: C:\INFILE.TXT
- Output file name: C:\OUTFILE
- Conversion mode: Text format file -> record sequential file
- Conversion type: Mixed (3 bytes character format, 2 bytes of hexadecimal format, 3 bytes character format)

```
MOVE LOW-VALUE TO COBF-INF.
MOVE "C:\INFILE.TXT" TO COBF-INPUT-FILENAME.
MOVE "C:\OUTFILE" TO COBF-OUTPUT-FILENAME.
MOVE "B" TO COBF-CONVERT-COND.
MOVE 3 TO COBF-CONVERT-NUM.
SET  COBF-CONVERT-TYPE-CHAR(1) TO TRUE.
MOVE 3 TO COBF-CONVERT-LEN(1).
SET  COBF-CONVERT-TYPE-TXTBIN(2) TO TRUE.
MOVE 2 TO COBF-CONVERT-LEN(2).
SET  COBF-CONVERT-TYPE-CHAR(3) TO TRUE.
MOVE 3 TO COBF-CONVERT-LEN(3).
CALL "COB_FILE_CONVERT" USING BY REFERENCE COBF-INF.
```

6.8.4.2 File load function

Function

The file load function creates a record sequential, relative, or indexed file based on a variable length record sequential file. The command also adds (extends) record data from a variable length record sequential file to an existing record sequential, relative, or indexed file. When the error occurs during file extension, the output file is returned to the state it was in before the file is extended.

Specification method

Calling format

```
CALL  "COB_FILE_LOAD" USING BY REFERENCE COBF-INF.
```

Data to be set for calling

COBF-INPUT-FILENAME

Specify the input file and folder name of a record sequential file.

COBF-OUTPUT-FILENAME

Specify the output file and folder name of the file to be created or extended. If an existing file is specified but not extended, an error occurs. If the specified file does not exist when extended, an error occurs.

COBF-OUTPUT-ATTR

Specify the attribute of the file to be created or extended, using the following characters.

- "S" : Record sequential file
- "R" : Relative file
- "I" : Indexed file

COBF-OUTPUT-RECFM

Specify the record format of the file to be created or extended, using the following characters.

- "F" : Fixed length format
- "V" : Variable length format

COBF-OUTPUT-RECLEN

Specify the maximum record length of the file to be created or extended. When using variable length data, specify the maximum record length.

COBF-OUTPUT-KEYNUM

Specify the number of data items used as record key when the indexed file is created or extended.

COBF-OUTPUT-USE-EXTKEY

Specify whether the attribute is specified for the data items used as record key when the indexed file is created or extended, using the following characters. The attribute is specified with COBF-OUTPUT-EXT-KEY-TYPE.

- "Y" : The data item attribute is used
- " " : The data item attribute is not used

Specify "Y" if the data item to be used as record key is in the UCS-2 or UTF-32 encoding form.

COBF-LOAD-COND

Specify whether to create or extended the file using the following characters.

- "C" : Create
- "E" : Extend

Specify the following information for each record key:

COBF-OUTPUT-OFFSET

Specify the relative location (byte count) of the record key data item, where the first byte is byte 0.

COBF-OUTPUT-KEYLEN

Specify the length of the record key data item in bytes.

COBF-OUTPUT-KEYCONT

Specify the key configuration, using the following characters.

- "C" : Indicates a key definition continues.
- "E" : Indicates the end of a key definition.

COBF-OUTPUT-KEYDUP

Specify "D" to permit duplicate record keys. This parameter is valid only when E is specified for COBF-OUTPUT-KEYCONT.

COBF-OUTPUT-EXT-KEY-TYPE

Specify the attribute of the data item used as the record key when "Y" is specified for COBF-OUTPUT-USE-EXTKEY, using the following condition name.

Condition name	Value	Meaning
COBF-EXT-KEY-TYPE-PICX	1	Item of not PIC N
COBF-EXT-KEY-TYPE-PICN	2	Item of PIC N with UCS-2 encoding form
COBF-EXT-KEY-TYPE-PICN32	3	Item of PIC N with UTF-32 encoding form

Return code

The return code for this function is received in the special register PROGRAM-STATUS.

Return code	Meaning
0	File loading was successful.
-1	File loading was unsuccessful.

When return code -1 is returned, details of the error are moved to the OBF-MESSAGE character string.

Note

- If the input file contains a record that is longer that the maximum record length of the output file, an error occurs when the function is executed.
- In indexed file extension mode, a record with a key value that is smaller than the maximum key value can be written out.

Example

1) Create an indexed file from a record sequential file.

- Input file name: C:\INFILE
- Output file name: C:\IXDFILE
- File organization: Indexed file
- Record format: Variable length
- Record length: 80 bytes
- Main record key: Consisting of two data items. The record key value can be duplicated.
  - Record key 1: 5 bytes beginning from 0-byte offset
  - Record key 2: 5 bytes beginning from 10-byte offset
- Alternate record key: 5 bytes beginning from 5-byte offset

```
MOVE LOW-VALUE TO COBF-INF.
MOVE "C:\INFILE" TO COBF-INPUT-FILENAME.
MOVE "C:\IXDFILE" TO COBF-OUTPUT-FILENAME.
MOVE "I" TO COBF-OUTPUT-ATTR.
MOVE "V" TO COBF-OUTPUT-RECFM.
MOVE 80 TO COBF-OUTPUT-RECLEN.
MOVE 3 TO COBF-OUTPUT-KEYNUM.
MOVE 0 TO COBF-OUTPUT-OFFSET(1).
MOVE 5 TO COBF-OUTPUT-KEYLEN(1).
MOVE "C" TO COBF-OUTPUT-KEYCONT(1).
MOVE 5 TO COBF-OUTPUT-OFFSET(2).
MOVE 10 TO COBF-OUTPUT-KEYLEN(2).
MOVE "E" TO COBF-OUTPUT-KEYCONT(2).
MOVE "D" TO COBF-OUTPUT-KEYDUP(2).
MOVE 5 TO COBF-OUTPUT-OFFSET(3).
MOVE 5 TO COBF-OUTPUT-KEYLEN(3).
MOVE "E" TO COBF-OUTPUT-KEYCONT(3).
MOVE "C" TO COBF-LOAD-COND.
CALL "COB_FILE_LOAD" USING BY REFERENCE COBF-INF.
```

2) Add the content of a record sequential file to a relative file.

- Input file name: C:\INFILE
- Output file name: C:\RELFILE
- File organization: Relative file
- Record format: Fixed length
- Record length: 80 bytes

```
MOVE LOW-VALUE TO COBF-INF.
MOVE "C:\INFILE" TO COBF-INPUT-FILENAME.
MOVE "C:\RELFILE" TO COBF-OUTPUT-FILENAME.
MOVE "R" TO COBF-OUTPUT-ATTR.
MOVE "F" TO COBF-OUTPUT-RECFM.
MOVE 80 TO COBF-OUTPUT-RECLEN.
MOVE "E" TO COBF-LOAD-COND
CALL "COB_FILE_LOAD" USING BY REFERENCE COBF-INF.
```

6.8.4.3 File unload function

Function

The file unload function creates a variable length record sequential file based on a record sequential, relative, or indexed file.

Specification method

Calling format

```
CALL  "COB_FILE_UNLOAD" USING BY REFERENCE COBF-INF.
```

Data to be set for calling

COBF-INPUT-FILENAME

Specify the input file and folder name of a record sequential file or relative file or indexed file.

COBF-INPUT-ATTR

Specify input file organization using the following characters.

Value	Meaning
"S"	Record sequential file
"R"	Relative file
"I"	Indexed file

COBF-INPUT-RECFM

Specify the record format of the input file using the following characters.

Value	Meaning
"F"	Fixed length
"V"	Variable length

COBF-INPUT-RECLEN

Specify the record length of the input file. When using variable length data, specify the maximum record length. This parameter cannot be specified when the input file is an indexed file.

COBF-OUTPUT-FILENAME

Specify the output file and folder name of a record sequential file to be created. If an existing file is specified, an error occurs.

Return code

The return code for this function is received in the special register PROGRAM-STATUS.

Return code	Meaning
0	File unloading was successful.
-1	File unloading was unsuccessful.

When return code -1 is returned, details of the error are moved to the COBF-MESSAGE character string.

Example

Create a record sequential file from a relative file.

- Input file name: C:\RELFILE
- Output file name: C:\OUTFILE
- File organization: Relative file
- Record format: Fixed length
- Record length: 80 bytes

```
MOVE LOW-VALUE TO COBF-INF.
MOVE "C:\RELFILE" TO COBF-INPUT-FILENAME.
MOVE "R" TO COBF-INPUT-ATTR.
MOVE "F" TO COBF-INPUT-RECFM.
MOVE 80 TO COBF-INPUT-RECLEN.
MOVE "C:\OUTFILE" TO COBF-OUTPUT-FILENAME.
CALL "COB_FILE_UNLOAD" USING BY REFERENCE COBF-INF.
```

6.8.4.4 File sort function

Function

The file sort function sorts the records in a file in ascending or descending order using a specified data item in the record as a key, and the sorted records are output to a variable length record sequential file.

Specification method

Calling format

```
CALL  "COB_FILE_SORT" USING BY REFERENCE COBF-INF.
```

Data to be set for calling

COBF-INPUT-FILENAME

Specify the input file and folder name of the file to be sorted. The specified file must be a record sequential, line sequential, relative, or indexed file.

COBF-INPUT-ATTR

Specify the file organization to be sorted, using the following characters.

Value	Meaning
"S"	Record sequential file
"L"	Line sequential file
"R"	Relative file
"I"	Indexed file

COBF-INPUT-RECFM

Specify the record format of the file to be sorted, using the following characters.

Value	Meaning
"F"	Fixed length
"V"	Variable length

COBF-INPUT-RECLEN

Specify the maximum record length of the file to be sorted, using the following characters. When using a variable length data item, specify the maximum record length. This parameter cannot be specified when the input file is an indexed file.

COBF-OUTPUT-FILENAME

Specify the output file and folder name of the record sequential file to which the sorted records are to be output. If an existing file is specified, an error occurs.

COBF-SORT-KEYNUM

Specify the number of keys to be sorted.

Note

Up to 64 can be specified.

COBF-SORT-OFFSET

Specify the relative location of the data item used as a key from the top of the record.

COBF-SORT-KEYLEN

Specify the size of the data item used as a key. PIC 1 () BIT specified for a key item indicates a mask value corresponding to the size made up of 1 byte.

COBF-SORT-KEYATTR

Specify the item attribute of a key. The following condition names are used in the specifications of an item attribute.

Condition Name	Value	Meaning
COBF-SORT-KEYATTR-PIC1	1	PIC 1()
COBF-SORT-KEYATTR-PIC1B	2	PIC 1() BIT
COBF-SORT-KEYATTR-PICX	3	PIC X()

Condition Name	Value	Meaning
COBF-SORT-KEYATTR-PICN	4	PIC N() (*1)
COBF-SORT-KEYATTR-PIC9	5	PIC 9()
COBF-SORT-KEYATTR-PICS9	6	PIC S9()
COBF-SORT-KEYATTR-PICS9L	8	PIC S9() LEADING
COBF-SORT-KEYATTR-PICS9T	10	PIC S9() TRAILING
COBF-SORT-KEYATTR-PICS9LS	12	PIC S9() LEADING SEPARATE
COBF-SORT-KEYATTR-PICS9TS	14	PIC S9() TRAILING SEPARATE
COBF-SORT-KEYATTR-PIC9PD	15	PIC 9() PACKED-DECIMAL
COBF-SORT-KEYATTR-PICS9PD	16	PIC S9() PACKED-DECIMAL
COBF-SORT-KEYATTR-PIC9B	17	PIC 9() BINARY
COBF-SORT-KEYATTR-PICS9B	18	PIC S9() BINARY
COBF-SORT-KEYATTR-PIC9C5	19	PIC 9() COMP-5
COBF-SORT-KEYATTR-PICS9C5	20	PIC S9() COMP-5
COBF-SORT-KEYATTR-PICC1	21	COMP-1
COBF-SORT-KEYATTR-PICC2	22	COMP-2

*1: The character encoding form follows compile option ENCODE. The sort processing that makes the data item have a different character encoding than that specified with the compile option ENCODE is not allowed.

COBF-SORT-KEYSEQ

Specify the sort order, using the following characters.

Value	Meaning
"A"	Sort records in ascending order.
"D"	Sort records in descending order.

Return code

The return code for this function is received in the special register PROGRAM-STATUS.

Return code	Meaning
"A"	Sort records in ascending order.
"D"	Sort records in descending order.

When return code -1 is returned, details of the error are moved to the COBF-MESSAGE character string.

Example

Sort a relative file and output the sorted records to OUTFILE.

- Input file name C:\RELFILE
- Output file name C:\OUTFILE
- Sort condition :
  - First Key :
    - item attribute: NATIONAL

            - Offset in record: 0
            - Size: 2
            - Sortorder: Ascending order
        - Second Key:
            - Item attribute: Alphanumeric
            - Offset in record: 10
            - Size: 5
            - Sortorder: Descending order
    - File attribute :
        - File organization: Relative file
        - Record format: Variable length
        - Record length: 80 bytes

```
MOVE LOW-VALUE TO COBF-INF.
MOVE "C:\RELFILE" TO COBF-INPUT-FILENAME.
MOVE "R" TO COBF-INPUT-ATTR.
MOVE "V" TO COBF-INPUT-RECFM.
MOVE 80 TO COBF-INPUT-RECLEN.
MOVE "C:\OUTFILE" TO COBF-OUTPUT-FILENAME.
MOVE 2 TO COBF-SORT-KEYNUM.
SET  COBF-SORT-KEYATTR-PICN(1) TO TRUE.
MOVE 0 TO COBF-SORT-OFFSET(1).
MOVE 2 TO COBF-SORT-KEYLEN(1).
MOVE "A" TO COBF-SORT-KEYSEQ(1).
SET  COBF-SORT-KEYATTR-PICX(2) TO TRUE.
MOVE 10 TO COBF-SORT-OFFSET(2).
MOVE 5 TO COBF-SORT-KEYLEN(2).
MOVE "D" TO COBF-SORT-KEYSEQ(2).
CALL "COB_FILE_SORT" USING BY REFERENCE COBF-INF.
```

6.8.4.5 File reorganization function

Function

The file reorganization function erases the empty blocks in an indexed file, and outputs the reorganized content to another indexed file. The file size of a reorganized indexed file is smaller than before reorganization.

Specification method

Calling format

```
CALL  "COB_FILE_REORGANIZE"  USING BY REFERENCE COBF-INF.
```

Data to be set for calling

COBF-INPUT-FILENAME

Specify the input file and folder name of the indexed file to be reorganized.

COBF-OUTPUT-FILENAME

Specify the output file and folder name of the indexed file created by reorganization.

Return code

The return code for this function is received in the special register PROGRAM-STATUS.

Return code	Meaning
0	File reorganization was successful.
-1	File reorganization was unsuccessful.

When return code -1 is returned, details of the error are moved to the COBF-MESSAGE character string.

Example

Reorganize an indexed file and output the reorganized file to OUTFILE.

- Input file name: C:\IXDFILE
- Output file name: C:\OUTFILE

```
MOVE LOW-VALUE TO COBF-INF.
MOVE "C:\IXDFILE" TO COBF-INPUT-FILENAME.
MOVE "C:\OUTFILE" TO COBF-OUTPUT-FILENAME.
CALL "COB_FILE_REORGANIZE" USING BY REFERENCE COBF-INF.
```

6.8.4.6 File copy function

Function

The file copy function copies a file.

Specification method

Calling format

```
CALL  "COB_FILE_COPY" USING BY REFERENCE COBF-INF.
```

Data to be set for calling

COBF-INPUT-FILENAME

Specify the input file and folder name of the copy source file.

COBF-OUTPUT-FILENAME

Specify the output file and folder name of the copy destination file.

Return code

The return code for this function is received in the special register PROGRAM-STATUS.

Return code	Meaning
0	File copying was successful.
-1	File copying was unsuccessful.

When return code -1 is returned, details of the error are moved to the COBF-MESSAGE character string.

Note

Wild card characters (?, *) cannot be included in the specified in the input or output file names.

Example

Copy C:\INFILE to C:\OUTFILE

```
MOVE LOW-VALUE TO COBF-INF.
MOVE "C:\INFILE" TO COBF-INPUT-FILENAME.
MOVE "C:\OUTFILE" TO COBF-OUTPUT-FILENAME.
CALL "COB_FILE_COPY" USING BY REFERENCE COBF-INF.
```

6.8.4.7 File move function

Function

The file move function moves a file.

Specification method

Calling format

```
CALL  "COB_FILE_MOVE" USING BY REFERENCE COBF-INF.
```

Data to be set for calling

COBF-INPUT-FILENAME

Specify the input file and folder name of the move source file.

COBF-OUTPUT-FILENAME

Specify the output file and folder name of the move destination file.

Return code

The return code for this function is received in the special register PROGRAM-STATUS.

Return code	Meaning
0	File moving was successful.
-1	File moving was unsuccessful.

When return code -1 is returned, details of the error are moved to the COBF-MESSAGE character string.

Note

Wild card characters (?, *) cannot be included in the specified file name.

Example

Move C:\INFILE to C:\OUTFILE.

```
MOVE LOW-VALUE TO COBF-INF.
MOVE "C:\INFILE" TO COBF-INPUT-FILENAME.
MOVE "C:\OUTFILE" TO COBF-OUTPUT-FILENAME.
CALL "COB_FILE_MOVE" USING BY REFERENCE COBF-INF.
```

6.8.4.8 File deletion function

Function

The file deletion function deletes a file.

Specification method

Calling format

```
CALL  "COB_FILE_DELETE" USING BY REFERENCE COBF-INF.
```

Data to be set for calling

COBF-INPUT-FILENAME

Specify the input file and folder name of the file to be deleted.

Return code

The return code for this function is received in the special register PROGRAM-STATUS.

Return code	Meaning
0	File deletion was successful.
-1	File deletion was unsuccessful.

When return code -1 is returned, details of the error are set in COBF-MESSAGE by character string.

Note

No wild card (?, *) cannot be included in the specified file name.

Example

Delete C:\INFILE

```
  MOVE LOW-VALUE TO COBF-INF.
  MOVE "C:\INFILE" TO COBF-INPUT-FILENAME.
  CALL "COB_FILE_DELETE" USING BY REFERENCE COBF-INF.
```

6.9 How to Use Other File Systems

With NetCOBOL, in addition to the COBOL file system included in the product package, certain non-COBOL file systems can be used by specifying a specific character string for the file name.

The usable file systems are listed below.

Btrieve

Btrieve is a record management system from Pervasive Software Inc.

External File Handler

The External File Handler's interface is the same format as the Micro Focus file handler.

The range of I/O functions that can be used in the External File Handler depends on the file system used.

The I/O function ranges usable by each of the COBOL file system and Btrieve are listed in "Table 6.9 Functional differences between file systems".

Table 6.9 Functional differences between file systems

Classification	Item		COBOL file system	Btrieve (**1)
File	Maximum file size (number of bytes)	Record sequential file	System limit	256 G
		Record sequential file (BSAM specification)	System limit	-

Classification	Item		COBOL file system	Btrieve (**1)
		Line sequential file	System limit	-
		Line sequential file (BSAM specification)	System limit	-
		Relative file	1 G	-
		Index file	System limit	256 G
Record	Record format	Fixed-length record format	O	O
		Variable-length record format	O	O
	Maximum record length (number of bytes)	Fixed-length record format	32,760	32,760
		Variable-length record format	32,760	32,760
	Minimum record length (number of bytes)	record sequential	1	4
		line sequential	0	-
		Relative	1	-
		Index	Size up to items that comprise key	Size up to items that comprise key
File management entry	SELECT clause	OPTIONAL specification	O	O
	ASSIGN clause	File identifier specification	O	O
		File identifier literal specification	O	O
		Data name specification	O	O
		DISK specification	O	-
	FILE STATUS clause	File status	O	O (**2)
	LOCK MODE clause	AUTOMATIC	O	O
		EXCLUSIVE	O	O
		MANUAL	O	O
	RECORDED KEY clause	Maximum number of specifiable data items	254 (**3)	204 (**4)
		Maximum total data length (number of bytes) specifiable	254	255
	ALTERNATE RECORD KEY clause	Maximum number of specifiable ALTERNATE RECORD KEY clause	125	118
		Maximum number of specifiable data items	254 (**5)	203 (**4)
		Maximum total data length (number of bytes) specifiable	254	255
	RELATIVE KEY phrase	Maximum value of specifiable data items	9,223,372,036,854,775,807	-

Classification	Item		COBOL file system	Btrieve (**1)
	Record key item (**5)	Alphanumeric	O	O
		National	O	O
		Unsigned zoned decimal	O	O (**6)
		Signed zoned decimal	-	O (**6)
		Unsigned packed decimal	O	O
		Signed packed decimal	-	O
		Unsigned binary (Normal ascending order storage format)	O	O
		Signed binary (Normal ascending order storage format)	-	O
		Unsigned binary (Reverse-order storage format)	-	O
		Signed binary (Reverse-order storage format)	-	O
Statement	READ statement	WITH LOCK specification	O	O
	START statement	Entire key specification	O	O
		Partial key specification	O	O
	DELETE statement	Deletion of the record with duplicated key values	O	O (**7)
	UNLOCK statement		O	O
Function	Thread	Single thread	O	O
		Multithread	O	O
	Transaction management	Transaction start instruction	-	O (**8)
		COMMIT instruction	-	O (**8)
		ROLLBACK instruction	-	O (**8)
	Recovery		O (**9)	O (**9)

O: Supported

-: Not supported

- **1
  Btrieve is used based on the information of "Pervasive PSQL V10 SP1".
- **2
  Btrieve does not return FILE STATUS=02.
- **3
  The total sum of the number of data items specifiable in the RECORD KEY clause and ALTERNATE RECORD KEY clause is up to 255 in the COBOL file system.

- **4
  The total sum of the number of data items specifiable in the RECORD KEY clause and ALTERNATE RECORD KEY clause is up to 204 in Btrieve.

- **5
  If unsupported data items are defined in record keys, the execution results are undefined.

- **6
  The NUMERIC type of the Btrieve file is equivalent to the signed zoned decimal data item data type without the SEPARATE specification in COBOL. However, the internal format of NUMERIC is called "88 consortium format", different from the internal format of COBOL 97. To use a signed zoned decimal data item that was defined without the SEPARATE phrase as the input or output item, data must be converted before reading and writing. For details on data conversion, see the topic "External Decimal Data Form Conversion" later in this chapter.

- **7
  In the Btrieve, if the DELETE statement of random access mode is used to delete the record of which file position directive is defined, the file position directives up to this file position directive become optional. The file position directive also becomes optional when one of the following statements is executed after the READ statement of the sequential access mode is executed with the subkey and the second or later record from the beginning of the duplicated subkey is read.

  - REWRITE statement of random access mode

  - DELETE statement of random access mode

  - WRITE statement of random access mode

- **8
  The function provided with the Btrieve must be called with the CALL statement. To perform the transaction operation with the Btrieve, the program support of AG-TECH Inc must be purchased separately.

- **9
  The COBOL file system provides the recovery function of the index file that can no longer be accessed. The Btrieve provide the recovery feature or various tools combined with the transaction management function.

6.9.1 Btrieve Files

Btrieve files can be used as record sequential and indexed files.

Please refer to "Table 6.9 Functional differences between file systems" for the functions available when using Btrieve files.

6.9.1.1 Specifying the Btrieve File Environment

This section explains how to use Btrieve files.

To specify that the Btrieve Record Manager should be used instead of the COBOL file system, add the string "BTRV" to the file-reference-identifier in the ASSIGN clause of the file control entry.

File-Identifier Literal to ASSIGN clause

If a file-identifier literal is specified as the file-reference-identifier, specify the file-identifier literal in the following format:

```
                                  ┌ no-specification ┐
ASSIGN TO "[path-name]file-name  {                    }  "
                                  └ ,BTRV            ┘
```

file name

Specify the file name or folder name of the input-output target file.

no-specification

The COBOL file system is used.

BTRV

The Btrieve Record Manager is used.

Data-Name for File-Reference-Identifier to ASSIGN clause

If a data name is specified as the file-reference-identifier, use the above format in the literal defined as the VALUE of the data name or that is moved to the data name.

```
                                   { no-specification  }
MOVE "[path-name]file-name         {                   }  " TO data-name
                                   { ,BTRV             }
```

DISK as File-Reference-Identifier to ASSIGN clause

If the character string DISK is specified as the file-reference-identifier, the Btrieve Record Manager cannot be used. The COBOL file system is used.

File-Identifier as File-Reference-Identifier to ASSIGN clause

If a file-identifier is specified as the file-reference-identifier, you add the "BTRV" string to the environment variable definition.

The environment variable specification format is:

```
                                             { no-specification  }
Environment-variable=[path-name]file-name    {                   }
                                             { ,BTRV             }
```

6.9.1.2 External Decimal Data Form Conversion

To handle external decimal data items created without the SIGN IS SEPARATE clause, two methods of converting the internal format are prepared.

- To convert item from NetCOBOL to Btrieve file format (88 consortium)
- To convert items from 88 consortium to NetCOBOL format

For the internal format of the NUMERIC data type for Btrieve, refer to the "BTRIEVE Programmer's Manual" by Btrieve Technologies, Inc.

Conversion Subroutines

- To convert items from 88consortium to COBOL format, use the following call:

  ```
  CALL "#DEC88TOFJ" USING [BY REFERENCE] name
  ```

  name is an elementary or group item containing external decimal data items not using the SIGN SEPARATE clause.

- To convert items from COBOL to 88consortium, use the following call:

  ```
  CALL "#DECFJTO88" USING [BY REFERENCE] name
  ```

  name is an elementary or group item containing external decimal data items with no SIGN SEPARATE clause.

Subroutine Restrictions

- Conversions are only performed for external decimal data items that do not have a SIGN SEPARATE specification.
- When the name is a group item and the following items are included in the subordinate items, the item will not be converted:
  - Items other than external decimal data items that do not have a SIGN SEPARATE specification.
  - Items subordinate to items that specify the REDEFINES clause as well as the item itself.
  - Items that specify the RENAMES clause.
- In the following cases, the execution results are not guaranteed:
  - A data name is used to define the target program of the CALL statement. (Even if the data name value is either "#DEC88TOFJ" or "#DECFJTO88".)

- "#DEC88TOFJ" or "#DECFJTO88" is called by a form different from the above-mentioned description form (E.g. Two or more names are specified in the USING phrase, or a BY CONTENT phrase is specified.).
- The name is a group item and the item uses the OCCURS clause with DEPENDING specification in a subordinate item.
- When data items are defined in the constant paragraph.

Note

- Use these subroutines immediately before passing a record to a Btrieve file or immediately after receiving one. COBOL cannot handle 88consortium format numbers. This means that the results of statements that 'operate', 'compare' or 'move as external decimal data item' are not guaranteed.
- The value of 88consortium format numbers is only guaranteed when contained in group items moved to other records.
- You cannot use the sort and merge functions with Btrieve files.
- Incorrect functions will be called if you receive external reference errors on "#DECFJTO88" or "#DEC88TOFJ" when linking.

Example

```
*>   :
 FILE-CONTROL.
      SELECT file-1  ASSIGN TO "filename.BTRV"
                     ORGANIZATION IS INDEXED
                     ACCESS MODE SEQUENTIAL
                     RECORD KEY IS mainkey OF record-data.
 DATA DIVISION.
 FILE SECTION.
 FD file-1.
 01 record-data.
   02 mainkey    PIC 9(4).
   02 data1      PIC S9(8).
   02 data2      PIC X(50).
 WORKING-STORAGE SECTION.
 01 workarea.
   02 mainkey    PIC 9(4).
   02 data1      PIC S9(8).
   02 data2      PIC X(50).
PROCEDURE   DIVISION.
*>       :
*> Writing in Btrieve file.
     OPEN OUTPUT file-1.
     CALL "#DECFJTO88" USING workarea.    *>... *1
     WRITE record-data FROM workarea.
     CLOSE file-1.
*>     :
*> Reading from Btrieve file.
     OPEN INPUT file-1.
     MOVE 6 TO mainkey OF record-data.
     START file-1.
     READ file-1 INTO workarea.
     CALL "#DEC88TOFJ" USING workarea.    *>... *2
     DISPLAY "data1 :" data1 OF workarea.
     CLOSE file-1.
```

*1 : Converts all external decimal data items that have no SIGN SEPARATE clause to 88consortium format, before the record is written to Btrieve.

*2 : Converts all external decimal data items that have no SIGN SEPARATE clause from 88consortium format into COBOL format, after the record is read from Btrieve.

6.9.1.3 Note

If you specify AUTOMATIC in the LOCK MODE clause, you cannot release the lock for unlocked records, even by executing the WRITE, REWRITE, DELETE, or START statements. However, you can release the lock for locked records by executing the REWRITE or DELETE statement.

6.9.2 External File Handler

The external file handler interface can be used to provide support for COBOL record sequential file, line sequential file, relative file and indexed file syntax. This feature is provided for compatibility with the equivalent Micro Focus COBOL feature so the interface details are not documented. This section describes how you configure your NetCOBOL system to use a DLL external file handler.

Usage

- When defining a data-name as a file-reference-identifier

  Specify ",EXFH" following the file-name to be allocated when setting the environment variable information.

  ```
  file-identifier=file-name,EXFH
  ```

- When defining a file-identifier literal as a file-reference-identifier

  Specify ",EXFH" following the file-name to be allocated when defining the file-identifier literal in the program.

  ```
  ASSIGN TO "file-name,EXFH".
  ```

- When defining a data name as a file-reference-identifier

  Specify ",EXFH" following the file-name to be allocated when defining the data-name in the program.

  ```
  MOVE  "file-name,EXFH" TO data-name.
  ```

Specification method

When an external file handler is used, DLL name must be defined. There are two methods for providing this information:

- Specifying the external file handler for the execution environment

  The following environment variables are set at execution time:

  ```
  @CBR_EXFH_API=entry-point-name
  ```

  entry-point-name : The entry point name of the EXFH file system implementation.

  ```
  @CBR_EXFH_LOAD=DLL-name
  ```

  DLL-name : DLL name of the EXFH file system implementation.

  Both an absolute path and a relative path can be specified for the path of the file. When a relative path is used, it is relative to the current folder.

  Note

  When @CBR_EXFH_LOAD is not specified, "*entry-point-name*.dll" is treated as DLL name.

  Example 1) When the entry point name of the EXFH file system implementation is "flsys" and DLL name is "filesys.dll", you would specify:

  ```
  @CBR_EXFH_API=flsys
  @CBR_EXFH_LOAD=filesys.dll
  ```

  Example 2) When the entry point name of the EXFH file system implementation is "file" and DLL name is "file.dll", you can specify:

```
@CBR_EXFH_API=file
```

- Specifying the external file handler on a per-file basis

  The info-file is created and is allocated as follows:

```
file-identifier=file-name,EXFH,INF(info-file)
```

The info-file is a text file with the following content:

```
[EXFH]
@CBR_EXFH_API=entry-point-name
@CBR_EXFH_LOAD=DLL-name
```

entry-point-name : The entry point name of the EXFH file system implementation.

DLL-name : DLL name of the EXFH file system implementation.

Note

When @CBR_EXFH_LOAD is not specified, "*entry-point-name*.dll" is treated as DLL name.

Example) When the file name is "Afile", the info-file name is "aflsys.inf", the entry point name is "sflsys" and DLL name is "afilesys.dll", you would specify:

```
file-identifier=Afile,EXFH,INF(aflsys.inf)
```

The content of "aflsys.inf":

```
[EXFH]
@CBR_EXFH_API=aflsys
@CBR_EXFH_LOAD=afilesys.dll
```

Cautions

- The external file handler to be used must be a DLL file. (An OBJ file cannot be used, unlike Micro Focus COBOL.)
- An external file handler cannot be used from a COBOL application compiled with the Unicode (compile option RCS(UCS2)).
- When using an external file handler from a COBOL application which was compiled for multithreading, the external file handler must also support multithreading.
- When using an external file handler, FILE STATUS values which are returned from the external file handler are set up as they are in the data items defined in the FILE STATUS clause. Consequently, they may be different from the standard FILE STATUS values. Refer to "Appendix B I-O Status List"
- The START statement, with FIRST specified, for an indexed file is not supported.

6.10 Recovering Indexed Files

COBOL provides two functions for recovering unusable indexed files: the indexed file recovery function and the simple indexed file recovery function.

6.10.1 Indexed File Recovery Function (CFURCOV)

The indexed file recovery function refreshes normal sections from the beginning of the file, and outputs abnormal sections to another file. This allows you to recover an unusable indexed file (unusable because it was not closed normally).

The indexed file recovery function works in the same manner as the Recovery command of the COBOL File Utility.

If you attempt to open an unusable indexed file, the system returns an I-O status of 39 or 90.

Coding Format

```
#include "f4agfutc.h"  /* function-declaration */
       signed _int64 CFURCOV(
              char *ixdfilename,
              char *blkdatname,
              char *message);
```

ixdfilename

Specify the address of the area storing the name (character string) of the indexed file to be recovered. The character string must end with a NULL (0x00) or blank (0x20). If a file name includes a space or comma (,), the file name must be specified within double quotation marks.

blkdatname

Specify the address of the area storing the name (character string) of the file where unrecoverable records are written. The character string must end with a NULL (0x00) or blank (0x20). Specify 0 if a file for writing unrecoverable records is unnecessary. If a file name includes a space or comma (,), the file name must be specified within double quotation marks.

message

Specify the address of the area storing a message that indicates the execution result of the indexed file recovery function. Allocate the storage area to the calling source (512 bytes required). Specify 0 if a message is unnecessary.

Return Value

The code corresponding to the message is returned as the execution result of the indexed file recovery function. See "Table 6.10 Codes and messages returned by the indexed file recovery function (and simple recovery function)" for codes and messages.

Example

```
#include "f4agfutc.h"
void callcobfrcov(void)
       {
              char ixdfilename[512] = "c:\\ixdfile\0";
              char blkdatname[512] = "c:\\blkdat\0";
              char message[512];
              CFURCOV(ixdfilename,blkdatname,message);
              return;
       }
```

6.10.2 Simple Indexed File Recovery Function

The simple indexed file recovery function resets flags in an indexed file that is in an unusable state so that the file is enabled. Unlike the indexed file recovery function, the simple indexed file recovery function does not recover abnormal sections in the indexed file. Consequently, accessing the indexed file after resetting the flag can result in an error because of erroneous data.

If you attempt to open an unusable indexed file, the system returns an I-O status of 39 or 90.

Coding Format

```
#include "f4agfutc.h"  /* function-declaration */
       signed _int64 CFURCOVS(
              char *ixdfilename,
              char *message);
```

ixdfilename

Specify the address of the area storing the name (character string) of the indexed file to be recovered. The character string must end with a NULL (0x00) or blank (0x20). If a file name includes a space or comma (,), the file name must be specified within double quotation marks.

message

Specify the address of the area storing a message that indicates the execution result of the simple indexed file recovery function. Allocate the storage area to the calling source (512 bytes required). Specify 0 if a message is unnecessary.

Return Value

The code corresponding to the message is returned as the execution result of the simple indexed file recovery function. For codes and messages, see "Table 6.10 Codes and messages returned by the indexed file recovery function (and simple recovery function) ".

Example

```
#include "f4agfutc.h"
void callcobfrcovs(void)
       {
              char ixdfilename[512] = "c:\\ixdfile\0";
              char message[512];
              CFURCOVS(ixdfilename,message);
              return;
       }
```

6.10.3 Notes

When writing a C routine, include the following file to support calling COBOL from a C program:

- f4agfutc.h

For more information on compiling a C program that calls a COBOL program, refer to "9.3.4 Compiling Programs".

When linking, use the LINK command to link the file:

- f4agfutc.lib

For more information on linking a C program that calls a COBOL program, refer to "9.3.5 Linking Programs".

When you execute, set up the following environment:

- Include, in the PATH environment variable, the folder containing the following files:
    - f4agfutc.dll, f4agfuty.dll, f4agfrm.dll
- The entry information given below is required for use from a program that specifies DLOAD in a compiler option. See "5.4.2 Entry Information for Subprograms".

```
[ENTRY]
CFURCOV=F4AGFUTC.DLL
CFURCOVS=F4AGFUTC.DLL
```

6.10.4 Examples of Calling from COBOL

The following example recognizes an unusable file at OPEN time and calls the indexed file recovery function.

Example

It is assumed that a file for storing unrecoverable data and the return message are not required:

```
000010 IDENTIFICATION DIVISION.
000020  PROGRAM-ID. RECOVER.
000030 ENVIRONMENT DIVISION.
000040  CONFIGURATION SECTION.
000050  SPECIAL-NAMES.
000060     ENVIRONMENT-NAME  ENV-NAME
000070     ENVIRONMENT-VALUE ENV-VALUE.
```

```
000080  INPUT-OUTPUT SECTION.
000090   FILE-CONTROL.
000100     SELECT IXDFILE ASSIGN TO FILE1
000110       ORGANIZATION IS INDEXED
000120       RECORD KEY   IS IXDFILE-RECKEY
000130       FILE STATUS  IS FSDATA.
000140 DATA DIVISION.
000150  FILE SECTION.
000160  FD  IXDFILE.
000170  01  IXDFILE-REC.
000180      03  IXDFILE-RECKEY PIC X(8).
000190      03  IXDFILE-DATA   PIC X(20).
000200  WORKING-STORAGE SECTION.
000210  77  IXDFILE-NAME PIC X(512).
000220  77  BLKFILE-NON  PIC S9(9) COMP-5 VALUE 0.
000230  77  MESSAGE-NON  PIC S9(9) COMP-5 VALUE 0.
000240  77  FSDATA       PIC XX.
000250  77  OPEN-FSDATA  PIC XX.
000260 PROCEDURE DIVISION.
000270     OPEN I-O IXDFILE.
000280     MOVE FSDATA TO OPEN-FSDATA.
000290     CLOSE IXDFILE.
000300         EVALUATE OPEN-FSDATA
000310             WHEN "00"
000320               GO TO MAIN-PROCESS
000330             WHEN "39"
000335             WHEN "90"
000340* Fetch the file name
000350               MOVE ALL SPACE TO IXDFILE-NAME
000360               DISPLAY "FILE1" UPON ENV-NAME
000370               ACCEPT IXDFILE-NAME FROM ENV-VALUE
000380* Call the indexed file recovery function
000390               CALL "CFURCOV" USING BY REFERENCE IXDFILE-NAME
000400                                    BY VALUE BLKFILE-NON
000410                                    BY VALUE MESSAGE-NON
000420               EVALUATE TRUE
000430                   WHEN PROGRAM-STATUS <= 1
000440                       GO TO MAIN-PROCESS
000450                   WHEN OTHER
000460                       GO TO PROCESS-END
000470               END-EVALUATE
000490             WHEN OTHER
000500               GO TO MAIN-PROCESS
000510         END-EVALUATE.
000520 MAIN-PROCESS.
000530     OPEN I-O IXDFILE.
      *           :
000580     CLOSE IXDFILE.
000590 PROCESS-END.
000600     EXIT PROGRAM.
```

The following example calls the indexed file recovery function when a file for storing unrecoverable data and a message are necessary:

```
000010 IDENTIFICATION DIVISION.
000020  PROGRAM-ID. SAGYOU2.
000030 ENVIRONMENT DIVISION.
000040  CONFIGURATION SECTION.
000050  SPECIAL-NAMES.
000060     ENVIRONMENT-NAME  ENV-NAME
000070     ENVIRONMENT-VALUE ENV-VALUE.
000080  INPUT-OUTPUT SECTION.
000090   FILE-CONTROL.
000100     SELECT IXDFILE ASSIGN TO FILE1  ORGANIZATION IS INDEXED
```

```
000110       RECORD KEY   IS IXDFILE-RECKEY  FILE STATUS  IS FSDATA.
000120 DATA DIVISION.
000130  FILE SECTION.
000140  FD  IXDFILE.
000150  01  IXDFILE-REC.
000160      03  IXDFILE-RECKEY PIC X(8).
000170      03  IXDFILE-DATA   PIC X(20).
000180  WORKING-STORAGE SECTION.
000190  77  IXDFILE-NAME PIC X(512).
000200  77  BLKFILE-NAME PIC X(11) VALUE "C:\BLKNDAT ".
000210  77  MESSAGE-AREA PIC X(512).
000220  77  FSDATA       PIC XX.
000230  77  OPEN-FSDATA  PIC XX.
000240 PROCEDURE DIVISION.
000250     OPEN I-O IXDFILE.
000260     MOVE FSDATA TO OPEN-FSDATA.
000270     CLOSE IXDFILE.
000280         EVALUATE OPEN-FSDATA
000290             WHEN "00"
000300               GO TO GYOUMU-KAISHI
000310             WHEN "39"
00315              WHEN "90"
000320* Fetch the file name
000330               MOVE ALL SPACE TO IXDFILE-NAME
000340               DISPLAY "FILE1" UPON ENV-NAME
000350               ACCEPT IXDFILE-NAME FROM ENV-VALUE
000360* Call the indexed file recovery function
000370               CALL "CFURCOV" USING BY REFERENCE IXDFILE-NAME
000380                                    BY REFERENCE BLKFILE-NAME
000390                                    BY REFERENCE MESSAGE-AREA
000400               EVALUATE TRUE
000410                   WHEN PROGRAM-STATUS <= 1
000420                       GO TO GYOUMU-KAISHI
000430                   WHEN OTHER
000440                       GO TO GYOUMU-TEISHI
000450               END-EVALUATE
000470             WHEN OTHER
000480               GO TO GYOUMU-TEISHI
000490         END-EVALUATE.
000500 GYOUMU-KAISHI.
000510     OPEN I-O IXDFILE.
      *           :
000560     CLOSE IXDFILE.
000570 GYOUMU-TEISHI.
000580     EXIT PROGRAM.
```

6.10.5 Codes and Messages

The section explains the codes and massage returned by the indexed file recovery.

Table 6.10 Codes and messages returned by the indexed file recovery function (and simple recovery function)

Code(Decimal)	Message
0	n records were restored. (*1)
1	There were n records that could not be restored. (*2)
10	Recovery file not found.
11	Recovery file not an indexed file.
12	Could not access the recovery file.

Code(Decimal)	Message
13	Unrecoverable data store file exists. (*2)
14	Other process accessing the file.
15	Other process recovering the file.
16	Insufficient disk space. (*2)
128	Insufficient memory space.
131	Record in recovery file not found. (*2)
132	Contradiction in the file information.
136	Name of recovery file is wrong.
137	Name of unrecoverable data store file is wrong.

*1 The simple recovery function does not return the message.

*2 The simple recovery function does not return this code and message.

6.11 COBOL File Access Routine

The COBOL File Access Routine consists of API (Application Program Interface) functions for accessing all types of COBOL file.

This routine supports development and operation of 64-bit (x64) applications that access the COBOL files.

For details on this routine, refer to the "COBOL File Access Routine User's Guide".

Chapter 7 Printing

This chapter explains how to print data by line or form, and provides information about types of printing methods, print characters, form overlay patterns, forms control buffers (FCB), and form descriptors.

7.1 Types of Printing Methods

Use a print file or presentation file to print data from a COBOL program. This section outlines how to print data using these files, and explains print character types, form overlay patterns, forms control buffers (FCB), and form descriptors. The print function you use depends on the printer being used. Refer to the user manual for each specific printer.

Note

Data that is output using a print file must be defined in advance in a data item for display (USAGE IS DISPLAY). Data that contains an invalid binary code may not be printed normally.

7.1.1 Outline of Printing Methods

There are two types of print files, files with a FORMAT clause and those without a FORMAT clause.

Use a print file without a FORMAT clause to print data in line mode, how to overlay data line by line with a form overlay pattern, and how to print data by setting print information with an FCB.

By using a print file with a FORMAT clause, data in forms with form descriptors can be printed.

This chapter classifies print and presentation files in the following groups:

(1) Print file without a FORMAT clause (How to print data in line mode)

(2) Print file without a FORMAT clause (How to use a form overlay pattern and FCB)

(3) Print file with a FORMAT clause

(4) Presentation file

"Table 7.1 Characteristics, advantages, and uses of each printing method" lists the characteristics, advantages, and uses of printing methods (1) to (4).

Table 7.1 Characteristics, advantages, and uses of each printing method

File Type		(1)	(2)	(3)	(4)
Characteristics	Data can be printed in line mode.	B	B	B	C
	Data can be overlaid with form overlay patterns.	C	B	B	B *1
	Data in forms can be printed with form descriptors.	C	C	B	B
Advantage	Simple program coding.	A	B	B	A
	Simple forms printing.	B	A	A	A
	Existing form descriptors generated by other systems can be used.	C	C	A	A
	Various types of print information can be specified in programs.	C	A	A	B
	Forms overlay print mode can be set.	C	B	C	C
Use	Print forms.	B	A	A	A

A : Can be used (suitable).

B : Can be used.

C : Cannot be used.

*1 Can be printed only when an overlay pattern name is specified in form descriptors or is specified in a printer information file. For details, refer to the "*PowerFORM RTS online manual*" or "*FORM RTS online manual*".

The following table lists related products.

Table 7.2 Related products

Related Product	File Type			
	(1)	(2)	(3)	(4)
FORM (**1)	-	Needed (**2)	Needed	Needed
PowerFORM (**3)	-	Needed	Needed	Needed
FORM overlay option (**1)	-	Needed	Needed	Needed
PowerFORM RTS	-	-	Needed	Needed

**1 : This is a product that operates by 32 bits Windows. This cannot be installed in 64-bit support environment.

**2 : Used to create an overlay.

**3 : Included in FORM.

(1) Print File without a FORMAT Clause (How to print data in line mode)

By using a print file without a FORMAT clause, data can be sent in line mode to a printer. During output, a logical page size, line feed, and page alignment can be specified.

For details on how to use a print file when printing data in line mode, see "7.2 How to print data in line mode using"

(2) Print File without a FORMAT Clause (How to use a form overlay pattern and FCB)

With a print file, a page of output data can be overlaid with a form overlay pattern, or print control information can be specified with a FCB. When specifying that a page of output data be overlaid with a form overlay pattern, use a control record. When specifying print control information with a FCB, write a FCB control statement in a runtime initialization file.

For details on form overlay patterns, see "7.1.3 Form Overlay Patterns". For details on FCB, see "7.1.4 Forms Control Buffers(FCB)". For how to use print files, see "7.3 How to use a form overlay pattern and FCB".

(3) Print File with a FORMAT Clause

For a print file with a FORMAT clause, specify the FORMAT clause in print file definitions in a program. By using a print file with a FORMAT clause, data in forms can be printed with partitioned form descriptors. Forms can be printed with the form overlay patterns described above and FCB. FORM RTS or PowerFORM RTS is required, however, to print forms using a print file with a FORMAT clause.

For details on form descriptors, see "7.1.6 Form Descriptors". For details on print files with form descriptors, see "7.4 Using Print Files with a FORMAT clause".

(4) Presentation File

With a presentation file, data in forms defined in form descriptors can be printed. It differs from the print file with a FORMAT clause in that this file can use form descriptors that are not in partition format. However, the form overlay pattern and FCB control changed from the program and the line record cannot be output.

Before using a presentation file, the file must be defined and a chart record must be created with a WRITE statement.

When a chart record is output, forms in the format defined in the form descriptors are printed. The attributes of output data defined in the form descriptors can then be changed.

The "FORM RTS" is required for document printing with the presentation file.

7.1.2 Print Characters

Specify the print attributes of print characters (such as size, font, style, form, direction and space) in the CHARACTER TYPE clause of data description entry. The MODE-n, mnemonic-name, and print mode name can be specified in the CHARACTER TYPE clause. The following lists the print attributes available in each form.

The CHARACTER TYPE clause is required for printing of national language entries.

Option	Attribute of print characters					
	Size	Font	Style	Form	Direction	Space
CHARACTER TYPE MODE-n	O	-	-	O *2	-	O *3
CHARACTER TYPE mnemonic-name	O	O	-	O	O	O *3
CHARACTER TYPE print mode name	O	O	O *1	O	O	O

*1 Specify the "FONT-nnn" in FONT of PRINTING MODE clause.

*2 Specify the "BY mnemonic-name" after "MODE-n."

*3 Determined depending on the print character size and form.

- If the MODE-1, MODE-2 or MODE-3 is specified in the CHARACTER TYPE clause, the size of their print characters us set to 12 points, 9 points or 7 points, respectively.

- If a mnemonic-name is specified in the CHARACTER TYPE clause, it can be printed with the print attributes indicated by the function name that has been associated with the mnemonic-name by the function-name clause of SPECIAL-NAMES paragraph. For function name details, see the "CHARACTER TYPE clause" of "NetCOBOL Language Reference."

- When specifying a print mode name in the CHARACTER TYPE clause, define the print attributes by associating them with the print mode name in the PRINTING MODE clause of SPECIAL-NAMES paragraph. The characters can be printed out with the defined print attributes. For details about how to write a PRINTING MODE clause, see the "PRINTING MODE clause" of "NetCOBOL Language Reference." The following explains the available print attributes.

Available sizes

3.0 to 300.0 points

Method

The following explains how to specify the character size. However, if a device font having a fixed size is selected, the characters are printed out in the points (character size) of the font installed on the printer.

For the specification details, see the "NetCOBOL Language Reference."

Specification	Character size
MODE-1/MODE-2/MODE-3	12, 9 or 7 points
Function name associated with mnemonic-name	12, 9 or 7 points
Print mode name (If SIZE is specified in PRINTING MODE clause)	3.0 to 300.0 points (in units of 0.1 point) If the character size is omitted, the characters are printed out according to the setup of character space. If both the character size and space are omitted, the following defaults are used. - National data item : Printed in 12-point character size. - Alphanumeric data item : Printed in the specified size if the "5.4.1.35 @CBR_PrinterANK_Size(Specification of ANK character size)" environment variable information is valid(*). If not, the characters are printed in 7.0-point size.

* : Applies to items that do not use the CHARACTER TYPE clause and the PRINTING POSITION clause.

Note

- Specify the font with the environment variable information @PrinterFontName. If the environment variable information or the font is not specified, CourierNew is normally selected. Refer to "5.4.1.60 @PrinterFontName(Set the Font Used for Print Files)".
- If an outline font or a TrueType font is specified for environment information, the print format is a scaleable font and data can be printed with 3.0 to 300.0 points in units of 0.1 point.

Available Fonts

Font's keywords used are MINCHOU, MINCHOU-HANKAKU, GOTHIC, GOTHIC-HANKAKU, GOTHIC-DP, and FONT-NUMBER.

Method

Specification	Font
MODE-1/MODE-2/MODE-3	MINCHOU
Function name associated with mnemonic-name	MINCHOU/ GOTHIC The default font is MINCHOU.
Print mode name (If FONT is specified in PRINTING MODE clause)	MINCHOU/ MINCHOU-HANKAKU/ GOTHIC/ GOTHIC-HANKAKU/, GOTHIC-DP/FONT-NUMBER The default font for alphanumeric data items is GOTHIC, and the default for national data items is MINCHOU.

For the specification details, see the "NetCOBOL Language Reference."

Print Rules

When FONT-NUMBER is specified, it is printed in the font of the font face name associated with each FONT-NUMBER in the font table specified in environment variable @CBR_PrintFontTable. The FONT-NUMBER indicates "FONT-nnn" specified for the PRINTING MODE clause.

When font table name is not specified, or the font face name associated with FONT-NUMBER in the font table is not specified, all are printed by MINCHOU typeface (usual style of type) regardless of the value of FONT-NUMBER.

Refer to the following for details and the specification method of the font table.

- "5.4.1.36 @CBR_PrintFontTable(Specify the font table used for a print file)"
- "5.4.1.68 File identifier(Specify the printer information file and various parameters used for the program)"
- "5.4.1.69 File identifier(Specify the printer and various parameters used for the program)"
- "7.1.14 Font table"

If you use a print file without a FORMAT clause, and the font name is set in the environment variable @PrinterFontName, data is printed using the specified font. If the font name is omitted, CourierNew is used as the default.

Refer to "5.4.1.60 @PrinterFontName(Set the Font Used for Print Files)" for more information.

For a print file with a FORMAT clause and presentation file, refer to the "*FORM RTS online manual*" or "*PowerFORM RTS online manual*" (pformapi.hlp).

Note

The data is printed with a fixed pitch and each character is left-aligned at each character position. When the proportional font with different width of each character is used, the print result may seem to have a space after a narrow character. Thus, it is recommended that the font having a single constant width of the character is used in the print file function.

Available Style

Styles used are REGULAR, BOLD, ITALIC, and BOLD-ITALIC. The style can be specified for the font in which FONT-NUMBER is specified. The default font is REGULAR.

Method

For the specification, see the "7.1.14 Font table".

Note

- When fonts other than the outline font are selected, it is printed in the font by the style of the font regardless of the specification of the style.
- It is necessary to specify FONT-NUMBER for the font to specify the character style in the font table. When the font name is specified for the PRINTING MODE clause, all styles become regulars.

Available Print Character Forms

Print character forms are em-size and em-size tall and wide; double size, en-size, and en-size tall and wide; and double size.

Method

Specification	Font
Function name associated with mnemonic-name of MODE-1/MODE-2/MODE-3	em-size tall and wide; double size, en-size, and en-size double size The default form is em-size.
Function name associated with mnemonic-name	em-size tall and wide; double size, en-size The default form is em-size.
Print mode name (If FORM is specified in PRINTING MODE clause)	em-size tall and wide; double size, em-size, en-size, and en-size tall and wide; and double size The default form is em-size.

For the specification details, see the "NetCOBOL Language Reference."

Direction of Print Characters

Characters are printed horizontally.

Print Character Spaces

Table 8.3 lists spaces determined according to the sizes and forms of print characters.

Method

Specification	Font
MODE-1/MODE-2/MODE-3	"Table 7.3 Relationship between print character sizes/forms and character spaces" lists spaces determined according to the sizes and forms of print characters.
Function name associated with mnemonic-name	
Print mode name (If PITCH is specified in PRINTING MODE clause)	Character spaces of 0.01 to 24.00 cpi are specified in units of 0.01 cpi. The default character space for alphanumeric data items is 10.0 cpi, and the default for national data items is 6.0 cpi.

For the specification details, see the "NetCOBOL Language Reference."

Table 7.3 Relationship between print character sizes/forms and character spaces

Character Size	Character Form (Unit: cpi)							
	Em-size	En-size	Tall char	En-size tall char	Wide size char	En-size wide char	Double size	En-double size
MODE-1 (12 point)	5	10	5	-	2.5	-	2.5	5
MODE-2 (9 point)	8	16	8	-	4	-	4	8
MODE-3 (7 point)	10	-	10	-	5	-	5	-
A (9 point)	5	10	5	10	2.5	5	2.5	5
B (9 point)	20/3	40/3	20/3	40/3	10/3	20/3	10/3	20/3
X-12P (12 point)	5	10	5	10	2.5	5	2.5	5
X-9P (9 point)	8	16	8	16	4	8	4	8
X-7P (7 point)	10	20	10	20	5	10	5	10
C (9 point)	7.5	15	7.5	15	3.75	7.5	3.75	7.5
D-12P (12 point)	6	12	6	12	3	6	3	6
D-9P (9 point)	6	12	6	12	3	6	3	6

For the meanings of the codes for character sizes in this table, refer to the "NetCOBOL Language Reference."

7.1.3 Form Overlay Patterns

A form overlay pattern is used to set fixed sections in forms in advance, such as ruled lines and headers. Forms can be printed by overlaying a page of output data with a form overlay pattern.

A form overlay pattern is generated from a screen image by FORM or PowerFORM. A form overlay pattern can be shared with multiple programs, and form overlay patterns generated by other systems can also be used.

For more information about how to generate form overlay patterns, refer to "*FORM manual*" and" FORM on-line help", or "PowerFORM Getting Started" and the "PowerFORM on-line help". For printing forms when using form overlay patterns, see "7.3 How to use a form overlay pattern and FCB"

7.1.4 Forms Control Buffers(FCB)

A forms control buffer (FCB) defines the number of lines on one page, line space, and column start position. By using an FCB, the number of lines on one page, line space, and column start position can be changed.

When an FCB name is specified with an I control record or when the default FCB name is specified in the initialization file (@DefaultFCB_Name=FCBxxxx), FCB information can be specified externally. Specify the FCB information in the initialization file as follows:

```
FCBxxxx=FCB-control-statement
```

xxxx is the FCB name specified in the I control record or FCB name (part of xxxx) specified for execution environment information @ DefaultFCB_Name=FCBxxxx.

The syntax of an FCB control statement is:

```
[LPI ((Line-space [,number-of-lines])[,(Line-space [,number-of-lines])]...)]
     [,CHm (Line-number [,line-number]...)[,CHm (Line-number [,line-number]...)]...]
           ⎧ SIZE(    ⎧ Page-length  ⎫ )                              ⎫
           ⎪          ⎩ 110          ⎭                                ⎪
           ⎪          ⎧ A3    ⎫                                       ⎪
    [,    <           ⎪ A4    ⎪                                        >    ]
           ⎪ FORM(   <  B4     >  [,    ⎧ PORT  ⎫ ])                  ⎪
           ⎪          ⎪ A5    ⎪         ⎩ LAND  ⎭                     ⎪
           ⎩          ⎪ B5    ⎪                                       ⎭
                      ⎩ LT
```

LPI information

Lines per inch can be specified from the top of the paper sequentially in a format of (line space, number of lines). LPI units of 6, 8 and 12 can be specified for the line spacing.

CH information

Line number can be specified to CHANNEL-01 through 12. Normally CHANNEL-01 is used for identifying the first printable line.

SIZE information

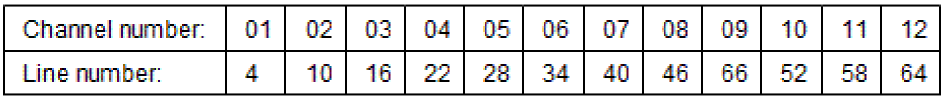
Specify the longitudinal size of the form in units of 1/10 inch using the SIZE operand. Specifiable values are 35 through 159 (3.5 inches through 15.9 inches). The default is 110 (11 inches).

FORM information

For the FORM operand, specify a standard size. With standard sizes, the longitudinal size is uniquely determined by the orientation of the form: PORT (Portrait: in longitudinal direction) or LAND (Landscape: in horizontal direction).

The default values of the I control record FCB and the default FCB are:

- 6 lpi x 11 inches (66 lines)



Channel number:	01	02	03	04	05	06	07	08	09	10	11	12
Line number:	4	10	16	22	28	34	40	46	66	52	58	64

If the default value or the Form size are mismatched, it is necessary to specify an FCB which matches the Form size of the environment variable (@DefaultFCB_Name) or I control record.

Refer to runtime environment variable "5.4.1.51 @DefaultFCB_Name(Specification of default FCB name)" for details on specifying the default FCB name.

Refer to "7.3 How to use a form overlay pattern and FCB" for details on specifying I control records.

FCB Necessities

FCB is information required for the COBOL runtime system to perform page control and line control according to the physical form size.

The COBOL runtime system automatically determines where in a page the print data output based on a print request (ADVANCING clause of the WRITE statement) by the COBOL program should be printed (page control). The system also automatically determines whether form feeds are required (line control).

For instance, suppose the user performs forms printing as follows:

- Physical form size: A4
- Paper orientation: Portrait
- Top of form: Line 1
- Line spacing: 6 lines per inch

Specify an FCB control statement as follows:

```
LPI ((6)), CH1 (1), FORM (A4, PORT)
```

The COBOL runtime system analyzes this FCB control statement and performs the following in accordance with the request by COBOL WRITE statement with the ADVANCING specification:

- When ADVANCING PAGE (forms feed) is requested by a WRITE statement, the system feeds one form and points to line 1— the top of form specified by the CH1 operand.
- When ADVANCING n LINES is requested by a WRITE statement, the system calculates the number of lines based on 6 lines per inch —the line spacing specified by the LPI operand, and advances as many lines as specified.
- Based on the information specified by the FORM operand, the system calculates the number of lines that can be printed on a form when an A4 form is used in portrait mode. Therefore, even if ADVANCING PAGE (forms feed request) is not specified explicitly in the COBOL WRITE statement, the system automatically feeds a form in response to a request to print data exceeding the number of lines that can be printed on a form.

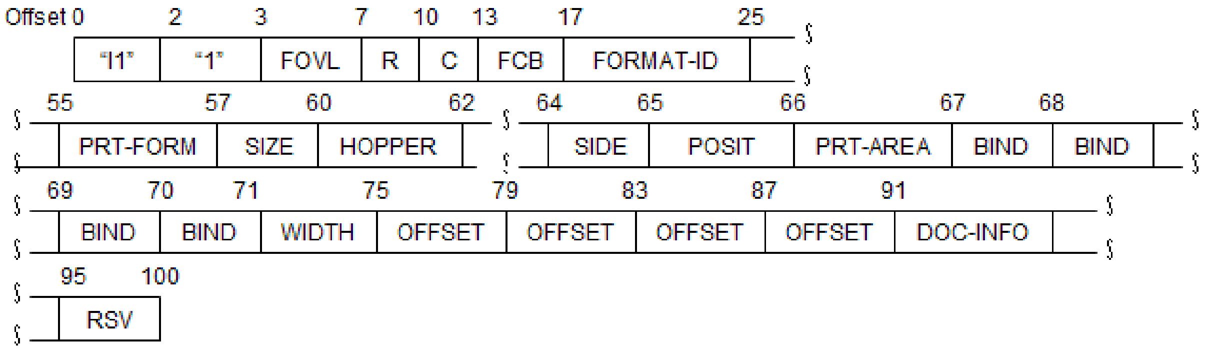
Thus, to allow a COBOL program to do forms printing, the FCB corresponding to the physical form size actually used must be specified. Note that if the specified FCB does not match the physical form size, print results may not be obtained for the intended design.

7.1.5 I and S control records

For the detailed information about the combination of set values in each record area, field explanation, and the areas not covered in this manual, see the "NetCOBOL Language Reference." Also, see the manuals of each I control records printer as the available functions may vary depending on the printer used.

I control records

The following shows a format of I control records.



```
01 I-control-record.
   03 Identifier                PIC X(2) VALUE "I1".
   03 Form                      PIC X(1) VALUE "1".
   03 Overlay-name              PIC X(4).           *> FOVL
   03 Overlay-repeat-count      PIC 9(3).           *> R
```

```
03 Copy-count                   PIC 9(3).                  *> C
03 FCB-name                     PIC X(4).                  *> FCB
03 Screen-form-descriptor-name  PIC X(8).                  *> FORMAT-ID
03                              PIC X(30).
03 Print-form                   PIC X(2).                  *> PRT-FORM
03 Form-size                    PIC X(3).                  *> SIZE
03 Form-free-port               PIC X(2).                  *> HOPPER
03                              PIC X(2).
03 Print-face                   PIC X.                     *> SIDE
03 Print-positioning            PIC X.                     *> POSIT
03 Print-inhibit-area           PIC X.                     *> PRT-AREA
03 Binding-margin-direction.                               *> BIND
   04                              PIC X OCCURS 4 TIMES.
03 Binding-margin-width         PIC 9(4).                  *> WIDTH
03 Print-origin.
   04                              PIC 9(4) OCCURS 4 TIMES. *> OFFSET
03 Document-name-ID-information PIC X(4).                   *> DOC-INFO
03                              PIC X(5) VALUE SPACE.       *> RSV
```

A phrase of "specification during file open" means the system print environment to be used when a file is opened.

If the FORMAT clause has been specified for a print file, the set value of form descriptor is made valid. If the form descriptor is omitted, the information stored in the printer information file is made valid. For details of printer information file, see the "*MeFt online manual*."

FOVL (Form overlay pattern name)

Specifies a form overlay pattern name to be used. If an overlay group is specified, the first overlay pattern is processed with a single overlay. If this field is left blank, the specification during file open is made valid.

R (Form overlay pattern repeat count)

Specifies the number of repeat times of form overlay pattern (0 to 255). The system considers it to be the same as the copy count.

C (Copy count)

Specifies the number of copies of each page (0 to 255). If zero (0) is specified, the specification during file open is made valid.

Note

The copy count setup is made invalid for dual-side printing. Value one (1) is assumed. This option is valid only when the printer driver supports the copy function.

FCB (FCB name)

Specifies an FCB name. If this field is left blank, the FCB name specified in the "@DefaultFCB_Name" environment variable information is made valid. For the specification of environment variable information, see the "5.4.1.51 @DefaultFCB_Name(Specification of default FCB name)". If the default FCB name is omitted, the default information of COBOL runtime system is made valid. For the default information details, see "7.1.4 Forms Control Buffers(FCB)". The FCB name cannot be specified together with the form descriptor name.

FORMAT-ID (Form descriptor name)

Specifies a form descriptor name to which the information of I control record applies. The fixed format page is created by this setup. If this field is left blank, an undefined format page is created. The FORMAT-ID cannot be specified together with the FCB name.

Note

The form descriptor specification is valid only for a print file with the FORMAT clause specified. For its application details, see the "7.4.2 Program Specifications".

PRT-FORM (Print form)

Specifies a print form. The following lists the available values.

P(Portrait mode), L(Landscape mode), LP(Line printer mode), PZ (Compressed portrait mode), or LZ (Compressed landscape mode)

If this field is left blank, the SIZE form size must also be left blank. In such case, the specification during file open is made valid.

Notes on print files without FORMAT clause:

The PZ and LZ options are valid for the printers and printer drivers that support the 80% compression printing. If not supported, the PZ and LZ options are ignored and the P or L option is assumed. If the LP option is specified, the forms designed in the continuous form (or stock form) size are printed in the A4 form size with the compressed character and line spaces. In this case, however, the character size is not compressed. Some adjacent characters may be overlapped to each other.

The form overlay is not compressed if the PZ, LZ or LP is specified.

SIZE (Form size)

Specifies a form size. The following lists the available form size.

A3, A4, A5, B4, B5, LTR or form-name

If this field is left blank, the PRT-FORM (print form) must also be left blank. In such case, the specification during file open is made valid.

If the FORMAT clause is omitted for the print file, a character string consisting of up to three characters can be specified in addition to the above form size. Specify the character string by associating the "@PRN_FormName_xxx" environment variable information with the form name when changing the above form size dynamically. For the information about how to specify the environment variable information, see the "5.4.1.61 @PRN_FormName_xxx(Specify paper name)".

Note

A character string consisting of up to three characters cannot be specified for the print file with the FORMAT clause. When specifying all size of forms including stock forms using the print file with FORMAT clause, keep this field blank and specify the actual form size using the printer information file. For details, see the "*MeFt online manual*."

HOPPER (Form supply port)

Specify a hopper. The following lists the available values.

P1 (Main hopper 1), P2 (Main hopper 2), S (Subhopper), P (Any hopper)

If this field is left blank, the specification during file open is made valid.

Note

As the hopper is selected by the system automatically depending on the form size specified, the hopper cannot be changed by the I control codes from an application.

SIDE (Print side)

Specifies a print side. The following lists the available values.

F (Single-side printing) or B (Dual-side printing)

If this field is left blank, the specification during file open is made valid.

Note

If the dual-side printing is specified, the copy count setup is made invalid. Value one (1) is assumed.

POSIT (Print positioning)

Specifies a start print side during dual-side printing. The following lists the available values.

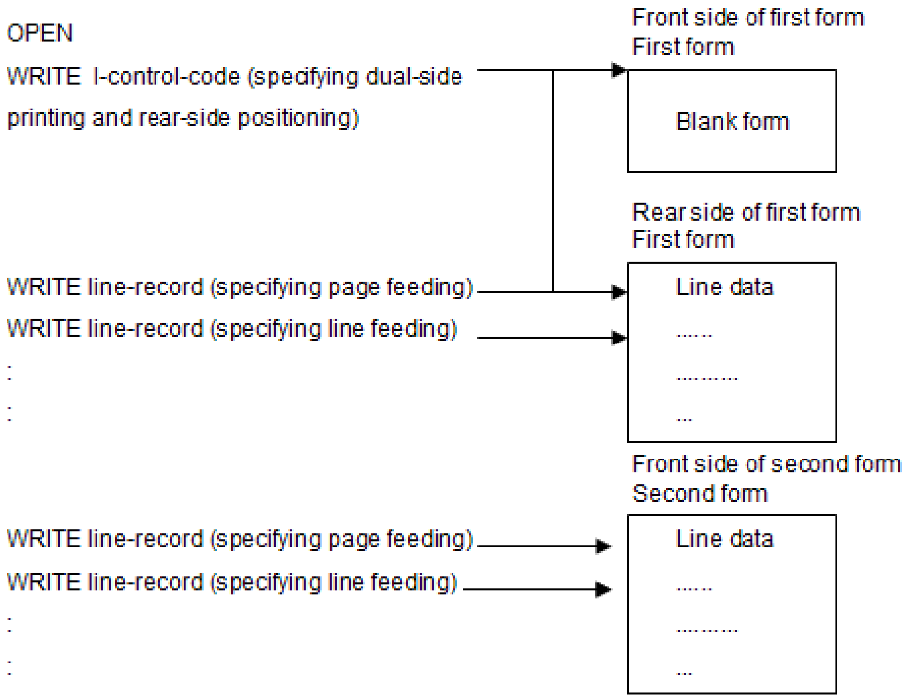
F (Front-side printing) or B (Back-side printing)

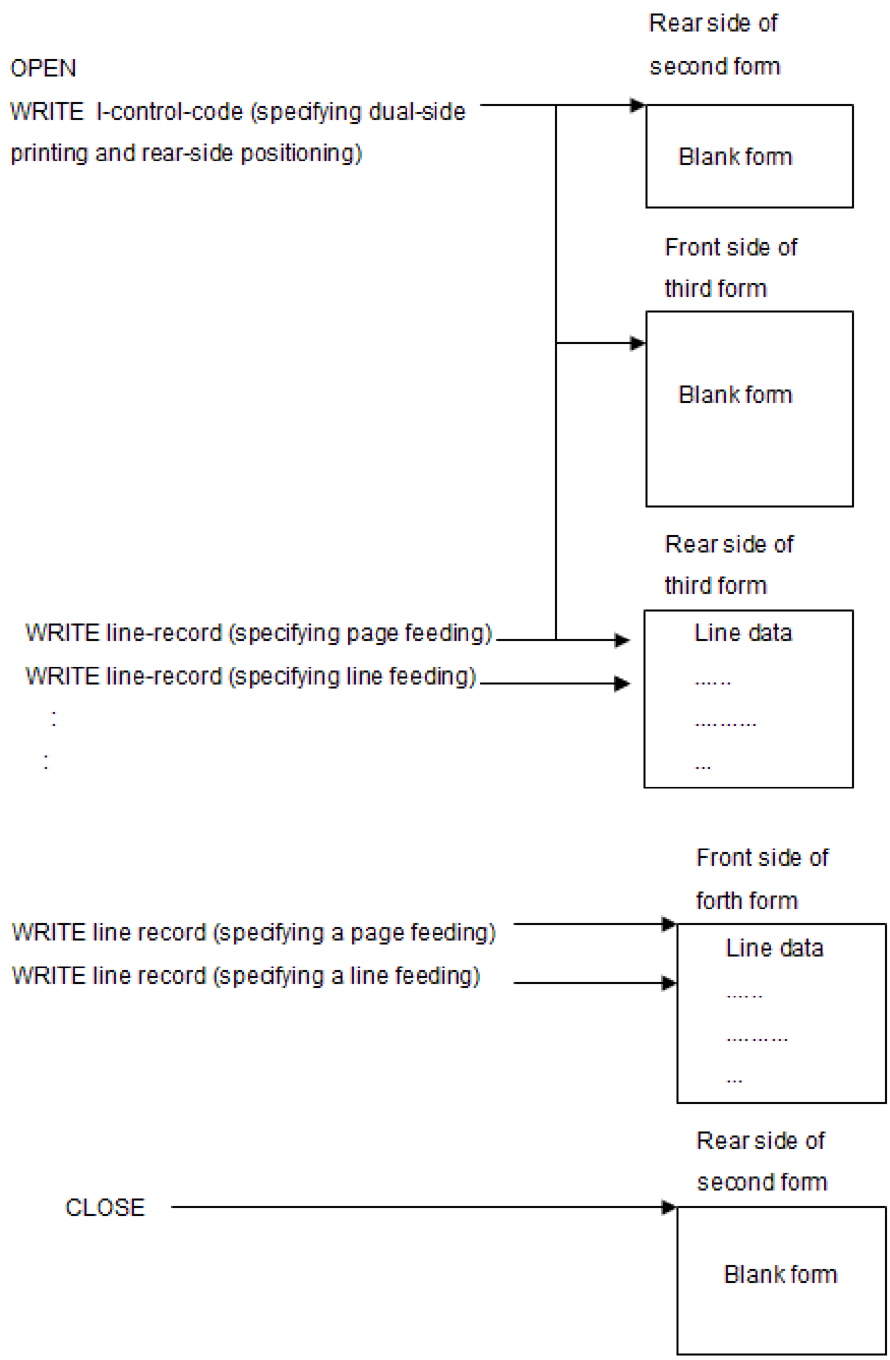
If this field is left blank, the front-side printing is assumed.

Note

- If a blank page printing is supported by the printer or printer driver and if the "Do not print blank forms" option is selected during this setup, control of POSIT print positioning function may fail. When using the POSIT function, always select the "Print blank forms" option.
- If the I control codes are output, the present print job ends and a new print job starts. Therefore, the present form is output and printing starts on a new form regardless of the front or back side printing immediately before output of I control codes.

Example


OPEN
WRITE I-control-code (specifying dual-side printing and rear-side positioning)
WRITE line-record (specifying page feeding)
WRITE line-record (specifying line feeding)
:
:
WRITE line-record (specifying page feeding)
WRITE line-record (specifying line feeding)
:
:
Front side of first form
First form
Blank form
Rear side of first form
First form
Line data
......
....
...
Front side of second form
Second form
Line data
......
....
...


Rear side of
second form
OPEN
WRITE I-control-code (specifying dual-side
printing and rear-side positioning)
Blank form
Front side of
third form
Blank form
Rear side of
third form
WRITE line-record (specifying page feeding)
WRITE line-record (specifying line feeding)
:
:
Line data
......
....
...
Front side of
forth form
WRITE line record (specifying a page feeding)
WRITE line record (specifying a line feeding)
Line data
......
....
...
Rear side of
second form
CLOSE
Blank form

PRT-AREA (Print inhibit area)

Specify to inhibit printing (L) or not (N). If this field is left blank, the specification during file open is made valid.

This option is ignored if a form descriptor is used.

BIND (Binding margin direction)

Specifies a binding margin direction when multiple continuous output pages are bound. As the binding margin direction is meaningful for multiple page output, the direction setup of the previous page is inherited if this field is left blank.

Notes:

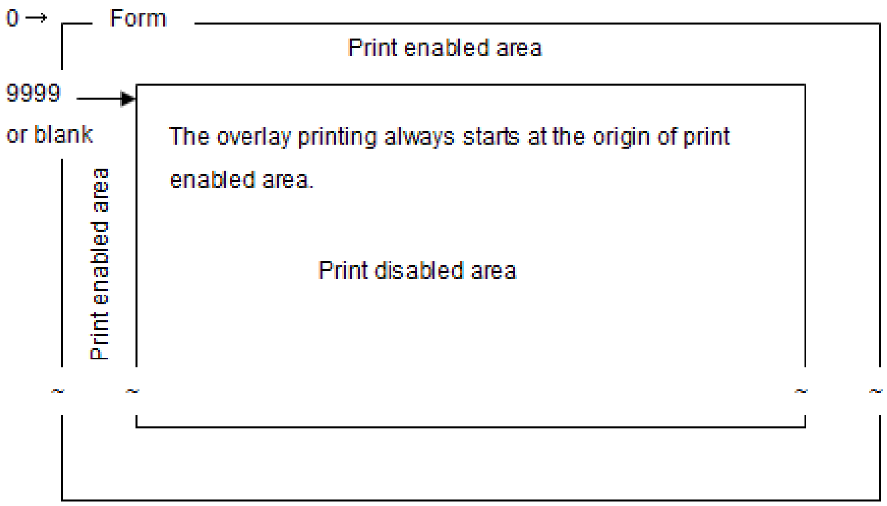
The binding margin direction of either L (Left) margin binding or U (Up) margin binding is valid. The R (Right) or D (Down) margin binding is ignored. The R and D margin binding must be considered by the application.

WIDTH (Binding margin width)

Specifies a binding margin width within a range of 0 to 9999 (in units of 1/1440 inch).

If the print file without FORMAT clause is used, the valid range depends on the offset of "@CBR_OverlayPrintOffset" environment variable information. If this information is set to VALID, the offset is valid for the form overlay and line records. If this information is set to INVALID or omitted, the offset is valid for line records only. In such case, the form overlay pattern is made invalid. (The line records and overlay are not combined correctly, and they may be shifted during printing.)

If the WIDTH is set to zero (0), printing starts at the left edge of the form. It may be in the print disabled area and some print data may be lost. Set the WIDTH to 9999 or blank to start printing at the left edge of print enabled area. For details, see the "7.1.13 Unprintable Areas".



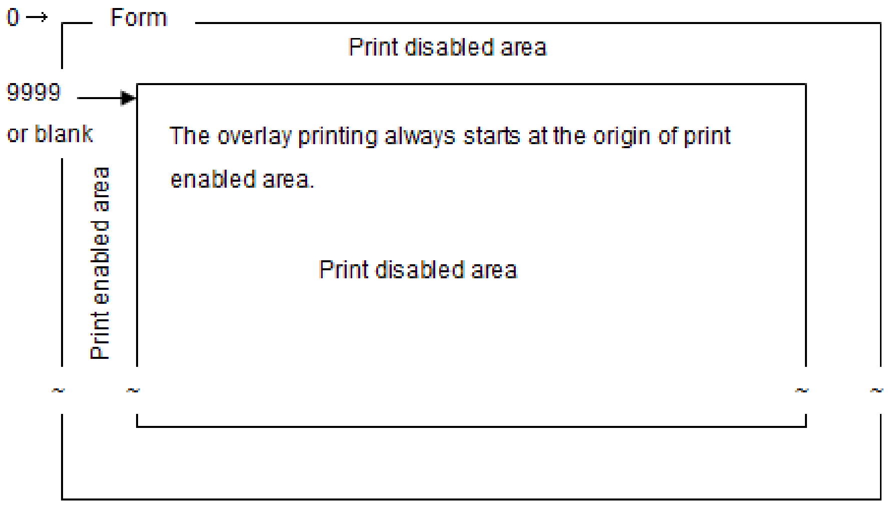
If the print file with FORMAT clause is used and if this field is set to 9999 or blank, the specification during file open is made valid.

OFFSET (Print origin)

Specifies the print origin within a range of 0 to 9999 (in units of 1/1440 inch).

If the print file without FORMAT clause is used, the valid range depends on the "@CBR_OverlayPrintOffset" environment variable information. If the information is set to VALID, the print origin is valid for the form overlay and line records. If it is set to INVALID or omitted, the print origin is valid for line records only. In such case, the origin is invalid in the form overlay pattern. (The line records and overlay are not combined correctly, and they may be shifted during printing.)

If the OFFSET is set to zero (0), printing starts at the left edge of the form. It may be in the print disabled area and some print data may be lost. Set the OFFSET to 9999 or blank to start printing at the left edge of print enabled area. For details, see "7.1.13 Unprintable Areas".



If the print file with FORMAT clause is used and if this field is set to 9999 or blank, the specification during file open is made valid. The OFFSET and PRT-FORM (Print form) can be specified simultaneously. However, if the LP (Line printer mode) is specified, the OFFSET is ignored. In the compressed print mode, the OFFSET must be specified in the length before being compressed.

DOC-INFO (Document name ID information)

Specifies the document name identification information. It can be a character string consisting of up to four alphanumeric characters. This value must be set in the "@CBR_DocumentName_xxxx" environment variable information during execution. This information name must be associated with the document name consisting of up to 128 bytes of alphanumeric, Japanese Kana and Kanji characters. For the format of environment variable information, see the "5.4.1.61 @PRN_FormName_xxx(Specify paper name)".

Note

The DOC-INFO field cannot be specified for the print file with FORMAT clause. When displaying a document name using the print manager, set this field to blank and specify the document name in the printer information file. For details, see the "*MeFt online manual*."

RSV (System reserved area)

This is the system reserved area. Keep the RSV blank.

S control records

The following shows a format of S control records.

Offset()	2	3	6	9	13	17	
"S1"	"0"	RSV	N	FOVL-1	FOVL-2		FOVL-n

```
 01  S-control-record.
       03  Identifier                   PIC  X(2)  VALUE  "S1".
       03  Form                         PIC  X(1)  VALUE  "0".
       03              r                PIC  X(3)  VALUE  SPACE.    *> RSV
       03  Overlay-count                PIC  9(3).                  *> N
       03  Overlay-name-1               PIC  X(4).                  *> FOVL-1
*>          :
       03  Overlay-name-n               PIC  X(4).                  *> FOVL-n
```

RSV (System reserved area)

This is the system reserved area. Keep the RSV blank.

N (Number of form overlay pattern names)

Specifies a number of form overlay pattern names.

FOVL-n (Form overlay pattern name)

Specifies a form overlay pattern name. However, the system uses the first form overlay pattern name only.

Valid range of control records

The I control record is valid between the time when it is output and when the next I control record is output. However, the form overlay pattern name specified by the I control record is valid until the next I control record or S control record is output.

The S control record is valid between the time when the record is output and when the next S control record or I control record is output.

Note

If an incorrect value is specified in a control record field, the program execution is interrupted and processing is terminated regardless of the FILE STATUS clause or error procedure setup.

7.1.6 Form Descriptors

When forms are designed with FORM or Power FORM, form descriptors are generated. COBOL enables data items defined in form descriptors to be included in a program so that forms can be printed with values set in the data items. Form overlay patterns can be included in form descriptors.

The FORM RTS or PowerFORM RTS is required when printing forms using form descriptors. FORM RTS or PowerFORM RTS requires a printer information file. For details of printer information files, refer to the "FORM RTS on-line help" or "*PowerFORM RTS online manual*" (pformapi.hlp).

For information on how to generate form descriptors, refer to the "*FORM Manual*" and "FORM on-line help" or "PowerFORM Getting Started" and "PowerFORM on-line help". For information about printing forms by using form descriptors see topics "7.4 Using Print Files with a FORMAT clause" or "7.5 Using Presentation Files (Printing Forms)".

Note

A file name except for the extension of file descriptor used in NetCOBOL must be an alphanumeric character string consisting of up to eight characters and beginning with an alphabetic character.

7.1.7 Special Registers

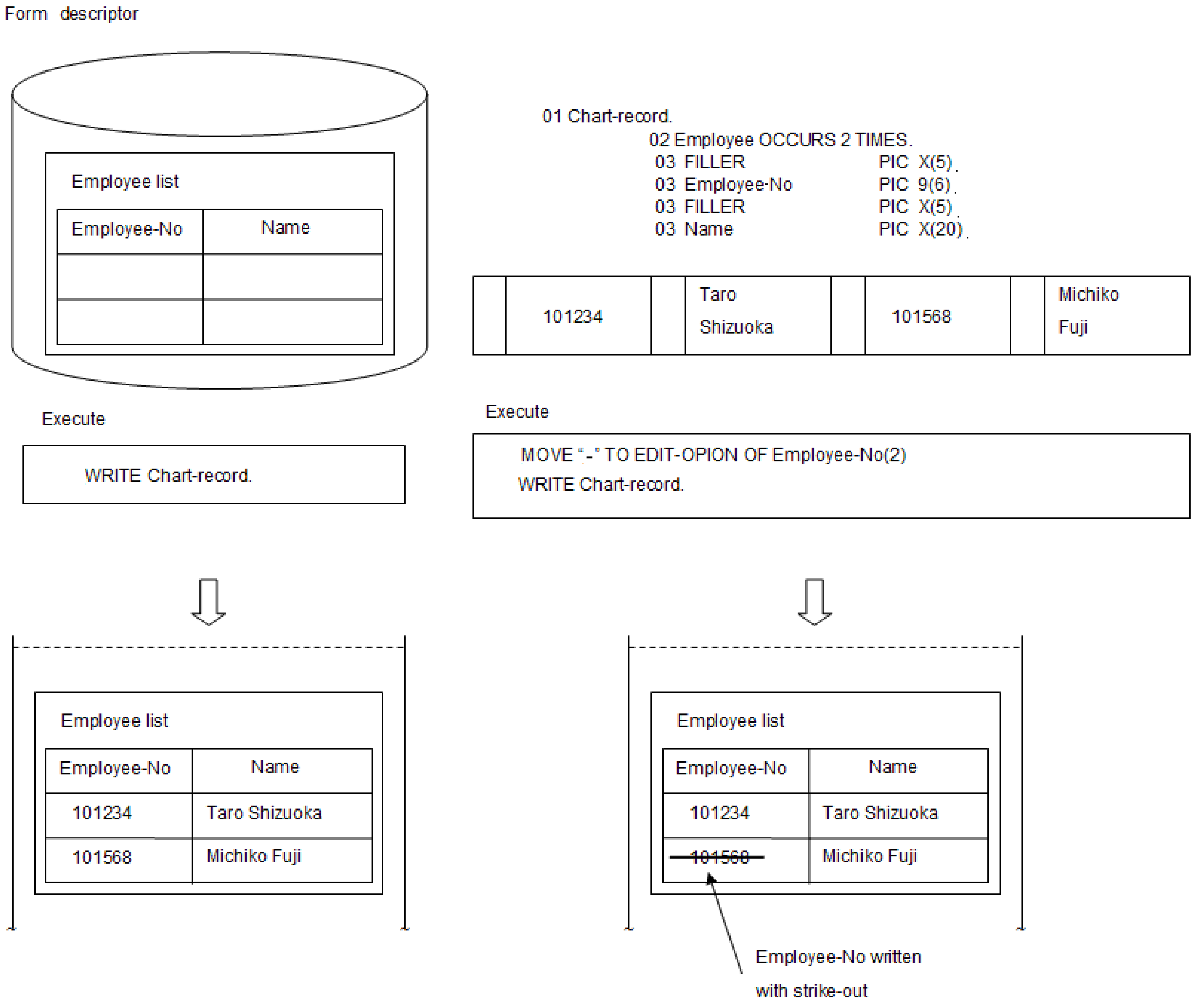
You can modify output data attributes for print files with the FORMAT clause, and presentation files, by using COBOL special registers.

The special registers for form functions are:

- EDIT-MODE: Specifies whether to use output processing or not.
- EDIT-OPTION: Specifies the bold/italic/underline attributes.
- EDIT-COLOR: Specifies colors.

You use these special registers by qualifying them with the data name defined in the form descriptor. For example, to set the color attribute of data name A, use "EDIT-COLOR OF A." For the values that you can set in these special registers, refer to the "*FORM RTS Online Manual*" or "*PowerFORM RTS Online Manual*" (pformapi.hlp).

Figure 7.1 Editing output by using the EDIT-OPTION special register



```
01 Chart-record.
   02 Employee OCCURS 2 TIMES.
    03 FILLER          PIC X(5).
    03 Employee-No     PIC 9(6).
    03 FILLER          PIC X(5).
    03 Name            PIC X(20).
```

```
WRITE Chart-record.
```

```
MOVE "-" TO EDIT-OPION OF Employee-No(2)
WRITE Chart-record.
```

Note

Form descriptors specified as having no item control fields are not able to use the special registers. Note that you cannot use form descriptors with item control fields and form descriptors without item control fields in the same program.

7.1.8 Arrangement Coordinates of Print Characters

Printer resolutions are usually expressed in dots per inch so line and column of character positions need to match the printer's (virtual) lines of columns dots. The system uses one of two methods to calculate the print coordinates characters lines and columns:

Dividing the print resolution (DPI) by the line or character spacing (LPI and CPI). When the resolution cannot be divided by using line spacing (LPI) or character spacing (CPI), the remaining dot is rounded down.

Similar to method 1, but the print position is corrected for rounding effects each print inch. This is done when the resolution cannot be divided using the line space (LPI) and character space (CPI).

For print files without a FORMAT clause you can specify which method should be used in the @CBR_PrintTextPosition environment variable.

Refer to "5.4.1.38 @CBR_PrintTextPosition(Specify method of calculating character arrangement coordinates)", for more details on the two methods of calculating the coordinates of print characters.

7.1.9 Print Information File

For print files without a FORMAT clause you can configure certain print information in a text file called the print information file.

The print information file contains alignment and document name information in the following format:

Format and set information

[PrintInformation]	<- Section name (fixed, and required)

TextAlign= { TOP
BOTTOM }
	<- Specify the upper/lower margin
	justification(may be omitted)

DocumentName= document-name	<- Specify the document name (may be omitted)

OverlayPrintOffset= { VALID
INVALID }
	<- Validate or invalidate the overlay
	offset move function (may be omitted)

PRTOUT= { LPTn:
COMn:
PRTNAME:printer-name }
	<- Specifies an output printer name.
	(may be omitted)

PAPERSIZE=paper-size-value	<- Value of Paper size. (may be omitted)
PRTFORM=print-format-value	<- Value of Print format. (may be omitted)
FOVLDIR=Foldrname-of-Overlay-file	<- Folder name in which Form overlay pattern file is stored. (may be omitted)
FOVLTYPE=Format-of-Overlay-file	<- File name of Form overlay pattern file. (may be omitted)
FOVLNAME=Overlay-file-name	<- File name of Form overlay pattern file. (may be omitted)
OVD-SUFFIX=Overlay-file-name(extension)	<- Suffix name of Form overlay pattern file. (may be omitted)

Section name

Section name "[PrintInformation]" identifies the print information file and must be specified.

TextAlign

When the line height is greater than the height of the printed characters you can specify whether the characters are aligned to the top or the bottom of the line. This instruction has priority over the specification of the environment variable information @CBR_TextAlign. Refer to the topic "5.4.1.44 @CBR_TextAlign(Specify alignment of print characters with either top or bottom of line)".

DocumentName

You can specify a document name to be displayed by the Windows Print Manager. The document name should be a combination of alphabet/numeral/kana/Japanese characters not exceeding 128 bytes. This specification will be disregarded if the environment variable information @CBR_DocumentName_xxxx is validly specified. Refer to "5.4.1.19 @CBR_DocumentName_xxxx(Specify I control record document name)".

OverlayPrintOffset

For a print file without a FORMAT clause, specify whether to validate (VALID) or invalidate (INVALID) the functions specified in the I control record, for forms overlay printing as well. The functions include the binding margin direction (BIND), binding margin width (WIDTH), and origin of printing (OFFSET). If this option is omitted, INVALID is used. This option takes precedence over the information specified in the environment variable @CBR_OverlayPrintOffset. Refer to the topic "5.4.1.34 @CBR_OverlayPrintOffset(I-Control record effects on overlay printing)".

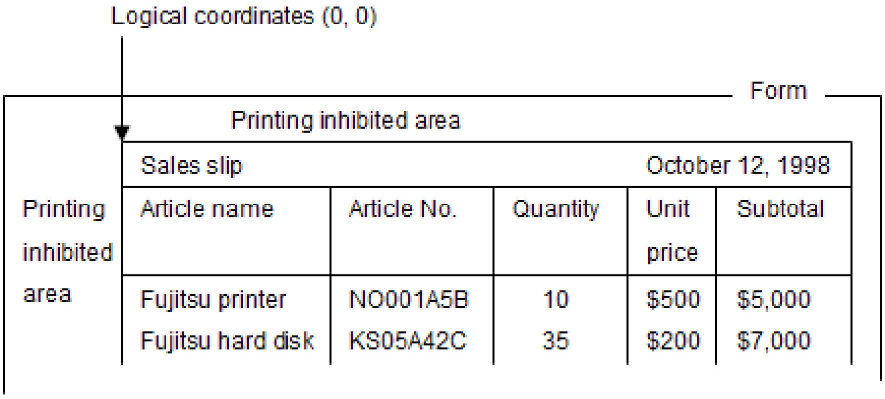
Information

This version allows VALID to be specified for this environment variable information to validate the functions which are specified in the I control record for both line records and forms overlay.

The following illustrates differences in print results between VALID and INVALID (default) mode, using an example of print results obtained by normal printing without using an origin offset, and printing with a one-inch origin offset in both the upper and left directions.

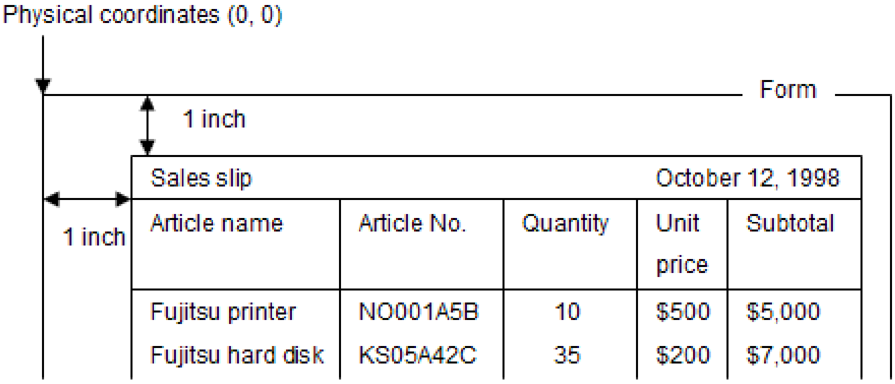
* : In the figure below, the ruled lines indicate a forms overlay pattern and the characters indicate line records.

- Result of printing without specifying an origin of printing



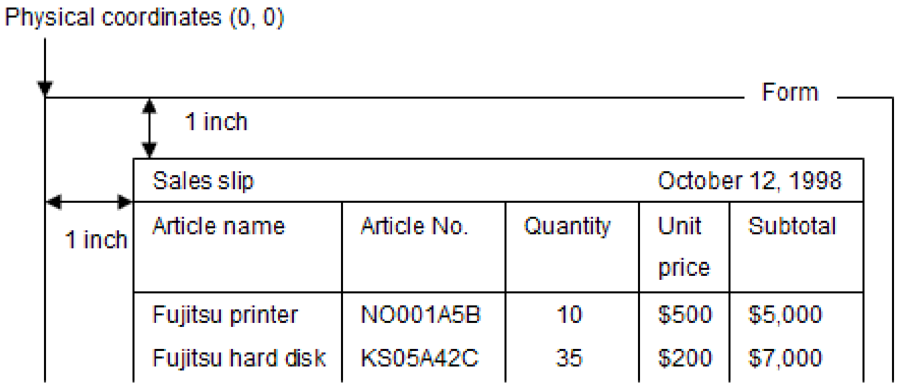
Sales slip				October 12, 1998
Article name	Article No.	Quantity	Unit price	Subtotal
Fujitsu printer Fujitsu hard disk	NO001A5B KS05A42C	10 35	$500 $200	$5,000 $7,000

- Result of printing with a one-inch origin offset each in the X and Y coordinates
  - VALID mode



Sales slip				October 12, 1998
Article name	Article No.	Quantity	Unit price	Subtotal
Fujitsu printer Fujitsu hard disk	NO001A5B KS05A42C	10 35	$500 $200	$5,000 $7,000

- INVALID mode



Sales slip				October 12, 1998
Article name	Article No.	Quantity	Unit price	Subtotal
Fujitsu printer	NO001A5B	10	$500	$5,000
Fujitsu hard disk	KS05A42C	35	$200	$7,000

PRTOUT

Specifies a printer to be used by the program. Specify it in the following cases:

- To change the destination (LPTn: "/" COMn: "/" PRINTNAME: printer-name) assigned to the file identifier
- To specify the destination which is made valid if the destination assignment to file identifier is omitted.

This option can change the destination without program modification and, therefore, it is useful in the following cases:

- If the destination is directly specified in the program using the data name setup or constant setup of ASSIGN clause.

The available destinations are the same as for the assignment of file identifier. Specify the local printer port name (LPTn:), communication port name (COMn:), or printer name (PRTNAME: printer-name) to which the actual form output printer is connected. However, the print information file name (INF) and font table name (FONT) cannot be specified.

The PRTOUT is optional. If omitted, the setup of file identifier name is made valid. If different local printer port names (LPTn:), communication port names (CPMn:) or printer names (PRTNAME: printer-name) are specified for the PRTOUT of file identifier and print information file, the PRTOUT setup of print information file is used. If both of them are omitted or if specified incorrectly, a runtime error occurs. For file identifier details, see the "5.4.1.69 File identifier(Specify the printer and various parameters used for the program)".

Note

The character code set of Print information file supports UTF-8. Please make the initialization file with UTF-8, and operate it by the Unicode program when you specify the printer name that contains the multi byte character. However, the specification of the printer name including the multi byte character is enabled with SJIS in a Japanese environment.

PAPERSIZE

Specify the size of the default form. The size that can be specified as following:

A3, A4, A5, B4, B5, LTR or form-name

For details, see the "SIZE (Form size)".

When this specification is omitted, a set value of the printer driver becomes effective.

Note

When the size of the default form is changed, the Forms Control records (FCB) should be defined and be specified according to the size of the form actually used. See the “7.1.4 Forms Control Buffers(FCB)” .

PRTFORM

Specify the print format for the default. The print format that can be specified as following:

P(Portrait mode), L(Landscape mode), LP(Line printer mode), PZ (Compressed portrait mode), or LZ (Compressed landscape mode)

For details, see the "PRT-FORM (Print form)".

When this specification is omitted, a set value of the printer driver becomes effective.

Note

- When the print form of default is specified, it is necessary to specify the size of the form. Please refer to "COBOL Language Reference" for the combination of the size of the form and the print form that can be specified.
- When the size of the default form is changed, the Forms Control records (FCB) should be defined and be specified according to the size of the form actually used. See the “7.1.4 Forms Control Buffers(FCB)” .

FOVLDIR

Specify the absolute path name of the form overlay pattern file storage destination folder. If this is omitted, the form overlay pattern cannot be printed. You cannot specify more than one folder.

This has priority over the "5.4.1.71 FOVLDIR(Set the Folder Containing Form Overlay Patterns)" runtime environment variable.

FOVLTYPE

If the 4 starting characters of the form overlay pattern file name are not "KOL5" (this is the default), specify the 4 starting characters of the file name in the format. If there is no format in the file name, specify the string "None".

Example

If the form overlay pattern file is "C:\FOVLDATA\KOL2OVD1.OVD"

```
FOVLDIR=C:\FOVLDATA

FOVLTYPE=KOL2

FOVLNAME=OVD1 or specify "OVD1" in the I/S control FOVL field and then add WRITE
```

This has priority over the "5.4.1.72 FOVLTYPE(Set the Format of the Form Overlay Pattern File)" runtime environment variable.

FOVLNAME

Omit the 4 starting characters of the format part of the overlay pattern file name. Specify the second half of the file name using a maximum of 4 characters. This is significant if the I/S control record is not output, or the overlay output is not specified in the I/S control record (the FOVL field is empty).

Example

If the form overlay pattern file is "C:\FOVLDATA\KOL2TEST.OVL"

```
FOVLDIR=C:\FOVLDATA

FOVLTYPE=KOL2

FOVLNAME=TEST, and (No WRITE for I/S control record or enter a space in the I/S control FOVL field
and then add WRITE)

OVD_SUFFIX=OVL
```

This has priority over the "5.4.1.73 FOVLNAME(Specifying Form overlay pattern file names)" runtime environment variable.

OVD_SUFFIX

If the file extension of the form overlay pattern file is not "OVD" (the default), specify a string that can be used as a file extension. If there is no file extension in the file name, specify the string "None".

Example

If the form overlay pattern file is "C:\FOVLDATA\KOL5ABCD"

```
FOVLDIR=C:\FOVLDATA

FOVLTYPE=KOL5

FOVLNAME=ABCD or specify "ABCD" and then WRITE in the FOVL field of the I/S control

OVD_SUFFIX=None
```

This has priority over the "5.4.1.74 OVD_SUFFIX(Set the Extension of the Form Overlay Pattern File)" runtime environment variable.

7.1.10 How to Determine Print File/Presentation File

COBOL offers several ways to create print files. These are: use a print file without FORMAT clause, a print file with FORMAT clause or presentation file.

The compiler recognizes the different styles of print file by recognizing the presence of specific elements in the source program. These are:

[1] FORMAT clause

[2] PRINTER specified in the ASSIGN clause

[3] PRINTER-n in the specified ASSIGN clause

[4] LINAGE clause

[5] ADVANCING specified in the WRITE statement

[6] File reference "GS-file identifier"

The table below shows the file type determined by the presence of the above elements [1] through [6].

File type	[1]	[2]	[3]	[4]	[5]	[6]
Print file without FORMAT clause	No	Yes (note)	Yes (note)	Yes (note)	Yes (note)	No
Print file with FORMAT clause	Yes	No	No	No	*	No
Presentation file	*	No	No	No	No	Yes

Yes : Element that determines the file type

*: Element can be included but doesn't affect the file type.

No : Cannot be included in that file type.

Note

The presence of [2], [3], [4] or [5] determines that the file is a print file without a FORMAT clause.

7.1.11 Cautions under Service Range (for printing)

When running a COBOL program under the Windows system service, the service starts up a daemon process with the system account and operates the COBOL program as a background job.

In this case, since no printer environment exists to the system account, the COBOL runtime system is unable to acquire the printer name and other information. Therefore, the form print function for the print file and presentation file does not function correctly.

- To run a COBOL program using a print file, with or without the FORMAT clause or a presentation file (destination: PRT) as a system service, explicitly specify the name of a printer to be assigned to the output destination of the file. If the default printer, local printer port name (LPTn:), or communication port name (COMn:) is specified for the output destination, data may not be printed because of a file allocation error.

Example

Examples of assignment for files that can be printed under the service range

1)Print file without FORMAT clause

- Specify an ASSIGN clause.
    - SELECT print-file ASSIGN TO S-PRTF.
- Specify environment variable information.
    - PRTF = PRTNAME: FUJITSU VSP4620A

2)Print file with FORMAT clause

- Specify a printer information file.
    - PRTDRV FUJITSU VSP4620A

Check the respective service specifications to determine whether the service runs with a specific system account or user account.

In addition, depending on the service, a login can be set up with a specific user account by configuring the startup in the control panel service. This creates a printable environment by setting the user account for when the service printer driver is installed.

- When Print file is used under the service range, the paper size is set as follows:
    - The size in printer properties in case of under the service range
    - The size in printer information file at the usual application program operation

7.1.12 Forms Design

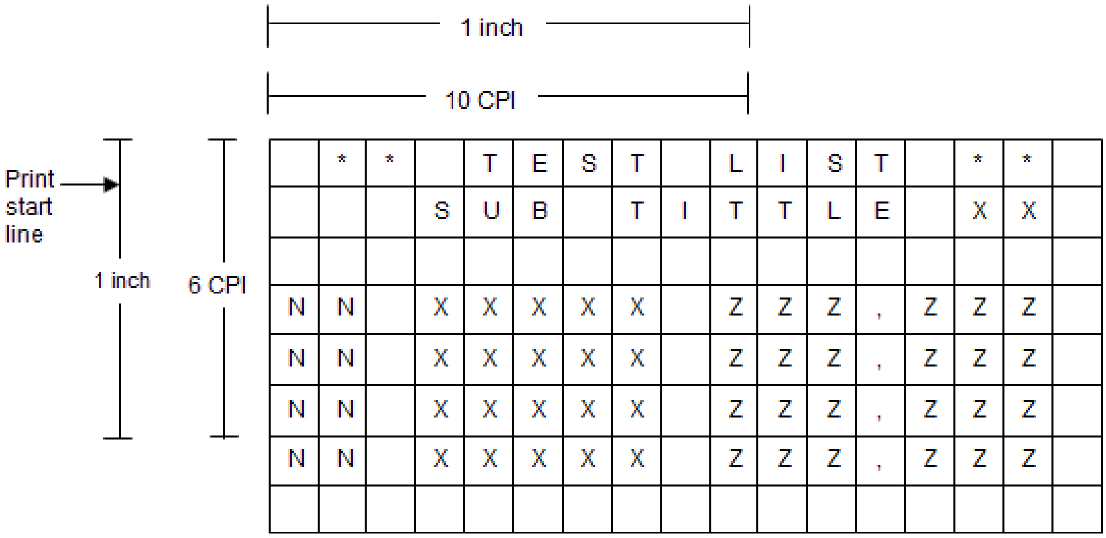
The concepts of lines and columns are indispensable for COBOL forms printing. Particularly when a print file without the FORMAT clause is used for printing forms mainly consisting of line records, detailed forms designing is required before a program is actually created.

Design forms in an actual print image by using design paper such as spacing charts (or a tool such as FORM that can display a grid image), then create a program based on this forms design.

Make the spacing chart boxes in the vertical direction correspond to line control of COBOL WRITE with ADVANCING or the LPI or CH operand of the FCB control statement. Make the boxes in the horizontal directions correspond to the number of digits of the PICTURE clause, character spacing (CPI) of the CHARACTER TYPE clause, or horizontal skip information of the PRINTING POSITION clause. When forms are designed, clarify how many lines should be printed per inch in the vertical direction, and how many characters per inch in the horizontal direction.

Example

[Example of spacing chart]



	*	*		T	E	S	T		L	I	S	T		*	*	
			S	U	B		T	I	T	T	L	E		X	X	
N	N		X	X	X	X	X		Z	Z	Z	,	Z	Z	Z	
N	N		X	X	X	X	X		Z	Z	Z	,	Z	Z	Z	
N	N		X	X	X	X	X		Z	Z	Z	,	Z	Z	Z	
N	N		X	X	X	X	X		Z	Z	Z	,	Z	Z	Z	

[FCB control statement]

```
LPI ( ( 6 ) ) , CH1 ( 1 ) , FORM ( ...
```

[COBOL program]

```
    SPECIAL-NAMES.
      PRINTING MODE PM1 IS FOR ALL
        AT PITCH 10.00 CPI.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT PRINT-FILE ASSIGN S-SYS001.
 DATA DIVISION.
 FILE SECTION.
 FD PRINT-FILE.
    01 PRINT-RECORD PIC X(80)
                    CHARACTER TYPE IS PM1.
 WORKING-STORAGE SECTION.
 01 PRINT-DATA  PIC X(80).
 PROCEDURE DIVISION.
*>               :
      OPEN OUTPUT PRINT-FILE.
      MOVE "   **  TEST LIST  **" TO PRINT-DATA.
      WRITE PRINT-RECORD FROM PRINT-DATA
                         AFTER ADVANCING PAGE.
*>               :
      CLOSE PRINT-FILE.
      STOP RUN.
```

7.1.13 Unprintable Areas

Number of printable characters on a form

The maximum number of characters that can be printed on a form is determined by the form size (horizontal direction) and character pitch (CPI). For instance, suppose continuous forms each measuring 15 inches by 11 inches are used with 10 CPI. Then, making a simple calculation by multiplying 15 inches (form size in the horizontal direction) by 10 CPI allows you to determine that up to 150 characters can be printed on a form.

As noted later, however, continuous forms have tractor sprocket holes on both right and left sides, and printing is physically disabled in an area about 1.4 inches wide in total on the right and left sides of each form (the size depends on the printer type). Thus, not all 15 inches in the horizontal direction can be printed. An area about 13.6 inches wide determined by subtracting the unprintable area can actually be printed, i.e., up to 136 characters can be printed on a form.

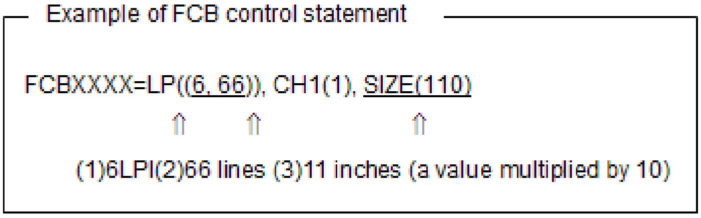
Number of printable lines on a form

The number of lines that can be printed is determined by the definition in the FCB control statement.

The FCB control statement specifies the vertical size of physical forms set in the printer, line spacing (LPI), and channel port. The number of lines that can be printed on a form is determined by the form size and line spacing defined by the FCB control statement.

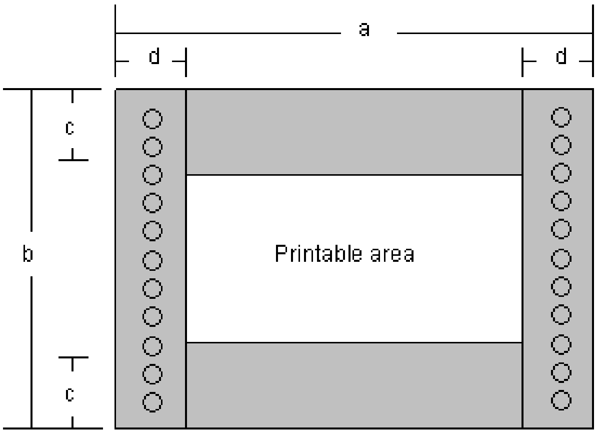
An example of the FCB control statement for using continuous forms of 15 by 11 inches is shown below:

- Line spacing: 6 LPI
- Maximum lines printable on a form: 66
- Vertical form size: 11 inches (Specify a value multiplied by 10.)



Example of FCB control statement

```
FCBXXXX=LP((6, 66)), CH1(1), SIZE(110)
              ⇑     ⇑              ⇑
    (1)6LPI(2)66 lines (3)11 inches (a value multiplied by 10)
```



The settings made before shipment are as follows (adjustable):

a: Form size (horizontal)

b: Form size (vertical)

c: Unprintable area

d: Unprintable area

Note

Some printers, due to their hardware specification, have physically unprintable areas on the upper, lower, right, and left sides of each form. The printer drivers that support these printers inhibit printing on these areas (they shift data to printable areas or discard data). The size of the physically unprintable area varies depending on the printer. The user should refer to the instruction manual prepared for the printer used and design forms in consideration of unprintable areas. Note that the numbers of printable characters and lines provided above are just standard values and are not fixed values.

7.1.14 Font table

The font table is a text-format file which defines the font information for form output using a print file.

The font information must be specified by associating the font face name and print style of print characters with the font number.

If a printing mode name is specified in the CHARACTER TYPE clause and if a font number is specified in the PRINTING MODE clause which defines the print mode name, a font table having the font number information is required.

The following shows a format of font table.

Format and setup information

[Font number]	<- Section name. (It must be specified for each font number.)
FontName=font-face-name	<- Specifies a type face name of the font. (Optional)
Style= { <u>R</u> / B / I / BI }	<- Specifies a print style. (Optional)

Section name

The section name (or font number) identifies the font information associated with the font number. The font information needs to be set for each font number.

Specify the font number ("FONT-nnn") written in the COBOL source program. The character string must consist of single-byte alphanumeric characters.

Example

```
[FONT-001]
```

FontName

Specifies a font used for character printing as the font face name. The default is the Courier New font (the normal font).

Note

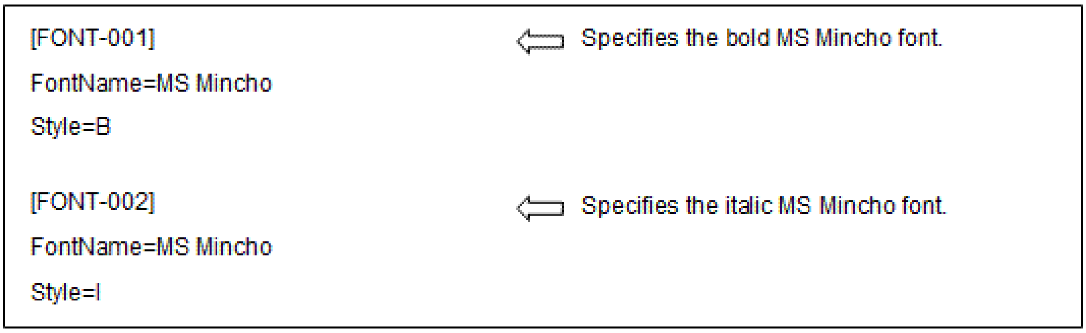
- Specify the font face name using up to 32 bytes of alphanumeric or national characters.
- Specify a typeface name for a font face name in the window displayed by the following operations:
  - Select Fonts in the Control Panel.
  - Double-click the relevant font from the displayed font list.
- The character code set of Font table supports UTF-8. Please make the initialization file with UTF-8, and operate it by the Unicode program when you specify the font face name that contains the multi byte character. However, the specification of the font face name including the multi byte character is enabled with SJIS in a Japanese environment.

Style

Specifies a style of font used for printing. The default is the regular(R) style.

- R: regular face
- B: Bold face
- I: Italic face

- BI: Bold and italic face



```
[FONT-001]                          ⇦  Specifies the bold MS Mincho font.
FontName=MS Mincho
Style=B

[FONT-002]                          ⇦  Specifies the italic MS Mincho font.
FontName=MS Mincho
Style=I
```

When using the font table in the print file, specify the font table name as the "@CBR_PrintFontTable" environment variable information or the file identifier.

Refer to "5.4.1.36 @CBR_PrintFontTable(Specify the font table used for a print file)", "5.4.1.68 File identifier(Specify the printer information file and various parameters used for the program)" and "5.4.1.69 File identifier(Specify the printer and various parameters used for the program)"

7.2 How to print data in line mode using

This section explains how to print data in line mode using a print file without a FORMAT clause. For a sample program using a print file, refer to the "Getting Started with NetCOBOL" guide.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. program-name.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
     [function-name IS mnemonic-name.]
INPUT-OUTPUT SECTION.
 FILE-CONTROL.
      SELECT file-name
        ASSIGN TO PRINTER
       [ORGANIZATION IS SEQUENTIAL]
       [FILE STATUS IS input-output-status]
        .
DATA DIVISION.
 FILE SECTION.
 FD  file-name
     [RECORD record-size]
     [LINAGE IS logical-page-configuration-specification]
     .
 01  record-name  [CHARACTER TYPE IS {MODE-1 | MODE-2 | MODE-3 | mnemonic-name}].
      record-description-entry.
 WORKING-STORAGE SECTION.
 [01  input-output-status    PIC  X(2).]
01  data-name  [CHARACTER TYPE IS { MODE-1 | MODE-2 | MODE-3 | mnemonic-name} ].]
PROCEDURE DIVISION.
      OPEN OUTPUT  file-name.
      WRITE  record-name  [FROM  data-name]  [AFTER  ADVANCING  ...].
      CLOSE  file-name.
 END PROGRAM program-name.
```

7.2.1 Outline

Define a print file in the same manner as a record sequential file, and do the same processing as would be required to create a record sequential file. The following can be specified with a print file without a FORMAT clause:

- Logical page size (LINAGE clause in the file description entry)
- Character size, font, form, direction, and space (CHARACTER TYPE clause in the data description entry)
- Line feed and page alignment (ADVANCING phrase in the WRITE statement)

7.2.2 Program Specifications

This section details program descriptions in a print file using data in line mode for each COBOL division.

ENVIRONMENT DIVISION

In the ENVIRONMENT DIVISION, write the relation between the function-name and mnemonic-name (print characters are indicated in the program) and define a print file.

Relating the Function-Name to the Mnemonic-Name

Relate the function-name indicating the size, font, form, direction, and space of a print character to the mnemonic-name. Specify the mnemonic-name in the CHARACTER TYPE clause when defining data items in records and work data items. For function-name types, refer to the "NetCOBOL Language Reference."

Defining Print Files

Table 7.4 lists information required to specify a file control entry

	Location	Information Type	Details and Use of Specification
Required	SELECT clause	File name	Write the name of a file to use in a COBOL program, conforming to the rules of COBOL user-defined words.
	ASSIGN clause	File- reference-identifier	Write PRINTER, PRINTER-n,ACCESS-NAME, local-printer-port-name (LPTn:) or serial-printer-port-name (COMn:). When PRINTER is specified, data is output to the system default printer.
Optional	ORGANIZATION clause	Keyword indicating file organization	Write SEQUENTIAL.
	FILE STATUS clause	Data-name	Write the data-name defined as a 2-byte alphanumeric data item in the WORKING-STORAGE or LINKAGE section. The input-output execution result is set for this data-name. (*1)

(*1) For value to be set, see "Appendix B I-O Status List".

DATA DIVISION

In the DATA DIVISION, define record definitions and the definitions of data-names specified in a file control entry. Write record definitions in file and record description entries. Table below lists information required to write a file description entry.

Table 7.5 Information to be specified in a file description entry

	Location	Information Type	Details and Use of Specification
Optional	RECORD clause	Record size	Define the size of the printable area.

	Location	Information Type	Details and Use of Specification
	LINAGE clause	Logical page configuration	Define the number of lines per page, top and bottom margins, and the footing area starting position. When a data-name is specified in this clause, the data can be changed in the program.

PROCEDURE DIVISION

Execute input-output statements in the following sequence:

1. OPEN statement with OUTPUT specified: Starts printing.
2. WRITE statement: Outputs data.
3. CLOSE statement: Stops printing.

OPEN and CLOSE Statements

Use an OPEN statement once at the start of printing and a CLOSE statement once at the end of printing.

WRITE Statement

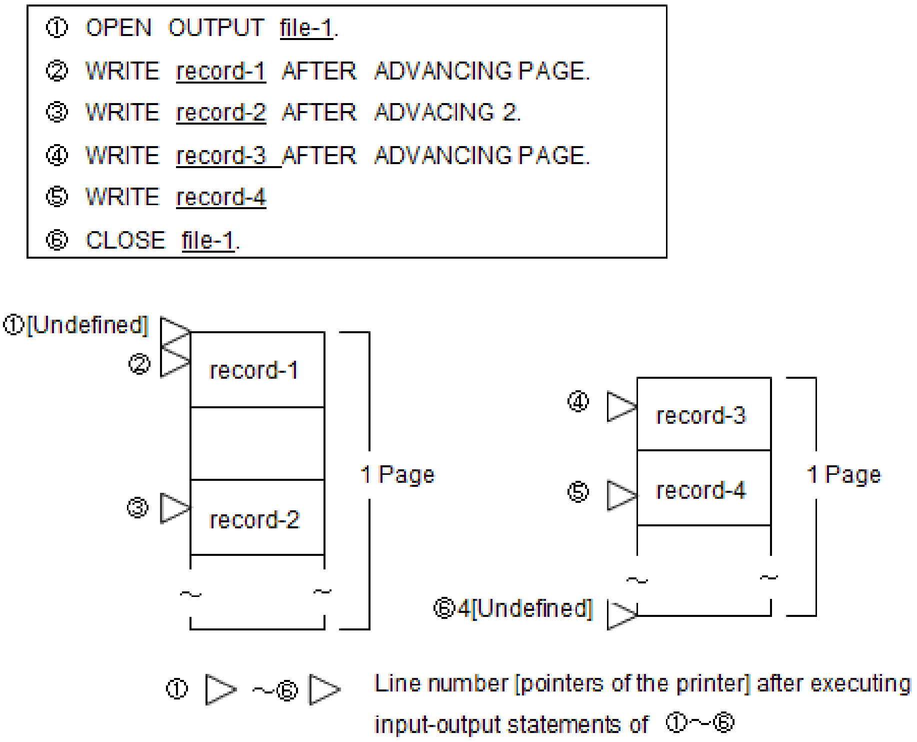
One WRITE statement writes one line of data.

Specify PAGE in the ADVANCING phrase of the WRITE statement for page alignment. When the number of lines is written, the page is advanced by the specified number of lines.

The AFTER ADVANCING and BEFORE ADVANCING phrases are used to specify whether data is output before or after page alignment or a line feed. When the ADVANCING phrase is omitted, "AFTER ADVANCING 1" is the default.

Control of print lines with AFTER ADVANCING phrase is shown in the following example.

Figure 7.2 Print lines with AFTER ADVANCING



```
① OPEN  OUTPUT  file-1.
② WRITE  record-1  AFTER  ADVANCING PAGE.
③ WRITE  record-2  AFTER  ADVACING 2.
④ WRITE  record-3  AFTER  ADVANCING PAGE.
⑤ WRITE  record-4
⑥ CLOSE  file-1.
```

①[Undefined]
②
record-1
③
record-2
1 Page
④
record-3
⑤
record-4
1 Page
⑥4[Undefined]
① ▷ ~⑥ ▷
Line number [pointers of the printer] after executing
input-output statements of ①~⑥

Note

- When writing "WRITE A FROM B", define the CHARACTER TYPE clause for B but not for A. If the CHARACTER TYPE clause is defined for both A and B, only the specification of B is valid.
- A new page is not started when the WRITE statement with the AFTER ADVANCING PAGE phrase immediately after the OPEN statement is executed.

Input-Output Error Processing

Refer to "6.6 Input-Output Error Processing".

7.2.3 Program Compilation and Linkage

There are no required compiler and linker options.

7.2.4 Program Execution

First, assign a printer. Assigning a printer depends on the contents of descriptions in the ASSIGN clause in the file control entry.

This section explains how to specify a printer in a program or initialization file, and gives the relation between the contents of descriptions in the ASSIGN clause and the output destination by using examples.

If the destination of a print file is other than a printer, contents are not assured.

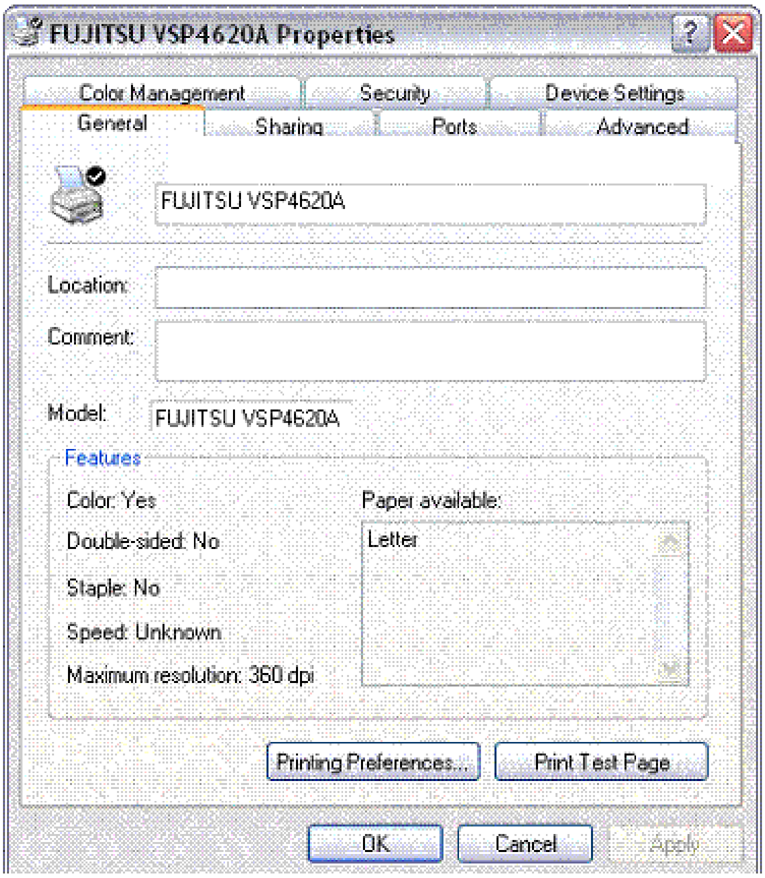
Specifying a Printer

A printer can be specified by the following methods:

- Printer name
- Local printer port name ("LPTn:") or Serial port name ("COMn: "). Where "n" is a number from 1-9.

Use the information in the Details tab of the Printer Properties dialog box to obtain the printer name. To access the Details tab, click on the Start button and then select SettingsàPrinters. Highlight a printer and select Properties from the File menu. Click on the Details tab.

The Printer Properties dialog box (with the General tab selected) is shown below.



FUJITSU VSP4620A

Location:

Comment:

Model: FUJITSU VSP4620A

Features

Color: Yes

Double-sided: No

Staple: No

Speed: Unknown

Maximum resolution: 360 dpi

Paper available:

Letter

Printer name

The printer name is the "PRTNAME:" name added to the beginning of "printer name" displayed in the edit box right to the printer mark of [General] sheet of [Properties] dialog.

Example

```
PRTNAME: FUJITSU VSP4620A
PRTNAME: Canon LBP-A404E
```

It is necessary to specify "\\server name\" for the name of the printer when connecting with the printer in the network.

Example

```
PRTNAME:\\PRTSVR\FUJITSU VSP4620A
PRTNAME:\\PRTSVR\Canon LBP-A404E
```

Information

The printer name can be acquired conveniently by using the Printer Name Selection dialog box that is displayed by [Printer Name Selection] dialog from the [Environment] menu of the Runtime Environment Setup Tool window.

Note

The name of a network printer is not displayed. Always specify the printer name from the [Printer Name Selection] dialog box.

Local printer port name / Serial port name

Specify a local printer port name or serial port name in the list box in the printing ahead of the [Ports] sheet of the [Properties] dialog.

Example

```
"LPT1:"
```

Printer name

The printer name means the name produced by attaching "PRTNAME:" to the beginning of an arbitrary printer name defined in the printer name edit box in the Printer Properties dialog box.

Example

```
PRTNAME: FUJITSU VSP4620A
PRTNAME: Canon LBP-A404E
```

The printer name displayed in the printer name edit box can also be specified when the system is connected to a network printer.

Example

```
PRTNAME: \\NMSVR20\FUJITSU VSP4620A
```

Information

The printer name can be acquired conveniently by using Printer Name Selection dialog box, displayed from the Runtime Environment Setup Tool window by selecting [Printer] from the [Environment] menu.

Local printer port name / Serial port name

Specify a local printer port name or serial port name in the following manner:

Example

```
"LPT1:"
```

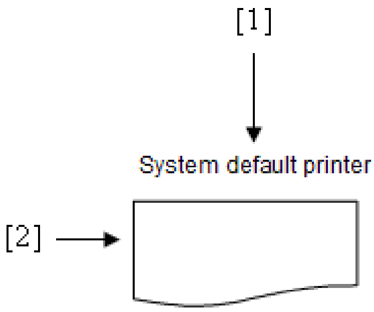
Examples of Programs

- When PRINTER is Specified in the ASSIGN Clause

  Data is sent to the system default printer (*1).

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. A.
 ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT print-file ASSIGN TO PRINTER. *>---[1]
 DATA DIVISION.
  FILE SECTION.
  FD print-file.
  01 outrec PIC X(125).
 PROCEDURE DIVISION.
```

```
     OPEN OUTPUT  print-file.            *>--+
     WRITE outrec AFTER ADVANCING 2.     *>  |[2]
     CLOSE print-file.                   *>--+
     EXIT PROGRAM.
 END PROGRAM A.
```



*1 System default printer:

"Set as Default" is checked for the printer.

To send data to a printer connected to a specific port, follow the method explained below.

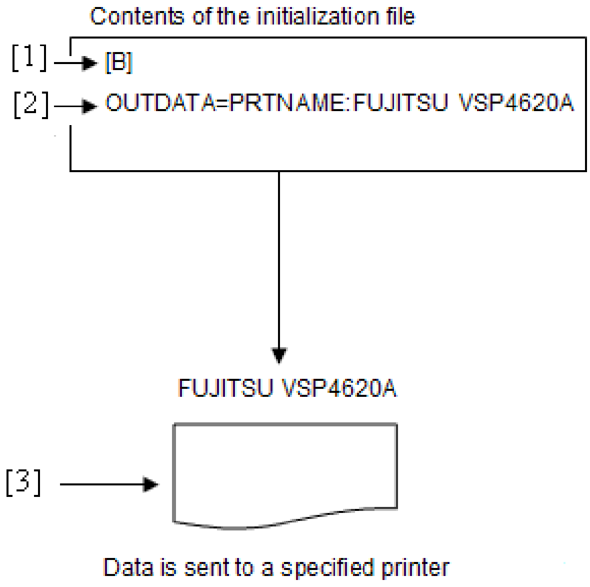
- When a File-Identifier is Specified in the ASSIGN Clause

  Define the name of the output destination printer, local printer port name, or serial port name with the file-identifier as the environment variable information name. For details on setting environment variable information, refer to "5.3 Setting Runtime Environment Information".

  A file assignment error occurs if no printer is assigned to the file-identifier. An example of when the initialization file is used is shown in the following figure.

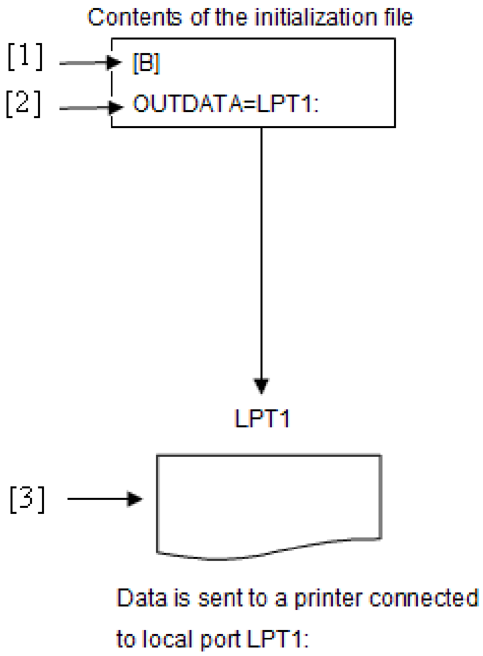
  When a printer name is specified :

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. B.                         *>---[1]
 ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT print-file ASSIGN TO OUTDATA. *>---[2]
 DATA DIVISION.
  FILE SECTION.
  FD print-file.
  01 outrec PIC X(125).
 PROCEDURE DIVISION.
     OPEN OUTPUT  print-file.            *>--+
     WRITE outrec AFTER ADVANCING 2.     *>  |[3]
     CLOSE print-file.                   *>--+
     EXIT PROGRAM.
 END PROGRAM B.
```


Contents of the initialization file
[1]
[B]
[2]
OUTDATA=PRTNAME:FUJITSU VSP4620A
FUJITSU VSP4620A
[3]
Data is sent to a specified printer

When a local port name is specified :

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. B.                          *>---[1]
 ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT print-file ASSIGN TO OUTDATA. *>---[2]
 DATA DIVISION.
  FILE SECTION.
  FD print-file.
  01 outrec PIC X(125).
 PROCEDURE DIVISION.
     OPEN OUTPUT  print-file.             *>--+
     WRITE outrec AFTER ADVANCING 2.      *>  |[3]
     CLOSE print-file.                    *>--+
     EXIT PROGRAM.
 END PROGRAM B.
```

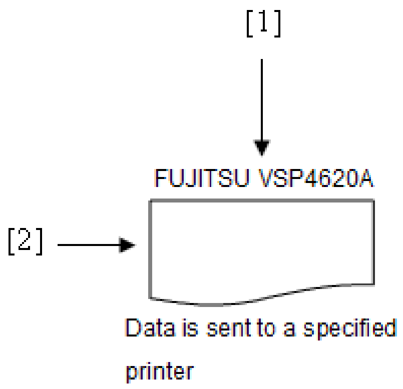

Contents of the initialization file
[1]
[B]
[2]
OUTDATA=LPT1:
LPT1
[3]
Data is sent to a printer connected
to local port LPT1:

- When a File-Identifier Literal is Specified in the ASSIGN Clause

  Data is sent to a printer specified in the file-identifier literal or to a printer connected to a local or serial port specified in the file-identifier literal.

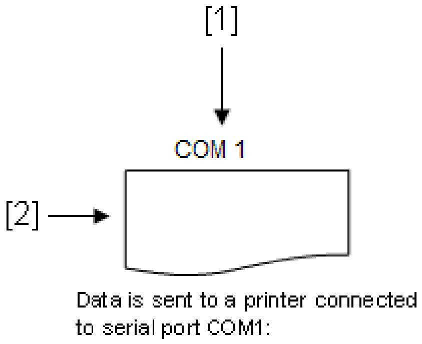
  When a printer name is specified :

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. C.
 ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT print-file
      ASSIGN TO "PRTNAME:FUJITSU VSP4620A".  *>---[1]
 DATA DIVISION.
  FILE SECTION.
  FD print-file.
  01 outrec PIC X(125).
 PROCEDURE DIVISION.
     OPEN OUTPUT  print-file.                *>--+
     WRITE outrec AFTER ADVANCING 2.         *>  |[2]
     CLOSE print-file.                       *>--+
     EXIT PROGRAM.
 END PROGRAM C.
```



When a serial port name is specified :

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. C.
 ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT print-file
      ASSIGN TO "COM1:".  *>---[1]
 DATA DIVISION.
  FILE SECTION.
  FD print-file.
  01 outrec PIC X(125).
 PROCEDURE DIVISION.
     OPEN OUTPUT  print-file.            *>--+
     WRITE outrec AFTER ADVANCING 2.     *>  |[2]
     CLOSE print-file.                   *>--+
     EXIT PROGRAM.
 END PROGRAM C.
```



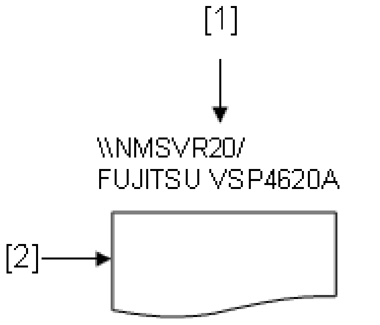
- When a Data-Name is Specified in the ASSIGN Clause

  Data is sent to a printer specified in the data-name or to a printer connected to a local or serial port specified in the data-name. A file assignment error occurs if the data-name is left blank.

  When a printer name is specified :

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. D.
 ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
```

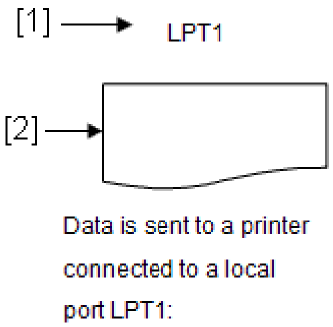
```
   FILE-CONTROL.
     SELECT print-file
      ASSIGN TO data-name-1.           *>---[1]
 DATA DIVISION.
  FILE SECTION.
  FD print-file.
  01 outrec PIC X(125).
  WORKING-STORAGE SECTION.
  01 data-name-1 PIC X(30).
 PROCEDURE DIVISION.
     MOVE "PRTNAME:\\NMSVR20\FUJITSU VSP4620A"
           TO data-name-1.             *>---[1]
     OPEN OUTPUT  print-file.          *>--+
     WRITE outrec AFTER ADVANCING 2.   *>  |[2]
     CLOSE print-file.                 *>--+
     EXIT PROGRAM.
 END PROGRAM D.
```



Data is sent to a specified printer

When a local port name is specified :

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. D.
 ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT print-file
      ASSIGN TO data-name-1.           *>---[1]
 DATA DIVISION.
  FILE SECTION.
  FD print-file.
  01 outrec PIC X(125).
  WORKING-STORAGE SECTION.
  01 data-name-1 PIC X(30).
 PROCEDURE DIVISION.
     MOVE "LPT1:" TO data-name-1.      *>---[1]
     OPEN OUTPUT  print-file.          *>--+
     WRITE outrec AFTER ADVANCING 2.   *>  |[2]
     CLOSE print-file.                 *>--+
     EXIT PROGRAM.
 END PROGRAM D.
```



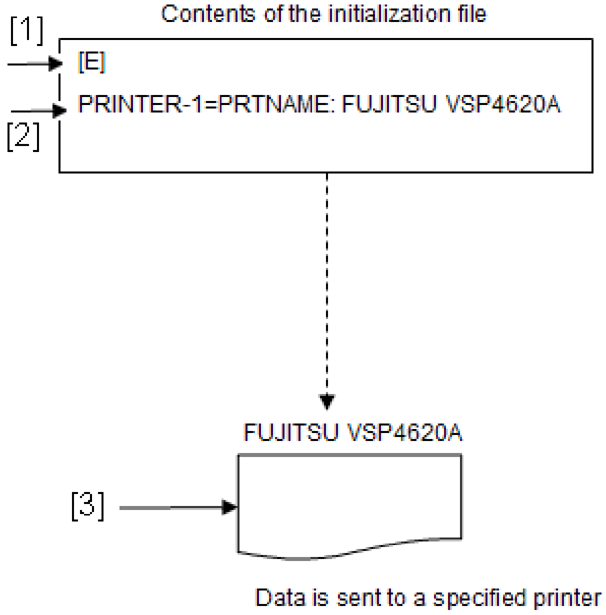
- When Character String PRINTER-n is Described in ASSIGN Clause

Set the printer name by using the character string PRINTER-n(n=1 to 9) as the environment variable name. For instructions on how to set the environment variable information, refer to the "5.3 Setting Runtime Environment Information".

If a printer is not allocated to PRINTER-n, an allocation error of the file occurs. The following show examples using a runtime initialization file.

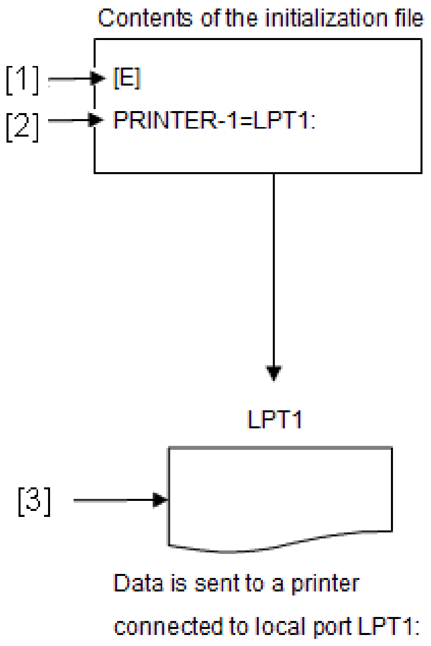
When a printer name is specified :

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. E.                      *>---[1]
 ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT print-file
      ASSIGN TO PRINTER-1.            *>---[1]
 DATA DIVISION.
  FILE SECTION.
  FD print-file.
  01 outrec PIC X(125).
  WORKING-STORAGE SECTION.
  01 data-name-1 PIC X(30).
 PROCEDURE DIVISION.
     OPEN OUTPUT  print-file.           *>--+
     WRITE outrec AFTER ADVANCING 2.    *>  |[2]
     CLOSE print-file.                  *>--+
     EXIT PROGRAM.
 END PROGRAM E.
```


Contents of the initialization file
[1]
[E]
PRINTER-1=PRTNAME: FUJITSU VSP4620A
[2]
FUJITSU VSP4620A
[3]
Data is sent to a specified printer

When a local port name is specified :

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. E.                      *>---[1]
 ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT print-file
      ASSIGN TO PRINTER-1.            *>---[1]
 DATA DIVISION.
  FILE SECTION.
  FD print-file.
  01 outrec PIC X(125).
  WORKING-STORAGE SECTION.
  01 data-name-1 PIC X(30).
 PROCEDURE DIVISION.
     OPEN OUTPUT  print-file.          *>--+
     WRITE outrec AFTER ADVANCING 2.   *>   |[2]
     CLOSE print-file.                 *>--+
     EXIT PROGRAM.
 END PROGRAM E.
```


Contents of the initialization file
[1]
[E]
[2]
PRINTER-1=LPT1:
LPT1
[3]
Data is sent to a printer
connected to local port LPT1:

7.3 How to use a form overlay pattern and FCB

This section explains how to use a form overlay pattern and FCB in a print file without a FORMAT clause. It uses the following code as an example.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. program-name.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
   [function-name IS mnemonic-name-1]
    CTL IS mnemonic-name-2.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT file-name
     ASSIGN TO PRINTER
     [ORGANIZATION IS SEQUENTIAL]
     [FILE STATUS IS input-output-status].
DATA DIVISION.
 FILE SECTION.
 FD file-name
    [RECORD record-size]
    [LINAGE IS logical-page-configuration-specification].
01 line-record-name [CHARACTER TYPE IS {MODE-1 | MODE-2 | MODE-3 | mnemonic-name-1}].
   Contents of the line record
   02 data-name-1 … .
      :
01 control-record-name.
   Contents of the line record
   02 data-name-2 … .
      :
WORKING-STORAGE SECTION.
[01 input-output-status PIC X(2).]
[01 data-name-3 [CHARACTER TYPE IS {MODE-1 | MODE-2 | MODE-3 | mnemonic-name-1}].]
PROCEDURE DIVISION.
     OPEN OUTPUT file-name.
     WRITE control-record-name AFTER ADVANCING mnemonic-name-2.
     WRITE line-record-name AFTER ADVANCING PAGE.
    [WRITE line-record-name [FROM data-name-3] [AFTER ADVANCING ~].]
     CLOSE file-name.
END PROGRAM program-name.
```

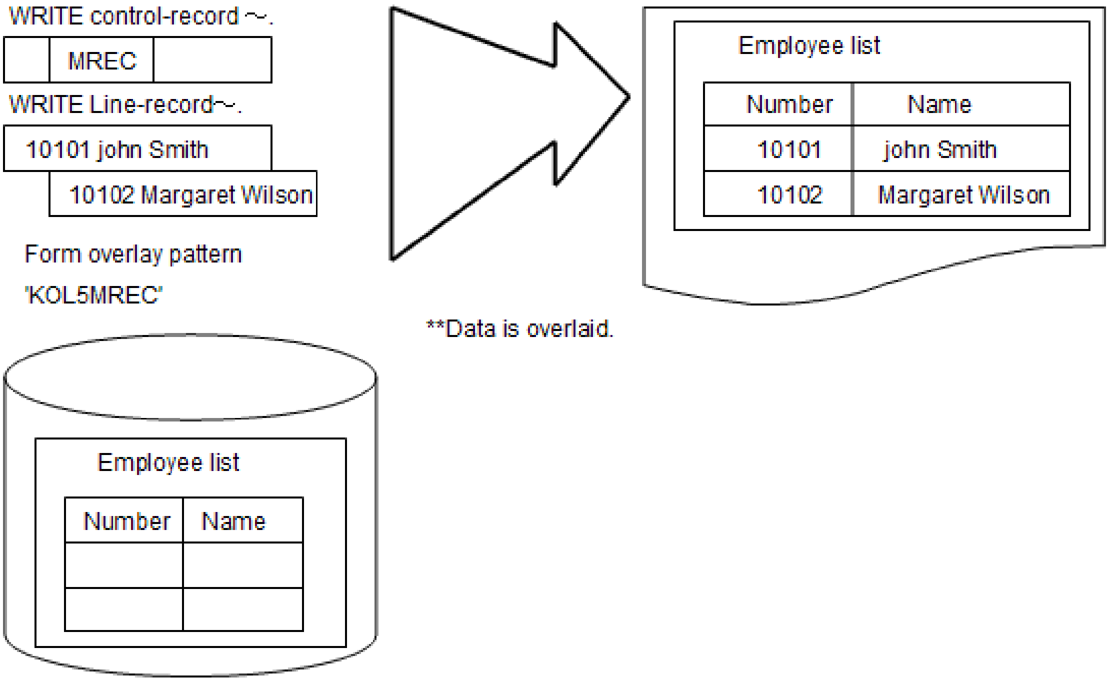
7.3.1 Outline

Use a control record to employ a form overlay pattern with a print file.

As with ordinary data, a control record is created with a WRITE statement. If a control record is created with a form overlay pattern name specified, the data and form overlay pattern are overlaid on the subsequent page.

The size, font, form, direction, and space of a print character are defined with a COBOL program and form overlay pattern. For character settings, see "7.1.2 Print Characters," and also refer to the "FORM HELP" or "PowerFORM HELP."

Figure 7.3 Defining print characters



7.3.2 Program Specifications

This section describes the code required in each of the COBOL divisions when you use form overlay patterns.

ENVIRONMENT DIVISION

In the ENVIRONMENT DIVISION, write the relation between the function-name and mnemonic-name, and define a print file.

Relating the Function-Name to the Mnemonic-Name

To enter a control record to the function-name CTL, relate the mnemonic-name. Specify this mnemonic-name in the WRITE statement when creating the control record.

When specifying a print character with a CHARACTER TYPE clause, relate the function-name indicating the size, font, form, direction, and space of a print character to the mnemonic-name. For function-name types, refer to the "NetCOBOL Language Reference."

Defining Print Files

Define a print file in a file control entry. For details of descriptions in a file control entry, see "Table 7.4 lists information required to specify a file control entry".

DATA DIVISION

In DATA DIVISION, define the record definitions and the definitions of data-names used in the ENVIRONMENT DIVISION.

Defining Records

Define a record in file and record description entries. For details of descriptions in a file description entry, see "Table 7.5 Information to be specified in a file description entry". Define the following records in the record description entry:

Line Records

Define a record to print data edited in a program. Multiple line records can be written.

The contents of one line record are printed sequentially from the left margin of the printable area. Specify the size of a line record so that it is written within the printable area.

The size of a print character can be specified in the CHARACTER TYPE clause in the data description entry. For contents that can be specified in the CHARACTER TYPE clause, see "7.1.2 Print Characters."

Control Records

Two types of control records are used for printing, I and S control records. The format of each control record is shown "7.1.5 I and S control records".

PROCEDURE DIVISION

Execute input-output statements in the following sequence:

1. OPEN statement with OUTPUT specified: Starts printing.
2. WRITE statement: Outputs data.
3. CLOSE statement: Stops printing.

OPEN and CLOSE Statements

Use an OPEN statement once at the start of printing and a CLOSE statement once at the end of printing.

WRITE Statement

Use a WRITE statement when creating a control or line record.

A line record is written in the same manner as when using a WRITE statement to create data in line mode. See "7.2.2 Program Specifications."

To write a control record, write the mnemonic-name related to the function-name CTL in the ADVANCING phrase.

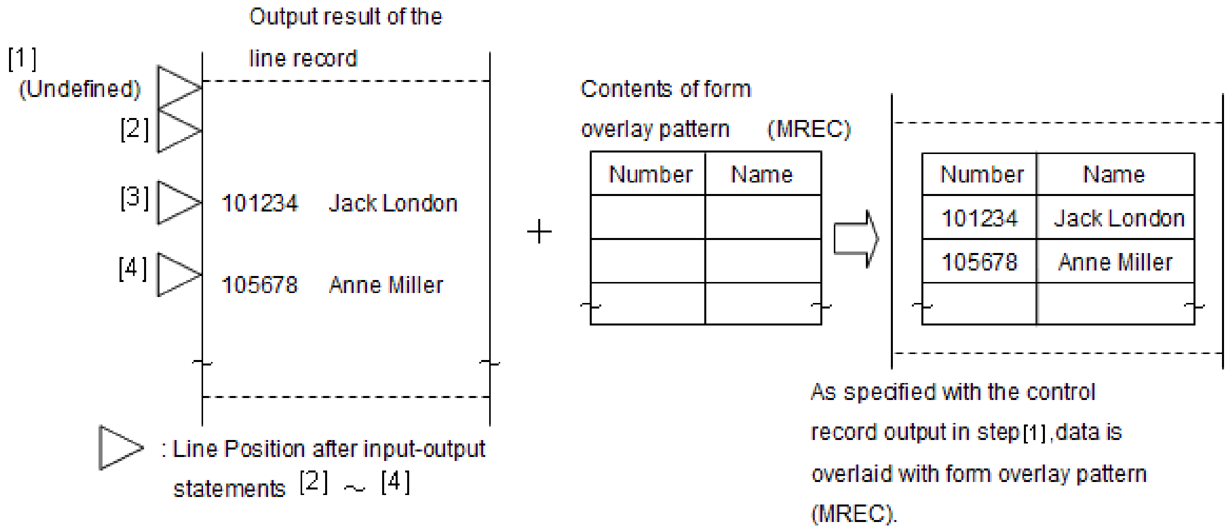
To overlay data created by a line record with a form overlay pattern, write a control record with a form overlay pattern name specified.

To not overlay data with a form overlay pattern, write a control record with a blank entered as the form overlay pattern name.

After a control record is written, the format of the output page is set according to the contents of the control record. When a control record is written, however, no line record can be written to the current page. Thus, to write a line record immediately after a control record, the AFTER ADVANCING PAGE phrase must be specified.

```
*>    :
  FILE SECTION.
  FD file-1.
 01  line-record PIC X(80).
  01 control-record.
    02 FOVL         PIC X(4).
 WORKING-STORAGE SECTION.
 01 employee-number PIC 9(6).
 01 name            PIC X(20).
 PROCEDURE DIVISION.
  MOVE  "MREC"     TO FOVL.
  WRITE  control-record AFTER ADVANCING mnemonic-name.  *> [1]
  MOVE  SPACE      TO line-record.
  WRITE  line-record    AFTER ADVANCING PAGE.           *> [2]
  MOVE  101234     TO employee-number.
  MOVE  "Jack London" TO name.
  WRITE  line-record    AFTER ADVANCING 2.              *> [3]
  MOVE  105678     TO employee-number.
  MOVE  "Anne Miller" TO name.
  WRITE line-record    AFTER ADVANCING 1.               *> [4]
*>    :
```

Figure 7.4 Flow of control with a form overlay pattern


Output result of the line record
[1] (Undefined)
[2]
[3]
[4]
101234 Jack London
105678 Anne Miller
+
Contents of form overlay pattern (MREC)
Number
Name
Number
Name
101234
Jack London
105678
Anne Miller
: Line Position after input-output statements [2] ~ [4]
As specified with the control record output in step[1],data is overlaid with form overlay pattern (MREC).

Input-Output Error Processing

Refer to "6.6 Input-Output Error Processing".

7.3.3 Program Compilation and Linkage

There are no required compiler and linker options.

7.3.4 Program Execution

This section explains how to execute a program using a form overlay pattern and FCB.

7.3.4.1 Programs Using Form Overlay Patterns

When a form overlay pattern is used with a print file, the following settings are required:

- Specify the form overlay pattern storage folder in the FOVLDIR environment variable or in the FOVLDIR key of the print information file. If the FOVLDIR setting is omitted, the form overlay pattern is not printed. For more details refer to "5.4.1.71 FOVLDIR(Set the Folder Containing Form Overlay Patterns)" or "7.1.9 Print Information File". In the case of a print file with a FORMAT clause, use FOVLDIR for the printer information file.

- If the extension of the form overlay pattern file is other than "OVD", specify the extension used in the OVD_SUFFIX environment variable or in the OVD_SUFFIX key of the print information file. If the file has no extension, specify "None". For more details refer to "5.4.1.74 OVD_SUFFIX(Set the Extension of the Form Overlay Pattern File)" or "7.1.9 Print Information File". In the case of a print file with a FORMAT clause, use OVD-SUFFIX for the printer information file.

- If the first four characters of the form overlay pattern file name are not "KOL5", specify the first four characters of the file name in the FOVLTYPE environment variable or in the FOVLTYPE key of the print information file. If there is no format in the file name, specify the string "None". For more details refer to "5.4.1.72 FOVLTYPE(Set the Format of the Form Overlay Pattern File)" or "7.1.9 Print Information File".

An example of an initialization file, containing the environment variable settings for a program "A" using a form overlay, is shown below:

Example

```
[A]
FOVLDIR=C:\FOVLDIR                  (1)
OVD_SUFFIX=                         (2)
FOVLTYPE=KOL6                       (3)
```

1. Set the folder where the form overlay pattern is stored using the FOLVDIR environment variable.
2. The extension of the form overlay pattern file is OVD.
3. The first four characters of the form overlay pattern file are "KOL6".

Output of Forms Overlay Pattern

Printers that can Output Forms Overlay Patterns

The software overlay function using the graphic device interface (GDI) of Windows is supported. Therefore, a printer having a printer driver compatible with Windows can print forms overlay patterns.

7.3.4.2 Programs Using an FCB

When you use an FCB with a print file, write an FCB control statement in the runtime initialization file.

For the specification format and content of an FCB control statement, see "7.1.4 Forms Control Buffers(FCB)."

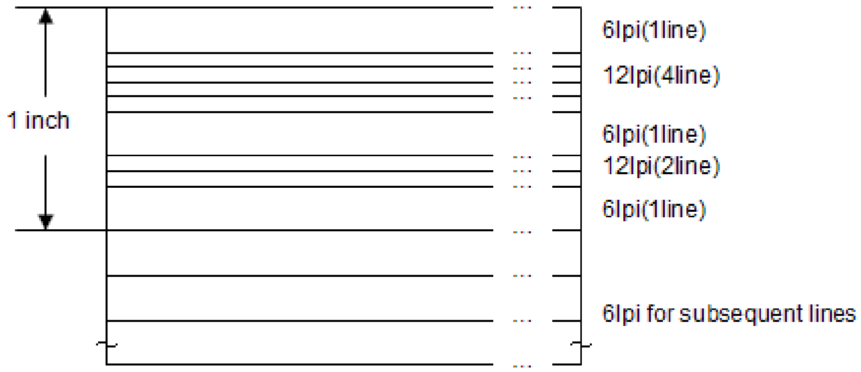
An error occurs if an FCB name is specified in the I control record, but no applicable FCB control statement is found in the runtime initialization file.

For example, the FCB control statement, in a runtime initialization file, for a program A with "FCB1" set as the FCB name, is:

```
[A]
FCBFCB1=LPI((6,1),(12,4),(6,1),(12,2),(6,1))
```

The output format of a page set by the above FCB control statement is shown below:

Figure 7.5 Page layout specified in FCB control statement



7.4 Using Print Files with a FORMAT clause

This section explains how to use partitioned form descriptors by using a print file with a FORMAT clause.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. program-name.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
   [function-name IS mnemonic-name.]
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT file-name
     ASSIGN TO file-reference-identifier
     [ORGANIZATION IS SEQUENTIAL]
     FORMAT IS form-descriptor-name-area
     GROUP IS item-group-name-area
     [FILE STATUS IS input-output-status-1 input-output-status-2].
DATA DIVISION.
 FILE SECTION.
 FD file-name
   [RECORD record-size]
   [CONTROL RECORD IS control-record-name.]
 01 line-record-name
    [CHARACTER TYPE IS {MODE-1 | MODE-2 | MODE-3 | mnemonic-name}].
    02 data-name-1 … .
        :
 01 control-record-name.
    02 data-name-2 … .
        :
     COPY form-descriptor-name OF XMDLIB.
 01 chart-record-name.          *> --+ Expanding a COPY statement
    02 data-name-3 … .           *> --+
WORKING-STORAGE SECTION.
 01 form-descriptor-name-area PIC X(8).
 01 item-group-name-area      PIC X(8).
[01 input-output-status-1     PIC X(2).]
[01 input-output-status-2     PIC X(4).]
[01 data-name-4 …
   [CHARACTER TYPE IS {MODE-1 | MODE-2 | MODE-3 | mnemonic-name}].]
PROCEDURE DIVISION.
    OPEN OUTPUT file-name.
    MOVE form-descriptor-name TO form-descriptor-name-area.
    MOVE item-group-name TO item-group-name-area.
    WRITE chart-record-name [AFTER ADVANCING …].
    WRITE control-record-name.
     :
    MOVE SPACE TO form-descriptor-name-area item-group-name-area.
    WRITE line-record-name [FROM data-name-4] AFTER ADVANCING PAGE.
      :
    CLOSE file-name.
    EXIT PROGRAM.
END PROGRAM program-name.
```

7.4.1 Outline

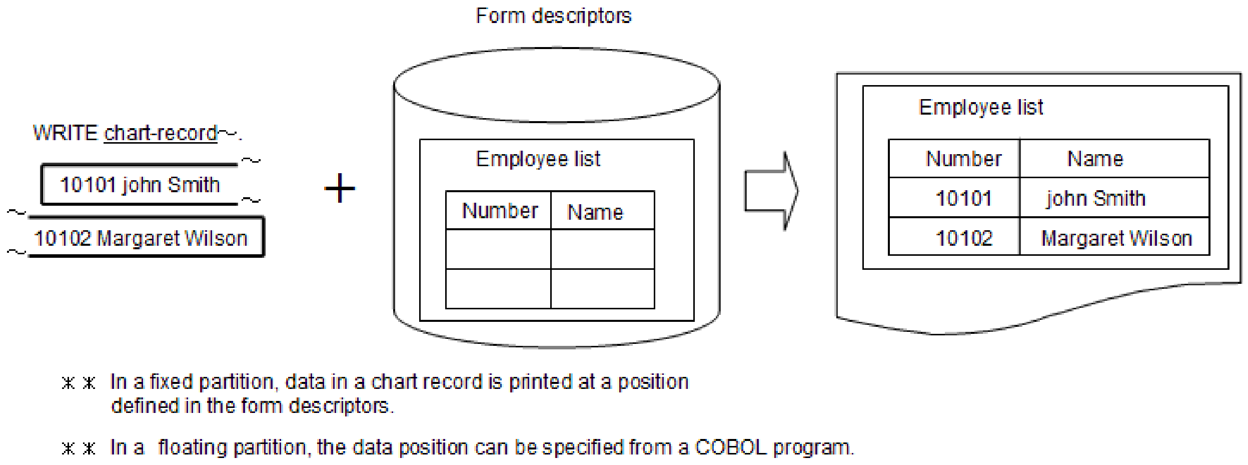
Use a chart record to print forms using partitioned form descriptors.

Define a partition (item group) designated in the form descriptors in a chart record. Multiple lines contained in a partition can be printed out by a single execution of WRITE statement. A single page of the form consists of one or more partitions. The print configuration of the page depends on the output sequence of chart records and line records. You do not have to write the definition statement of a chart record, as it can be fetched from the form descriptors with a COBOL COPY statement.

As with ordinary data, create a chart record with a WRITE statement. The size, font, form, direction, and space of print characters can be indicated in the COBOL program and form descriptors.

For character settings, refer to "7.1.2 Print Characters", the "FORM" manual and "FORM online help", or the "PowerFORM Getting Started" manual and PowerFORM on-line help.

Figure 7.6 Flow of control with form descriptors


Form descriptors
WRITE chart-record~.
10101 john Smith
10102 Margaret Wilson
+
Employee list
Number
Name
Employee list
Number
Name
10101
john Smith
10102
Margaret Wilson
⁎⁎ In a fixed partition, data in a chart record is printed at a position defined in the form descriptors.
⁎⁎ In a floating partition, the data position can be specified from a COBOL program.

Note

- The following form descriptors cannot be used with print files with a FORMAT clause. Use the presentation file form print function.
  - Labels
  - New block of column setting partition
  - New frame of frame partition
  - Output range specification in rectangular domain
  - Bottom information setting of partition form
- When creating a form descriptor, use the following rules for the names to be set or referred to in the COBOL program:
  - Specify the form descriptor name using up to eight single-byte alphanumeric characters.
  - Specify the item group name and partition name using up to six single-byte alphanumeric characters.
  - Specify the item name by following the syntax rules of COBOL user-defined word.
- To output the forms that contain national items in UTF-32 encoding, convert the form descriptor to the UTF-32 form descriptor. For the conversion of the form descriptor, see "I.4 CNVMED2UTF32 Command".

7.4.1.1 Fixed and floating partitions

There are two types of partitions: fixed partitions and floating partitions.

Fixed partitions

A fixed partition has the fixed print position in each page. The partition is printed out in the position specified by the form descriptor.

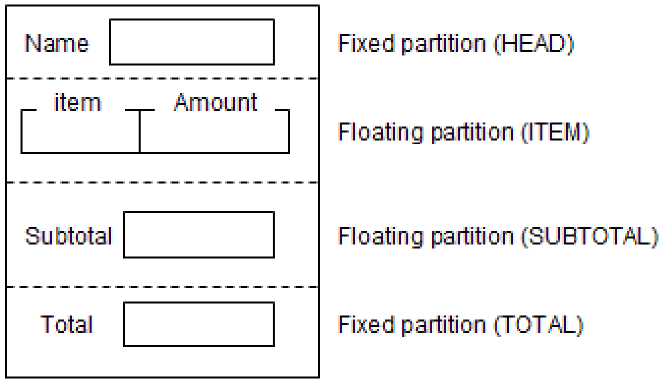
Floating partitions

A floating partition has a print position in a page which is determined by the output sequence. The partition is printed out in the position which has been determined by the execution of WRITE statements.

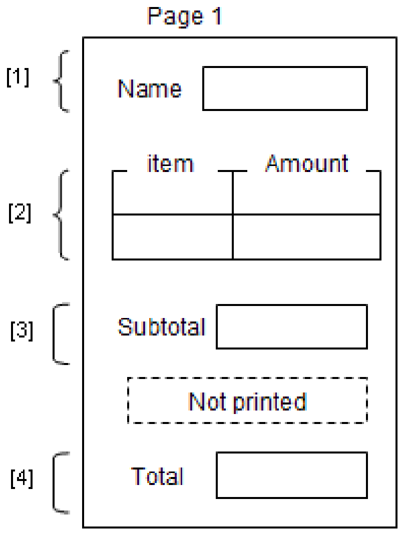
Example

The following shows the WRITE statement of chart records and gives an output example of fixed and floating partitions to be output.

Form descriptor. Estimate (ESTIMATE.PMD)



```
      MOVE   "ESTIMATE" TO form-descriptor-name-area
      MOVE   "HEAD"   TO  item-group-name-area
      WRITE   ESTIMATE                            *> [1]
      MOVE   "ITEM"   TO  item-group-name-area
      WRITE   ESTIMATE                            *> [2]
      WRITE   ESTIMATE                            *> [2]
      MOVE   "SUBTOTAL" TO  item-group-name-area
      WRITE   ESTIMATE                            *> [3]
      MOVE   "TOTAL" TO  item-group-name-area
      WRITE   ESTIMATE                            *> [4]
```



The floating partition is printed out in the WRITE statement execution sequence (or after the number of lines specified by the ADVANCING phrase if any has been fed). The area surrounded by dotted lines is not printed out as the TOTAL partition is printed at the fixed position defined by the form descriptor.

Combination of partitions and line records

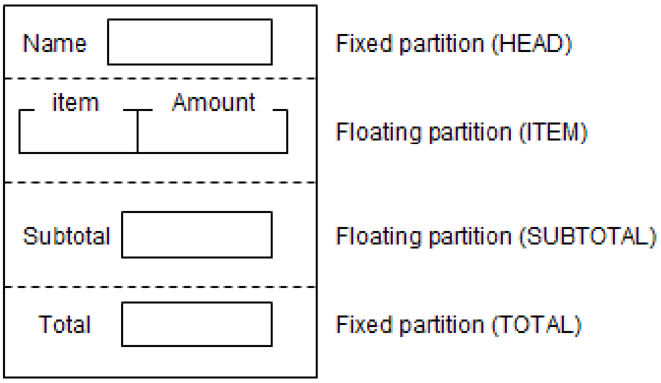
The partitions defined in the form descriptor can be printed out using the chart records. Also, normal line records can be output to any position of the same page.

The partitions and line records are printed out in the WRITE statement execution sequence (or after the number of lines specified by the ADVANCING phrase if any has been fed).

Example

The following shows the WRITE statements specifying a combination of chart records and line records and gives an output example of fixed partitions, floating partitions and line data.

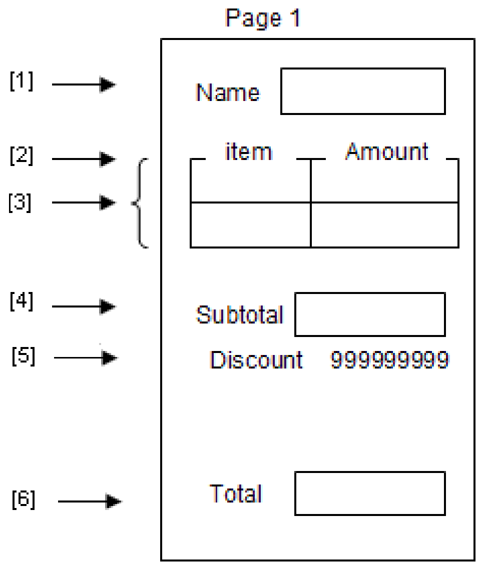
Form descriptor. Estimate (ESTIMATE.PMD)



[line record]

```
01 line-record.
   02 discount PIC 9(9).
```

```
     MOVE  "ESTIMATE" TO form-descriptor-name-area
     MOVE  "HEAD"  TO  item-group-name-area
     WRITE  ESTIMATE                         *>[1]
     MOVE  "ITEM"  TO  item-group-name-area
     WRITE  ESTIMATE                         *>[2]
     WRITE  ESTIMATE                         *>[3]
     MOVE  "SUBTOTAL" TO  item-group-name-area
     WRITE  ESTIMATE                         *>[4]
     MOVE SPACE TO form-descriptor-name-area
     MOVE SPACE TO item-group-name-area
     WRITE line-record                       *>[5]
     MOVE  "ESTIMATE" TO form-descriptor-name-area
     MOVE  "TOTAL" TO  item-group-name-area
     WRITE  ESTIMATE                         *>[6]
```



Note

If a line record or a floating partition is output to the fixed partition print area, the fixed partition to be printed in that position cannot be printed out in the same page. The page is fed and the fixed partition is printed out in the position of the next page.

7.4.2 Program Specifications

This section explains how to write programs that use form descriptors in a print file with a FORMAT clause.

ENVIRONMENT DIVISION

In the ENVIRONMENT DIVISION, write the relation between the function-name and mnemonic-name (print characters are indicated in the program) and define a print file.

Relating the Function-Name to the Mnemonic-Name

To specify a print character with a CHARACTER TYPE clause, relate the function-name indicating the size, form, direction, and space of a print character to the mnemonic-name. For function-name types and type styles, refer to the "NetCOBOL Language Reference."

Defining Print Files with FORMAT clause

Define a print file in a file control entry. Table below lists the information required to write a file control entry.

Table 7.6 Information to be specified in a file control entry

	Location	Information Type	Details and Use of Specification
Required	SELECT clause	File name	Write the name of a file to use in a COBOL program, conforming to the rules of COBOL user-defined words.
	ASSIGN clause	File- reference-identifier	Write a file-identifier, file-identifier literal, or data-name. Use a file reference code to assign a printer information file to be used by Form RTS at execution time.
	FORMAT clause	Data-name	Specify a data-name defined as an 8-byte alphanumeric data item in the WORKING-STORAGE or LINKAGE section. Use this data-name to set the form descriptor name.

	Location	Information Type	Details and Use of Specification
	GROUP clause	Data-name	Specify a data-name defined as an 8-byte alphanumeric data item in the WORKING-STORAGE or LINKAGE section. Use this data-name to set the name of the item group defined in the form descriptor.
Optional	FILE STATUS clause	Data-name	Write the data-name defined as a 2-byte alphanumeric data item in the WORKING-STORAGE or LINKAGE section. The input-output execution result is set for this data-name. For value to be set, see "Appendix B I-O Status List". For detail information, specify a 4-bytes alphanumeric data item.

How you assign a printer information file at execution time depends on whether a file-identifier, file-identifier literal, or data-name is specified for a file-reference-identifier.

What you specify for a file-reference-identifier depends on when the name of the printer information file is determined. Specify a file-identifier literal if the name of the printer information file is determined during COBOL program generation and not changed afterwards. Specify a file-identifier if the name of the printer information file is undetermined during COBOL program generation or it is to be determined every program execution time. Specify a data-name to determine the name of the printer information file in a program.

DATA DIVISION

In the DATA DIVISION, define record definitions and the definitions of data used in the ENVIRONMENT DIVISION.

Defining Records

Define a record in file and record description entries. Table below lists information required to write a file description entry.

Table 7.7 Information to be specified in a file description entry

	Location	Information Type	Details and Use of Specification
Optional	RECORD clause	Record size	Define the size of the printable area.
	CONTROL RECORD clause	Control record name	Specify a control record name.

In a record description entry, line, control, and chart records can be defined. For information about how to define and use line and control records, see "7.3.2 Program Specifications."

A record description statement defined in a chart record can be fetched from form descriptors with a COPY statement with IN/OF XMDLIB specified as a library-name at compile time. For details of the content to be expanded with the COPY statement, see "7.5.4 Program Specifications."

PROCEDURE DIVISION

Execute input-output statements in the following sequence:

1. OPEN statement with OUTPUT specified: Starts printing.
2. WRITE statement: Writes data.
3. CLOSE statement: Stops printing.

OPEN and CLOSE Statements

Use an OPEN statement once at the start of printing and a CLOSE statement once at the end of printing.

WRITE Statement

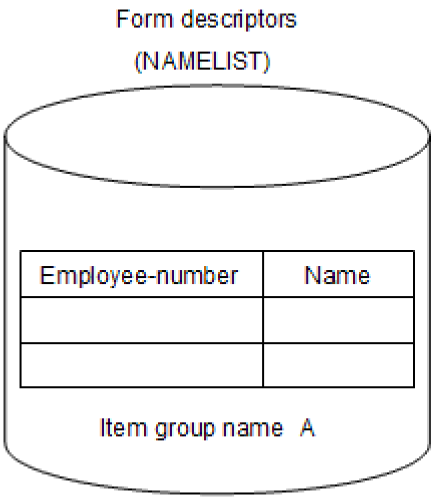
WRITE statements can output: line records, control records, and chart records. The output of these records makes the page either a fixed form page or an irregular form page. In a fixed form page, the layout and chart record are defined by form descriptors and the chart record defined by the form descriptor or the line record can be printed. Form descriptors do not define irregular form pages, therefore only line records can be printed because they behave in the same manner as print files without a FORMAT clause.

When chart records and line records exist together on a fixed form page, the form descriptors defining the chart records should be set in the CONTROL RECORD clause of the file description paragraph.

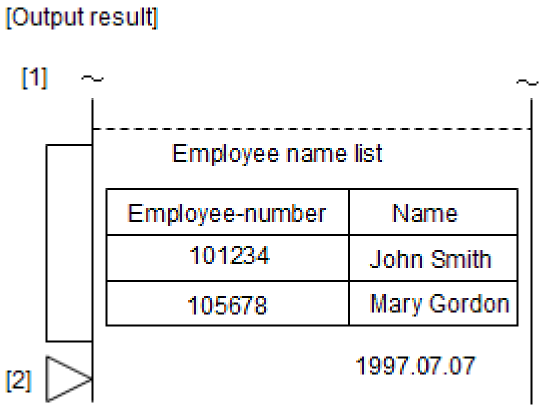
A page becomes an irregular form page, if the form descriptor name in the control record is blank when the OPEN statement is executed. A page is a fixed form page if either the data name specified in the FORMAT clause, or in the control record, is set to the name of a descriptor containing a chart record.

The fixed or irregular form of these pages is maintained on subsequent pages as long as a WRITE statement changing the form of the page is not executed.

Pages can be changed if new form descriptors are added. In this instance, AFTER ADVANCING is specified for a WRITE statement immediately after the output of the control record. See "7.2.2 Program Specifications" for the ADVANCING specification of a WRITE statement.



```
     MOVE  101234 TO employee-number OF  NAMELIST(1).
     MOVE  105678 TO employee-number OF  NAMELIST(2).
     MOVE "John Smith"     TO  name  OF  NAMELIST(1).
     MOVE "Mary Gordon"    TO  name  OF  NAMELIST(2).
     MOVE "NAMELIST" TO form-descriptor-name-area.
     MOVE "A" TO item-group-name-area.
     WRITE NAMELIST  AFTER ADVANCING PAGE.              *>[1]
     MOVE SPACE   TO form-descriptor-name-area.
     MOVE "1997.07.07" TO  print-data  OF line-record.
     WRITE line-record AFTER ADVANCING 3.               *>[2]
```



Note

When executing a WRITE statement without the AFTER ADVANCING PAGE phrase, the printing start line position is not validated. In this case, printing is started from the first printable line of the printer.

Input-Output Error Processing

Refer to "6.6 Input-Output Error Processing".

7.4.3 Program Compilation and Linkage

Compilation

Select the compiler option -m, and specify the folder name of the form descriptors storage file. For more details refer to "3.5.2.12 -m(Specify the FORM descriptor file folder)".

If multiple form descriptors are used and their extensions are different from each other, select the environment variable SMED_SUFFIX and specify the extension. Refer to "1.2.1 Setting Environment Variables" for more details.

The data of encode UTF-32 when output to the printer

It is necessary to input the form descriptor for UTF-32 to the COPY statement when compiling.

Confirm the specification of the above-mentioned compiler option and the environment variable.

When any of the following is specified, a national item becomes the printer output of encode UTF-32.

a. The ENCODING phrase of encode UTF-32 is specified for the COPY statement when the record definition is taken from the form descriptor.

b. The ENCODING phrase of encode UTF-32 is specified for the file description entry when taking it as a subordinate record definition of the file description entry.

c. Encode UTF-32 is specified for a national item.

The relation of strength becomes a.> b.> c.

Linkage

There are no libraries that need to be linked.

7.4.4 Program Execution

To execute a program that uses form descriptors with a print file, FORM RTS or PowerFORM RTS requires a printer information file. For details about how to generate a printer information file, see "7.5.6 Generating Printer Information Files."

The following environments must be set to execute a program that uses form descriptors with a print file:

- Add the folder containing the FORM RTS or the PowerFORM RTS to environment variable PATH.
- Generate a printer information file to be used by FORM RTS or PowerFORM RTS, and assign the file according to the contents of the ASSIGN clause. For information about how to assign a file, refer to "6.7.1 Assigning Files", because it is the same as assigning ordinary files. For details on a printer information file and how to generate it, refer to the FORM RTS online help or PowerFORM RTS online help (pformapi.hlp).

Information

When a printer information file is specified with a relative path name, it is retrieved in the following sequence:

1. Folder set for environment variable MEFTDIR
2. Current folder

Example

The following is an example showing form descriptors used with a print file.

Contents of the ASSIGN clause in a COBOL program:

```
ASSIGN TO PRTFILE
```

Contents of the runtime initialization file:

```
    :
PRTFILE=C:\DIR\MEFPRC
    :
```

Assign a printer information file to the file-identifier specified in the ASSIGN clause of a COBOL program.

Note

To use a form overlay pattern in print files with a FORMAT clause, set the file extensions, the path of the form overlay storage folder, and the form overlay pattern file names, in the printer information file. The values set in the environment variables FOVLDIR and OVD_SUFFIX have no effect.

7.5 Using Presentation Files (Printing Forms)

This section explains how to print forms using a presentation file. The following sample program printing a form using the presentation file function.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. program-name.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
  SELECT file-name
    ASSIGN TO GS-file-reference-identifier
    SYMBOLIC DESTINATION IS "PRT"
    FORMAT IS form-descriptor-name-area
    GROUP IS item-group-name-area
   [PROCESSING MODE IS processing-mode-area]
   [UNIT CONTROL IS special-control-information-area]
   [FILE STATUS IS input-output-status-1 input-output-status-2].
DATA DIVISION.
 FILE SECTION.
 FD file-name.
 COPY form-descriptor-name OF XMDLIB.
 01 chart-record-name.          *> --+ Expanding a COPY statement
    02 data-name-1 … .           *> --+
 WORKING-STORAGE SECTION.
 01 form-descriptor-name-area        PIC X(8).
 01 item-group-name-area             PIC X(8).
[01 processing-mode-area             PIC X(2).]
[01 special-control-information-area PIC X(6).]
[01 input-output-status-1            PIC X(2).]
[01 input-output-status-2            PIC X(4).]
PROCEDURE DIVISION.
    OPEN OUTPUT file-name.
   [MOVE output-specification    TO EDIT-MODE   OF data-name-1.]
   [MOVE highlight-specification TO EDIT-OPTION OF data-name-1.]
   [MOVE output-color            TO EDIT-COLOR  OF data-name-1.]
    MOVE form-descriptor-name TO form-descriptor-name-area.
    MOVE item-group-name TO item-group-name-area.
   [MOVE processing-type TO processing-mode-area.]
   [MOVE control-information TO special-control-information-area.]
    WRITE chart-record-name.
      :
    CLOSE file-name.
    EXIT PROGRAM.
END PROGRAM program-name.
```

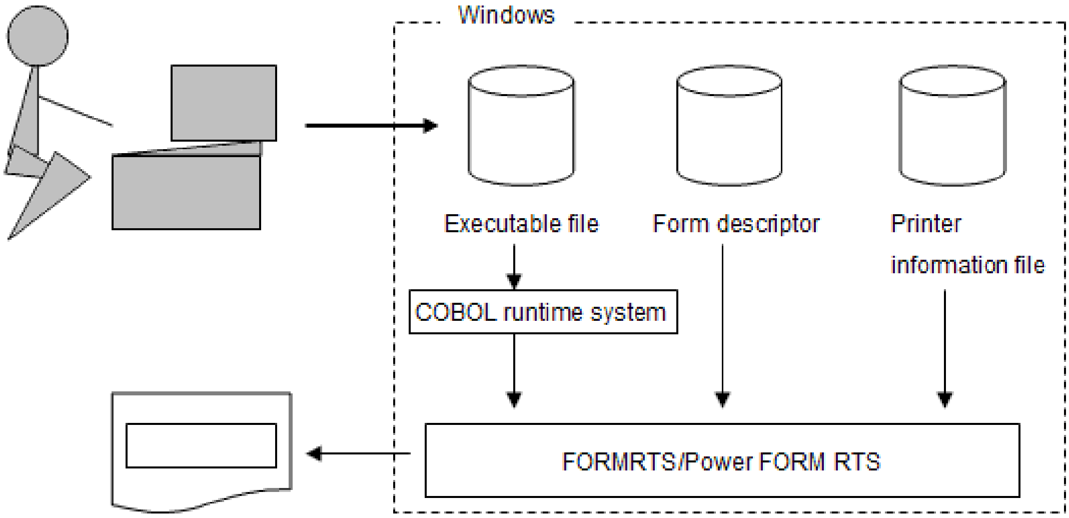
7.5.1 Outline

The presentation file function prints forms in the format defined with FORM or Power FORM (form descriptors). Form descriptors are generated with a screen image.
(FORM is a product that operates by 32 bits Windows. This cannot be installed in 64-bit support environment.)

Data items defined in form descriptors can be included in a COBOL program at compile time by using the COBOL COPY statement. Thus, you do not have to write data item definitions for print forms in a COBOL program.

A form descriptor and FORM RTS are required to use the presentation file function. The relationships between these factors are shown in the figure below.

Use in local environment:



7.5.2 Work Procedures

To print forms with the presentation file module, form descriptors, COBOL source programs, and printer information files are required.

Generate form descriptors and COBOL source programs before compile time, and print information files before execution.

The following is the standard order of work procedures for printing forms with the presentation file module:

1. Generate form descriptors with FORM or Power FORM.
2. Generate COBOL source programs by using a text editor.
3. Compile and link COBOL source programs to generate executable programs.
4. Generate printer information files with a text editor.
5. Execute the executable programs.

7.5.3 Generating Form Descriptors

This section describes the information to specify when creating form descriptors to be used with the presentation file function.

For detailed FORM functions and how to use FROM, refer to "*FORM manual and FORM online help*".

For detailed PowerFORM functions and how to use PowerFORM, refer to "PowerFORM Getting Started" and PowerFORM on-line help.
Table below lists the information to be set up when creating form descriptors.

Table 7.8 Information to be set to generate form descriptors

Information Type		Details and Use of Specification
Required	File name	Specify the name of a file in which form descriptor is stored.
	Definition size	Specify the forms size with the numbers of lines and columns.
	Descriptors format	Specify the format.
	Data item	Specify the data item to set print data. This item name is used as a data-name when writing a COBOL program.
	Item group or Partition	Put one or more items to be printed at a single run of print processing into one item group. This item group name is used when writing a COBOL program.
Optional	Item control field(*1)	Specify a 5-byte item control field if the contents of form descriptors need to be changed with a special register in a COBOL program.

*1 :The item control field is information appended to data items defined in form descriptors, and is classified into three input-output options:

- Sharable (3 bytes)
- Non-sharable (5 bytes)
- None

When a special register in a COBOL program is used, non-sharable (5 bytes) must be specified.

Note

- When creating a form descriptor, use the name to be set or referred to by the COBOL program in the following rules.
    - Specify a form descriptor name using up to eight single-byte alphanumeric characters.
    - Specify a group item name and a partition name using up to six single-byte alphanumeric characters.
    - Specify a group item name by following the syntax rules of COBOL user-defined words.
- A national data item must convert the data of encode UTF-32 and convert the form descriptor for UTF-32 when you output the printer. Please refer to "I.4 CNVMED2UTF32 Command" for the conversion of the form descriptor.

7.5.4 Program Specifications

This section explains how to write programs when printing forms with the presentation file module for each COBOL division.

ENVIRONMENT DIVISION

ENVIRONMENT DIVISION defines a presentation file. In the presentation file, as with defining an ordinary file, write a file control entry in the FILE-CONTROL paragraph of the INPUT-OUTPUT section.

Table below lists the contents to be written in the file control entry. These information values can be determined regardless of the contents of form descriptors generated with FORM or Power FORM.

Table 7.9 Information to be specified in a file control entry

	Location	Information Type	Details and Use of Specification
Required	SELECT clause	File name	Write the name of a file to use in a COBOL program, conforming to the rules of COBOL user-defined words.
	ASSIGN clause	File- reference-identifier	Specify this item in the format of "GS- file-identifier". This file-identifier is the environment variable to set the path name of the printer information file used by the connection product at execution.

	Location	Information Type	Details and Use of Specification
	FORMAT clause	Data-name	Specify a data-name defined as an 8-byte alphanumeric data item in the WORKING-STORAGE or LINKAGE section. For this data-name, specify the name of form descriptors during forms printing.
	GROUP clause	Data-name	Specify a data-name defined as an 8-byte alphanumeric data item in the WORKING-STORAGE or LINKAGE section. For this data-name, specify the item group name or partition name to be output during forms printing.
	SYMBOLIC DESTINATION clause	Specification of the output destination	Specify "PRT".

Table 7.10 Information to be specified in a file control entry (cont.)

	Location	Information Type	Details and Use of Specification
Optional	FILE STATUS clause	Data-name	Write the data-name defined as a 2-byte alphanumeric data item in the WORKING-STORAGE or LINKAGE section. The input-output execution result is set for this data-name. For value to be set, see "Appendix B I-O Status List". For detail information, specify a 4-byte alphanumeric data item.
	PROCESSING MODE clause	Data-name	Specify the data-name defined as a 2-byte alphanumeric data item in the WORKING-STORAGE or LINKAGE section. For this data-name, set in input-output processing type during forms input-output. See "Table 7.11 Input-output processing types and specification values".
	UNIT CONTROL clause	Data-name	Specify the data-name defined as a 6-byte alphanumeric data item in the WORKING-STORAGE or LINKAGE section. For this data-name, set the control information of input-output processing at printing. See "Table 7.11 Input-output processing types and specification values".

Table 7.11 Input-output processing types and specification values

Processing Type	Value	Control Information	
Printer control	"CT"	Form feed	"PAGE"
Partition output	"PW"	Output after feeding nnn lines	"Annn"
		Feed nnn lines after output	"Bnnn"
		Output at line number nnn	"Pnnn"
Line movement output	"FW"	Move backward by nnn lines	"Annn"
		Move forward by nnn lines	"Snnn"

DATA DIVISION

In the DATA DIVISION, write chart record definitions and the definitions of item-names specified in the file control entry.

A record description statement defined in a chart record can be fetched from form descriptors with a COPY statement with IN/OF XMDLIB specified as a library-name. For details of the record description statement to be expanded, see "Program Specifications."

Note

When the EXTERNAL clause is specified for a presentation file, be sure to read the topic "9.2.5.2 Notes on using an external file".

PROCEDURE DIVISION

Like ordinary file processing, use input-output statements for printing forms. Execute input-output statements in the following sequence:

1. OPEN statement with I-O specified: Starts printing.
2. WRITE statement: Outputs data.
3. CLOSE statement: Stops printing.

OPEN and CLOSE Statements

Use an OPEN statement once at the start of printing and a CLOSE statement once at the end of printing.

WRITE Statement

One WRITE statement prints one form.

The name of form descriptors used for printing must be set for the data-name specified in the FORMAT clause before executing a WRITE statement.

With a WRITE statement, data items in an item group set for the data-name specified in a GROUP clause are eligible for printing. By setting a value in a special register before executing a WRITE statement, the data item attributes can be changed. For details about how to use a special register, see "7.1.7 Special Registers".

Input-Output Error Processing

Refer to "6.6 Input-Output Error Processing".

7.5.5 Program Compilation and Linkage

Compiling

Select compiler option -m, and specify the path name to the folder of the file containing the form descriptors. Refer to "3.5.2.12 -m(Specify the FORM descriptor file folder)".

If multiple file descriptors are used and if their extensions differ between them, specify an extension in environment variable SMED_SUFFIX. Refer to "1.2.1 Setting Environment Variables" for more details.

The data of encode UTF-32 when output to the printer

It is necessary to input the form descriptor for UTF-32 to the COPY statement when compiling.

Confirm the specification of the above-mentioned compiler option and the environment variable.

When any of the following is specified, a national item becomes the printer output of encode UTF-32.

a. The ENCODING phrase of encode UTF-32 is specified for the COPY statement when the record definition is taken from the form descriptor.

b. The ENCODING phrase of encode UTF-32 is specified for the file description entry when taking it as a subordinate record definition of the file description entry.

c. Encode UTF-32 is specified for a national item.

The relation of strength becomes a.> b.> c.

Linking

There are no libraries that need to be linked.

7.5.6 Generating Printer Information Files

This section describes how to print forms with the presentation file function. For details on printer information files and how to generate them, refer to the FORM RTS online help or PowerFORM RTS online help (pformapi.hlp).

Table below lists the information to enter in a printer information file.

Table 7.12 Information to be entered in a printer information file

Information Type	Details and Use of Specification
PRTDRV	Specify the device name of the output printer device.
MEDDIR	Specify the path name to the folder containing form descriptors.
MEDSUF	Specify the extension of form descriptor files. When the specification of the extension is omitted, the FORM RTS or the PowerFORM RTS default value is used.

7.5.7 Program Execution

The following environments must be set to execute a program that prints forms with the presentation file module:

- Add the folder containing the FORM RTS or the PowerFORM RTS to the PATH environment variable.
- Set the name of the printer information file with the file-identifier as an environment variable name.

Information

When a printer information file is specified with a relative path name, it is retrieved in the following sequence:

1. Folder set for environment variable MEFTDIR
2. Current folder

Chapter 8 Using Screens

This chapter explains how to display screens, and how to enter data from the displayed screens. This chapter also describes how to use presentation files, and how to use the screen handling function.

COBOL programs allow you to display a screen where you can enter data. This is referred to as screen input-output. A screen handling function is provided.

8.1 Using the Screen Handling Function

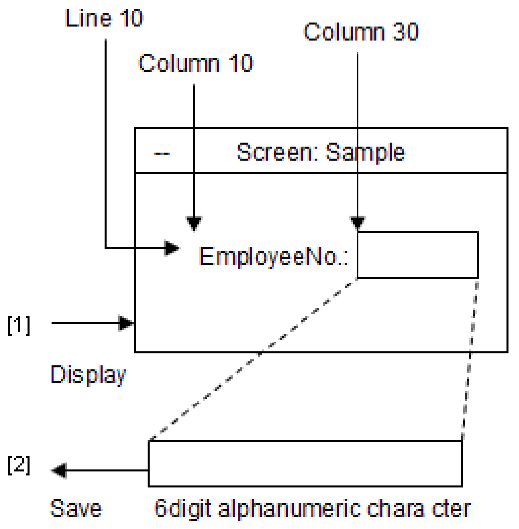
This section explains how to perform screen input-output using the screen handling function. A sample program using the screen handling function is provided for your reference.

8.1.1 Outline

The screen handling function displays a screen with a DISPLAY statement and inputs data from a screen with an ACCEPT statement.

The screen layout is defined in a screen data description entry in the SCREEN section in the DATA DIVISION. A screen item defined in the SCREEN section is arranged on the screen with the line and column numbers.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. SAMPLE.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 number-1   PIC X(6).
  01 employee-record.
    02 employee-no   PIC X(6).
    02               PIC X     VALUE ":".
    02 employee-name PIC X(12).
  SCREEN SECTION.
  01 screen-item-1  BLANK SCREEN.
    02                 VALUE "Employee No. :" LINE 10 COLUMN 10.
    02 screen-item-11 LINE 10  COLUMN 30
                      PIC X(6) USING number-1 AUTO.
 PROCEDURE DIVISION.
     DISPLAY  screen-item-1.     *>[1]
     ACCEPT   screen-item-1.     *>[2]
     MOVE  number-1 TO employee-no.
```



8.1.2 Screen Windows

One screen window to perform screen input-output with the screen handling function is generated per run unit. Screen windows are generated by the first ACCEPT or DISPLAY statement executed, and closed when the run unit is terminated.

Although the size of the logical screen of this window is normally 25 lines and 80 columns, it can be changed with environment variable @ScrnSize.

You can specify whether to close this window automatically or close after message confirmation with environment variable @WinCloseMsg.

The following table lists window attributes for screen handling that can be changed at execution time. For details about how to specify runtime environment information, see "5.3 Setting Runtime Environment Information".

Table 8.1 Changing window attributes for screen handling

Attribute	Runtime Environment Information	Setting Value	Meaning
Automatic window closing	@WinCloseMsg (*1)	ON	Closes the window after displaying a message.
		OFF	Closes the window without displaying a message.
Window size	@ScrnSize	(m,n)	Specifies the size of a logical screen with the number of columns (m) and lines (n).

*1 : This runtime environment information is also valid for console windows used with the ACCEPT/DISPLAY function.

8.1.3 User-Defined Function Keys

The screen handling function invalidates the input of specific function keys and corresponding processing in programs to specified function keys.

To use a function key when executing a program, use the key definition file.

Key definition file

If the key definition file is used, the end of screen data entry can be specified by a predefined function key. And the entry operations of numeric data items and numeric edited data items can be changed.

The key definition file specifies the validity of predefined key that ends on-screen data entry. When a valid data entry end key is used, the on-screen data entry is ended. The information about status keys 1 and 2 (defined as the key definition file in the data entries) is returned to the data entry specified by the CRT STATUS clause of SPECIAL-NAMES paragraph. If the data entry end key is made invalid, the end of data entry is not considered even when the key is used.

For specification of key definition files, see "5.4.1.40 @CBR_SCR_KEYDEFFILE(Specify key definition file for screen handling)".

If the key definition file name is omitted, the on-screen data entry is ended when the Enter key is used.

Description form of key definition file

Entries in the key definition file are defined as follows.

```
[COBOL.KBD]
ESC=11000
F01=01001
SHIFT+F01=11020
ENTER=2
[COBOL.ACCPMODE]
numeric_input=MODE2
```

Key entry definition (section name: [COBOL.KBD])

In the key definition, the following of each key are specified.

Active/Inactive

Value returned to CRT-STATUS-KEY-1

Value returned to CRT-STATUS-KEY-2

As a result, the user becomes to be able to customize how to receive the key pressing on the screen windows arbitrarily.

```
Key name = XYZZZ
   X: Active/Inactive flag ('1' is Active, '0' is Inactive)
   Y: Value of CRT-STATUS-KEY-1

 For example,
 '1' is indicated the CRT-STATUS-KEY-1 is 1,
 '2' is indicated the CRT-STATUS-KEY-1 is 2.
 ZZZ: Value of CRT-STATUS-KEY-2 ('000' to '999')

 ENTER=2
   When the ENTER key is used, it is considered to be the Tab key.
```

In the example, when the escape key is pressed while inputting data from the screen Windows, "1" is returned to CRT-STATUS-KEY-1, and 0 (Zero) is returned to CRT-STATUS-KEY-2. The pressing becomes invalid when the F01 key is pressed, and the data input is continued.

The section name of the key definition is [COBOL.KBD] fixation.

The ENTER key is treated as the TAB key when the following is specified in the key definition file.

```
ENTER=2
```

When this specification is done, it is necessary to set the key to end at least one input in the key definition file to complete the input operation.

Numeric and Numeric Edited Data entry method (section name: [COBOL.ACCPMODE])

The behavior when entering numeric and numeric edited data items can be controlled by specifying the following:

$$\text{numeric\_input=} \left\{ \begin{array}{l} \underline{\text{MODE1}} \\ \text{MODE2} \end{array} \right\}$$

MODE1: Ordinary entry. Automatic digit adjustment at the decimal point is not performed.

MODE2: Fixed decimal point position entry. Digit adjustment is performed when entering the decimal point.

Example

The differences caused by entry methods in the resultant data entered are shown by the following examples.

1) For PIC 9999 ([^] indicates the position of cursor)

	MODE1	MODE2

Initial display	`0000`
	`^`
		`0000`
		`^`

Enter "1"	`1000`
	` ^`
		`1000`
		` ^`

Enter "2"	`1200`
	`  ^`
		`1200`
		`  ^`

Enter "."	`1200`
	`  ^`
		`0012`

2) For PIC ZZ99.99 ([_] indicate a blank and [^] indicate the position of cursor)

	MODE1	MODE2
Initial display	_______ ^	__00.00 ^
Enter "1"	1______ ^	__10.00 ^
Enter "2"	12_____ ^	-----__12.00 ^
Enter "3"	123____ ^	_123.00 ^
Enter "4"	1234___ ^	1234.00 ^
Enter "5"	12345__ ^	1234.50 ^
Enter "6"	123456_ ^	1234.56 ^
Enter the Enter key	3456.00	1234.56

Please refer to the sample key definition file included in the product for the method of describing the key name.

Note

- The value of CRT-STATUS-KEY-2 used can be from '000' to '255'. When the value more than this is used as an effective key, the range which can be returned to the CRT STATUS phrase will be exceeded. Results will be undefined.
- If ENTER=2 is specified, another entry end key must have been set for termination of on-screen data entry. If ENTER=2 is not specified, the ENTER key can be used as the entry end key. A value other than 2 (Tab key) cannot be assigned to the ENTER key.
- If the key definition file has multiple [COBOL.KBD] sections or multiple [COBOL.ACCPMODE] sections, the first encountered section is made valid. If multiple information sets having the same key name exist, the information of the first encountered key name is made valid.
- The description of key definition file cannot have a space.
- The description of key definition file is case-sensitive.
- If the description of key definition file begins with a semicolon (;), it is considered to be a comment line.
- If a key other than the "Fxx" function key is set to invalid in the key definition file, the function of this key is executed when it is pressed.
- If a key is not specified in the key definition file, the default value of 01999 is used instead.

8.1.4 Program Specification

This section explains how to write programs using the screen handling function for each COBOL division.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. program-name.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SPECIAL-NAMES.
     [CURSOR     IS data-name-1]
     [CRT STATUS IS data-name-2].
 DATA DIVISION.
  WORKING-STORAGE SECTION.
 [01 data-name-1.]
   [02 line-number    PIC 9(3).]
   [02 column-number  PIC 9(3).]
 [01 data-name-2.]
   [02 status-key-1   PIC 9.]
   [02 status-key-2   PIC 9.]
   [02                PIC X.]
  SCREEN SECTION.
 [01 screen-item-1 ...]
 PROCEDURE DIVISION.
     DISPLAY  screen-item-1 ...
     ACCEPT   screen-item-1 ... [ON EXCEPTION ...].
 END PROGRAM program-name.
```

ENVIRONMENT DIVISION

The following information can be entered in the SPECIAL-NAMES paragraph:

- Data items for setting or receiving cursor positions in the CURSOR clause.
- Data items for receiving screen input-output status in the CRT STATUS clause. The following table lists values set for these data items.

Table 8.2 Screen input-output status values

Status Key 1 (First Character)	Status Key 2 (Second Character)	Meaning
"0"	"0"	A termination key was entered by the operator.
	"1"	The last item was entered.
"1"	X"00"-x"FF"	A user-defined function key was entered. (A function key number is set for status key2.) (*1)
"2"	X"00"-x"FF"	A system-defined function key was entered. (A function key number is set for status key2.) (*2)
"9"	X"00"	No input item was found. (Error)

*1 A user-defined function key is a function key defined in the key definition file.
*2 When the key definition file is used, the function given by the key definition file is assigned to Status key 2. If the function key is not defined, the function key number is returned to Status key 2.

DATA DIVISION

The SCREEN section is written at the end of the DATA DIVISION.

Define the screen item in the screen section by using the screen data entry.

The SCREEN section has literal, input, output, and update items. These items are classified based on coding of the screen data description entry.

The following table lists relationships between screen item attributes and clauses that can be specified in the screen data description entry. For information about how to code the screen data description entry, refer to the "COBOL Language Reference."

Data items defined in the BASED-STORAGE section cannot be specified in the SCREEN section.

Table 8.3 COBOL clauses specified for screen items

Purpose		COBOL Clause	Screen Item Attribute *1				*2	
			L	I	O	U	G	E
Emphasis	Display in high intensity	HIGHLIGHT clause	o	o	o	o	x	o
	Display in low intensity	LOWLIGHT clause	-	-	-	-	x	-
	Display with blinking	BLINK clause	-	-	-	-	x	-
	Display with underline	UNDERLINE clause	o	-	o	o	x	o
Color	Specify background color	BACKGROUND-COLOR clause	o	o	o	o	o	o
	Specify foreground color	FOREGROUND-COLOR clause	o	o	o	o	o	o
	Reverse background and foreground colors	REVERSE-VIDEO clause	o	o	o	o	x	o
Sound	Sound an audio tone	BELL clause	o	-	o	o	x	o
Expression format	Display a blank for zero	BLANK WHEN ZERO clause	x	-	o	o	x	o
	Specify justification	JUSTIFIED clause	x	o	o	o	x	o
	Specify operation sign position	SIGN clause	x	o	o	o	o	O
Display method	Specify full-screen erasure	BLANK SCREEN clause	o	-	o	o	o	o
	Specify partial screen erasure	ERASE EOS clause	o	-	o	o	x	o
	Specify full-line erasure	BLANK LINE clause	o	-	o	o	x	o
	Specify partial line erasure	ERASE EOL clause	o	-	o	o	x	o
	Specify non-display status	SECURE clause	x	o	x	x	o	o
Position	Specify column number	COLUMN NUMBER clause	o	o	o	o	x	o
	Specify line number	LINE NUMBER clause	o	o	o	o	x	o
Input	Specify input mode	FULL clause	x	o	-	o	o	o
	Specify input mode	REQUIRED clause	x	o	-	o	o	o
Others	Automatic cursor skip	AUTO clause	x	o	-	o	o	o
	Specify general characteristics	PICTURE clause	x	o	o	o	x	o
	Specify expression format	USAGE clause	x	o	o	o	o	o
	Specify literal item	VALUE clause	o	x	x	x	x	o

o : Can be specified

- : Can be specified, but not validated

x : Cannot be specified

*1 L: Literal, I: Input, O: Output, U: Update

*2 Define G (Group) as a group item, and E (Elementary) as an elementary item.

Note

If the same data items are specified in the FROM clause and TO clause, then the item assumed to be an update item.

PROCEDURE DIVISION

To display a screen, use a DISPLAY statement with a screen item defined.

After executing a DISPLAY statement, a screen specified in the screen item is displayed on the display unit. To input data from the screen, use an ACCEPT statement that specifies the screen item. Upon executing the ACCEPT statement, data can be input from the screen on the display unit. After input, by referring the values of screen input status set for data-names specified in the CRT STATUS clause in the SPECIAL-NAMES paragraph, appropriate processing can be selected.

8.1.5 Program Compilation and Linkage

No compiler options or additional libraries are required.

8.1.6 Program Execution

- To define a function key, specify the key definition file by the environment variable @CBR_SCR_KEYDEFFILE as shown in the following example. Refer to "5.4.1.40 @CBR_SCR_KEYDEFFILE(Specify key definition file for screen handling)" for detail.

Example

Case of using the key definition file

```
@CBR_SCR_KEYDEFFILE=SAMPLE.KBD
```

Contents of SAMPLE.KBD

```
[COBOL.KBD]
F01=11001
F02=11002
F03=11003
F04=11004
F05=01005
F06=01006
F07=01007
F08=01008
F09=01009
F10=01010
F11=01011
F12=01012
F13=11013
F14=11014
F15=11015
F16=11016
```

PF1 to PF4 and PF13 to PF16 are valid, but PF5 to PF12 are invalid.

- To change a window attribute, define the runtime environment information listed in "Table 8.1 Changing window attributes for screen handling". Then, execute the program.

Note

- The ACCEPT/DISPLAY statement and screen handling function operate windows with an ACCEPT or DISPLAY statement, but use different windows.
- The window used by the screen handling function cannot be made into an icon.
- The size of a logical screen must not be smaller than the sizes of data items defined in the screen items.
- When the data item specified for a DISPLAY or ACCEPT statement is larger than the number of columns of the logical screen, an error message (Warning) is output and the data is displayed on the line, truncated at the end of the line, with no wrapping to the next line.
- Input items are always underlined.
- An error occurs when a program is executed if a logical screen set for runtime environment information @ScrnSize is (number of columns + 1) * number of lines > 16,250.
- When the Close button is clicked in the screen window, or the Close command is selected from the pop-up menu in the screen window, a dialog box confirming whether to forcibly terminate the program is displayed. If forced termination is selected in this dialog box, the program ends.
- In this system, when the following keys are defined as disabled, the control of the keys is left to the system:
    - F10 (displaying the system menu)
    - ALT + F4 (terminating an application)
    - ALT + F6 (changing the active window)
- In this system, the control of pressing the following keys is left to the system regardless of whether they are enabled or disabled in the key definition file:
    - CTRL + ESC
    - ALT + ESC
    - CTRL + TAB
    - ALT + TAB

Chapter 9 Calling Subprograms (Inter-Program Communication)

This chapter explains how to call programs from other programs often referred to as inter-program communication. It covers the following topics:

- Overview of Calling Relationship
- Calling COBOL Program from COBOL Program
- Coupling with C Language Program
- Calling COBOL Program from Visual Basic

9.1 Outline of Calling Relationships

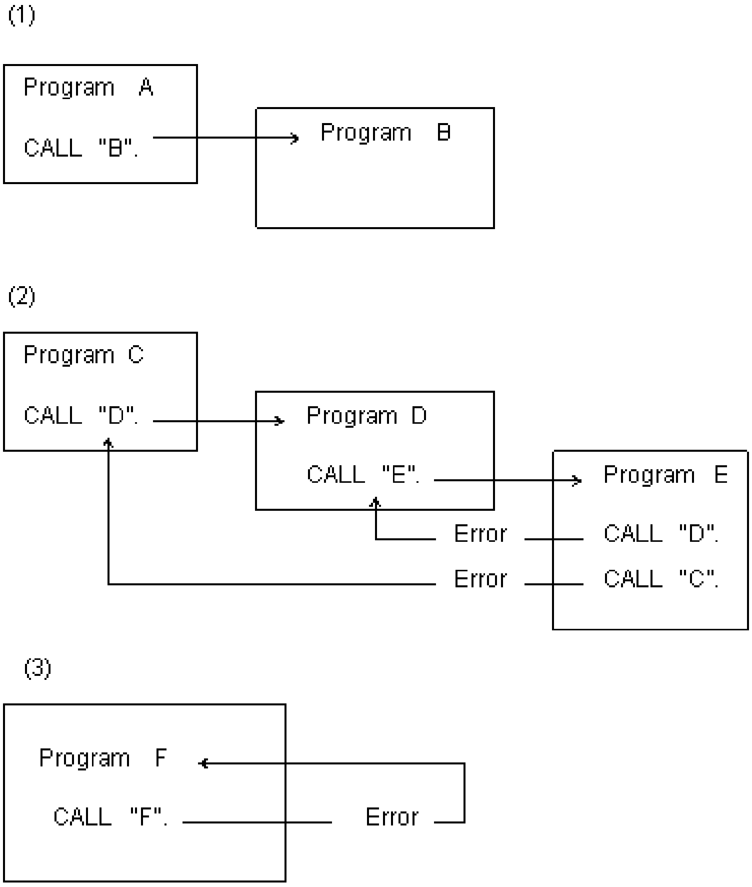
This section outlines program calling relationships and dynamic program structure.

9.1.1 Calling Relationship Forms

A COBOL programs can call other programs or can be called from other programs, as shown in "Figure 9.1 Calling relationship forms", form (1). This form is supported even if the programs are coded in other languages.

A COBOL program without a recursive attribute, however, cannot be called recursively, as shown in section (2) in Figure below. A COBOL program without a recursive attribute cannot call itself, as shown in section (3) in Figure below.

Figure 9.1 Calling relationship forms



9.1.2 COBOL Inter-Language Environment

The COBOL has runtime environment and run unit. This section explains that.

Runtime Environment and Run Unit

Generally, the runtime environment of COBOL programs is opened when needed and closed when no longer needed or at the end of a COBOL run unit.

However, when multithread programs are run, timing of closing of runtime environment is different. See "19.3.1 Execution Environment and Run Unit".

The COBOL run unit refers to the duration starting from when control is passed on to the COBOL main program and ending when it returns to the calling side (Figure below (a)) or until STOP RUN statement is executed (Figure below (b)). An exception is when calling COBOL programs from programs in other languages ("other language programs").

The COBOL main program refers to a COBOL program to which the control is passed on first during a COBOL run unit.

When calling COBOL programs from other language programs, call JMPCINT2 before calling the initial COBOL program, and call JMPCINT3 after calling the final COBOL program.

JMPCINT2 is a subroutine for processing initialization of COBOL programs. JMPCINT3 is a subroutine for closing the runtime environment of the COBOL programs.

If a COBOL program is called from an other-language program without calling JMPCINT2, the COBOL program called becomes the COBOL main program. Therefore, the opening and closing of the COBOL system occurs every time the COBOL program is called, greatly degrading execution efficiency (Figure below (c)).

However, by calling JMPCINT2, the period until JMPCINT3 is called can be the COBOL run unit and the execution environment is not closed until JMPCINT3 is called. (Figure below (d))

For information on how to call JMPCINT2 and JMPCINT3, see "G.2 Subroutines Used to Link to Another Language".

Processing Performed while the Runtime Environment is Opened or Closed

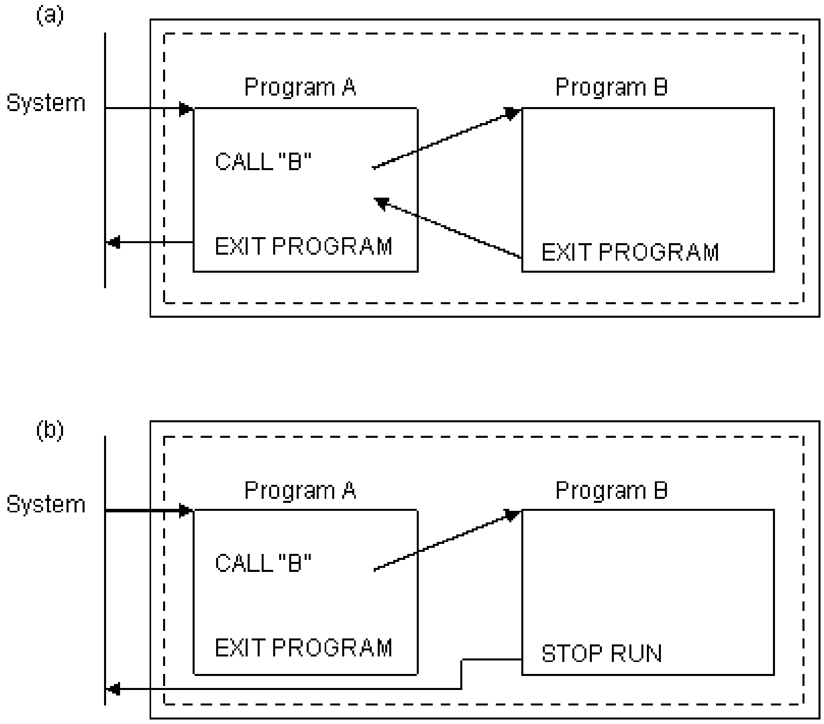
When the runtime environment is opened, information on executable initialization files required for COBOL programs to run is fetched. When the runtime environment is closed, the resources used by COBOL programs are freed.

The processing performed when the runtime environment is closed includes closing the files used by the ACCEPT/DISPLAY function, closing open files, closing external files, freeing external data, freeing factory objects, and freeing object instances left unreleased.

For instance, when programs are used as shown in (c) in the figure below, the runtime environment of programs A and B differs from that of program C. Therefore, the external data for programs A and B is located at a different area from that for program C. In addition, the caution given below applies to this kind of usage. Therefore, use programs as shown in (d) in the figure.

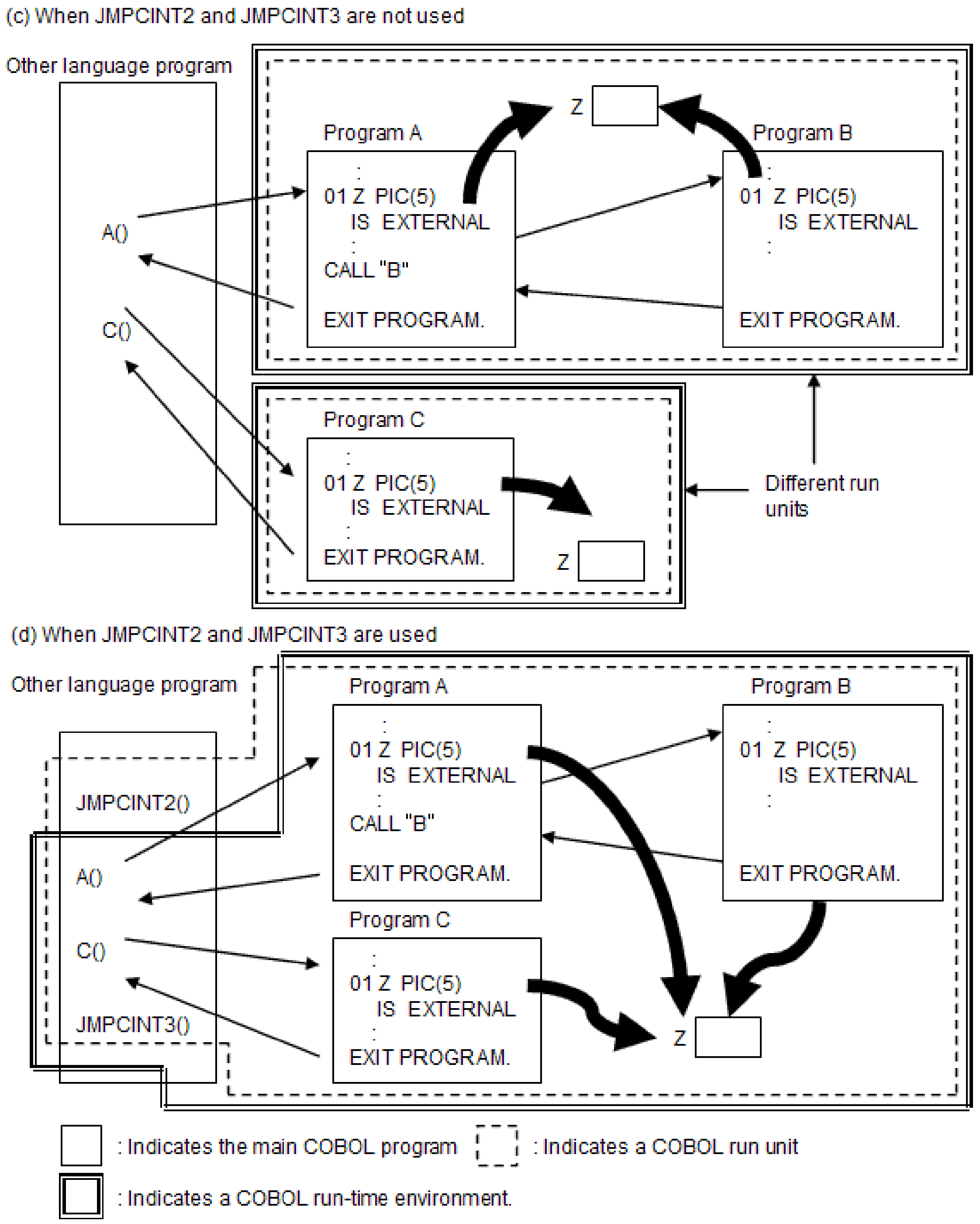
COBOL Programs Only

Figure 9.2 COBOL run units



Calling COBOL Programs from Other Language Programs

Figure 9.3 (cont.) COBOL run units



(c) When JMPCINT2 and JMPCINT3 are not used

(d) When JMPCINT2 and JMPCINT3 are used

Note

In the following case, do not call the COBOL run unit several times from other language program:

- When using external data or external file
- If unconditional close is made to an opened file
- When STOP RUN statement is executed with other than the COBOL main program
- When using the object-oriented programming function

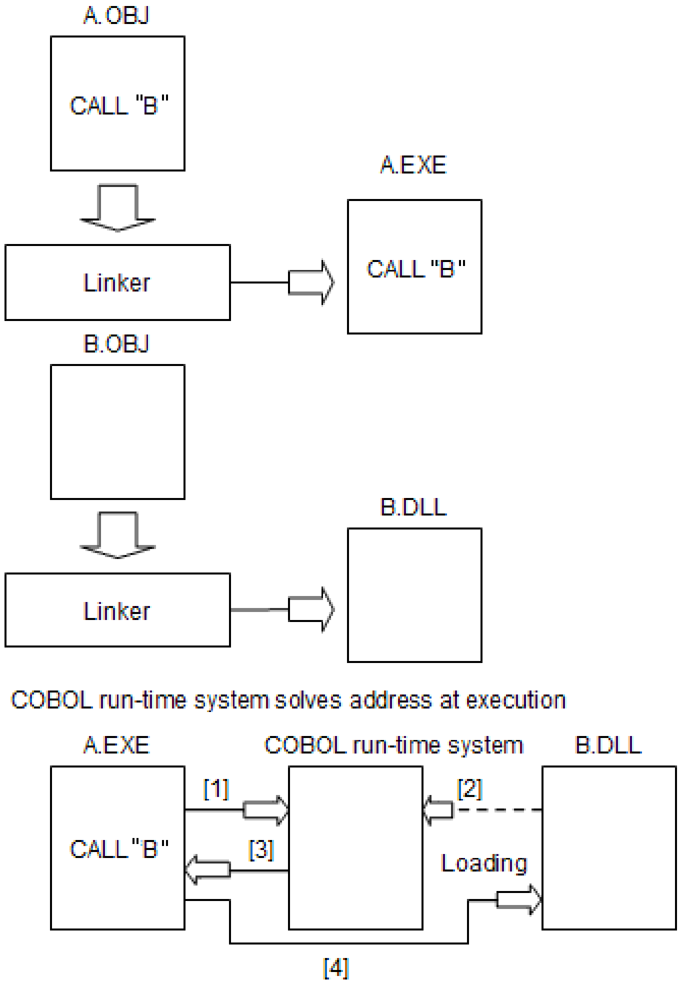
9.1.3 Dynamic Program Structure

This section describes the features of dynamic program structure, subprogram entry information and cautions.

9.1.3.1 Features of Dynamic Program Structure

If dynamic program structure is used, a subprogram is loaded to virtual memory by the COBOL runtime system when the subprogram is actually called by a CALL statement. Moreover, once loaded, this subprogram can be deleted from virtual memory with a CANCEL statement. This achieves faster startup of the executable file than the simplified structure or dynamic link structure where all the subprograms are loaded at the startup of the executable file. This simultaneously saves virtual and actual memory use. Calling of subprograms, however, becomes slower as it is made via the COBOL runtime system. For information on how to use dynamic program structure of multithread, see "19.10.3 Dynamic structure".

Figure 9.4 Dynamic Program Structure



Description of the figure:

[1] to [4] shows the processing order.

- [1]: COBOL runtime system is called.
- [2]: COBOL runtime system loads Program B.
- [3]: Returns to Program A.
- [4]: Branches to Program B.

To execute programs of dynamic structure, subprogram entry information is required. This information is not required, however, if the DLL file name of the subprogram is in the form "Program name. DLL". Consequently, it is recommended to create a DLL for each subprogram, and name the file "Program name. DLL".

Users of this program structure should fully understand the overall structure before implementing it. (Check "9.1.3.3 Cautions" below.)

9.1.3.2 Subprogram Entry Information

Entry information is necessary if the structure of a program to be run is of dynamic structure. For the specifying format of entry information, refer to "5.4.2 Entry Information for Subprograms".

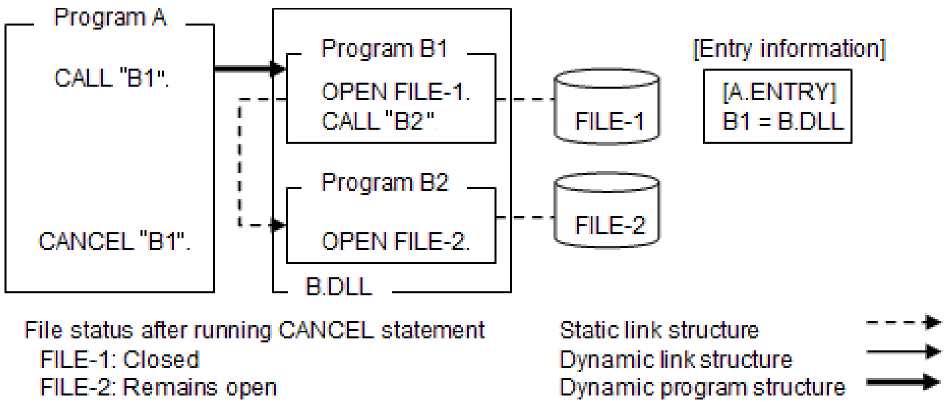
9.1.3.3 Cautions

This section describes cautions required when using dynamic structure.

- Apart from the runtime initialization program, the status of programs that are called, by the CALL statement, is the same as the status when control was returned from the previous call. However, if a CALL is made after the execution of a CANCEL statement, the program returns to its initial status.
- If a program is called simultaneously by dynamic program structure and dynamic link structure, you may not be able to delete the program from virtual memory using the CANCEL statement.
- If an identifier is specified in a CALL statement, the maximum valid length for the program name contained in the identifier is 255 bytes - starting from the first byte of the identifier. Characters after byte 256 are disregarded. Also, leading spaces before the program name are disregarded.
- When executing the CANCEL statement to the program using database, note the following point.
  When using the remote database access function (ODBC), the cursor that has been opened in a subprogram specified by the CANCEL statement will be closed, and the SQL statement reserved by the PREPARE statement will be released. However, when using the precompiler, the cursor will remain opened, and the SQL statement remains reserved by the PREPARE statement even after the CANCEL statement. To prevent this situation, close the cursor, and release the SQL statement reserved by the PREPARE statement before executing the CANCEL statement.
- When using the CANCEL statement in an environment in which a dynamic program structure and simplified or dynamic link structure coexist, note the following:
  When the CANCEL statement is executed, the DLL is removed from virtual memory. Accordingly, the file or cursor opened using a subprogram that is called using a subprogram specified in the CANCEL statement may be left open. Since operation in this event is not guaranteed, be sure to close the file or cursor opened by the subprogram before the CANCEL statement is executed.
- Figures below illustrate the effects of CANCEL statements on opened files.

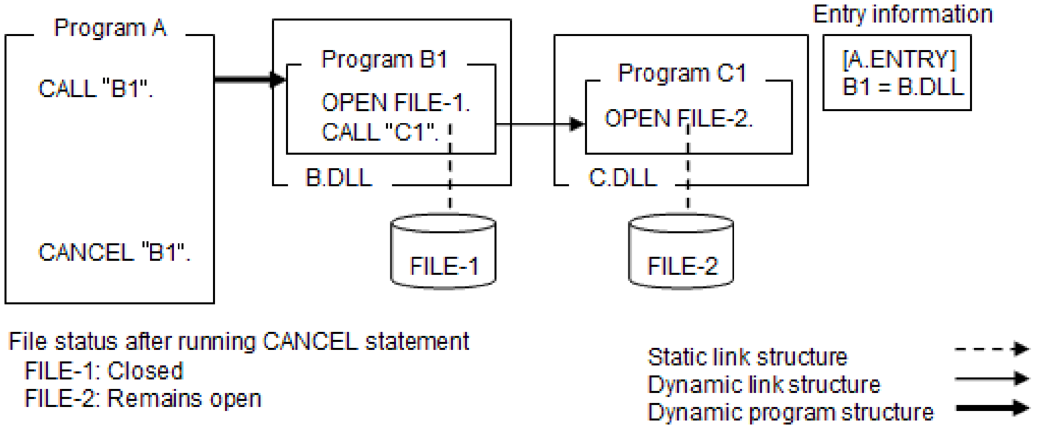
  When a subprogram in the simplified structure has been called by the subprogram to be canceled:

Figure 9.5 Effects of CANCEL when using a static link structure



When a subprogram in the dynamic link structure has been called by the subprogram to be canceled:

Figure 9.6 Effects of CANCEL when using a dynamic link structure



Information

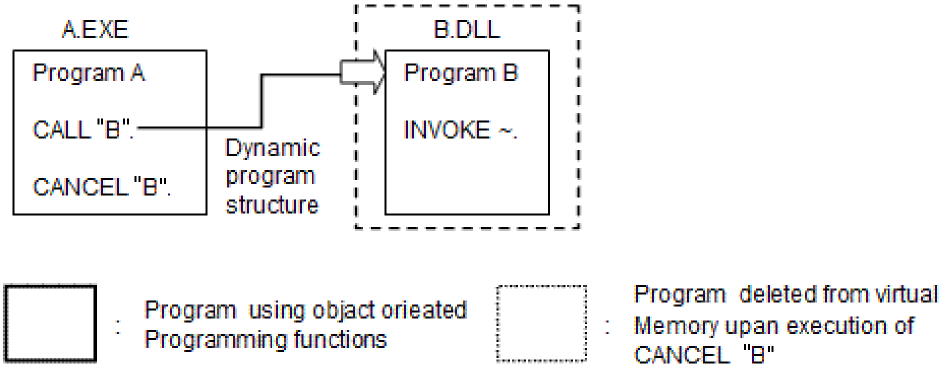
In the above example, B.DLL and C.DLL are loaded when "CALL "B1"" is executed, and B. DLL and C. DLL are deleted from the virtual memory when "CANCEL "B1"" is executed.

- Do not use CANCEL to delete programs that use object-oriented programming functions. The following examples illustrate some situations where this guideline applies.

Example

Since Program B, which uses object-oriented programming functions, would be deleted from virtual memory upon execution of "CANCEL "B"", this CANCEL statement cannot be used.

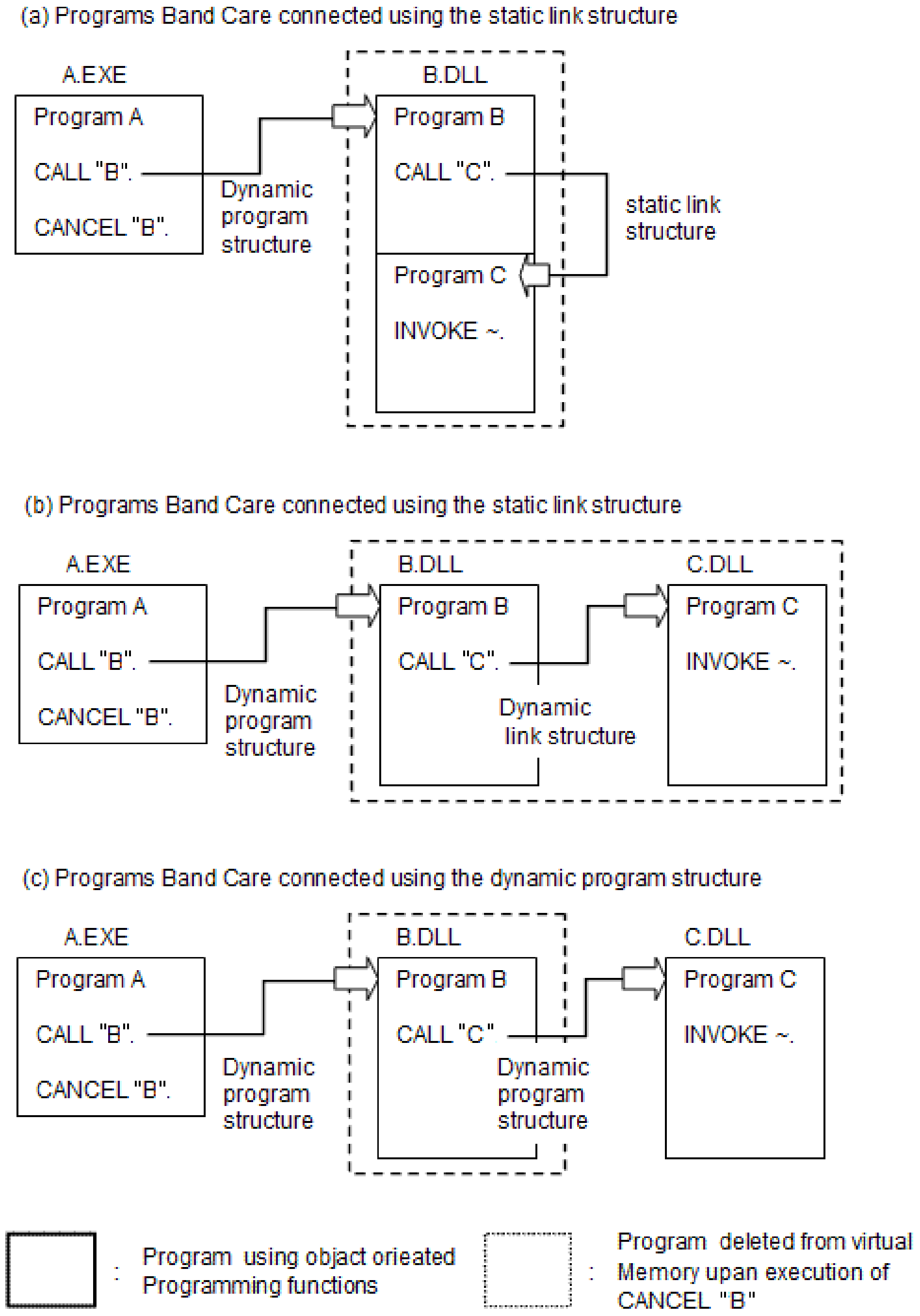
Figure 9.7 Cannot CANCEL a program using OO functions - simple case.



Program C uses OO programming functions, and is called by Program B. In diagram (a) of Figure below, Programs B and C are connected using a static link structure, and in diagram (b) they are connected using a dynamic link structure. In these cases, (a) and (b), program C is deleted from virtual memory when program B is cancelled, so the CANCEL statement is not valid with these program structures.

However, in this case, the CANCEL statement can be used if Programs B and C are connected using dynamic program structure as illustrated in diagram (c).

Figure 9.8 Avoiding canceling programs that use OO functions.



(a) Programs Band Care connected using the static link structure

(b) Programs Band Care connected using the static link structure

(c) Programs Band Care connected using the dynamic program structure

9.2 Calling COBOL Programs from COBOL Programs

This section explains how to call other COBOL programs (subprograms) from COBOL programs (calling programs).

9.2.1 Calling Method

To call subprograms from COBOL programs, use the CALL statement specifying the subprogram name. There are two methods of calling programs, depending on whether the subprogram name is known when the program is created or is determined when the program is executed.

- If the subprogram name is known when the program is created

  Use the program name literal to specify the program name directly in the CALL statement.

- If the subprogram name is determined when the program is executed

  Specify a data name in the CALL statement and move the program name to the data name before executing the CALL statement.

  Calling programs by specifying data names in the CALL statement makes the program structure dynamic between calling programs and subprograms.

  For details of program structure, refer to "4.3 Program Structure".

9.2.2 Secondary Entry Points

The beginning of a program procedure is the primary entry point, and an entry point set in the middle of a procedure is a secondary entry point.

Executing the CALL statement with a program name specified executes a subprogram from the primary entry point.

To execute a subprogram from a secondary entry point, specify the name of the secondary entry point in the CALL statement the same way you specify the program name.

To set a secondary entry point in a COBOL program, write an ENTRY statement. When a program is sequentially executed, the ENTRY statement is skipped. An ENTRY statement cannot be written in internal programs.

9.2.3 Returning Control and Exiting Programs

To return control from subprograms to calling programs, execute the EXIT PROGRAM statement. When the EXIT PROGRAM statement is executed, control returns immediately after the CALL statement is executed by the calling program.

To quit execution of all COBOL programs, execute the STOP RUN statement. When the STOP RUN statement is executed, control returns to the calling source of the COBOL main program.

9.2.4 Passing Parameters

Parameters can be passed between calling programs and subprograms.

In a calling program, write data items defined in the FILE, WORKING-STORAGE, or LINKAGE sections in the USING phrase of the CALL statement. In a subprogram, write data-names to receive parameters in the USING phrase of the PROCEDURE DIVISION header or ENTRY statement.

The order of data-names entered in the USING phrase of the CALL statement of the calling program corresponds to that of data names written in the USING phrase of the called subprogram. Data names need not be the same between the calling program and subprogram. The attribute, length, and number of corresponding data items, however, should be the same.

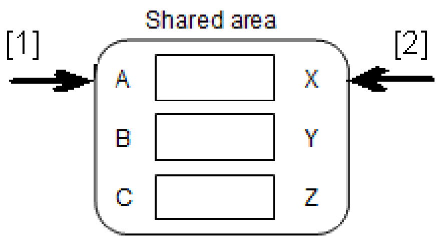
Order of data-names between calling and called programs

[Program A]

```
WORKING-STORAGE SECTION.
 01  A   PIC X.
 01  B   PIC X(10).
 01  C.
     02  D PIC X(5).
     02  E PIC X(5).
 PROCEDURE DIVISION.
     CALL "B" USING  A B C.    *>[1]
```

[Program B]

```
LINKAGE SECTION.
 77  X   PIC X.
 77  Y   PIC X(10).
 77  Z   PIC X(10).
 PROCEDURE DIVISION USING  X Y Z.   *>[2]
```



When the contents of parameters of the calling program should not be changed by the subprogram, write "BY CONTENT data-name" in the USING phrase of the CALL statement.

BY CONTENT data-name in the USING phrase of the CALL statement

[Program A]

```
WORKING-STORAGE SECTION.
 77  PARM-A  PIC X(5).
 PROCEDURE DIVISION.
     MOVE "ABCDE" TO PARM-A.
     CALL "B" USING BY CONTENT PARM-A.  *>[1]
```

[1] The contents of PARM-A ("ABCDE") remain unchanged when control returns from program B.

[Program B]

```
LINKAGE SECTION.
 77  PARM-B  PIC X(5).
 PROCEDURE DIVISION USING PARM-B.
     MOVE "12345" TO PARM-B.
```

Note the following points for subprograms to receive parameters correctly:

- Define data items to receive parameters in the LINKAGE section of the subprogram.
- Describe data items to receive parameters in a USING phrase of the PROCEDURE DIVISION header or of a ENTRY statement on the subprogram.
- The number of parameters specified for a CALL statement of a calling program and the number of parameters described in the USING phrase of the PROCEDURE DIVISION header, or of an ENTRY statement, in the subprogram must be the same. Also, the lengths of corresponding parameters must be the same.

If an error is found in descriptions, the programs cannot be operated correctly. When programs are compiled and executed, these errors can be checked within the following range:

Checked Item	Compile time	Runtime
Errors in data item definition for receiving parameters of LINKAGE section.	Yes	--
Description of data item for receiving parameters of USING phrase.	Yes (*1)	--
Disagreement of the number and length of parameters.	Yes (*2)	Yes (*2)

*1 : Descriptions with errors may not be checked when the parameters described in the USING phrase of PROCEDURE DIVISION header and USING phrase of the ENTRY statement differ.

*2 : You must specify the CHECK compile option when you compile the program. CALL statements that call internal programs are checked when compiling; CALL statements that call external programs are checked at run time. For details, refer to "Using the CHECK Function" in the "NetCOBOL Debugging Guide".

Note

- When calling COBOL programs from COBOL programs, you cannot pass parameters with "USING BY VALUE."
- When object reference data items are passed between calling programs and subprograms, USAGE clause of object reference data item must be agreement.

9.2.5 Sharing Data

By specifying the EXTERNAL clause in the data or file description entry, the data area can be shared among multiple external programs. Specifying the EXTERNAL clause gives the data or file the external attribute. Data having the external attribute is external data, and a file having the external attribute is an external file.

Using this function enables data transfer between programs without using arguments (parameter transfer function). However, use this function carefully because it makes it hard to see the area referencing or setting structure.

The EXTERNAL clause cannot be specified in a file record entry, but a data item defined in the record entry of an external file becomes external data.

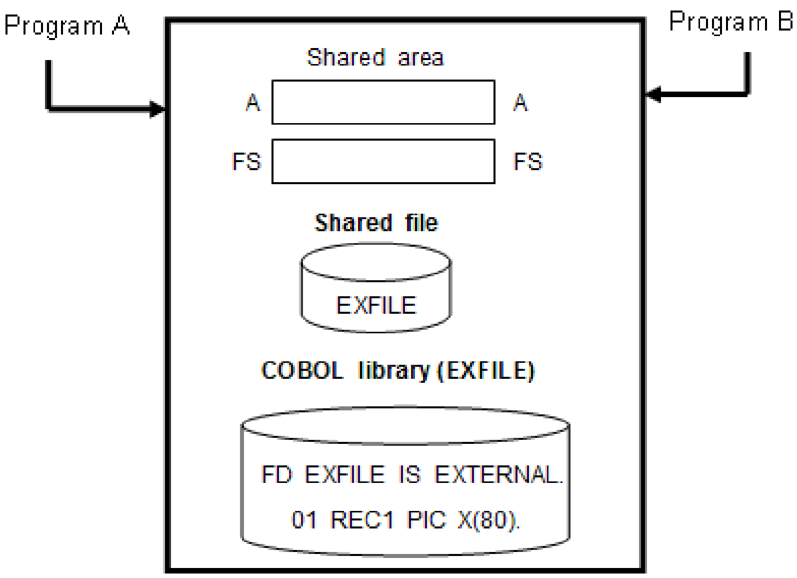
Generating the definitions of external data or files as a COBOL library, then including the data in programs with the COPY statement can improve program maintainability.

[Program A]

```
  SELECT EXFILE
    ASSIGN TO "C:\A.DAT"
    FILE STATUS IS EXFS.
DATA DIVISION.
FILE SECTION.
   COPY EXFILE.
WORKING-STORAGE SECTION.
01 EXDATA01 IS EXTERNAL.
  02 A  PIC X(5).
01 EXFS IS EXTERNAL.
  02 FS PIC X(2).
PROCEDURE DIVISION.
    MOVE SPACE TO A.
    OPEN INPUT EXFILE.
    CALL "B"
```

[Program B]

```
  SELECT EXFILE
    ASSIGN TO "C:\A.DAT"
    FILE STATUS IS EXFS.
DATA DIVISION.
FILE SECTION.
 COPY EXFILE.
WORKING-STORAGE SECTION.
01 EXDATA01 IS EXTERNAL.
  02 A  PIC X(5).
01 EXFS IS EXTERNAL.
  02 FS PIC X(2).
PROCEDURE DIVISION.
    MOVE "ABCDE" TO A.
    READ EXFILE.
```



9.2.5.1 Notes on using external data

External data is checked for the maximum or minimum area length (variable length data item) used. For external data consisting of group items, the attributes of individual data items making up the external data are not checked. Therefore, careless use may cause data exceptions or abnormal execution results. To avoid this problem, use a COBOL library, so that the same record structure is used between programs that share data.

An external data area is reserved when control is once passed to the program in which the external data is written, and freed when the run unit of the runtime system ends. That is, the external data area is not freed even if a CANCEL statement deletes the program. Be careful when external data is used in a program that is called repeatedly.

9.2.5.2 Notes on using an external file

- Two or more programs can share an external file. A program other than the program that executed an OPEN statement can perform input-output processing for the file.
- An external file can be handled by programs the same way as normal files without an external attribute. However, the external file must be defined with the same attribute among programs that share it. Since one file is shared by multiple programs, its attribute must essentially be defined as being consistent among them.
- To define an external file with the same attribute means to match the definition of the items indicated in the general rules in "FILE-CONTROL," in the "*COBOL Reference Manual*".
- Since many attribute items of an external file should be the same, it is recommended to use a COBOL library wherever possible. Since these items are checked at run time, an error may occur at the last link stage and cause a turn-back of development.
- Once a program in which an external file is written is executed, the record area and control area of the external file are not freed, even if the program is deleted by the CANCEL statement. These areas are freed when the run unit of the runtime system ends. (These areas of a normal file without an external attribute are freed when the program in which the file is written is deleted.) Exercise caution when using an external file in a program that is called repeatedly.

9.2.6 Return Codes

When control is returned to calling programs from subprograms, return code values can be passed either using a RETURNING data item or the PROGRAM-STATUS (or RETURN-CODE) special register.

You specify the RETURNING phrase in the CALL statement and in the heading of the procedure division of the subprogram. The data type and length of the data items specified in the RETURNING phrases of calling and called programs must match.

[Program A]

```
  WORKING-STORAGE SECTION.
  01 RTN-ITM  PIC S9(2) DISPLAY.
 PROCEDURE DIVISION.
     CALL "B" RETURNING RTN-ITM.
     IF RTN-ITM NOT = 0 THEN
*>      :
```

[Program B]

```
  LINKAGE SECTION.
  01 RTN-CD  PIC S9(2) DISPLAY.
 PROCEDURE DIVISION RETURNING RTN-CD.
     IF ERROR-CODE > 0    *> error occurred
       THEN
         MOVE 99 TO RTN-CD
       ELSE
         MOVE  0 TO RTN-CD
     END-IF
```

Note

If RETURNING is specified in the CALL statement the value of the special register PROGRAM-STATUS in the calling program remains unchanged.
Also, if RETURNING is specified in the CALL statement, the subprogram has to set a value in the RETURNING item. If no value is set, the value of the RETURNING item in the calling program is undefined.

Since the PROGRAM-STATUS special register is implicitly declared as "PIC S9(18) COMP-5", the user does not have to define it in a program.

When a subprogram sets a value in the PROGRAM-STATUS special register, the value is set in the PROGRAM-STATUS special register of the program that called the subprogram.

[Program A]

```
*>      :
     MOVE 0 TO PROGRAM-STATUS.
     CALL "B".
     IF PROGRAM-STATUS NOT = 0 THEN
*>      :
```

[Program B]

```
     IF ERROR-CODE > 0 THEN  *> error occurred
        MOVE 99 TO PROGRAM-STATUS
        EXIT PROGRAM
     END-IF.
```

Note that, if your program structure relies on PROGRAM-STATUS being communicated from lower to higher levels in the program hierarchy, you should not use RETURNING in intermediate levels of the hierarchy. Using RETURNING prevents PROGRAM-STATUS being passed between those programs.

When the COBOL program executes STOP RUN, it is returned as a resetting value to the program where the value of special register PROGRAM-STATUS of the program that executes STOP RUN statement called the COBOL program. When STOP RUN statement is executed by the method procedure without special register PROGRAM-STATUS, 0 is returned as a resetting value.

Note

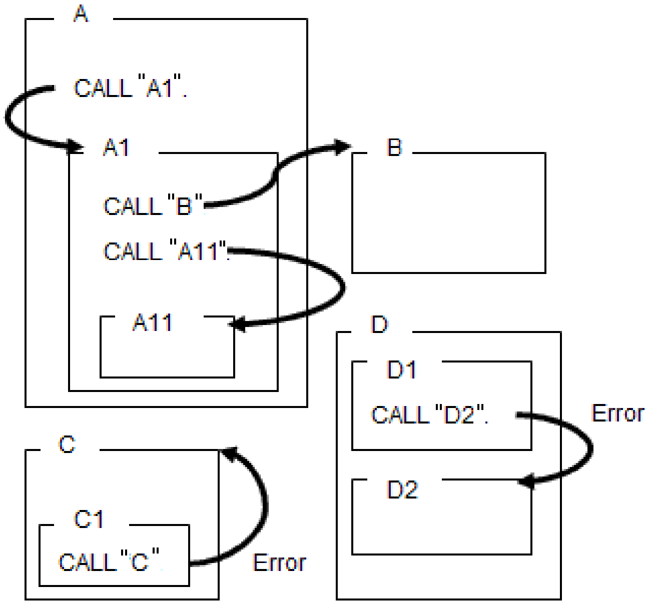
Please do not describe RETURNING phrase in the COBOL program called first when you use STOP RUN statement by the COBOL program. The type of the resetting value is acquired at STOP RUN statement execution time (value of special register PROGRAM-STATUS) when the resetting value returns by EXIT PROGRAM statement specification for RETURNING phrase the description of RETURNING phrase according to the type (value of RETURNING phrase) and the resetting value might not be able to be acquired differing, and correctly.

9.2.7 Internal Programs

COBOL programs are classified into external and internal programs based on the program structure. The outermost program not included in other programs is an external program. Programs directly or indirectly included in an external program are internal programs.

An external program (A) can call its internal program (A1). The internal program (A1) can call another external program (B) or its internal program (A11). However, internal programs (C1 and D1) cannot call outside programs (C and D2) other than common programs.

Figure 9.9 Calling sequence

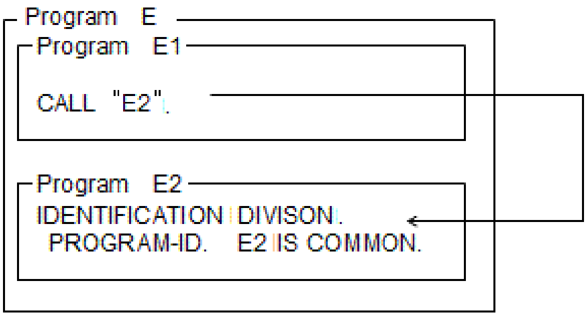


Common Programs

To call an internal program from an outside program, specify a COMMON clause in the PROGRAM-ID paragraph of the called internal program.

The program with COMMON specified is a common program, and it can be called from internal programs not including the program with COMMON specified.

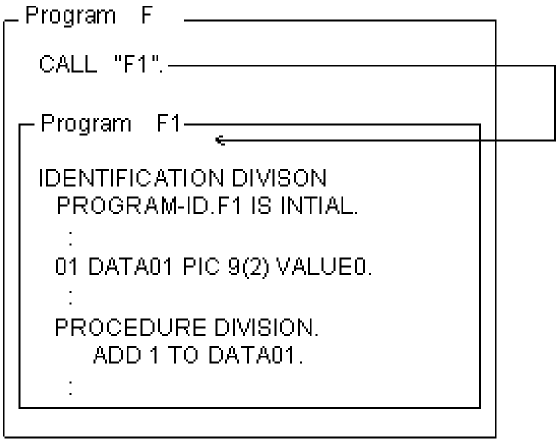
Figure 9.10 Using the COMMON clause



Initial Programs

To place a program in the initial state whenever it is called, specify an INITIAL clause in the PROGRAM-ID paragraph. This program is an initial program. When the initial program is called, the program is always placed in the initial state.

Figure 9.11 An INITIAL program



When program F1 is called, the value of DATA01 is always 0.

Valid Scope of Names

To use data items defined by external programs in the internal programs, specify the GLOBAL clause in the data description entry. Normally, names are valid only within the same program, but internal programs can use data items with the GLOBAL clause specified.

External programs cannot use data items with GLOBAL clause specified in the internal programs.

9.2.8 Notes

- If compiler option ALPHAL is used both for the calling program and the subprogram, the program names are treated as follows:
  - Calling program: Program names specified using a literal in the CALL statement are treated as uppercase.
  - Subprogram: Program names written in the PROGRAM-ID paragraph are treated as uppercase.

  Refer to "A.2.1 ALPHAL(lowercase handling (in the program))" for more details.
- When compiling the calling program and subprogram, specify the same compiler option for each, ALPHAL or NOALPHAL. Specify compiler option NOALPHAL when you want lowercase letters to be distinguished in your program names.

- When compiling a main program, compiler option MAIN must be specified. When compiling subprograms, compiler option NOMAIN must be specified.
  See "A.2.24 MAIN(main program/sub-program specification)" for more details.

- When two or more programs with the same program name in the run unit of COBOL exist, operation is not guaranteed.
  When the program with the same program name at execution time is called against the specification, it operates as follows.

  - When COBOL source program is different though it is the same program name, the message of execution time of JMP0032I-U is output.

  - When date-compiled is different though it is the same COBOL source program, the message of execution time of JMP0032I-U is output.

  - The state called last time cannot be retained when the same program is included in two or more DLL files, and the operation not intended might be done.

9.3 Calling and Being Called by C Programs

This section explains how to call C programs (functions) from COBOL programs, and how to call COBOL programs from C programs (functions). In this section, C programs (functions) are simply called C programs.

9.3.1 Calling C Programs from COBOL Programs

This section explains how to call C programs from COBOL programs.

[COBOL program]

```
IDENTIFICATION DIVISION.
  PROGRAM-ID.  PROGRAM-NAME.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  DATA1.
    02  ELEMENT1  PIC 9(4).
    02  ELEMENT2  PIC X(10).
  01  DATA2       PIC S9(4) COMP-5.
  01  DATA3       PIC X(1).
 PROCEDURE DIVISION.
     CALL "FUNCTION_NAME"  WITH C LINKAGE
                           USING DATA1 DATA2
                                 BY VALUE  DATA3.
     IF PROGRAM-STATUS = 0 THEN
        DISPLAY "RETURN-CODE : 0".
END PROGRAM PROGRAM-NAME.
```

[C program]

```
typedef struct
{
       char element1[4];
       char element2[10];
} structure_name;

long long int FUNCTION_NAME(
       structure_name *argument1,
       short int      *argument2,
       char           *argument3)
{
       long int function_value = 0;
       //  :
       return(function_value);
}
```

9.3.1.1 Calling Method

To call C programs from COBOL programs, specify function names in the CALL statement of COBOL. When the return statement is executed in the called C program, control returns immediately after the COBOL CALL statement.

9.3.1.2 Passing Parameters

To pass parameters from COBOL programs to C programs, specify data-names in the USING phrase of the CALL statement.

The parameters to be passed to the C program are the addresses or the values of the data items. Specify parameters in the USING phrase of the CALL statement.

The relation between the description of the USING phrase and the contents of parameters is explained below.

When BY REFERENCE Data-Name is Specified: Item Address

The COBOL program passes, to the C program, the address of the specified data item. Declare a pointer having a data type corresponding to the attribute of the parameter to be passed as a dummy argument in the C program. For correspondence between COBOL and C data types, see "Table 9.1 Correspondence between COBOL data items and C data types".

When BY CONTENT Data-Name (or Literal) is Specified: Item Address

The COBOL program passes, to the C program, the address of an area containing the value of the specified data item. Declare a pointer having the data type corresponding to the attribute of the parameter to be passed as a dummy argument in the C program.

Changing the contents of the area pointed to by this argument in the C program does not change the contents of the data item in the COBOL program.

When BY VALUE Data-Name is Specified: Contents of the Data item

The COBOL program passes to the C program the contents of the specified data item. Changing the contents of the argument in the C program does not change the contents of the data item in the COBOL program.

Different coding for USING specification

This section explains the differences between the BY REFERENCE and BY CONTENT specifications, and between BY REFERENCE and BY VALUE specifications, using program samples.

- Difference between BY REFERENCE and BY CONTENT specifications

  Consider the following example of a COBOL program calling a C program with BY REFERENCE and BY CONTENT parameters.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID.   MAINCOB.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
  01 PRM1  PIC S9(9) COMP-5.
  01 PRM2  PIC S9(9) COMP-5.
 PROCEDURE DIVISION.
     MOVE 10 TO  PRM1.
     MOVE 10 TO  PRM2.
     CALL "SUBC" WITH C LINKAGE
                 USING BY REFERENCE PRM1
                       BY CONTENT   PRM2.
     DISPLAY "PRM1=" PRM1.
     DISPLAY "PRM2=" PRM2.
```

```
#include <windows.h>

long long int SUBC(
       long int  *p1,
       long int  *p2)
{
       *p1 = *p1 +10;
       *p2 = *p2 +10;

       return(0);
}
```

Execution Result

```
PRM1=+000000020

PRM2=+000000010
```

When the C program is called from the COBOL program shown above, the content of PRM1 with the BY REFERENCE specification is updated to 20, while the content of PRM2 with the BY CONTENT specification remains at 10. This indicates that the parameter passed using BY REFERENCE updates the data of the calling program if the parameter value is changed in the called program. It also indicates that the parameter passed using BY CONTENT does not affect data in the calling program even if the parameter value is changed in the called program. The same applies when the called program is a COBOL program.

- Difference between BY REFERENCE and BY VALUE specifications

  Consider the following example of a COBOL program calling a C program with BY REFERENCE and BY VALUE parameters.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID.   MAINCOB.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
  01 PRM1  PIC S9(9) COMP-5.
  01 PRM2  PIC S9(9) COMP-5.
 PROCEDURE DIVISION.
     MOVE 10 TO  PRM1.
     MOVE 10 TO  PRM2.
     CALL "SUBC" WITH C LINKAGE
                 USING BY REFERENCE PRM1
                       BY VALUE     PRM2.
     DISPLAY "PRM1=" PRM1.
     DISPLAY "PRM2=" PRM2.
```

```
#include <windows.h>

long long int SUBC(
       long int  *p1,
       long int   p2)
{
       *p1 = *p1 +10;
        p2 =  p2 +10;

       return(0);
}
```

Execution Result

```
PRM1=+000000020

PRM2=+000000010
```

While BY REFERENCE passes the address of an item, BY VALUE passes the value of the item directly. Therefore, as with the BY CONTENT specification, the BY VALUE specification does not cause the content of the parameter to be changed in the calling program, even when the content of the corresponding parameter in the called program is changed. Note that the coding in the called C program varies depending on whether it receives an address or a value.

Information

If the BY specification is omitted, BY REFERENCE is assumed.

9.3.1.3 Return Codes(Function Values)

Use the PROGRAM-STATUS special register or RETURNING phrase to receive return codes (function values) from C programs.

Using the RETURNING phrase

If you want to use the RETURNING phrase, you need to make sure that the data type returned by the C program corresponds to the type of the item in the RETURNING phrase. Please refer to "9.3.3 Correspondence of COBOL and C Data Types" for more information.

[COBOL Program]

```
  WORKING-STORAGE SECTION.
  01 AGRP.
     02  AITEM1 PIC X(10).
     02  AITEM2 PIC X(20).
  77 B          PIC S9(4) COMP-5.
  01 RTN-ITM    PIC S9(4) COMP-5.
 PROCEDURE DIVISION.
     CALL "C" WITH C LINKAGE
              USING AGRP B
              RETURNING RTN-ITM.
     IF RTN-ITM NOT = 0 THEN
*>       :
```

[Function C]

```
#include <windows.h>

typedef struct
{
       char aitem1[10];
       char aitem2[20];
}agrp;

short int C(agrp *agrpp,short int *b)
{
       return(0);
}
```

Note

The delivery of the value of item described in RETURNING clause becomes a delivery of the content of data name. Therefore, the match as the COBOL program can be taken by describing to return the content of group item even on C source side.

[COBOL Program]

```
  WORKING-STORAGE SECTION.
  01 AGRP.
     02  AITEM1 PIC X(10).
     02  AITEM2 PIC X(20).
  77 B          PIC S9(4) COMP-5.
  01 RTN-ITM.
    02 RITEM1   PIC X(10).
    02 RITEM2   PIC S9(4) COMP-5.
 PROCEDURE DIVISION.
     CALL "C" WITH C LINKAGE
              USING AGRP B
              RETURNING RTN-ITM.
     IF RITEM1 = SPACE AND
        RITEM2 = 0 THEN
*>       :
```

[Function C]

```
#include <windows.h>
#include <string.h>

typedef struct
{
       char aitem1[10];
       char aitem2[20];
}agrp;

typedef struct rtnitm
{
       char      ritem1[10];
       short int ritem2;
};

struct rtnitm C (agrp *agrpp,short int *b)
{
       struct rtnitm rttbl;
       strcpy(&(rttbl.ritem1[0]),"          ");
       rttbl.ritem2=0;

       return(rttbl);
}
```

Using PROGRAM-STATUS

If you want to use the PROGRAM-STATUS, C function values must be of the long long int type.

[COBOL Program]

```
  WORKING-STORAGE SECTION.
  01 AGRP.
     02  AITEM1 PIC X(10).
     02  AITEM2 PIC X(20).
  77 B          PIC S9(4) COMP-5.
 PROCEDURE DIVISION.
     CALL "C" USING AGRP B
     IF PROGRAM-STATUS = 0 THEN
*>      :
```

[Function C]

```
#include <windows.h>

typedef struct
{
       char aitem1[10];
       char aitem2[20];
}agrp;

long long int C(agrp *agrpp, short int *b)
{
       return(0);
}
```

Note

- To receive function values apart from long long int type values, specify RETURNING phrase of the CALL statement. Refer to the following example of how to call a short int type C program.
  The attributes of special register PROGRAM-STATUS corresponds to the long long int type of C. Consequently, if a C program of

long int type or short int type is called, referring to the value of the special register PROGRAM-STATUS, may not provide the correct function value.

```
  01 SHORT-RET   PIC  S9(4) COMP-5.
*>     :
 PROCEDURE DIVISION.
     CALL "Cprog" RETURNING SHORT-RET.
     IF SHORT-RET = 0 THEN
*>     :
```

- When you call void type C programs, the PROGRAM-STATUS special register is updated to an unfixed value. To prevent the PROGRAM-STATUS special register being updated, describe a dummy data item (PIC S9(18) COMP-5) in RETURNING phrase as shown in the example.

```
  01  DUMMY-RET   PIC  S9(18) COMP-5.
*>     :
 PROCEDURE DIVISION.
     CALL "Cprog" RETURNING DUMMY-RET.
```

9.3.2 Calling COBOL Programs from C Programs

This section explains how to call COBOL programs from C programs, as illustrated in the example below.

```
function_name()
{
       typedef struct
       {
              char element1[4];
              char element2[10];
       }structure_name;

       extern void JMPCINT2();
       extern void JMPCINT3();
       extern long int program_name(structure_name *, short int *);
       structure_name arg_1;
       short int      arg_2;

       JMPCINT2();
       if (program_name(&arg_1,&arg_2)){
              printf("Error\n");
       }else{
              printf("Return Code = 0\n");
       }
       JMPCINT3();
       return(0);
}
```

```
 IDENTIFICATION DIVISION.
    PROGRAM-ID.  program_name.
 DATA DIVISION.
 LINKAGE SECTION.
 01 DATA-NAME-1.
   02 ELEMENT-1 PIC 9(4).
   02 ELEMENT-2 PIC X(10).
 77 DATA-NAME-2  PIC S9(4) COMP-5.
 PROCEDURE DIVISION USING DATA-NAME-1 DATA-NAME-2.
     MOVE 0 TO PROGRAM-STATUS.
     EXIT PROGRAM.
 END PROGRAM program_name.
```

9.3.2.1 Calling Method

To call COBOL programs from C programs, specify COBOL program names in the C function-calling format. When the EXIT PROGRAM statement is executed in the called COBOL program, control returns immediately after the C program function call.

9.3.2.2 Passing Parameters

To pass arguments from C programs to COBOL programs, specify arguments with C function calls. Arguments to be passed to COBOL programs must be storage addresses.

The COBOL subprograms specify data-names in the USING phrase of the PROCEDURE DIVISION header or ENTRY statement. The contents of the areas pointed to by the C arguments are received in the COBOL items.

A CONST type specified can be designated in the declaration or definition of a variable at the address specified by an argument. When this happens, do not change the contents of the area at the address specified by the argument.

9.3.2.3 Return Codes(Function Values)

Items in the Procedure Division RETURNING phrase and PROGRAM-STATUS special register are passed to C programs as function values. You need to ensure that the data types of RETURNING phrase items correspond to the C data types (refer to [1] and [2] of the following figure) used. See "9.3.3 Correspondence of COBOL and C Data Types" below.

[Function C]

```
C()
{
       typedef struct
       {
              char aitem1[10];
              char aitem2[20];
       }agrp;

       extern void JMPCINT2();
       extern void JMPCINT3();
       extern short int COB(agrp *agrpp, short int *b);  // [1]
       agrp      prm1;
       short int prm2;

       JMPCINT2();
       if (COB(&prm1,&prm2)==0){
              printf("Return Code = 0\n");
       }
       JMPCINT3();
}
```

[Program COB]

```
    PROGRAM-ID.  COB.
 DATA DIVISION.
 LINKAGE SECTION.
 01 AGRP.
    03 AITEM1 PIC X(10).
    03 AITEM2 PIC X(20).
 77 B         PIC S9(4) COMP-5.
 01 RTN-ITM   PIC S9(4) COMP-5.  *>[2]
 PROCEDURE DIVISION USING AGRP B
                   RETURNING RTN-ITM.
     MOVE 0 TO RTN-ITM.
     IF RTN-ITM > 0 THEN  *> error occurred
       MOVE 99 TO RTN-ITM.
```

Note

- If you use the RETURNING phrase in the procedure division header, note that values set in the PROGRAM-STATUS special register are not passed to the calling C program.
- When a value of entry with the RETURNING clause is passed, the data name contents are passed. If using a group item, see "9.3.1 Calling C Programs from COBOL Programs".

To pass function values using the PROGRAM-STATUS special register, the C program must receive the function values using the long long int type .

[Function C]

```
C()
{
       typedef struct
       {
              char aitem1[10];
              char aitem2[20];
       }agrp;

       extern void JMPCINT2();
       extern void JMPCINT3();
       extern long long int COB(agrp *agrpp, short int *b);
       agrp      prm1;
       short int prm2;

       JMPCINT2();
       if (COB(&prm1,&prm2)==0){
              printf("Return Code = 0\n");
       }
       JMPCINT3();
}
```

[program COB]

```
 PROGRAM-ID.  COB.
DATA DIVISION.
LINKAGE SECTION.
01 AGRP.
   03 AITEM1 PIC X(10).
   03 AITEM2 PIC X(20).
77 B         PIC S9(4) COMP-5.
PROCEDURE DIVISION USING AGRP B.
    MOVE 0 TO PROGRAM-STATUS.
```

9.3.3 Correspondence of COBOL and C Data Types

When mixing COBOL and C programs, you determine the types of data passed between the programs of the two languages. "Table 9.1 Correspondence between COBOL data items and C data types" lists the corresponding data types.

For information about COBOL internal formats, refer to the "NetCOBOL Language Reference". For C internal formats, refer to C language manuals.

Note

- In particular, when using COBOL internal Boolean data items and C bit fields, note the storage area locations.

- Note the following about transferring character strings:
    - Because no terminating character (\0) is set for a character string received from a COBOL program, the length of the character string must be known for processing of the string.
    - Space characters must be added to a character string to be sent to a COBOL program, so that the character string length matches the variable area length of the COBOL program.
      Do not set a terminating character (\0) at this time.

Table 9.1 Correspondence between COBOL data items and C data types

COBOL Data item	C Data Type	COBOL Coding Example	C Declaration Example	Size
Alphabetic or Alphanumeric	char, char array type or struct (structure type)	77 A PIC X.	Char A;	1 byte
		01 B PIC X(20).	Char B[20];	20 bytes
External decimal (*1)	char array type, or struct (structure type)	77 C PIC S9(5) SIGN IS LEADING SEPARATE.	Char C[6];	6 bytes
		01 D PIC S9(9) SIGN IS TRAILING SEPARATE.	Char D[10];	10 Bytes
Binary (*2) (*3)	unsigned char	01 E USAGE IS BINARY-CHAR UNSIGNED.	unsigned char E;	1 byte
	short int	01 F PIC S9(4) COMP-5. or 01 F USAGE IS BINARY-SHORT SIGNED.	short int F;	2 bytes
	long int	77 G PIC S9(9) COMP-5. or 77 G USAGE IS BINARY-LONG SIGNED.	long int G;	4 bytes
	long long int	77 H PIC S9(9) COMP-5. or 77 H USAGE IS BINARY-DOUBLE SIGNED.	long long int H;	8 bytes

Group item (*4)	char, char array type, or struct (structure type)	01 IGRP
		02 I1 PIC S9(4) COMP-5.
		02 I2 PIC X(4).
			Struct
			{ short int I1;
			char I2[4];
			}IGRP;
				6 bytes

Internal floating-point
(single-precision /
double-precision)
	float
	double
		01 J COMP-1.
		01 K COMP-2.
			Float J;
			Float K;
				4 bytes
				8 bytes

- *1 The internal format of COBOL external decimal data items are character strings consisting of characters indicating signs and numeric characters. In the C program, therefore, these are treated as character data not as numeric data. To handle external decimal data items as numeric data in the C program, the data type must be converted in the C program.

- *2 Binary data items map to the C short int or long int types, depending on the number of digits as follows:

  - Up to 4 digits: short int

  - 5 to 9 digits: long int

  - 10 to 9 digits: long long int

  A binary data item with a fraction part is converted to a floating-point data item, then passed, as in the example below:

  [COBOL program]

```
WORKING-STORAGE SECTION.
 01 H     PIC 9V9 COMP-5.
 01 FLOAT COMP-1.
 PROCEDURE DIVISION.
     MOVE H TO FLOAT.
     CALL "C" USING FLOAT.
```

  [Function C]

```
long long int C(float *h)
{
       printf("float = %f\n",*h);
       return(0);
}
```

- *3 When declaring group items as structure, note the storage boundary of variables included in the structure. To align the data item storage boundary with COBOL, specify the SYNCHRONIZED clause in the data description entry. For details, refer to the "COBOL Language Reference." For C variable storage boundary alignment, refer to C language manuals.

- *4 When the USAGE IS COMP-5 item receives a value from a C language short, long int type, or long long int type, and the value exceeds the maximum digits of the PICTURE clause, the result may not be as intended or desired. In this case, use a USAGE IS BINARY-SHORT SIGNED, USAGE IS BINARY-LONG SIGNED or USAGE IS BINARY-DOUBLE SIGNED item as these types are implemented so they always take the same number of digits as the corresponding C language types.

Information

- To call COBOL programs from C++ programs (extensions CPP and CXX) created with Visual C++, use the following "extern "C" " routines:

  [C++ Program]

```
#include <windows.h>

extern "C" void JMPCINT2();
extern "C" void JMPCINT3();
extern "C" long long int COBSUB(int *P);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nShowCmd)
{
       int prm1=0;

       JMPCINT2();
       COBSUB(&prm1);
       JMPCINT3();

       return(0);
}
```

  [COBOL Program]

```
IDENTIFICATION DIVISION.
  PROGRAM-ID.  COBSUB.
 DATA DIVISION.
 LINKAGE SECTION.
 01 C-PRM PIC S9(9) COMP-5.
 PROCEDURE DIVISION USING C-PRM.
     DISPLAY C-PRM.
     *>      :
     EXIT PROGRAM.
```

- To call C++ programs created with Visual C++ from COBOL programs, use the following "extern "C" " routines:

  [COBOL program]

```
 IDENTIFICATION DIVISION.
  PROGRAM-ID.  COBMAIN.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 COPRM1.
   02 COBPRM-1 PIC X(10).
   02 COBPRM-2 PIC X(20).
 77 COBPRM2    PIC S9(9) COMP-5.
 PROCEDURE DIVISION.
     CALL "CSUB" USING COBPRM-1 COBPRM-2.
```

  [C++ Program]

```
#include <windows.h>
#include <stdio.h>

typedef struct
{
       char prm1[10];
       char prm2[20];
}argp;

extern "C" long long int CSUB(argp *argpp,int *b)
{
       printf("argp = %s : %d\n",argpp,*b);
       return;
}
```

9.3.4 Compiling Programs

This section describes points to be aware of when compiling mixed language (C and COBOL) applications.

- C program names can be case sensitive so use the NOALPHAL COBOL compiler option to prevent the compiler converting all program names to upper case. Refer to "A.2.1 ALPHAL(lowercase handling (in the program))".

  For example, (a) and (b)will use the program name "abc" when NOALPHAL is used. But with ALPHAL the program name will be "ABC".

```
  a) CALL "abc".

  b) PROGRAM-ID. abc.
```

- CALL statements that use a data name are not affected by the ALPHAL option.

  Calls program "abc" regardless of whether option ALPHAL or NOALPHAL is specified.

```
  MOVE  "abc" TO A.
  CALL A.
```

9.3.5 Linking Programs

This section describes how to link programs according to the calling conventions and different program structures.

Note

- If you call JMPCINT2 or JMPCINT3, you need to link with F4AGCIMP. LIB.
- When the COBOL program is linked with the C program, the executable file or DLL is recommended to be made by each support environment.

9.3.5.1 How to Link C Programs to Call COBOL Programs

The following gives examples of linking C programs , using the static link structure, dynamic link structure or dynamic program structure to call COBOL programs.

For more information on program structures, refer to "4.3 Program Structure".

[C Program]

```
#include <windows.h>

extern void JMPCINT2();
extern void JMPCINT3();
extern long long int  COBSUB(int *p);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nShowCmd)
{
    int prm1;
    //   :
    JMPCINT2();
    COBSUB(&prm1);
    JMPCINT3();
    //   :
}
```

[COBOL Program]

```
000000 @OPTIONS SRF(FREE)
IDENTIFICATION DIVISION.
  PROGRAM-ID.  COBSUB.
 DATA DIVISION.
 LINKAGE SECTION.
 01 B  PIC S9(9)  COMP-5.
 PROCEDURE DIVISION  USING B.
    DISPLAY B.
    *>   :
    EXIT PROGRAM.
```

Linking with Static Link Structure

To link a C program calling a COBOL program with the static link structure use a command of the following format:

```
LINK Cprog.obj COBSUB.OBJ F4AGCIMP.LIB LIBCMT.LIB Windows-function-import-library /OUT:Cprog.EXE
```

- Cprog.obj : Object file of C program
- COBSUB.OBJ : Object file of COBOL program COBSUB
- F4AGCIMP.LIB : COBOL runtime system import library

- LIBCMT.LIB : C runtime library
- Windows-function-import-library : Other import libraries required to support the Windows functions used by the C program (e.g. USER32.LIB).

Linking with Dynamic Link Structure

To create a DLL of the COBOL program to be called by the C program, use a link command of the following form:

```
LINK COBSUB.OBJ F4AGCIMP.LIB /DLL /NOENTRY /OUT:COBSUB.DLL
```

- COBSUB.OBJ : Object file of COBOL program COBSUB
- F4AGCIMP.LIB : Import library of COBOL runtime system

To create the executable for the C program calling the COBOL program use a link command of the form:

```
LINK Cprog.obj F4AGCIMP.LIB LIBCMT.LIB COBSUB.LIB  /OUT:Cprog.EXE
```

- Cprog.obj : Object file of C program
- F4AGCIMP.LIB : Import library of COBOL runtime system
- LIBCMT.LIB : C runtime library
- COBSUB.LIB : Import library of COBOL program COBSUB

Linking with Dynamic Program Structure

Linking with dynamic program structure requires some additional code in the calling C program to load the COBOL .DLL file.

[C Program]

```
#include <windows.h>

extern void JMPCINT2();
extern void JMPCINT3();
long long int (far *COBSUB)(int *p);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nShowCmd)
{
    int prm1=0;
    HANDLE COBHND;

    COBHND = LoadLibrary("COBSUB.DLL");
    (FARPROC)COBSUB=GetProcAddress(COBHND,"COBSUB");
    JMPCINT2();
    COBSUB(&prm1);
    JMPCINT3();
    FreeLibrary(COBHND);

}
```

To create the DLL for a COBOL program to be called from a C program use a link statement of the form:

```
LINK COBSUB.OBJ F4AGCIMP.LIB /DLL /NOENTRY /OUT:COBSUB.DLL
```

- COBSUB.OBJ : Object file of COBOL program COBSUB
- F4AGCIMP.LIB : Import library of COBOL runtime system

To create the executable file for a C program calling a COBOL program use a link command of the form:

```
LINK Cprog.obj F4AGCIMP.LIB LIBCMT.LIB /OUT:Cprog.EXE
```

- Cprog.obj : Object file of C program
- F4AGCIMP.LIB : Import library of COBOL runtime system
- LIBCMT.LIB : C runtime library

9.3.5.2 How to Link a COBOL Program that Calls a C Program

The following gives examples of linking COBOL programs, using the static link structure, dynamic link structure or dynamic program structure to call C programs.

For more information on program structures, refer to "4.3 Program Structure".

Information

The generation of external reference information on C program has three methods. These are the method of specifying _declspec(dllexport) for function declaration, the method of specifying the /EXPORT option of link command and the method of specifying the EXPORTS statement of the module definition file. The following example is using the method of specifying _declspec(dllexport) for function declaration of C program.

[COBOL Program]

```
IDENTIFICATION DIVISION.
  PROGRAM-ID.  MAINCOB.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 B  PIC S9(9)  COMP-5.
 PROCEDURE DIVISION.
    CALL "CSUB" USING B.
    *>   :
    DISPLAY B.
    *>   :
    STOP RUN.
```

[C Program]

```
#include <windows.h>

_declspec(dllexport)
long long int CSUB(int *b)
{
       printf("b=%d\n",*b);
       return;
}
```

C Calling ConventionsLinking with Static Link Structure

To link a COBOL Program to call a C program using the static link structure use a link command of the form:

```
LINK MAINCOB.OBJ CSUB.OBJ F4AGCIMP.LIB LIBCMT.LIB
Windows-function-import-library /OUT:MAINCOB.EXE
```

- MAINCOB.OBJ : Object file of COBOL program MAINCOB
- CSUB.OBJ : Object file of C program CSUB
- F4AGCIMP.LIB : Import library of COBOL runtime system
- LIBCMT.LIB : C runtime library
- Windows-function-import-library : Import libraries for Windows functions being used by the C program (e.g. USER32.LIB)

Linking with Dynamic Link Structure

To create a DLL for a C program to be called by a COBOL use a link command of the form:

```
LINK CSUB.OBJ LIBCMT.LIB Windows-function-import-library /DLL /OUT:CSUB.DLL
```

- CSUB.OBJ : Object file of C program CSUB
- LIBCMT.LIB : C runtime library
- Windows-function-import-library : Import libraries of Windows functions being used by the C program (e.g. USER32.LIB)

To create the executable file for a COBOL program calling a C program use a link command of the form:

```
LINK MAINCOB.OBJ F4AGCIMP.LIB LIBCMT.LIB CSUB.LIB /OUT:MAINCOB.EXE
```

- MAINCOB.OBJ : Object file of COBOL program MAINCOB
- F4AGCIMP.LIB : Import library of COBOL runtime system
- LIBCMT.LIB : C runtime library
- CSUB.LIB : Import library of C program CSUB

Linking with Dynamic Program Structure

To create a DLL for a C program to be called by a COBOL program use a link command of the form:

```
LINK CSUB.OBJ LIBCMT.LIB Windows-function-import-library /DLL /OUT:CSUB.DLL
```

- CSUB.OBJ : Object file of C program CSUB
- LIBCMT.LIB : C runtime library
- Windows-function-import-library : Import libraries of Windows functions being used by the C program (e.g. USER32.LIB)

To create the executable file for a COBOL program calling a C program use a link command of the form:

```
LINK MAINCOB.OBJ F4AGCIMP.LIB LIBCMT.LIB /OUT:MAINCOB.EXE
```

- MAINCOB.OBJ : Object file of COBOL program MAINCOB
- F4AGCIMP.LIB : Import library of COBOL runtime system
- LIBCMT.LIB : C runtime library

9.3.6 Executing Programs

When executing programs note the following:

- When calling COBOL programs from C programs, no COBOL runtime options can be specified for arguments of functions that call COBOL programs. The runtime options are treated the same as other arguments and are not handled as COBOL runtime options even if specified.
- Do not use the exit function to unconditionally terminate C programs called from COBOL programs.
- Do not use the STOP RUN statement to terminate COBOL programs called from C programs with JMPCINT2.
- Unlike C character strings, no null characters are inserted automatically at the end of COBOL character strings.
- Do not end the program executing STOP RUN statement when the program that contains the program made by Java and C# is called.

Chapter 10 Using ACCEPT and DISPLAY Statements

This chapter offers tips on using the ACCEPT and DISPLAY statements, including I-O destination types and specification methods, using console windows, message boxes, and program using files, and entering current date and time. Additionally, this chapter describes fetching command line arguments and environment variable handling.

10.1 ACCEPT/DISPLAY Function

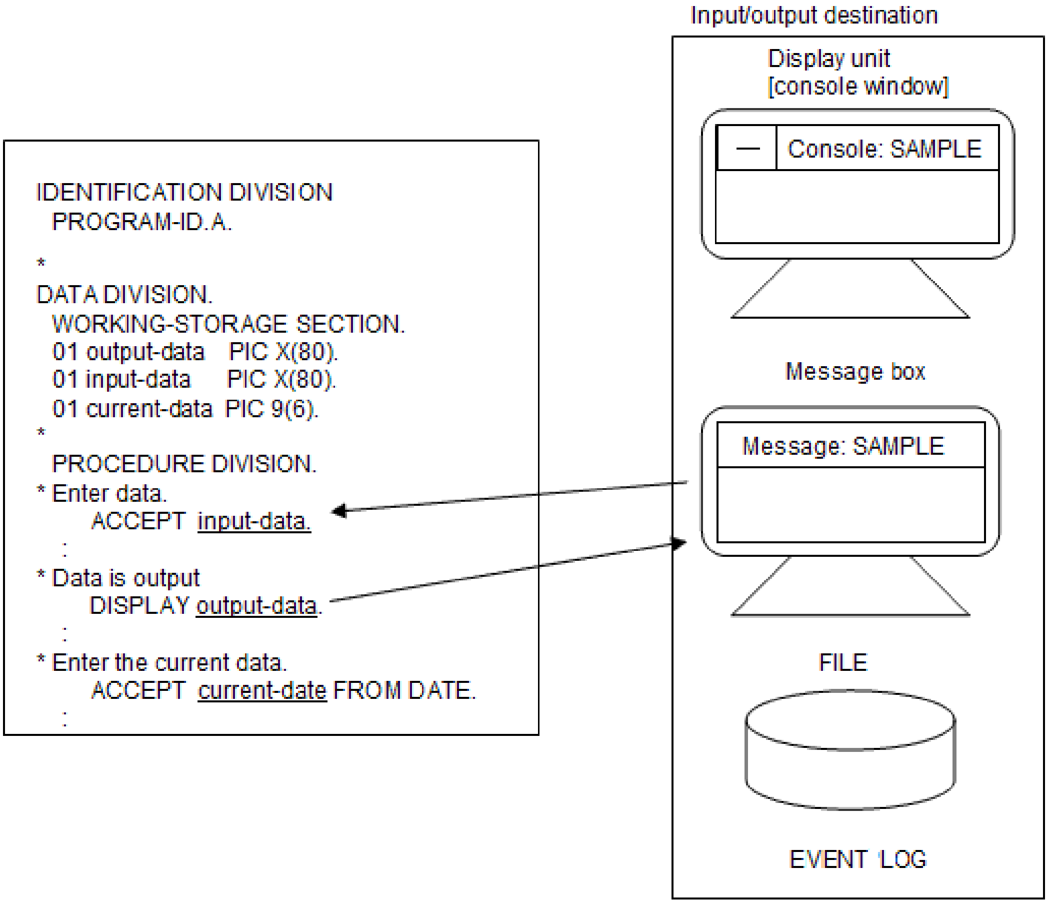
This section describes how to accept and display data in console windows, message boxes, Event Log, and files, using the ACCEPT and DISPLAY statements. Sample programs using this function are provided for your reference.

10.1.1 Outline

With the ACCEPT/DISPLAY function, data is input and output with console windows, message boxes, and files. Additionally, the current date and time can be read from the system.

Use the ACCEPT statement to input data, and use the DISPLAY statement to output data.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. A.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 output-data  PIC X(80).
01 input-data   PIC X(80).
01 current-date PIC 9(6).
*>
PROCEDURE DIVISION.
*> Enter data.
    ACCEPT input-data.     *> <--[1]
*> Data is output
    DISPLAY output-data.   *> --> [2]
*> Enter the current date
    ACCEPT current-date FROM DATE.
```



```
IDENTIFICATION DIVISION
  PROGRAM-ID.A.
*
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 output-data    PIC X(80).
  01 input-data     PIC X(80).
  01 current-data  PIC 9(6).
*
  PROCEDURE DIVISION.
* Enter data.
       ACCEPT  input-data.
     :
* Data is output
       DISPLAY output-data.
     :
* Enter the current data.
       ACCEPT  current-date FROM DATE.
     :
```

10.1.2 Input/Output Destination Types and Specification Methods

The input/output destination of data depends on:

- ACCEPT statement FROM specification
- DISPLAY statement UPON specification
- Compiler option specifications
- Setting of runtime environment information

"Table 10.1 Input/output destinations of the ACCEPT/DISPLAY function" lists the relationship between these specifications and input/output destinations.

Table 10.1 Input/output destinations of the ACCEPT/DISPLAY function

	FROM or UPON Specification	Compiler Option to be Specified	Runtime Environment Information Setting	Input/Output Destination
1)	None or function-name SYSIN/SYSOUT	None	-	Display unit (console window) (*2)
		SSIN (runtime environment information name) SSOUT (runtime environment information name)	Enter a file name for the runtime environment information name	File (*1)
		None	Set EVENTLOG in @CBR_DISPLAY_SYSOUT _OUTPUT	Event Log (*4) (*5)

	FROM or UPON Specification	Compiler Option to be Specified	Runtime Environment Information Setting	Input/Output Destination
2)	Function-name SYSERR	-	Enter a blank for @MessOutFile	Display unit (message box, command prompt, or system console) (*3)
			Enter a file name for @MessOutFile	File
			Set EVENTLOG in @CBR_DISPLAY_SYSERR_OUTPUT	Event Log (*5)
3)	Function-name CONSOLE	-	-	Display unit (console window) (*2)
			Set EVENTLOG in @CBR_DISPLAY_CONSOLE_OUTPUT	Event Log (*4) (*5)

- *1 : When SYSIN and SYSOUT are specified for compiler options SSIN and SSOUT as environment variable names, the input source and output destinations are system standard input and output.
- *2 : There are three types of console windows as follows:
  - COBOL console window (This window is created by COBOL.)
  - Command prompt window
  - System console window

  Two or more windows cannot be used simultaneously in all the programs that work during the process. The window to use is specified by the compiler options of the main program and the setting of the environment variable information.
- *3 : A message box is used if the COBOL console window is used as a console window. Standard errors are written to this message box if the command prompt window and the system console window are in use.
- *4 : Data cannot be entered in the Event Log.
- *5 : Data cannot be output to any other destinations (file, console, etc.) while writing to the Event Log.

10.1.3 Handling of Unicode data

Unicode data can be input or output using the ACCEPT or DISPLAY statement. Caution is required when the operand is a group item including a national data item.

```
WORKING-STORAGE SECTION.
01 PERSONAL-DATA.
  02 NAME PIC N(8).
  02 TEL PIC 9(10).
*>
PROCEDURE DIVISION.
    DISPLAY PERSONAL-DATA.    *> <--[1]
    DISPLAY NAME TEL.         *> <--[2]
```

Unicode encoding forms differ depending on the class. When a group item includes a national data item that is displayed using a DISPLAY statement, an illegal character appears (as shown in [1]). In this case, display each elementary item separately (as shown in [2]).

When data is read to a group item where a national item is included using the ACCEPT statement, the character-code of the data becomes the character-code of the alphanumeric item of the character-code specified by the ENCODE option.

When a file is specified for the input-output destination of the ACCEPT or DISPLAY statement, the encoding form of the file is UTF-8.

Note

The encoding form differs from that of a line sequential file. (UTF-16 little-endian is assumed when the record definition class is unique to national data item, as described below.)

Therefore, note the difference in the code when using a simple input-output module for line sequential files. See section "20.4.8 ACCEPT/DISPLAY statement".

10.1.4 Reading/Writing Data with Console Windows

The console window is used when data is read from the display device by the Accept/Display function, and data is displayed. Each run unit of COBOL programs has only one console window.

This section explains how to write, compile, link, and execute programs, and provides an example of the simplest coding using console windows.

10.1.4.1 Console Windows

The COBOL console window is a window generated by COBOL based on GUI. The window is generated when the first ACCEPT or DISPLAY statement of a run unit is executed, and deleted normally when the run unit ends. To use the window, specify MAIN (WINMAIN) in a compiler option for the main program when it is a COBOL application. When the main program is a C or other language program, specify environment variable information @CBR_CONSOLE = COBOL (default).

Note

- The sub option WINMAIN is the default of the compiler option MAIN. Similarly, the value SYSTEM is the default of environment variable information @CBR_CONSOLE when the compiler option MAIN (WINMAIN) is specified or the main program is a C or other language program.
- Environment variable information @CBR_CONSOLE = COBOL may be specified with the compiler option MAIN (MAIN). In this case, when an application is started using a method other than input to the command prompt window, such as by clicking an icon, a system console is generated soon. In addition, a COBOL console is also generated for input-output use.

The attributes of console windows can be changed with the COBOL runtime initialization file. Table below lists the attributes that can be changed and the specification methods.

Table 10.2 Changing the attributes of console windows

Attribute	Runtime Environment Information	Setting Value	Meaning
Automatic window closing	@WinCloseMsg (*1)	ON	Closes the window after displaying a message
		OFF	Closes the window without displaying a message
Window size	@CnslWinSize	(m,n)	Specifies the size of a window with the number of columns (m) and the number of lines (n)
Number of lines for data to be retained	@CnslBufLine	Number of lines	Data for the number of specified lines is retained and can be browsed by vertically scrolling the window
Font	@CnslFont	(Font name, Font size)	Specifies the font of consol window

*1 This runtime environment information is also valid for windows used with the screen operation function.

10.1.4.2 Command Prompt Windows

You can activate an application program from the command prompt window of Windows. You can use the activated window as a console window. You can use redirection when you run this way. The input/output targets correspond as follows:

ACCEPT statement	Standard input
DISPLAY statement specifying the name that corresponds to the function name SYSOUT and CONSOLE	Standard output
DISPLAY statement specifying the name that corresponds to the function name SYSERR	Standard error

If you use the Command Prompt Window, you must specify the appropriate compiler option or environment variable. If the main program is a COBOL program, you must specify the compiler option MAIN(MAIN) to the main program. Also, if the main program is written in another language (such as C), it is necessary to make a "main type" application (such as a main function), and to set the environment variable @CBR_CONSOLE=SYSTEM.

Note

- If the main program is COBOL, and the compiler option MAIN(WINMAIN) is specified, and if the main program is in another language (such as C) and is a WinMain function, the prompt window that activated an application program cannot be used as a console window, even if the environment variable @CBR_CONSOLE=SYSTEM is set. In this case, a system console window is generated, apart from the activated window.
- If the compiler option MAIN(MAIN) is specified, @CBR_CONSOLE defaults to system.
- When you click on the Close button of a command prompt window, the COBOL application is closed by force, and cannot recover system resources or perform "close" processing on open files.

10.1.4.3 System Console Window

The system console window is generated when an application is activated and while the application is being executed. Its design is similar to the command prompt window. You can specify font, layout and the color of the window in the Console Window Properties dialog box of the Control panel. Though this setting is for every logged-in user, it can be changed (as can the COBOL console window).

To use the system console window, specify the compiler option MAIN(WINMAIN) and the environment variable @CBR_CONSOLE=SYSTEM to a main program. If an execution file is specified to use the command prompt window, and is activated in a window other than the command prompt window, the system console window is used as a console window.

Note

- When you click on the Close button of the system console window, the COBOL application is closed by force, and cannot recover system resources or perform "close" processing on open files.
- In the following cases, when an application is started from a command prompt, abnormal operation may occur. In these cases, use the Windows START command.
    - When the COBOL application is compiled to use the system console window (having specified the MAIN(WINMAIN) option).
    - When COBOL is called from a Windows application (as opposed to a console application).

```
START COBOL-APL.EXE
```

- The code-set of the system console is Shift-JIS. When the output character string is Unicode, it is converted from Unicode to Shift JIS. A conversion error might occur in the Unicode character that does not exist in shift JIS.

10.1.4.4 Program Specifications

This section explains the program descriptions for each COBOL division.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. program-name.
DATA DIVISION.
 WORKING-STORAGE SECTION.
  01  data-name PIC X(80).
PROCEDURE DIVISION.
     ACCEPT   data-name.
     DISPLAY  data-name.
     DISPLAY "nonnumeric-literal".
     EXIT PROGRAM.
END PROGRAM program-name.
```

ENVIRONMENT DIVISION

No specifications are required.

DATA DIVISION

In the DATA DIVISION, define data items to store input data and items to set output data. Define the data items with one of the following attributes:

- Group item
- Alphabetic data item
- Alphanumeric data item
- Binary data item
- Internal decimal data item
- External decimal data item
- Alphanumeric edited data item
- Numeric edited data item
- National data item (cannot be used in an ACCEPT statement)
- National edited data item (cannot be used in an ACCEPT statement)

PROCEDURE DIVISION

Use an ACCEPT statement to input data from a console window. Input data is stored in a data-name specified in an ACCEPT statement for the length (80 alphanumeric characters if defined as 01 input-data PIC X(80)) defined for the data-name.

If the length of input data is less than that of data to be stored, the input request is repeated.

For character data, the input request is repeated until the entered data satisfies the specified length.

For numeric data, the input request is repeated until the Enter or Return key is pressed.

Use a DISPLAY statement to output data to a console window.

When a data-name is specified in a DISPLAY statement, data stored in the data-name is output.

When a nonnumeric literal is specified in a DISPLAY statement, a specified character string is output.

10.1.4.5 Program Compilation and Linkage

Do not specify the compiler options SSIN and SSOUT.

Specify the compiler option MAIN(WINMAIN) to a main program when you use COBOL console window.

If you use the command prompt window or the system console window, specify the compiler option MAIN(MAIN).

10.1.4.6 Program Execution

Execute programs like ordinary programs. However, execute within the command prompt window when you use the command prompt window.

Data input is requested on a console window when an ACCEPT statement in a program is executed. Enter data as required.

Input data is set for the data item for the length (80 bytes) specified in an ACCEPT statement. If the length of input data is less than that of the data item, the input request is repeated.

For character data, the input request is repeated until the entered data satisfies the specified length.

For numeric data, the input request is repeated until the Enter or Return key is pressed.

When a DISPLAY statement in the program is executed, data is output to the console window.

Information

To associate data at one entry with one ACCEPT statement, use function-name CONSOLE. When function-name CONSOLE is used, space is filled if the length of input data is less than that of the data item.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. A.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
   CONSOLE IS CONS.
DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  data-1   PIC X(80).
PROCEDURE DIVISION.
    ACCEPT  data-1  FROM CONS.
    DISPLAY data-1  UPON CONS.
    EXIT PROGRAM.
END PROGRAM A.
```

Note

If you click on the Close button of a console window, and if you select the Close command from the pop-up menu of a console window, a dialog box to confirm if you want to close the program by force is displayed. The program is closed when you say you want to do this.

10.1.5 Writing Messages to Message Boxes

This section explains how to write messages to message boxes.

10.1.5.1 Message Boxes

Normally, COBOL program runtime messages are displayed in message boxes. With a COBOL program, messages other than runtime messages can also be displayed in the message boxes. Executing a DISPLAY statement opens a message box, and clicking on the OK button closes it.

Note

If you use the command prompt or the system console window, you cannot use a message box, and should instead specify the COBOL console window.

10.1.5.2 Program Specifications

This section explains program descriptions for processing files using the message box for each COBOL division.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. program-name.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SPECIAL-NAMES.
    SYSERR IS mnemonic-name.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  data-name  PIC X(80).
 PROCEDURE DIVISION.
     DISPLAY data-name UPON mnemonic-name.
     EXIT PROGRAM.
 END PROGRAM program-name.
```

ENVIRONMENT DIVISION

In the ENVIRONMENT DIVISION, associate a mnemonic-name with function-name SYSERR.

DATA DIVISION

In the DATA DIVISION, define data items to set output data. Define these data items with one of the following attributes:

- Group item
- Alphabetic data item
- Alphanumeric data item
- External decimal data item
- Alphanumeric edited data item
- Numeric edited data item
- National data item
- National edited data item

PROCEDURE DIVISION

To send a message to a message box, use a DISPLAY statement where a mnemonic-name associated with function-name SYSERR is specified in the UPON clause.

If a data-name is specified in a DISPLAY statement, data stored in the specified data-name is output.

If a nonnumeric literal is specified, the specified character string is output. The maximum length of the message that can be written at one time is the length of the data that can be displayed within the message box.

10.1.5.3 Program Compilation and Linkage

No specific compiler and linker options are required.

10.1.5.4 Program Execution

Execute programs as ordinary programs.

To output a message to a file, you should set @MessOutFile. See "5.4.1.56 @MessOutFile(Set a Message Output File)"

Note

- When you specify the MAIN(MAIN) compiler options, you should set @CBR_CONSOLE=COBOL.

10.1.6 Programs Using Files

This section explains how to write, compile, link, and execute programs, and provides an example of the simplest coding for file processing using the ACCEPT/DISPLAY function.

10.1.6.1 Program Specifications

This section explains the syntax, by COBOL division, for implementing ACCEPT/DISPLAY statements to access file data.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. program-name.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  data-name  PIC X(80).
 PROCEDURE DIVISION.
     ACCEPT  data-name.
     DISPLAY data-name.
     EXIT PROGRAM.
 END PROGRAM program-name.
```

ENVIRONMENT DIVISION

No specifications are required.

DATA DIVISION

In the DATA DIVISION, define data items to store input data and items to set output data. Define these data items with one of the following attributes:

- Group item
- Alphabetic data item
- Alphanumeric data item
- External decimal data item
- Alphanumeric edited data item
- Numeric edited data item
- National data item (cannot be used in an ACCEPT statement)
- National edited data item (cannot be used in an ACCEPT statement)

PROCEDURE DIVISION

At program execution, an input file is opened with the first ACCEPT statement, and an output file is opened with the first DISPLAY statement of the program. With subsequent ACCEPT and DISPLAY statements, data is read or output only.

After a file is opened (after executing the first ACCEPT and DISPLAY statements), the input/output destination cannot be changed.

The input and output files are closed upon termination of program execution.

Inputting Data

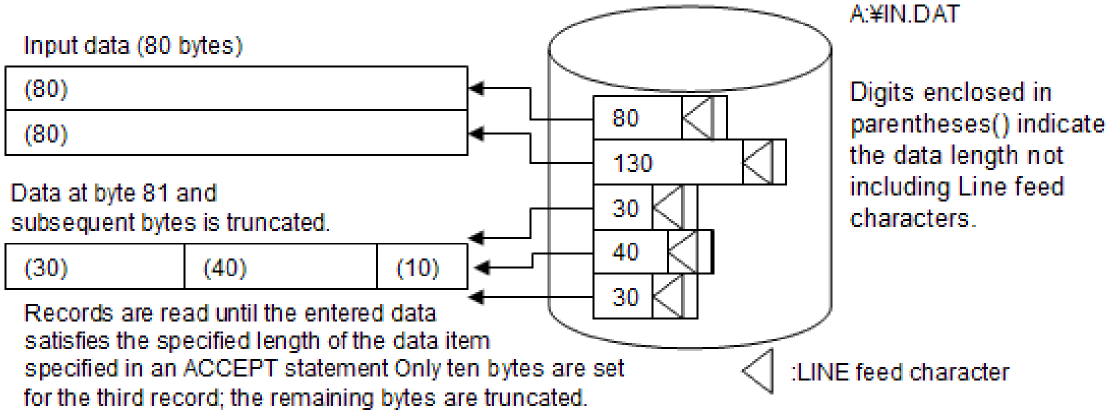
Use an ACCEPT statement to input data from a file.

Data at the byte immediately before the line feed character is handled as one record.

Input data is read by record. Input data is stored in a data-name specified in an ACCEPT statement for the length (80 alphanumeric characters if defined as 01 input-data PIC X(80)) defined for the data-name. If the length of input data is less than that of data to be stored, the next record is read and linked to the previously read data.

In this case, line feed characters are not treated as data. Records are read until the entered data satisfies the specified length.

Figure 10.1 Reading records


A:¥IN.DAT
Input data (80 bytes)
(80)
(80)
Data at byte 81 and subsequent bytes is truncated.
(30)
(40)
(10)
Records are read until the entered data satisfies the specified length of the data item specified in an ACCEPT statement Only ten bytes are set for the third record; the remaining bytes are truncated.
80
130
30
40
30
Digits enclosed in parentheses() indicate the data length not including Line feed characters.
:LINE feed character

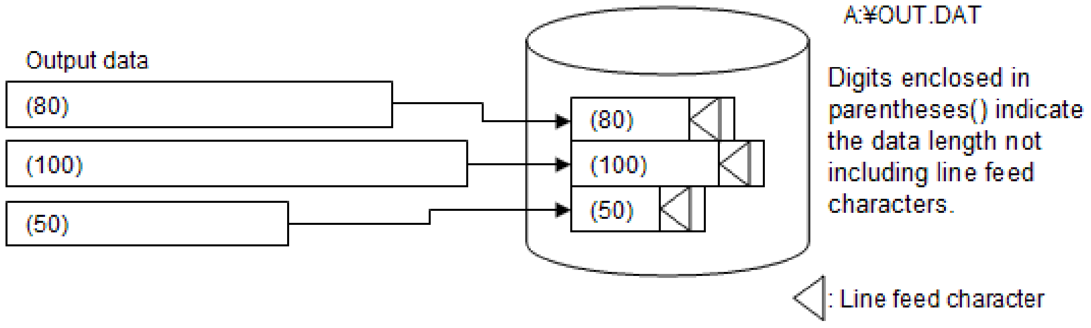
Outputting Data

Use a DISPLAY statement to send data to a file.

When a data-name is specified in a DISPLAY statement, the content stored in the data-name is output.

If a nonnumeric literal is specified in a DISPLAY statement, the specified character string is output. With one DISPLAY statement, the length of the line feed character, plus output data, is the length of data to be output.

Figure 10.2 Using DISPLAY to output data


A:¥OUT.DAT
Output data
(80)
(100)
(50)
(80)
(100)
(50)
Digits enclosed in parentheses() indicate the data length not including line feed characters.
: Line feed character

10.1.6.2 Program Compilation and Linkage

To enter data from a file with ACCEPT, specify compiler option SSIN. To send data to a file with DISPLAY, specify compiler option SSOUT. For more details, refer to "A.2.49 SSIN(ACCEPT statement data input destination)" and "A.2.50 SSOUT(DISPLAY statement data output destination)".

Example

```
SSIN(INDATA),SSOUT(OUTDATA)
```

Note

- When SYSIN is specified to compiler option SSIN, a console window is the data input destination of an ACCEPT statement, regardless of the specification of runtime environment variable SYSIN.
- When SYSOUT is specified to compiler option SSOUT, a console window is the data output destination of a DISPLAY statement, regardless of the specification of runtime environment variable SYSOUT.

- When the name that corresponds to function name CONSOLE is specified in FROM or UPON, a console window is the data input/output destination regardless of the specification of runtime environment variable SYSIN/SYSOUT.

10.1.6.3 Program Execution

Specify the names of files used for input-output in the runtime environment information specified for compiler options SSIN and SSOUT.

Example

```
INDATA=A:\IN.DAT
OUTDATA=A:\OUT.DAT
```

Note

- The input file is opened in input mode and used in shared mode. No records are locked when read.
- The output file is opened in output mode and used in exclusive mode.
- If a specified file already exists at the output destination, the file is recreated (previous data is deleted).
- Line feed characters are not handled as data.It is not allowed to change the I-O destination by using the environment variable processing function after opening a file (that is, after running the first ACCEPT and DISPLAY statements).
- Refer to "6.9 How to Use Other File Systems" for the maximum file size and the maximum record length.
- A virtual device (ex. NUL) cannot be specified for INPUT/OUTPUT.

10.1.6.4 File Output Extension Function for the DISPLAY Statement

With file output extension functions, Appending to Existing Files and Dummy Files can be used for file output with a DISPLAY statement.

Appending to Existing Files

For the environment variable name specified for the SSOUT compile option, specify ",MOD" after the data output destination file name.

Example

```
OUTDATA=A:\OUT.DAT,MOD
```

Note

If MOD is specified, and an existing file has the same name, information is added to the file. If no existing file has the same name, a new file is created.

Dummy files

When the output file is a dummy file, the output file is not generated even though the DISPLAY statement is executed. For more details, refer to "6.7.7 Dummy File".

For the environment variable name specified for the SSOUT compile option, specify ",DUMMY". The file name is optional and can be omitted.

Example

```
OUTDATA=A:\OUT.DAT,DUMMY
  or
OUTDATA=,DUMMY
```

Note

- A comma (,) must precede "DUMMY". If there is no preceding comma, "DUMMY" is treated as a file name.
- If a file name is specified with ",DUMMY", the file name is ignored.

COMMON NOTES

- If the file name contains a comma (,) it must be enclosed in double quotation marks (").
- If a character string other than MOD or DUMMY is specified after a comma (,) an error occurs in the initial execution of the DISPLAY statement.
- ",MOD" and ",DUMMY" can be specified at the same time. However, if ",MOD" and ",DUMMY" are specified at the same time, ",MOD" is invalid, and only ",DUMMY" is valid.

10.1.6.5 File Input Extension Function for the ACCEPT Statement

With file input extension functions, a file can be opened with each thread and a dummy file can be used for file input with an ACCEPT statement.

Dummy file

When the input file is a dummy file, the ACCEPT statement is executed without actually creating an input file. The value of the data items specified for the ACCEPT statement is not updated. For more details, refer to "6.7.7 Dummy File"

Specify ",DUMMY" for the environment variable name specified for the SSIN compile option.

The file name is optional and can be omitted.

Example

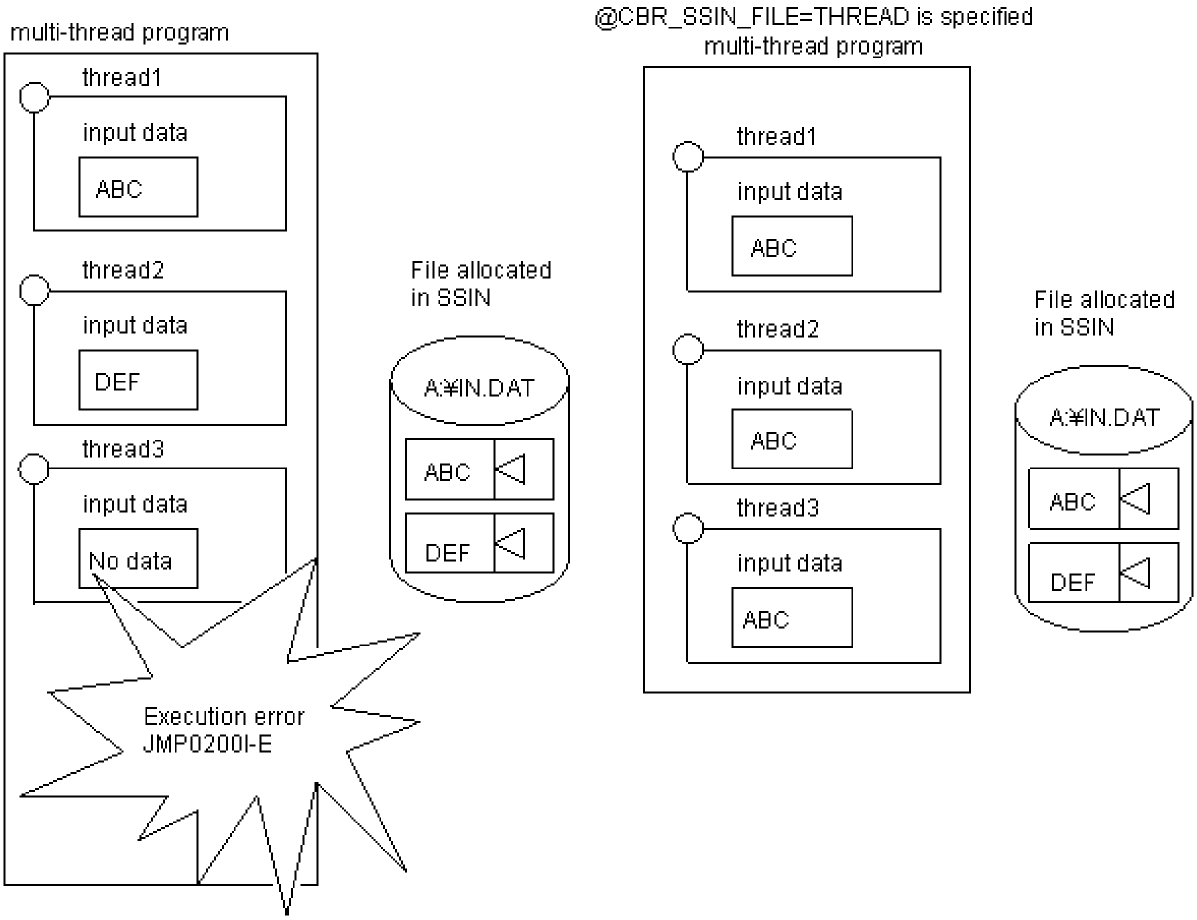
```
INDATA=A:\IN.DAT,DUMMY
  or
INDATA=,DUMMY
```

Note

- If the file name contains a comma (,) it must be enclosed in double quotation marks (").
- A comma (,) must precede "DUMMY". If there is no preceding comma, "DUMMY" is treated as a file name.
- If a file name is specified with ",DUMMY", the file name is ignored.
- If a character string other than DUMMY is specified after the comma (,) an error occurs in the initial execution of the ACCEPT statement.

File opening with multithread

A multithread program usually shares one input file, and the input file is opened by each thread that uses the input file open function. When the ACCEPT statement is executed from two or more threads, it is possible to enter a thread without input data because the order of execution depends on the thread control system order. JMP0200I-E is output if there is no input data.



Specify "THREAD" for environment variable @CBR_SSIN_FILE. See "19.7.2.1.1 Format for Specifying Runtime Environment Information".

Example

```
@CBR_SSIN_FILE = THREAD
```

10.1.7 Entering Current Date and Time

This section explains how to write, compile, link, and execute programs for entering the current date and time by using system clocks with the ACCEPT/DISPLAY function.

10.1.7.1 Programs Specifications

This section explains program specifications for entering the current date and time based on the system clock by using the ACCEPT statement, by each COBOL division.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. program-name.
```

```
DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  date-1         PIC 9(6).
 01  day-1          PIC 9(5).
 01  day-of-week-1  PIC 9(1).
 01  time-1         PIC 9(8).
PROCEDURE DIVISION.
    ACCEPT  date-1         FROM DATE.
    ACCEPT  day-1          FROM DAY.
    ACCEPT  day-of-week-1  FROM DAY-OF-WEEK.
    ACCEPT  time-1         FROM TIME.
```

ENVIRONMENT DIVISION

No specifications are required.

DATA DIVISION

In the DATA DIVISION, define the data items required to store input data.

PROCEDURE DIVISION

To input the current date and time, use an ACCEPT statement in which DATE, DAY, DAY-OF-WEEK, or TIME are written for the FROM specification.

10.1.7.2 Program Compilation and Linkage

No specific compiler and linker options are required.

10.1.7.3 Program Execution

Execute programs as ordinary programs.

When an ACCEPT statement in a program is executed, the current date and time are set for the data-name specified in an ACCEPT statement.

Example

12-23-1994 (Fri.) 14:15:45.00

<table>
<tr><th>Coding of FROM Specification</th><th>Contents Set for the Data-Name</th></tr>
<tr><td>FROM DATE</td><td>|9|4|1|2|2|3|</td></tr>
<tr><td>FROM DAY</td><td>|9|4|3|5|7|</td></tr>
<tr><td>FROM DAY-OF-WEEK</td><td>|5|</td></tr>
<tr><td>FROM TIME</td><td>|1|4|1|5|4|5|0|0|</td></tr>
</table>

Information

The CURRENT-DATE function can obtain four digits of the year. Refer to "D.3 Obtaining the Year Using the CURRENT-DATE Function" for detail.

10.1.8 Setting and Accepting a Given Date

This section explains how to write, compile, link, and execute programs for setting a given date using an environment variable and accepting it into your program using the ACCEPT statement.

If you need a 4 digit year, then you need to use the CURRENT-DATE function to retrieve the date as described in "D.4 Calculating Days from an Arbitrary Reference Date".

10.1.8.1 Program Specifications

The code to accept the date set in the environment variable looks like this, and is described below:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. program-name.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  data-name PIC 9(6).
 PROCEDURE DIVISION.
     ACCEPT  data-name  FROM  DATE.
```

ENVIRONMENT DIVISION

No specifications are required.

DATA DIVISION

In the DATA DIVISION, define the data items required to store input data. ACCEPT FROM DATE requires a 6 digit display numeric item (Year Month Day - 2 digits each)

PROCEDURE DIVISION

To accept the current date into your program, use an ACCEPT statement with the FROM DATE phrase.

10.1.8.2 Program Compilation and Linkage

No specific compiler and linker options are required.

10.1.8.3 Program Execution

In order to specify a date that is different from the system date, you need to specify the following environment variable:

```
@CBR_JOBDATE=Year.Month.Day
```

- Year can be two digits from 00 to 99 or four digits from 1900 to 2099
- Month is a two digit month from 01 to 12
- Day is a two digit day from 01 to 31

Years from 1900 to 1999 can be expressed in two or four digits. Years from 2000 must be expressed in four digits. When using the CURRENT-DATE function the runtime system will make 2 digit years into 19xx years.

Example

To set the date October 1, 1990:

```
@CBR_JOBDATE=90.10.01
```

ACCEPT Statement	Contents of Data-Name
ACCEPT data-name FROM DATE	901001

To set the date October 1, 2004:

```
@CBR_JOBDATE=2004.10.01
```

ACCEPT Statement	Contents of Data-Name
ACCEPT data-name FROM DATE	041001

(To retrieve the 4 digit year, use the CURRENT-DATE function.)

10.1.9 Programs that use the Event Log

This section explains how to write, compile, link, and execute programs that use the Event Log.

10.1.9.1 Program Specifications

The code required in each COBOL division in order to write to the Event Log is described below:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. program-name.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SPECIAL-NAMES.
     CONSOLE IS mnemonic-name.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 data-name PIC X(80).
 PROCEDURE DIVISION.
     DISPLAY data-name UPON mnemonic-name.
     DISPLAY " Nonnumeric literal" UPON mnemonic-name.
     EXIT PROGRAM.
 END PROGRAM program-name.
```

ENVIRONMENT DIVISION

Associate the mnemonic-name with SYSOUT, SYSERR, or CONSOLE.

DATA DIVISION

Define the data to be written to the Event Log

PROCEDURE DIVISION

To write to the Event Log, use a DISPLAY statement where the mnemonic-name associated with SYSOUT, SYSERR or CONSOLE is specified in UPON.

10.1.9.2 Program Compilation and Linkage

No specific compiler or linker options are required.

10.1.9.3 Program Execution

Specific environment variable settings are required at execution time when using UPON in DISPLAY statements to write to the Event Log.

For No UPON or SYSOUT

```
@CBR_DISPLAY_SYSOUT_OUTPUT = EVENTLOG(computer-name)
```

For SYSERR

```
@CBR_DISPLAY_SYSERR_OUTPUT = EVENTLOG(computer-name)
```

For CONSOLE

```
@CBR_DISPLAY_CONSOLE_OUTPUT = EVENTLOG(computer-name)
```

Note

- Each write to the Event Log is limited to 1024 characters.
- You cannot output to any other destinations (file, console, etc.) while writing to the Event Log.

- When writing to the Event Log, the Event Source Name is "NetCOBOL Application x64".
  When the Event Source Name is changed, "Addition and deletion of the registry key for the event log output sub routine" tool (COBSETER.exe) is used. The tool can only be used by an Administrators group user who has registry key access authority. Set the event-source-name as follows:

  If using No UPON or SYSOUT

  ```
  @CBR_DISPLAY_SYSOUT_EVENTLOG_SRCNAME = event-source-name
  ```

  If using SYSERR

  ```
  @CBR_DISPLAY_SYSERR_EVENTLOG_SRCNAME = event-source-name
  ```

  If using CONSOLE

  ```
  @CBR_DISPLAY_CONSOLE_EVENTLOG_SRCNAME = event-source-name
  ```

- When writing to the Event Log, the Event Level is "Information". The Event Level can be changed using the following environment variables:

  If using No UPON or SYSOUT

  ```
  @CBR_DISPLAY_SYSOUT_EVENTLOG_LEVEL = { I | W | E }
  ```

  If using SYSERR

  ```
  @CBR_DISPLAY_SYSERR_EVENTLOG_LEVEL = { I | W | E }
  ```

  If using CONSOLE

  ```
  @CBR_DISPLAY_CONSOLE_EVENTLOG_LEVEL = { I | W | E }
  ```

  - I : Information Event.

  - W : Warning Event.

  - E : Error Event.

- The Event Log can be output to other computers on the network. To do so, install this product (COBOL or COBOL runtime system) on the destination computer.

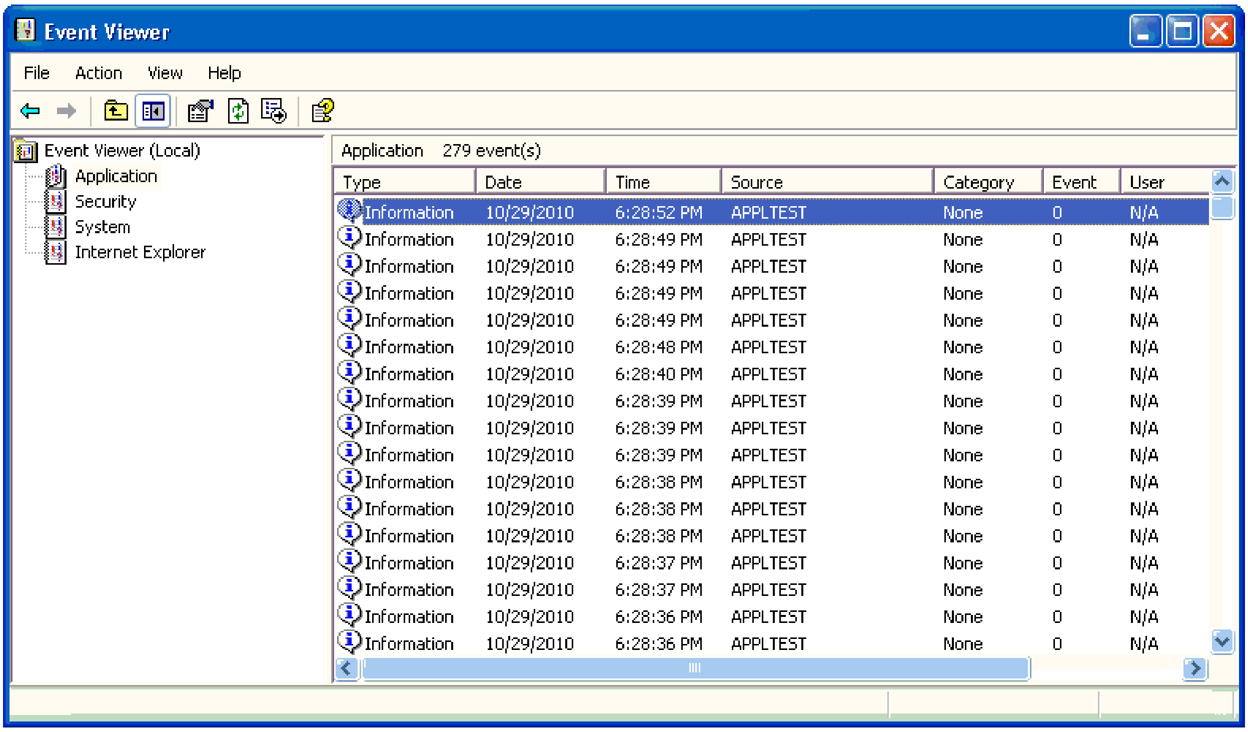
- If a write to the Event Log on the specified computer fails, an error message is generated in the Event Log of the executing computer and additional associated messages are output.

- Use the COB_REPORT_EVENT subroutine to output more detailed information to the Event Log. Refer to "18.2.2 Function that Outputs User-defined Information to the Event Log" for more information.

- You may need to set permissions to access the Event Log. For more information, refer to the Microsoft documentation for setting Event Log access rights.

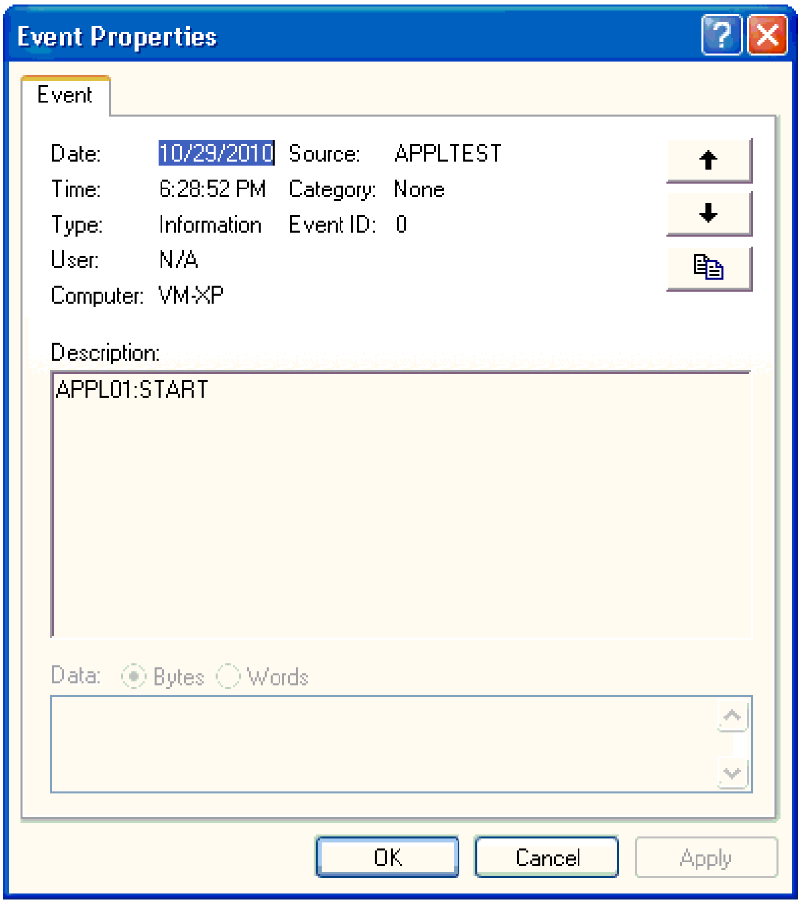
Example

Event Source Name : "APPLTEST"
Event Type : "W"
Data of DISPLAY statement : "APPL01:START"



Application 279 event(s)

Type	Date	Time	Source	Category	Event	User
Information	10/29/2010	6:28:52 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:49 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:49 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:49 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:49 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:48 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:40 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:39 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:39 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:39 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:38 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:38 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:38 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:37 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:37 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:36 PM	APPLTEST	None	0	N/A
Information	10/29/2010	6:28:36 PM	APPLTEST	None	0	N/A



Date: 10/29/2010 Source: APPLTEST
Time: 6:28:52 PM Category: None
Type: Information Event ID: 0
User: N/A
Computer: VM-XP

Description:

APPL01:START

Data: ● Bytes ○ Words

10.2 Fetching Command Line Arguments

This section explains how to access the number, and values, of arguments specified in the command line used to invoke an application. For example, the following shows a command line with three arguments.

10.2.1 Outline

You associate mnemonic names with two special function names: ARGUMENT-NUMBER and ARGUMENT-VALUE. The ARGUMENT-NUMBER name is used to retrieve the number of arguments in the command line, and to set the argument number for the next argument to be retrieved. The ARGUMENT-VALUE name is used for retrieving the argument values.

To obtain the number of arguments, you ACCEPT FROM the ARGUMENT-NUMBER name.

To set the argument number for the next argument value to be retrieved you DISPLAY the argument number UPON the ARGUMENT-NUMBER name.

To retrieve an argument value, you ACCEPT FROM the ARGUMENT-VALUE name.

A character string delimited with spaces or quotation marks (") is counted as one argument.

```
SAMPLE  J.SMITH  901234  SEATTLE
```

10.2.2 Program Specifications

This section explains program descriptions for each COBOL division when the command line argument-handling functions are used. The code below shows the syntax explained in the following topics.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID.  program-name.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SPECIAL-NAMES.
     ARGUMENT-NUMBER IS mnemonic-name-1
     ARGUMENT-VALUE  IS mnemonic-name-2.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  data-name-1 ... .
  01  data-name-2 ... .
  01  data-name-3 ... .
 PROCEDURE DIVISION.
     ACCEPT  data-name-1     FROM mnemonic-name-1.
    [DISPLAY numeric-literal UPON mnemonic-name-1.]
    [DISPLAY data-name-2     UPON mnemonic-name-1.]
     ACCEPT  data-name-3     FROM mnemonic-name-2
                                  [ON EXCEPTION ... ].
 END PROGRAM  program-name.
```

ENVIRONMENT DIVISION

Associate the following function-names with mnemonic-names:

- ARGUMENT-NUMBER
- ARGUMENT-VALUE

```
ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SPECIAL-NAMES.
     ARGUMENT-NUMBER IS mnemonic-name-1
     ARGUMENT-VALUE  IS mnemonic-name-2.
```

DATA DIVISION

Define data items to deliver values.

Table 10.3 Counts and values of arguments

Contents	Attribute
Number of arguments	Unsigned numeric data item
Argument position (not required if specified with a literal)	Unsigned numeric data item
Argument value	Fixed-length group item or alphanumeric data item

The following is a definition example of data items:

```
DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  number-of-arguments  PIC 9(2)  BINARY.
 01  argument-position    PIC 9(2)  BINARY.
 01  argument-value.
     02  argument-value-1  PIC X(10)  OCCURS 1 TO 10 TIMES
                           DEPENDING ON number-of-arguments.
```

PROCEDURE DIVISION

You find out the number of arguments that have been specified in the command line by using ACCEPT FROM the ARGUMENT-NUMBER mnemonic name.

The number of arguments are set in the data name specified in the ACCEPT statement.

To retrieve an argument value, specify the position of the argument using the DISPLAY statement ([1]) corresponding to the function name ARGUMENT-NUMBER. Then, retrieve the argument value using the ACCEPT statement ([2]) corresponding to the function name ARGUMENT-VALUE. The argument value is set in the data name specified in the ACCEPT statement. If a nonexistent argument position is specified (such as argument position 4 specified while only three arguments are used), an exception condition occurs. If an exception condition occurs, the statement ([3]), if written with ON EXCEPTION in the ACCEPT statement, is executed. For a program that does not execute the DISPLAY statement for positioning, the argument position is set to 1 when the program begins to run. Thereafter, the argument position is set to the number given below each time an ACCEPT statement is executed.

Argument positions are numbered from 0 to 99. Argument position 0 is a command name.

```
DISPLAY 5                  UPON mnemonic-name-1.     --- [1]
ACCEPT  argument-value-1(5)  FROM mnemonic-name-2    --- [2]
    ON EXCEPTION MOVE 5 TO error-number              --- [3]
                 GO TO error-processing
END-ACCEPT.
```

Note

- When a program references an argument value without executing the DISPLAY statement for positioning, the argument position is set to 1 at the beginning of program execution, and then set to the following position number each time an ACCEPT statement is executed.

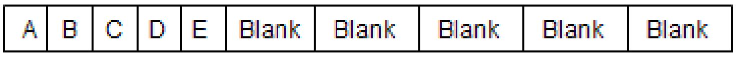
- The lengths of the argument values cannot be obtained.

- The rules for the COBOL MOVE statement are applied to the setting of data items for both the number of arguments and argument values. For example:

```
   01  number-1   PIC  9.
   01  argument-1 PIC  X(10).
PROCEDURE DIVISION.
       :
       ACCEPT number-1  FROM argument-number.   *>...  (1)
       ACCEPT argument-1 FROM argument-value.   *>...  (2)
```

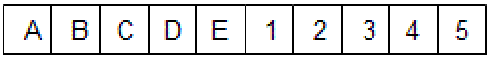
If statement (1) is executed when the number of arguments specified in the command is 10, the contents of "number-1" is 0.

If statement (2) is executed when the value of an argument to be fetched is "ABCDE", the content of "argument-1" is as follows:



A	B	C	D	E	Blank	Blank	Blank	Blank	Blank

If statement (2) is executed when the value of an argument to be fetched is "ABCDE12345FGHIJ", the content of "argument-1" is as follows:



A	B	C	D	E	1	2	3	4	5

10.2.3 Program Compilation and Linkage

No specific compiler and linker options are required.

10.2.4 Program Execution

Execute programs as ordinary programs.

Note

- The command-line retrieval functions should really only be used with main COBOL programs.
- With a COBOL program called from another COBOL program, the values of arguments returned are those of the arguments specified on the command line that activated the application.

10.3 Environment Variable Handling Function

This section explains how to refer to and update the values of environment variables. Environment variables explained in this section indicate runtime environment information set at program execution.

10.3.1 Outline

During program execution, the values of environment variables can be referred to and updated.

To refer to the value of an environment variable, use a DISPLAY statement where a mnemonic-name corresponding to function-name ENVIRONMENT-NAME is specified and an ACCEPT statement for which a mnemonic-name corresponding to function-name ENVIRONMENT-VALUE is specified.

To update the value of an environment variable, use a DISPLAY statement where a mnemonic-name corresponding to function-name ENVIRONMENT-NAME is specified and a DISPLAY statement where a mnemonic-name corresponding to function-name ENVIRONMENT-VALUE is specified.

10.3.2 Programs Specifications

This section explains program descriptions for each COBOL division when using the environment variables handling function. The code below shows the syntax explained in the following topics.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. program-name.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
```

```
  ENVIRONMENT-NAME  IS mnemonic-name-1
  ENVIRONMENT-VALUE IS mnemonic-name-2.
DATA DIVISION.
 WORKING-STORAGE SECTION.
01 data-name-1 ...
01 data-name-2 ...
PROCEDURE DIVISION.
   DISPLAY {"nonnumeric-literal" | data-name-1} UPON mnemonic-name-1.
   ACCEPT data-name-2 FROM mnemonic-name-2 [ON EXCEPTION ...].
   DISPLAY {"nonnumeric-literal" | data-name-1} UPON mnemonic-name-1.
   DISPLAY data-name-2 UPON mnemonic-name-2 [ON EXCEPTION ...].
END PROGRAM program-name.
```

ENVIRONMENT DIVISION

Associate the following function-names with mnemonic-names:

- ENVIRONMENT-NAME
- ENVIRONMENT-VALUE

DATA DIVISION

Define data items to deliver values.

Table 10.4 Attributes of environment variables

Contents	Attribute
Name of an environment variable (not required if specified with a literal)	Fixed-length group item or alphanumeric data item
Value of an environment variable (not required if specified with a literal)	Fixed-length group item or alphanumeric data item

PROCEDURE DIVISION

To refer to the value of an environment variable, first, specify the environment variable name to be referred to with a DISPLAY statement (see the line marked (1) in the code below) where the mnemonic-name corresponding to function-name ENVIRONMENT-NAME is specified. Then, refer to the value of the environment variable with an ACCEPT statement (2) where the mnemonic-name corresponding to function-name ENVIRONMENT-VALUE is specified.

Values of environment variables are set to the data name specified in the ACCEPT statement.

If the name of the environment variable to be referred to has not been specified or the name of a non-existing environment variable has been specified, an exception condition occurs. In this case, a statement (3) specified for ON EXCEPTION is executed.

To update the value of an environment variable, first, specify the environment variable name to be updated with a DISPLAY statement (4) using a mnemonic-name corresponding to function-name ENVIRONMENT-NAME. Then use a DISPLAY UPON statement (5), which uses a mnemonic-name corresponding to function-name ENVIRONMENT-VALUE, to set the value of this environment variable.

If the name of the environment variable to be updated has not been specified or the area to set the value of the environment variable cannot be assigned, an exception condition occurs. In this case, a statement (6) specified for ON EXCEPTION is executed.

```
 DISPLAY "TMP1"        UPON environment-variable-name   *>... (1)
 ACCEPT value-of-TMP1  FROM environment-variable-value  *>... (2)
   ON EXCEPTION                                         *>... (3)
     MOVE  occurrence-of-error TO error-code
 END-ACCEPT.
*>    :
 DISPLAY "TMP2"        UPON environment-variable-name.  *>... (4)
 DISPLAY value-of-TMP2 UPON environment-variable-value  *>... (5)
   ON  EXCEPTION                                        *>... (6)
     MOVE  occurrence-of-error TO error-code
 END-DISPLAY.
```

Note

The lengths of environment variables cannot be obtained.

10.3.3 Program Compilation and Linkage

No specific compiler and linker options are required.

10.3.4 Program Execution

Execute programs as ordinary programs.

Note

The value of an environment variable changed during program execution is valid only during program execution.

Chapter 11 Using SORT/MERGE Statements (Sort-Merge Function)

Sort rearranges the records in a file according to a certain sequence, while merge integrates the records in multiple files into one file. This chapter describes the sort/merge function, looking at the types of sort and merge processing.

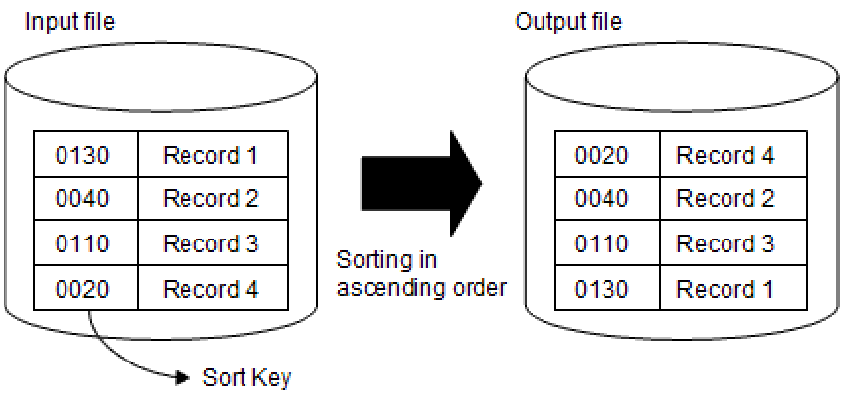
11.1 Outline of Sort and Merge Processing

This section outlines sort and merge processing.

Sort

Sort means that records in a file are rearranged in ascending or descending order using record information as a sort key. The records are rearranged by the collating sequence appropriate to the attribute of the key item.

Figure 11.1 Sorting records



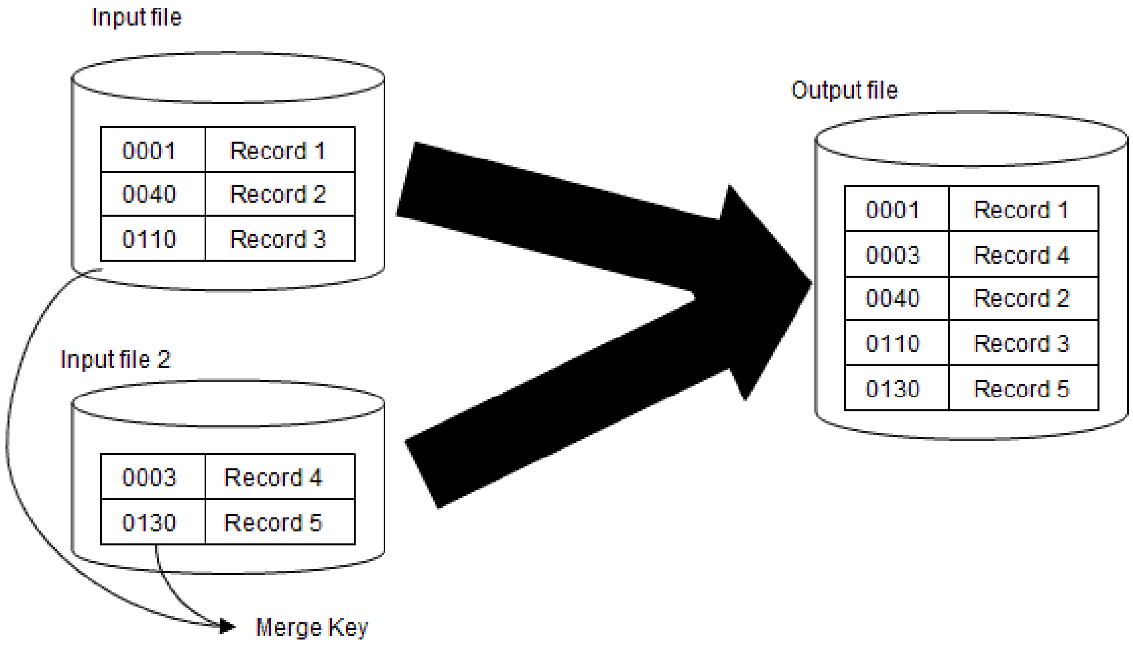
0130	Record 1
0040	Record 2
0110	Record 3
0020	Record 4

0020	Record 4
0040	Record 2
0110	Record 3
0130	Record 1

Merge

Merge means that records in multiple files with records sorted in ascending or descending order are integrated into one file.

Figure 11.2 records



0001	Record 1
0040	Record 2
0110	Record 3

0003	Record 4
0130	Record 5

0001	Record 1
0003	Record 4
0040	Record 2
0110	Record 3
0130	Record 5

11.2 Using Sort

This section explains the types of sort processing, and how to write, compile, link, and execute a program that uses sort processing.

Note

Please install PowerBSORT to use sort.

11.2.1 Types of Sort Processing

The following four types of sort processing are possible:

1. All records in the input file are sorted in ascending or descending order, then are written to the output file: (Input) file (Output) file
2. All records in the input file are sorted in ascending or descending order, then are handled as output data: (Input) file (Output) records
3. Specific records or data items are sorted in ascending or descending order, then are written to the output file: (Input) record (Output) file
4. Specific records or data items are sorted in ascending or descending order, and are handled as output data: (Input) record (Output) record

When sorting records in the input file (file to sort) without changing their contents, sort processing type 1 or 2 is normally used. When an input file is not used or the content of a record is changed, sort processing type 3 or 4 is used.

When writing sorted records to the output file without processing the contents of each record, sort processing type 1 or 3 is used. When the output file is not used or the content of each record is changed, sort processing type 2 or 4 is used.

11.2.2 Program Specifications

This section explains the contents of a program that sorts records for each COBOL division.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. program-name.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT sort-file-1   ASSIGN TO SORTWK01.
 [SELECT input-file-1  ASSIGN TO file-reference-identifier-1 ...]
 [SELECT output-file-1 ASSIGN TO file-reference-identifier-2 ...]
DATA DIVISION.
 FILE SECTION.
 SD sort-file-1 [RECORD record-size].
 01 sort-record-1.
   02 sort-key-1 ...
   02 data1 ...
 FD input-file-1 ...
 01 input-record-1 ...
    [Contents of input record]
 FD output-file-1 ...
 01 output-record-1 ...
    [Contents of output record]
PROCEDURE DIVISION.
    SORT sort-file-1 ON
         {ASCENDING | DESCENDING} KEY sort-key-1
         {USING input-file-1 | INPUT PROCEDURE IS input-procedure-1}
         {GIVING output-file-1 | OUTPUT PROCEDURE IS output-procedure-1}.
    EXIT PROGRAM.
input-procedure-1 SECTION.
```

```
    OPEN INPUT input-file-1.
input-start.
    READ input-file-1 AT END GO TO input-end.
    MOVE input-record-1 TO sort-record-1.
    RELEASE sort-record-1.
    GO TO input-start.
input-end.
    CLOSE input-file-1.
input-exit.
    EXIT.
output-procedure-1 SECTION.
    OPEN OUTPUT output-file-1.
output-start.
    RETURN sort-file-1 AT END GO TO output-end END-RETURN.
    MOVE sort-record-1 TO output-record-1.
    WRITE output-record-1.
    GO TO output-start.
output-end.
    CLOSE output-file-1.
output-exit.
    EXIT.
END PROGRAM program-name.
```

ENVIRONMENT DIVISION

The following file must be defined.

- Sort-merge file (SD)

  A work file for sort processing must be defined. The ASSIGN clause is assumed as a comment; up to 8 alphanumeric characters (the first character must be alphabetic) must be specified in it.

The following files are defined if required.

- Input file (FD)

  Define the same way as for ordinary file processing.

- Output file (FD)

  Define the same way as for ordinary file processing.

DATA DIVISION

Define the records of files defined in the ENVIRONMENT DIVISION.

PROCEDURE DIVISION

The SORT statement is used for sort processing. The contents of the SORT statement differ depending on the chosen processing method.

If you use the SORT-CORE-SIZE special register, you can restrict the capacity of the memory space used by PowerBSORT. This special register is a numerical item defined using PIC S9(8) BINARY implicitly. The settings value is a numerical value expressed in bytes.

- For file input, "USING input-file-name" must be specified.
- For record input, "INPUT PROCEDURE input-procedure-name" must be specified.
- For file output, "GIVING output-file-name" must be specified.
- For record output, "OUTPUT PROCEDURE output-procedure-name" must be specified.

The input procedure specified in INPUT PROCEDURE can pass records to be sorted one-by-one with the RELEASE statement.

The output procedure specified in OUTPUT PROCEDURE can receive sorted records one-by-one with the RETURN statement.

Multiple sort keys can be specified.

When all records are sorted, the sort result is set in special register SORT-STATUS. Special register SORT-STATUS need not be defined in the COBOL program since it is automatically generated.

By checking the SORT-STATUS value after the SORT statement is executed, COBOL program execution can continue even if sort processing terminates abnormally. Setting 16 in SORT-STATUS in the input or output procedure specified by the SORT statement terminates sort processing.

Table below lists the valid values for special register SORT-STATUS and their meanings.

Table 11.1 SORT-STATUS values and their meanings

Value	Meaning
0	Normal termination
16	Abnormal termination

Note

Any input and output files used must be closed during SORT statement execution.

Although this special register is equivalent to the SMSIZE() compile option and the smsize option, if more than one specified at the same time, the SORT-CORE-SIZE special register has the highest priority, the smsize option at the time of execution has the next highest priority, and the SMSIZE() compiler option has the third highest priority.

Example

```
    Special register     MOVE 102400 TO SORT-CORE-SIZE
    (102400=100 kilobytes)
    Compile option    SMSIZE(500K)
    Execution time option smsize300k
```

In this case, the SORT-CORE-SIZE special register value of 100 kilobytes has the highest priority.

11.2.3 Program Compilation and Linkage

Compiler option EQUALS may be specified if required. Refer to "A.2.14 EQUALS(same key data processing in SORT statements)".

When more than one record has the same sort key value in sort processing, EQUALS guarantees that the records are written in the same order as they are read.

Note

Using this compiler option degrades performance.

11.2.4 Program Execution

Execute the program that uses sort as follows:

1. Set environment variable BSORT_TMPDIR.

    - In sort processing, the temporary file is necessary.

2. Assign input and output files

    When input and output files are defined using file-identifiers, use these identifiers as environment variables to set the names of input and output files.

3. Execute the program

Note

Sort merge processing is usually done by the COBOL runtime system. If PowerBSORT is installed, PowerBSORT is used. When PowerBSORT is used the work file is made in the following folder (shown in priority order):

1. The folder specified by environment variable BSORT_TMPDIR
2. The folder specified by environment variable TEMP
3. The folder specified by environment variable TMP
4. The Windows system folder

11.3 Using Merge

This section explains the types of merge processing and how to write, compile, link, and execute a program that uses merge processing.

Note

Please install PowerBSORT to use merge.

11.3.1 Types of Merge Processing

The following two types of merge processing are possible:

1. Write all records in multiple files already sorted in ascending or descending order to the output file in ascending or descending order: (Input) file (Output) file
2. Processing all records in multiple files already sorted in ascending or descending order as output data in ascending or descending order: (Input) file (Output) records

When merged records are written to the output file without changing the record contents, merge processing type 1 is normally used. When an output file is not used or record contents are changed, merge processing type 2 is used.

11.3.2 Program Specifications

This section explains the contents of a program that uses merge for each COBOL division.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. program-name.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT merge-file-1  ASSIGN TO SORTWK01.
    SELECT input-file-1  ASSIGN TO file-reference-identifier-1 ...
    SELECT input-file-2  ASSIGN TO file-reference-identifier-2 ...
   [SELECT output-file-1 ASSIGN TO file-reference-identifier-3 ...]
DATA DIVISION.
 FILE SECTION.
 SD merge-file-1 [RECORD record-size].
 01 merge-record-1.
   02 merge-key-1 ...
   02 data1 ...
 FD input-file-1 ...
 01 input-record-1 ...
   [Contents of input record]
 FD input-file-2 ...
 01 input-record-2 ...
   [Contents of input record]
 FD output-file-1 ...
 01 output-record-1 ...
   [Contents of output record]
PROCEDURE DIVISION.
    MERGE merge-file-1 ON
          {ASCENDING | DESCENDING} KEY merge-key-1
             USING input-file-1 input-file-2 ...
          {GIVING output-file-1 | OUTPUT PROCEDURE IS output-procedure-1}.
    EXIT PROGRAM.
output-procedure-1 SECTION.
    OPEN OUTPUT output-file-1.
output-start.
    RETURN merge-file-1 AT END GO TO output-end END-RETURN.
    MOVE merge-record-1 TO output-record-1.
    WRITE output-record-1.
    GO TO output-start.
output-end.
    CLOSE output-file-1.
output-exit.
    EXIT.
END PROGRAM program-name.
```

ENVIRONMENT DIVISION

The following files must be defined:

- Sort-merge file

  A work file for merge processing must be defined. The ASSIGN clause is assumed as a comment; up to 8 alphanumeric characters (the first character of which must be alphabetic) must be specified in it.

- Input file

  All files to be merged must be defined

- Output file

  Define the same way as for ordinary file processing, if required.

DATA DIVISION

Define the records of files defined in the ENVIRONMENT DIVISION.

PROCEDURE DIVISION

The MERGE statement is used for merge processing. The contents of the MERGE statement differ depending on whether file or record output is to be used for merge processing.

If you use the SORT-CORE-SIZE special register when you install PowerBSORT, you can restrict the capacity of the memory space used by PowerBSORT. This special register is a numerical item defined using PIC S9(8) BINARY implicitly. The settings value is a numerical value expressed in bytes.

- For file output, "GIVING output-file-name" must be specified.
- For output record processing, "OUTPUT PROCEDURE output-procedure-name" must be specified.

The output procedure specified in OUTPUT PROCEDURE can receive merged records one-by-one by using the RETURN statement.

Multiple merge keys can be specified.

When all records are merged, the merge result is set in special register SORT-STATUS.

Unlike general data, special register SORT-STATUS need not be defined in the COBOL program since it is automatically generated. By checking the SORT-STATUS value after MERGE statement execution, COBOL program execution can continue even if merge processing terminates abnormally.

Setting 16 in SORT-STATUS in the output procedure specified by the MERGE statement terminates merge processing. The following table lists the possible values for special register SORT-STATUS and their meanings.

Table 11.2 SORT-STATUS values and their meanings

Value	Meaning
0	Normal termination
16	Abnormal termination

Note

Any input and output files used must be closed during MERGE statement execution.

Although this special register is equivalent to the SMSIZE() compile option and the smsize option, when more than one of these is specified at the same time, the SORT-CORE-SIZE special register has the highest priority, the smsize option at the time of execution has the next highest priority, and the SMSIZE() compiler option has the third highest priority.

Example

```
    Special register     MOVE 102400 TO SORT-CORE-SIZE
    (102400=100 kilobytes)
    Compile option   SMSIZE(500K)
    Execution time option smsize300k
```

In this case, the SORT-CORE-SIZE special register value of 100 kilobytes has the highest priority.

11.3.3 Program Compilation and Linkage

No specific compiler or linkage options are required.

11.3.4 Program Execution

1. Assign input and output files.

   When input and output files are defined using file-identifiers, use these identifiers as environment variables to set the names of input and output files.

2. Execute the program.

Chapter 12 System Program Functions

This chapter describes the functions that are useful when creating system programs. Included in this chapter are explanations of how to use pointers, the ADDR and LENG functions, and the PERFORM statement with a NO LIMIT phrase.

12.1 Types of System Program Functions

NetCOBOL has the following functions that are not included in the COBOL standards. These functions are called the system program functions in NetCOBOL. The following functions are useful when creating system programs:

- Pointer
- ADDR function and LENG function
- PERFORM statement with a NO LIMIT phrase

The following section outlines and explains the features of each function.

Pointer

An area with a specific address can be referenced and updated using a pointer.

For example, when a parameter identifying an area address is used to call a COBOL program from a program written in another language, the area contents at that address can be referenced or updated with a pointer in the COBOL program.

ADDR Function and LENG Function

The ADDR function can obtain the address of a data item defined by COBOL.

The LENG function can obtain the lengths of a data item and literal defined by COBOL in bytes.

For example, an area address or length can be passed as a parameter to call a program written in another language from a COBOL program.

PERFORM Statement with a NO LIMIT phrase

In NetCOBOL, the PERFORM statement can be written with a NO LIMIT phrase. This allows you to PERFORM a section of code indefinitely until an explicit exit, that you code, is encountered.

12.2 Using Pointers

This section explains how to use pointers.

12.2.1 Outline

A pointer is used to reference an area having a specific address. The following data items are required to use a pointer:

- Data item defined in BASED-STORAGE section (a-item)
- Data item whose attribute is pointer data item (b-item)

The pointer is normally used with the pointer qualifier (->). (a-item) is pointed to by (b-item). This is called pointer qualification, as shown in the following example:

```
(b-item) -> (a-item)
```

In this case, the contents of (a-item) are those of the area whose address is set in (b-item).

12.2.2 Program Specifications

This section explains the contents of a program that uses pointers for each COBOL division.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID.  program-name.
DATA DIVISION.
 BASED-STORAGE SECTION.
 01  data-name-1 ... [BASED ON pointer-1].
 77  data-name-2 ... [BASED ON pointer-2].
 WORKING-STORAGE SECTION.
 01  pointer-1  POINTER.
 LINKAGE SECTION.
 01  pointer-2  POINTER.
PROCEDURE DIVISION USING  pointer-2.
    MOVE [pointer-1 ->] data-name-1 ... .
    IF   [pointer-2 ->] data-name-2 ... .
END PROGRAM  program-name.
```

ENVIRONMENT DIVISION

No specifications are required.

DATA DIVISION

The data-names where addresses are specified for reference or update must be defined in the BASED-STORAGE section. The data-names for storing addresses (with the attribute of pointer data item (POINTER)) must also be defined in the FILE section, WORKING-STORAGE section, BASED-STORAGE section, and LINKAGE section.

Defining Data-Names in BASED-STORAGE Section

A data-name can be defined in the BASED-STORAGE section by using a data description entry the same way as defining ordinary data.

An actual area is not secured for a data-name defined in the BASED-STORAGE section during program execution. Thus, when referencing a data item defined in BASED-STORAGE, specify the address of the area to reference.

When the BASED ON clause is specified in a data description entry, the data-name can be used without pointer qualification since a pointer is implicitly given by the data-name specified in the BASED ON clause. When using a data-name where the BASED ON clause is not specified, pointer qualification is required.

PROCEDURE DIVISION

A data-name with a pointer given can be specified in such statements as the MOVE and IF statements just like an ordinary data-name.

12.2.3 Program Compilation and Linkage

No specific compiler and linkage options are required.

12.2.4 Program Execution

No specific environment settings are required.

12.3 Using the ADDR and LENG Functions

This section explains how to use the ADDR and LENG functions.

12.3.1 Outline

The ADDR function returns the address of a data item as a function value. The LENG function returns the size of a data item or literal in bytes.

12.3.2 Program Specifications

This section explains the contents for each COBOL division of a program that uses the ADDR and LENG functions.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. program-name.
DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  data-name-1 ... .
 01  pointer-1  POINTER.
 01  data-name-2.
   02  ... [ OCCURS  ...  DEPENDING ON ...].
 01  data-name-3  PIC 9(4) BINARY.
PROCEDURE DIVISION.
    MOVE FUNCTION  ADDR(data-name-1) TO pointer-1.
    MOVE FUNCTION  LENG(data-name-2) TO data-name-3.
END PROGRAM  program-name.
```

ENVIRONMENT DIVISION

No specifications are required.

DATA DIVISION

The data-names for storing functions values returned by the ADDR and LENG functions must be defined.

The attribute of a function value of the ADDR function is a pointer data item.

The attribute of a function value of the LENG function is a numeric data item.

PROCEDURE DIVISION

The ADDR and LENG functions can be specified in such statements as the MOVE and IF statements, the same as with ordinary data-names.

12.3.3 Program Compilation and Linkage

No specific compiler and linkage options are required.

12.3.4 Program Execution

No specific environment settings are required.

12.4 Using the PERFORM Statement with a NO LIMIT phrase

This section explains how to use the PERFORM statement with a NO LIMIT phrase.

12.4.1 Outline

In certain circumstances, using a conventional PERFORM statement to determine the condition that will terminate repeated processing of the target code, complex coding is required. Using the PERFORM statement with the NO LIMIT phrase can simplify the required coding.

When PERFORM with NO LIMIT is used, the target code is executed repeatedly until either an explicit EXIT PERFORM is executed or a branch statement indicating an explicit transfer of control is executed.

12.4.2 Program Specifications

This section explains the contents of each division of a COBOL program that uses the PERFORM statement with a NO LIMIT phrase.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID.  program-name.
PROCEDURE DIVISION.
    PERFORM WITH NO LIMIT
      IF  ...
        EXIT PERFORM
      END-IF
       :
    END-PERFORM.
END PROGRAM  program-name.
```

ENVIRONMENT DIVISION

No specifications are required.

DATA DIVISION

No specifications are required.

PROCEDURE DIVISION

WITH NO LIMIT must be specified for the PERFORM statement to prevent the system generating code to determine the at end condition. When PERFORM WITH NO LIMIT is specified, the group of statements is executed repeatedly until an explicit exit or transfer of control is encountered.

When you use the NO LIMIT phrase you must include statements in the performed code to determine when the iteration should end, and to exit the repeated processing. If you do not specify a statement to exit the repeat processing, processing continues indefinitely (in an infinite loop).

12.4.3 Program Compilation and Linkage

No specific compiler and linkage options are required.

12.4.4 Program Execution

No specific environment settings are required.

Chapter 13 Introduction to Object-Oriented Programming

This chapter introduces object-oriented programming. It looks at why OO programming has been incorporated into COBOL, considers the goals of object orientated programming, and explains the key concepts involved.

The sections are:

- Overview
- Why OO COBOL?
- Goals of object-oriented programming
- Concepts of object-oriented programming

13.1 Overview

Object-oriented (OO) programming introduces new concepts and terminology to the traditional COBOL programming model. To help you place these concepts in context, this chapter first reviews why OO and COBOL have been combined. It then lists the goals of OO programming and describes the key concepts that have been developed to meet those goals.

13.2 Why OO COBOL?

Before tackling the concepts of OO COBOL you may be asking why should an object-oriented version of COBOL be created and why should it be of interest to you? The following sections give you a brief answer to these questions and provide some groundwork for the object-oriented concepts.

13.2.1 COBOL is Alive!

COBOL is a living language. COBOL vendors have always sought to provide their customers access to the latest technology and techniques, and the COBOL standard has been updated regularly to support the best practices.

Object-oriented design and programming is now a widely accepted technique. It therefore makes sense that COBOL should facilitate the implementation of object-oriented designs.

13.2.2 Software Development Challenges

The major software development challenges are:

1. Providing the users what they really want.
2. Providing it in a reasonable timeframe.
3. Providing it to an acceptable quality standard.
4. Enhancing it throughout its life.

Solving these challenges in an optimal way requires that issues in the areas of people, processes, products and technology be addressed. Although you need to look to other sources for advice on people, processes and products, COBOL has long been recognized as one of the best technologies for business applications. By embracing the best software practices and supporting OO design, OO COBOL continues to be a major player in the technology area.

13.2.3 Best Software Practices

The 1985 COBOL standard brought structured programming constructs to the COBOL language. Two other software engineering techniques have been proven to contribute greatly to productivity, quality and maintainability. These are the techniques of information hiding and modularity.

13.2.3.1 Information Hiding

Information hiding is the principle of confining the availability of data, or sets of changeable design decisions, to a single module. Other routines wanting access to the information make calls, or send messages, to the owning module. The interface to the module is kept constant (or as constant as possible) while the structure of the data being handled or the code within the module is changed.

Implementing information hiding in non-OO COBOL has been difficult. For example using PERFORM to execute a logical function requires the function code to know all about the structure of the data. Even when a function has been placed in a separate module, complete record structures are often passed to and from the module.

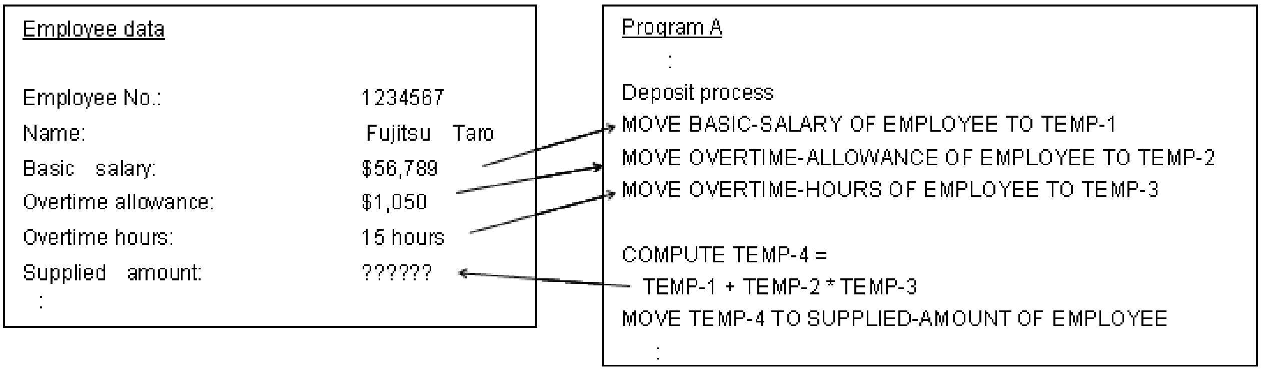
An information hiding capability is built into OO COBOL. The common OO term for this is "encapsulating" the data.

Example

1) No information hiding

Program A accesses the data directly.

Figure 13.1 Direct data access



Employee data

```
Employee No.:          1234567
Name:                  Fujitsu   Taro
Basic   salary:        $56,789
Overtime allowance:    $1,050
Overtime hours:        15 hours
Supplied   amount:     ??????
  :
```

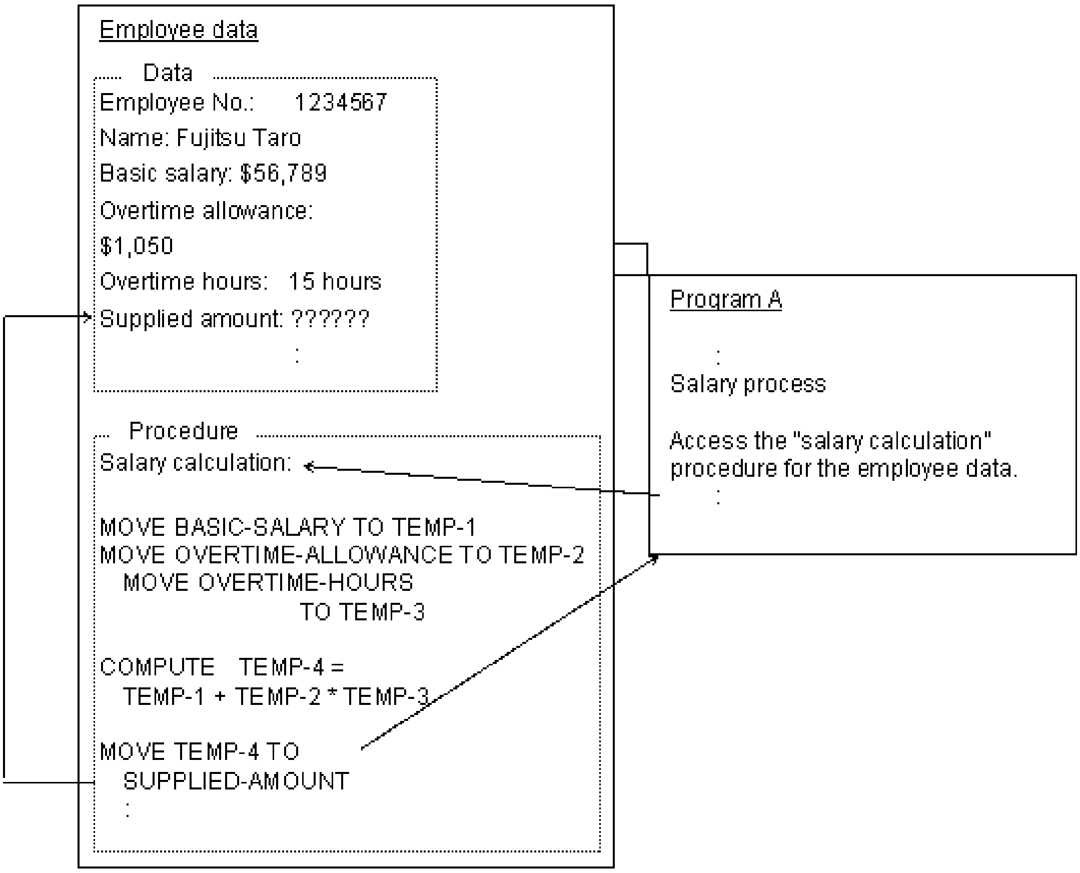
Program A

```
      :
Deposit process
MOVE BASIC-SALARY OF EMPLOYEE TO TEMP-1
MOVE OVERTIME-ALLOWANCE OF EMPLOYEE TO TEMP-2
MOVE OVERTIME-HOURS OF EMPLOYEE TO TEMP-3

COMPUTE TEMP-4 =
  TEMP-1 + TEMP-2 * TEMP-3
MOVE TEMP-4 TO SUPPLIED-AMOUNT OF EMPLOYEE
    :
```

2) Information hiding

Program A only affects the data by invoking the owning procedure (Salary calculation).

Figure 13.2 Data Hiding



13.2.3.2 Modularity

This is the principle of breaking the software up into discrete pieces according to common guidelines. The guidelines often vary from group to group, but typically will recommend that each module implements a single function and be limited to a certain number of lines of significant code.

Although a system may still have high overall complexity, breaking it up into modules helps people focus on smaller areas of the complexity.

COBOL has always supported a degree of modularity through the PERFORM and CALL statements. With the addition of methods to OO COBOL, modular thinking is strongly encouraged and the temptation to cross module boundaries is decreased.

13.2.4 Object-Oriented Design

One of the attractions of OO design is that it seeks to create a design model that directly matches real world objects. For example, a bank has customers and accounts, so an OO banking system has customer objects and account objects that contain the data and model the relevant behaviors of customers and accounts. Thus putting together an OO design should force a greater understanding of the users because the designers seek to build an accurate understanding of the real world objects. OO design should help development groups come closer to meet the first software challenge - that of delivering what the users really want or need.

You don't need to have an OO language to implement OO designs, but having an OO language simplifies the move from design to coding.

13.2.5 Code Reuse

One big attraction of OO programming is that more code can be made common and more code can be taken from one application to the next. You only have to write only the differential code. The features that provide this ability are classes and inheritance.

OO COBOL provides both these features. Fujitsu also provides a class browser to help you understand the classes and the inheritance structure.

13.2.6 Best Business Language

COBOL has long been recognized as the best language for implementing business applications. It continues to provide the benefits of readability, maintainability, solid file handling, and support for business data structures.

It makes sense to build on the wealth of COBOL expertise and years of investment in COBOL applications. As described in the paragraphs above OO COBOL gives you the ability to add all the benefits of OO programming to your COBOL foundation.

13.3 Goals of Object-Oriented Programming

Before explaining the concepts of OO programming it will help if you understand the goals that object-oriented programming seeks to achieve. The goals are listed below. The OO terms in bold are explained in detail in the "13.4 Concepts of Object-Oriented Programming" section.

Improve the match of the computer system to the real-world system by designing the system around real-world objects.

Reduce the impact of data structure changes by encapsulating the data. (Enforce information hiding.)

Reduce the impact of coding technique changes by modularizing functions in methods.

Gain greater code reuse by identifying related objects and allowing them (or more strictly the classes from which the objects are created) to share properties and behaviors through inheritance.

Gain greater code reuse by using to summarize common processes.

Gain greater coding productivity by providing the ability to write only the differential code when reusing code classes.

Increase program flexibility by allowing the same term to implement different behaviors depending on the context. (This feature is known as polymorphism.)

Ensure that incompatibilities in module interfaces are picked up quickly by checking interface conformance (at both compile and execution time).

13.4 Concepts of Object-Oriented Programming

13.4.1 Objects

At the center of object-oriented programming are objects. Objects model real-life objects, or things, with their attributes and behaviors. In software terms, objects are combinations of data and procedure code.

For example, a bank has customers. Customers have names, addresses, and bank accounts. They deposit money in and withdraw money from their accounts. An object-oriented view could have customer objects and account objects. Customer objects would have name and address attributes. Account objects would have balance and previous transaction attributes. The customer object might be part of the customer's interaction with an ATM and send messages to the account object to deposit or withdraw money. The system creates objects for every customer and every account.

Object Instances

In creating an object-oriented design, you identify groups of objects (such as customers and accounts) that have identical attributes and behaviors. These groups are called classes. You define the attributes (data) and behaviors (procedure code) for each class. When the object-oriented system needs to create a new customer, it copies the attributes from the appropriate customer class to create an "instance" of the customer object. Object instances are called "objects", "object instances" or just "instances". The instance is then populated with the data for that customer. The procedures defined for the class are associated with the new object.

13.4.2 Classes

When designing an object-oriented application, you define classes that are groups of objects that have common attributes and behaviors.

When implementing the object-oriented application you write code to define each class. The code contains data to represent the attributes and has procedures (called methods) to implement the behaviors. When you execute the application, the class definitions are used to create the object instances.

13.4.3 Abstract Classes

Abstract classes are simply classes that do not correspond to real-world objects. You can define abstract classes to contain code that is common across two or more of your real-world classes, or to handle functionality required by the computer system.

13.4.4 Factory Objects

In OO COBOL each class may have a special object associated with it, called the Factory Object. The factory object is responsible for creating new object instances for the class, destroying existing object instances and managing data associated with all object instances for that class.

For example, a factory object can maintain a count of the total number of instances.

13.4.5 Methods

The procedures associated with each class are called methods. Methods can be thought of as modules or subroutines. A class may have as many methods as it requires to manage its data.

13.4.6 Messages (Invoking Methods)

In OO terminology, objects communicate with one another by sending messages. In OO COBOL, messages are sent by invoking methods. Both conventional COBOL code and code within methods can invoke methods.

To invoke a method you identify the object that is to execute the method, supply the method name, and pass any parameters required by the method.

In OO COBOL you can either use the INVOKE statement (which is similar to a CALL) or use an in-line technique for invoking methods. The in-line technique looks like this:

```
MOVE <object-ID :: "method-name" (parameters)> TO target-item
```

The code within the "<" and ">" signs (these are not part of the syntax) causes the method to be invoked. The data returned by the method is moved to target-item.

13.4.7 Method Prototypes

You will often want to develop methods in different compilation units than the unit that invokes the method. However, you still want the compiler to check that your invocation of the method is correct - for example, that it provides the right number of parameters. OO COBOL lets you do this by providing a feature known as method prototypes.

Method prototypes look like method definitions but they do not have any procedure code. They only define the LINKAGE SECTION items and the PROCEDURE DIVISION USING ... RETURNING header. This enables the compiler and runtime system to do the necessary conformance checking.

13.4.8 Encapsulation

Encapsulation is the word generally used in OO design to describe information hiding. The data for an object can only be accessed and updated by the methods for that object. Other programs, or methods belonging to other objects, can only access an object's data by invoking the appropriate methods belonging to that object.

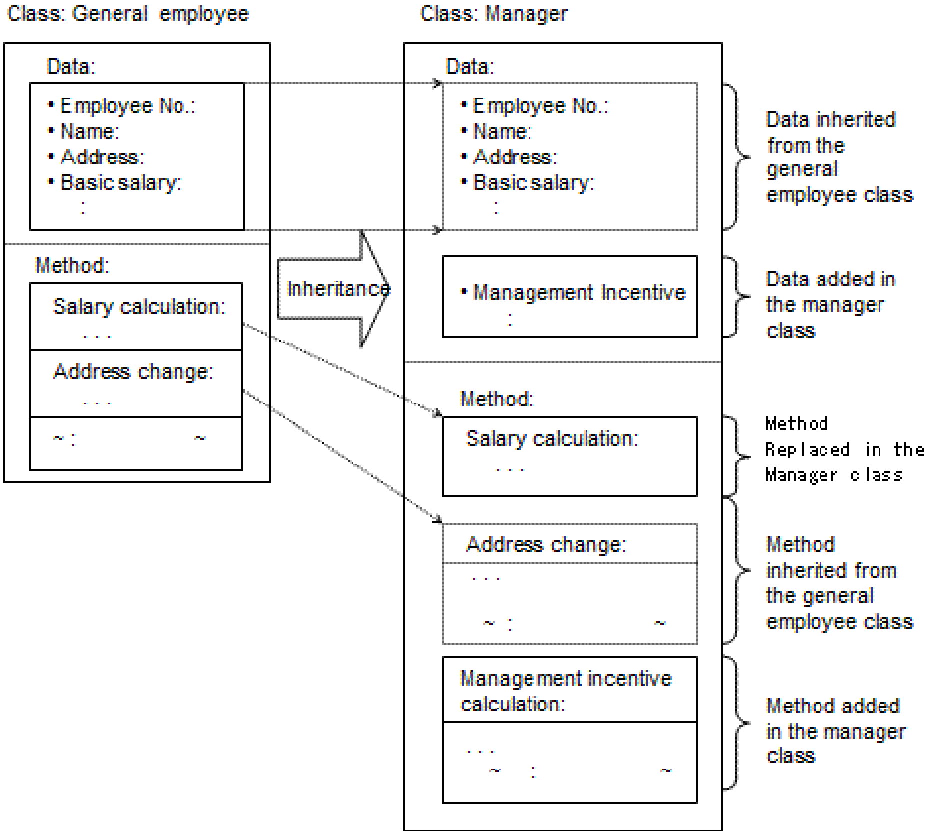
People talk about the data being "encapsulated" in the object.

13.4.9 Inheritance

A central feature of OO design is the concept of one class inheriting the properties of another class. By using this feature, objects that are similar to other objects, in all but a few details, can inherit all the features of the similar object and add or substitute code for the differences. This enables reuse of many code and design features.

For example, you could create a class for employees that contains all the data and methods for general employees and create a separate class for managers. Managers might have all the attributes of general employees but differ only in having a management incentive bonus. The manager class could inherit from the employee class and add the data and methods required to support the management incentive bonus - as illustrated in the following diagram:

Figure 13.3 Inheritance example



13.4.9.1 Parent, Child, Sub and Super Classes

We usually refer to the class that inherits the properties as the child or subclass, and the class from which the properties are being inherited as the parent or superclass.

The sub- and super- class terms have the advantage that they can refer to any class below or above the current class, not just the immediate parent or child.

13.4.9.2 Single Inheritance

The basic inheritance model has one parent for each child. This is referred to as a single inheritance model.

13.4.9.3 Multiple Inheritance

OO COBOL supports multiple inheritance where a child class can have one, two, or more parent classes. In this case, the child inherits properties from all its parents.

See the definition of the INHERITS clause in the "NetCOBOL Language Reference" for details of what happens if two or more of the parents contain methods with the same names.

13.4.10 Polymorphism

Polymorphism is a term used to describe using the same method name to perform slightly different operations on different objects or in different circumstances.

For example, you could have a "printObject" method defined for several different objects. The method would have the same general function - to print the contents of the object - but would differ in the details of the data and format used for the print report.

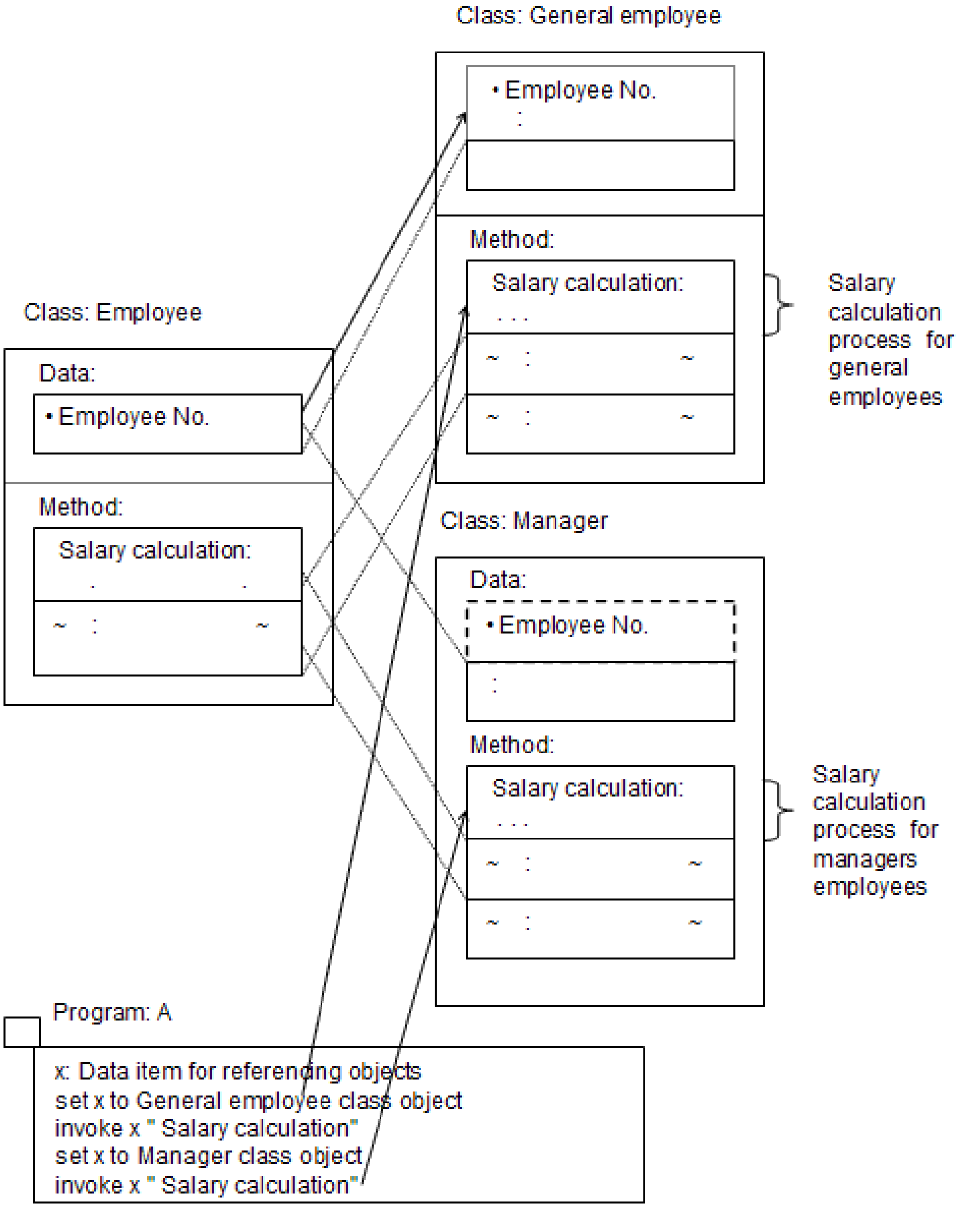
This can bring efficiencies in coding, as one line of code can request a similar function from several different objects, depending on the circumstances when the line is executed.

13.4.11 Binding

Binding is the process by which names in code are connected to actual data or procedures. Traditionally most binding happens at compile or link time. OO COBOL allows binding to happen at compile, link, load or run time. When the binding happens at run time it is referred to as dynamic binding, sometimes called late binding in other OO environments.

For example, you could have a general employee class and a manager employee class, both being children of an abstract employee class. Either, or both, might have their own method for calculating salaries. The diagram below illustrates how the system can only determine the correct bindings at run time. "x" is a special data item for holding references to objects.

Figure 13.4 Determining correct binding


Class: General employee
• Employee No.
:
Method:
Salary calculation:
. . .
~ : ~
~ : ~
Salary calculation process for general employees
Class: Employee
Data:
• Employee No.
Method:
Salary calculation:
. .
~ : ~
Class: Manager
Data:
• Employee No.
:
Method:
Salary calculation:
. . .
~ : ~
~ : ~
Salary calculation process for managers employees
Program: A
x: Data item for referencing objects
set x to General employee class object
invoke x " Salary calculation"
set x to Manager class object
invoke x " Salary calculation"

13.4.12 Conformance

Conformance is the term used in OO programming to state that two or more interfaces o match each other. It is generally used to describe one class conforming to another class. One class conforms to another class if you can take all the interfaces of the first class and execute them within the second class. I.e. The second class defines, through inheritance or otherwise, all the methods defined in the first class.

The inheritance rules generally ensure that superclasses conform to subclasses - because all the methods of the superclass are defined in the subclass.

Conformance impacts the COBOL programmer when handling object references and invoking methods. To ensure conformance between interfaces and objects the compiler checks:

That arguments passed to a method match the parameters specified in the USING phrase of the procedure division header.

That only valid references are moved to object references (when the object reference is constrained to refer only to a class or its subclasses).

For example, if a method requires three parameters A, B and C, conformance checking can warn you when you attempt to invoke the method with only two parameters. It can also inform you that you have declared data item B to be a different data type than the method expects.

Chapter 14 Basic Features of OO COBOL

This chapter explains how you implement the OO concepts introduced in "Chapter 13 Introduction to Object-Oriented Programming". The topics covered are:

- Overview of an OO COBOL application
- Defining classes
- Defining factory objects
- Defining objects
- Defining methods - including prototype methods
- Defining inheritance - including multiple inheritance
- Using repositories
- Working with objects - including types of object reference, predefined object identifiers and explicit and in-line method invocation
- Conformance checking
- Method binding
- Using properties.
- Using more advanced functions.

Overview of an OO COBOL Application

Designing an OO Application

In order to take full advantage of OO COBOL you need to learn and apply an OO design technique. Consult the latest literature to find a technique that works best for you. Your OO design framework will guide the structure for your OO COBOL application.

Before you start coding, you should also consider how you want to structure your sources. For example, you can define all the methods and data for a class within a single source file, or you can choose to build your class definition from prototype definitions and define the methods in separate source files. The former style may be convenient for small applications, the latter style - of putting methods in separate source files - is likely to be the style of choice for larger applications.

Once you know your design and your coding style you can start building your classes. Most people create their data definitions and prototype method definitions, then implement the method procedure code in an order that enables incremental testing.

Samples and Style Used in this Text

The samples in this text are based on (but will not always be identical to) code from the samples supplied with NetCOBOL (SAMPLE 15 through SAMPLE 20).

Both text and samples follow the style of using method prototypes within class definitions and putting the method data and procedure code in separate source files.

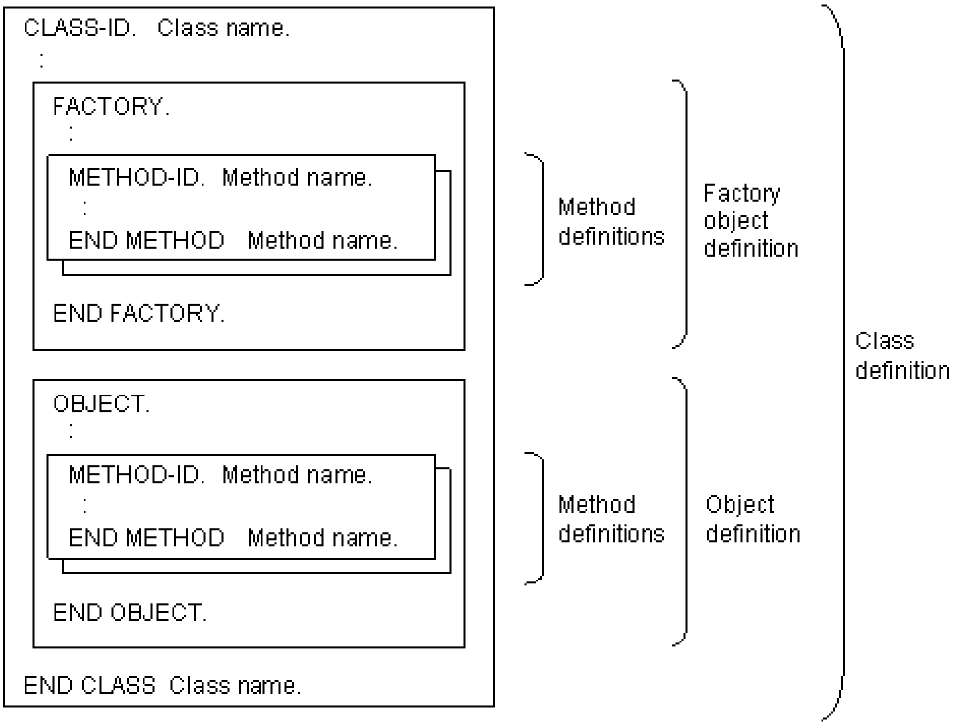
14.1 The OO COBOL Source Structure

OO COBOL adds the following elements to the standard COBOL program structure:

- Class definition
- Factory object definition
- Object definition
- Method definition

These all form part of the class definition that is structured as shown in Figure below:

Figure 14.1 OO COBOL Source Structure



Each of these new structures - class, factory, object and method -are based on the standard COBOL division structure of identification division, environment division, data division and procedure division. The only departure from this structure is that a class does not have its own data and procedure divisions - any data and procedure code required are defined in the factory and object definitions.

The table below provides a brief summary of these new structures:

Table 14.1 OO COBOL Structures

Class	Factory Object
IDENTIFICATION DIVISION. CLASS-ID. Class-name. [environment-division] [factory] [object] END CLASS Class-name.	IDENTIFICATION DIVISION. FACTORY. [environment-division] [data-division] [procedure-division] [method-definition]... END FACTORY.
Object	**Method**
IDENTIFICATION DIVISION. OBJECT. [environment-division] [data-division] [procedure-division] [method-definition]... END OBJECT.	IDENTIFICATION DIVISION. METHOD-ID. Method-name. [environment-division] [data-division] [procedure-division] END METHOD Method-name.

The IDENTIFICATION DIVISION headers are optional so you will often see these structures starting with the header lines: "CLASS-ID", "FACTORY", "OBJECT" and "METHOD-ID".

The following sections provide more detail about these structures. The equations below give a casual guide to the detail.

```
class = class admin. stuff + factory definition + object definition
class admin. stuff = CLASS-ID paragraph + INHERITS clause + REPOSITORY paragraph
factory definition = factory data + factory methods
object definition = object data + object methods
factory methods }  Usually written as method
object methods  }  prototypes
actual method code = separate source file + METHOD-ID. method-name OF class-name.
```

Once you know how to define classes, factories, objects and methods, you need to understand the ways of invoking methods, how conformance is checked and how binding works. These are explained in this chapter as well as the inheritance processes and the useful PROPERTY feature.

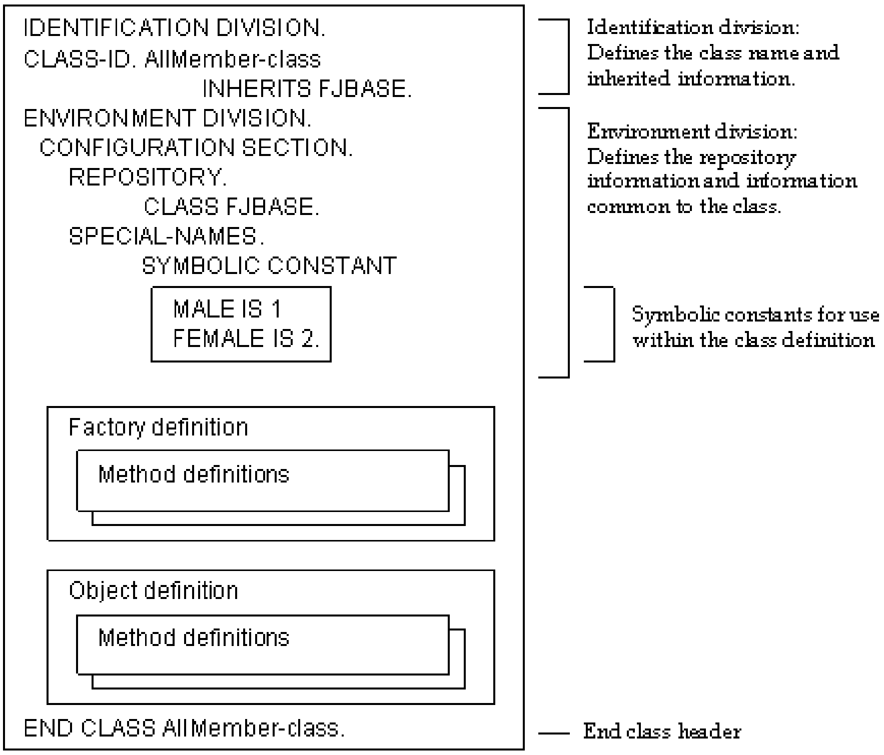
14.1.1 Defining Classes

As described in the previous chapter, a class defines the data and methods for the object instances that are created from that class. In addition to this object definition, OO COBOL also lets you define a factory object to manage the object instances and requires that you specify repository (and other) information in the class's ENVIRONMENT DIVISION.

The class can be thought of as a container for the factory object definition and the object definition.

Figure below shows the general structure of the class source - starting with the CLASS-ID paragraph and ending with the END CLASS terminator.

Figure 14.2 General structure of a Class source file


IDENTIFICATION DIVISION.
CLASS-ID. AllMember-class
INHERITS FJBASE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
CLASS FJBASE.
SPECIAL-NAMES.
SYMBOLIC CONSTANT
MALE IS 1
FEMALE IS 2.
Factory definition
Method definitions
Object definition
Method definitions
END CLASS AllMember-class.
Identification division: Defines the class name and inherited information.
Environment division: Defines the repository information and information common to the class.
Symbolic constants for use within the class definition
End class header

Data declared in the environment division in the class, that is, the class-names declared in the repository paragraph, function-names, mnemonic-names, and the symbolic-constants declared in the special-names paragraph are valid in all source definitions in the class (areas enclosed in bold lines in the diagram above).

14.1.2 Defining Factory Objects

Purpose of the Factory Object

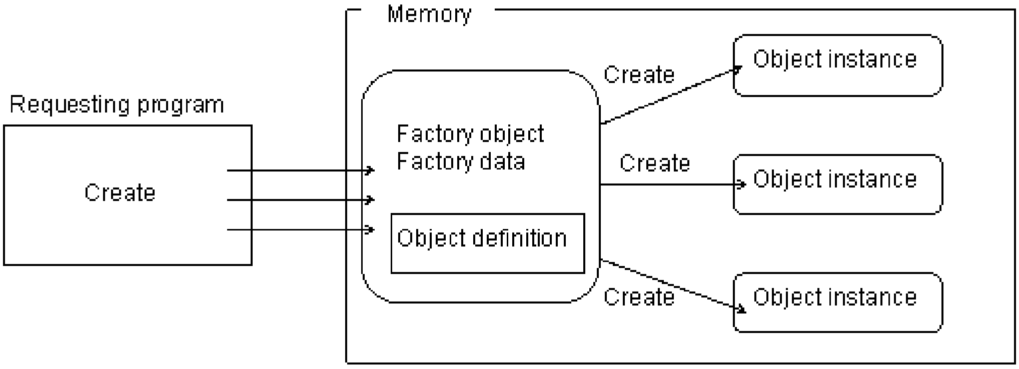
A single factory object is created for each class. It provides:

- Data common to objects - "factory data" (e.g. instance counts, group insurance premiums)
- Methods for handling the factory data
- Methods for object management (e.g. creation, data initialization)

The runtime system creates the factory objects when the application starts and removes them when the application ends. Thus, they are always available to the application.

For example, to create an object instance, the class's "create" method is invoked. The factory object receives this "create" message and creates the object instance.

Figure 14.3 Creating object instances



The single factory object can create multiple object instances (if there are multiple invocations of the "create" method). Thus, the factory object is a "factory" creating object instances.

Usually the object management methods, such as "create", are integrated into the class by inheriting the FJBASE class. This is a standard class provided with the Fujitsu OO COBOL system. See the "14.3.2 The FJBASE Class". You define factory data and methods for application-specific behavior, such as data that is the same for all objects, or special initialization operations.

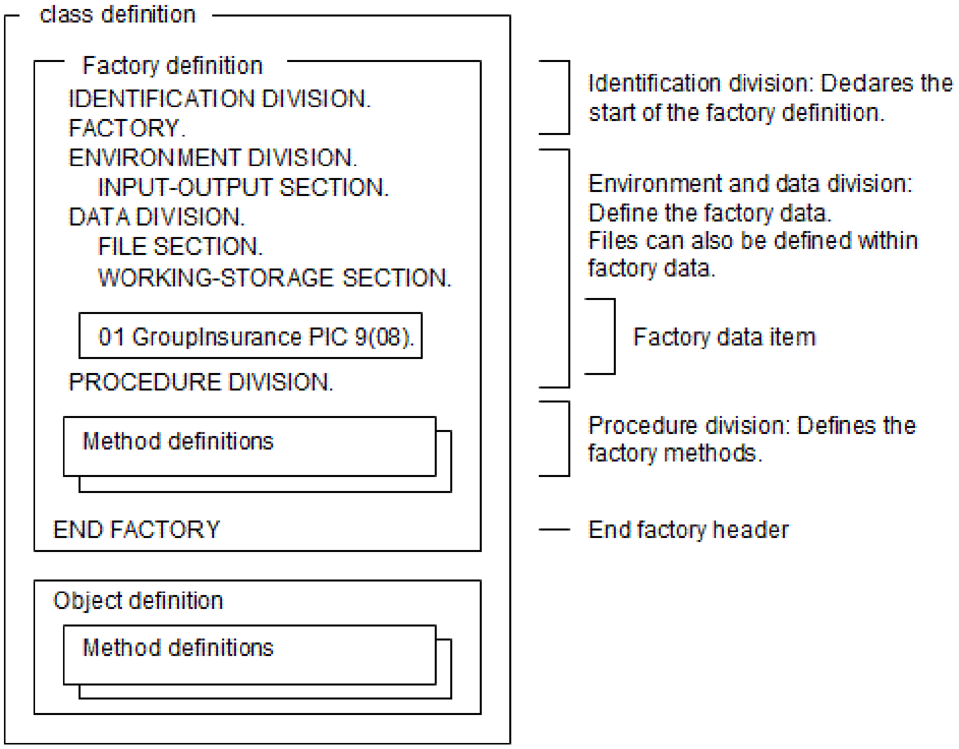
14.1.3 Defining the Factory Object Source

The source definition for the factory consists of:

- Identification division containing the FACTORY paragraph
- Environment and data divisions defining the factory data
- Procedure division containing the factory methods
- END FACTORY terminator.

Classes do not have to define a factory object so all the above items can be omitted from the class definition.

Figure 14.4 Factory source structure



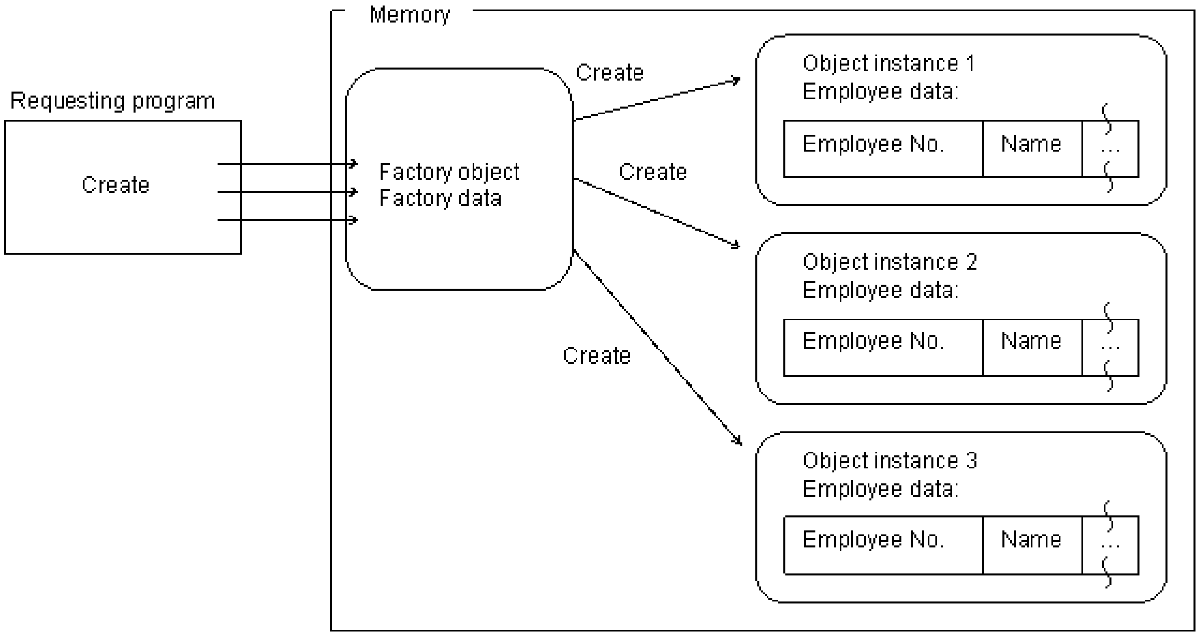
Data defined in the environment division and data division of the factory definition can only be accessed by the factory methods (i.e. the data items cannot be referenced outside the bold-line frame in the diagram above).

Purpose of Objects

Objects are the central elements of an OO system. They provide the essential properties and behaviors of the real world objects and processes that the system supports.

Essentially objects consist of encapsulated (or "hidden") data that can only be accessed and updated through the object (or class) methods. Many object instances can be created, each instance having the same structure of data and accessing the same methods, but containing different data values. For example each instance in the Figure below would be populated with different data.

Figure 14.5 Creating object instances

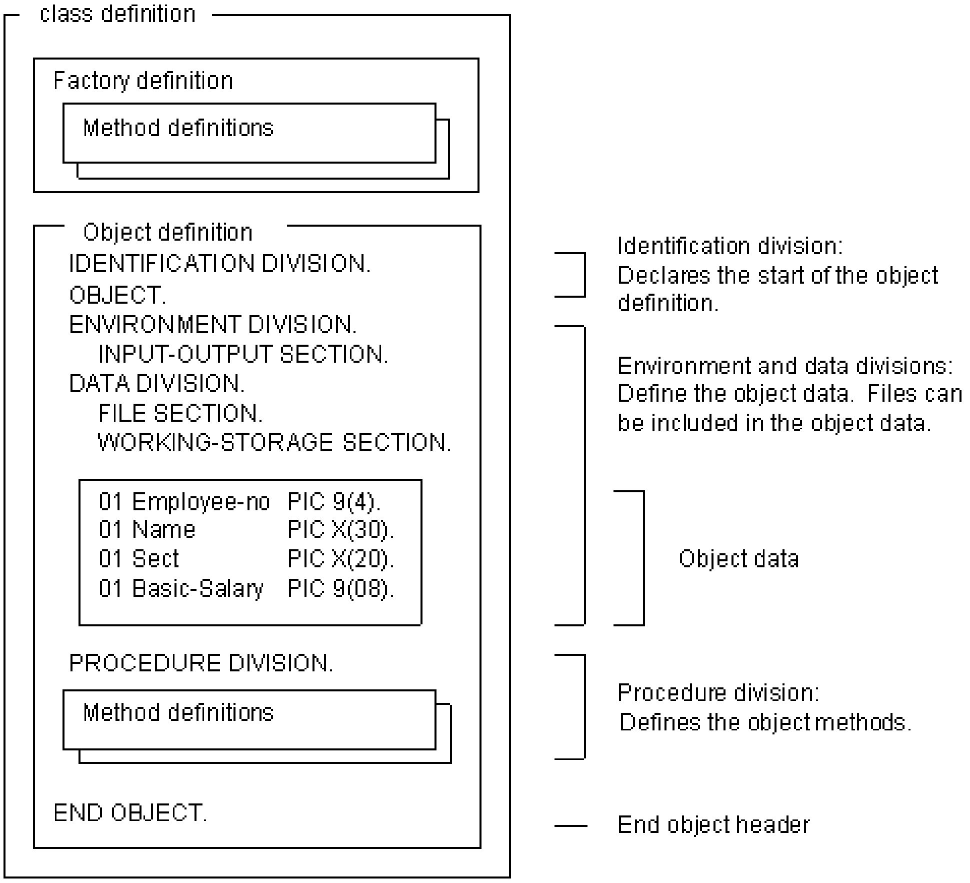


14.1.4 Defining the Object Source

As described in the overview, objects are defined using an identification division with the object header, environment and data divisions for the object data, and a procedure division containing the object methods.

The following diagram shows the source definition for an object.

Figure 14.6 Object source structure



```
IDENTIFICATION DIVISION.
OBJECT.
ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
DATA DIVISION.
   FILE SECTION.
   WORKING-STORAGE SECTION.

   01 Employee-no  PIC 9(4).
   01 Name         PIC X(30).
   01 Sect         PIC X(20).
   01 Basic-Salary PIC 9(08).

PROCEDURE DIVISION.

END OBJECT.
```

Identification division: Declares the start of the object definition.

Environment and data divisions: Define the object data. Files can be included in the object data.

Procedure division: Defines the object methods.

Data defined in the environment and data divisions of the object definition can only be accessed by the object methods (i.e. the data items cannot be referenced outside the bold-line frame in the diagram above).

14.1.5 Defining Methods

Purpose of Methods

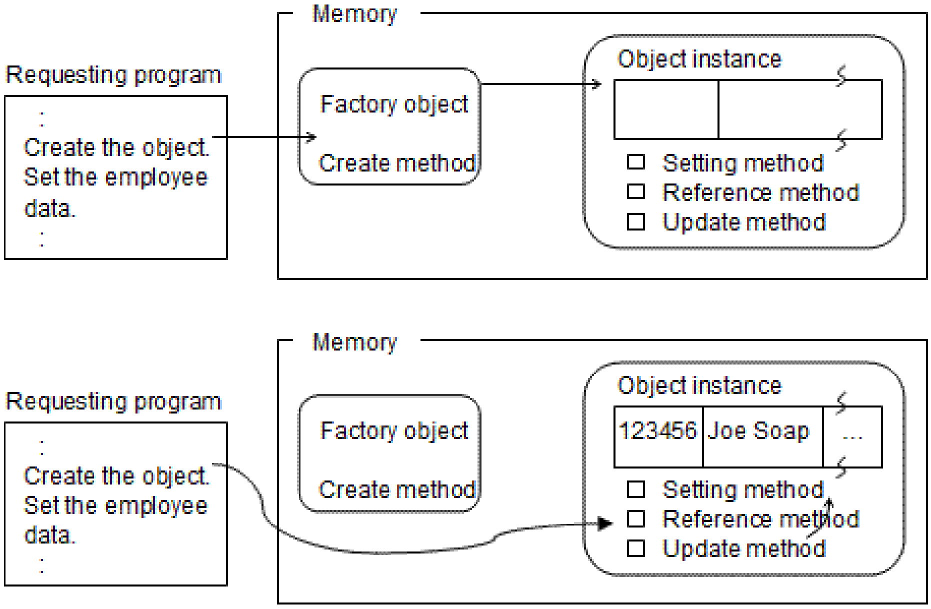
Methods have two roles:

- Perform the functions of the objects and application.
- Provide access to the object's data.

Because the object data are not accessible to routines outside the object's methods, you need to consider which data needs to be available to, and updateable by, other routines. You need to create methods to support that access.

For example, a factory method might be responsible for creating an object instance, while an object method actually receives and sets the object data values:

Figure 14.7 Creating an object and populating the data

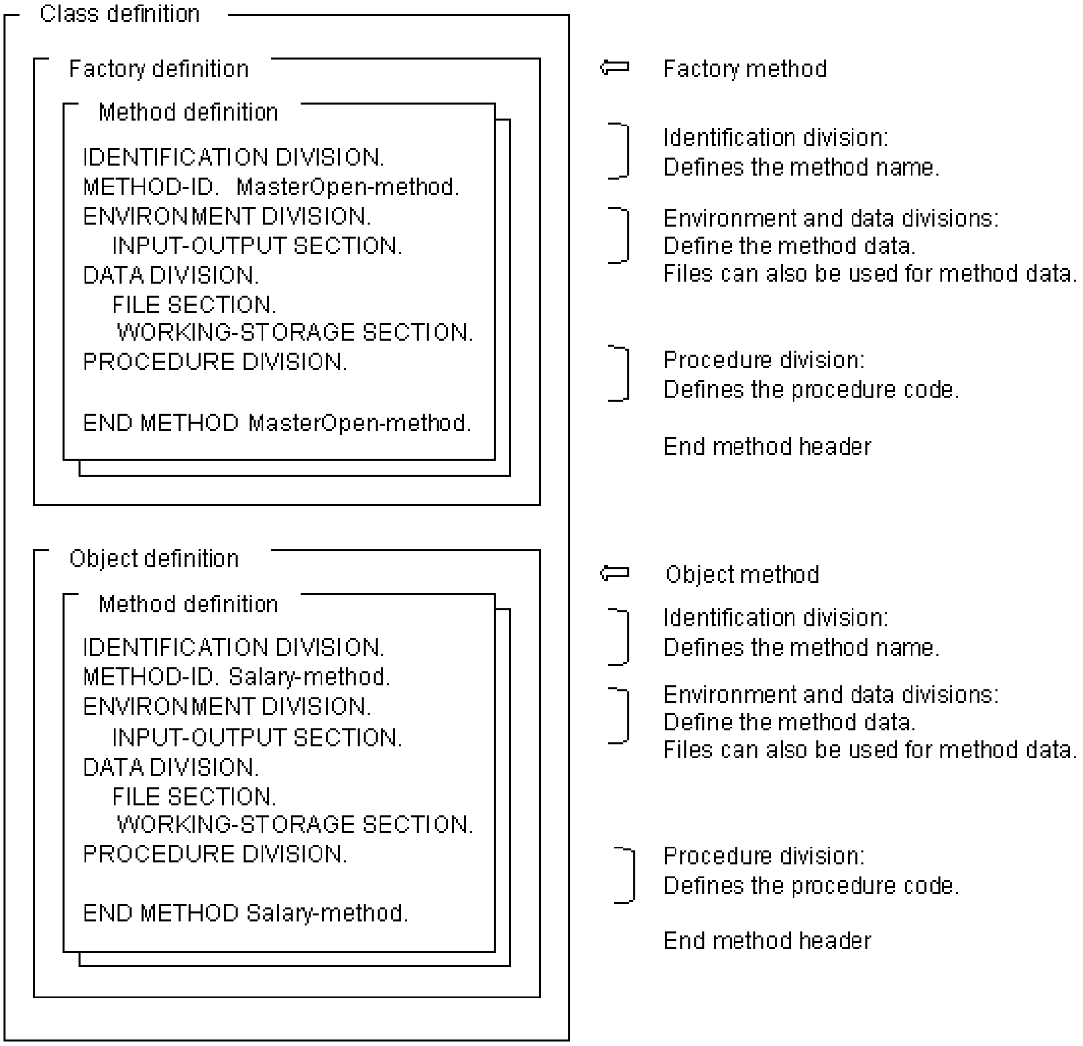


Defining the Method Source

Methods can be factory methods (defined in a factory definition) or object methods (defined in an object definition). Methods are like internal programs and have all four divisions: identification, environment, data and procedure. They have direct access to the object's data and may be activated recursively.

The syntax for factory and object methods is the same:

Figure 14.8 Method definitions



```
IDENTIFICATION DIVISION.
METHOD-ID.  MasterOpen-method.
ENVIRONMENT DIVISION.
    INPUT-OUTPUT SECTION.
DATA DIVISION.
    FILE SECTION.
    WORKING-STORAGE SECTION.
PROCEDURE DIVISION.

END METHOD MasterOpen-method.
```

```
IDENTIFICATION DIVISION.
METHOD-ID. Salary-method.
ENVIRONMENT DIVISION.
    INPUT-OUTPUT SECTION.
DATA DIVISION.
    FILE SECTION.
    WORKING-STORAGE SECTION.
PROCEDURE DIVISION.

END METHOD Salary-method.
```

Data defined within a method can only be accessed by the method code.

See "14.2.5 Invoking Methods" for details of how methods are invoked using the INVOKE statement.

14.1.6 PROTOTYPE Methods

As described in the overview, methods are often defined within the class as prototype methods. The purpose is to allow different people to work on methods for the same class.

A prototype method definition contains the minimum information required to compile the class and perform the appropriate conformance checks. To create a prototype method definition you code:

- The PROTOTYPE clause in the METHOD-ID paragraph.
- Linkage section definitions for any USING or RETURNING data.
- The PROCEDURE DIVISION USING ... RETURNING header.

For example:

Prototype method definition

```
IDENTIFICATION DIVISION.
 CLASS-ID. AllMember-class INHERITS FJBASE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
   REPOSITORY.
       CLASS FJBASE.
 IDENTIFICATION DIVISION.
 OBJECT.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
 PROCEDURE DIVISION.
 IDENTIFICATION DIVISION.                   *>--+
 METHOD-ID.  JobClassGet-method PROTOTYPE.  *>  |
 DATA DIVISION.                             *>  |PROTOTYPE method
  LINKAGE SECTION.                          *>  |Only describes the interface.
  01 L-EMPLOYEE-NUMBER  PIC 9(4).           *>  |
  01 L-JOB-CLASS        PIC 9(1).           *>  |
 PROCEDURE DIVISION USING L-EMPLOYEE-NUMBER *>  |
                   RETURNING L-JOB-CLASS.   *>  |
 END METHOD JobClassGet-method.             *>--+
 END OBJECT.
 END CLASS AllMember-class.
```

Then, in another source file, you create the complete code for the method. In the METHOD-ID paragraph you add the "OF class-name" clause (or "OF FACTORY OF class-name" if the prototype method is a factory method) - this defines the class whose prototype method this method substitutes. Also, add a REPOSITORY paragraph giving the name of the repository containing the class.

Method defined in a separate source file

JobClassGet-method (in a separate source file)

```
IDENTIFICATION DIVISION.
 METHOD-ID.  JobClassGet-method             *> Specify the class name for
             OF AllMember-class.            *> this object method.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
   REPOSITORY.                              *> You also need to specify
       CLASS AllMember-class.               *> the repository for the class
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  LINKAGE SECTION.                          *>--+Define the data for
  01 L-EMPLOYEE-NUMBER  PIC 9(4).           *>  |the method
  01 L-JOB-CLASS        PIC 9(1).           *>--+
 PROCEDURE DIVISION USING L-EMPLOYEE-NUMBER *>--+
                   RETURNING L-JOB-CLASS.   *>  |Create the procedure code
     *> Actual method code.                     |for the method
 END METHOD JobClassGet-method.             *>--+
```

Because the separate method specifies the class repository file, you should compile the class definition before compiling the separate method.

For more details, refer to "14.8 Using More Advanced Functions" and "Chapter 15 Developing OO COBOL Applications".

14.2 Working with Objects

In the previous sections in this chapter, you have learned how to define your OO COBOL system - the classes, objects, and methods. Now you need to learn how to work with the objects you create.

This section tells you how to create the object instances, how to reference these instances, how to invoke methods, and what determines the lifetime of the instances. Because you can't create an object instance without creating a reference to that object we will look at these two topics together.

14.2.1 Creating and Referring to Object Instances

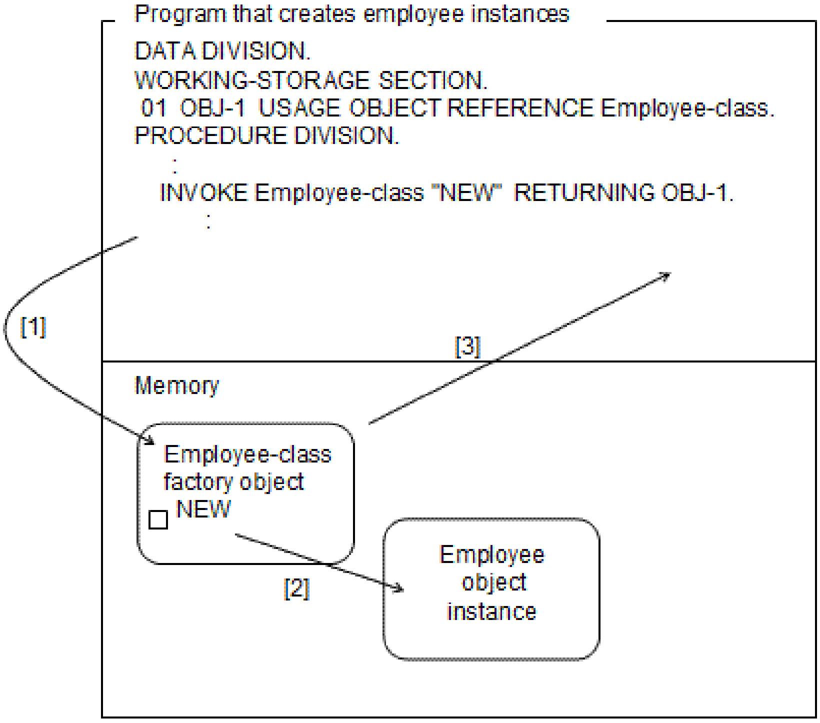
When you create an object instance, the OO COBOL system creates a handle for that instance - something that can be used to refer to and identify that particular instance. The handle is known as an object reference.

You store object references in data items specified with the USAGE OBJECT REFERENCE clause.

To create an object instance you use the factory method "New" and supply an object reference data item. The New method creates the new object instance and returns the object reference in the data item.

For example, consider the code to create a new instance of the Employee-class:

Figure 14.9 Creating a new instance of the employee class



```
DATA DIVISION.
WORKING-STORAGE SECTION.
 01  OBJ-1  USAGE OBJECT REFERENCE Employee-class.
PROCEDURE DIVISION.
    :
  INVOKE Employee-class "NEW"  RETURNING OBJ-1.
       :
```

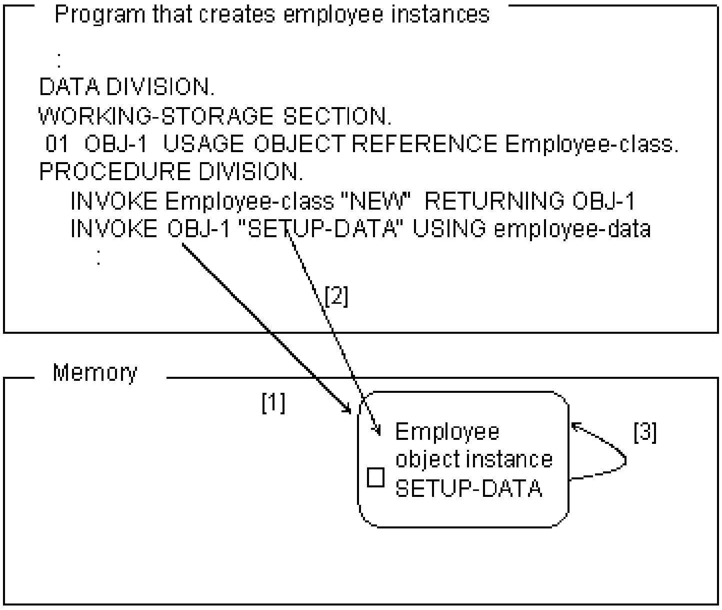
Program that creates employee instances
[1]
[3]
Memory
Employee-class factory object
□ NEW
[2]
Employee object instance

The creation process is:

- The creating program invokes the class's NEW method, which is implemented by the factory object for that class.
- The NEW method creates a new instance of the object defined in that class.
- The NEW method returns the object reference in the data item provided in the INVOKE statement.

Whenever you want to refer to the object instance, you use the object reference (OBJ-1 in the example above). So, for this example, to invoke the method to setup the employee data, you use the object reference to identify which object's method is the target of the INVOKE statement:

Figure 14.10 Populating the data of an object instance



```
  :
DATA DIVISION.
WORKING-STORAGE SECTION.
 01  OBJ-1  USAGE OBJECT REFERENCE Employee-class.
PROCEDURE DIVISION.
    INVOKE Employee-class "NEW"  RETURNING OBJ-1
    INVOKE OBJ-1 "SETUP-DATA" USING employee-data
      :
```

In the above diagram:

- The object reference data item, OBJ-1 identifies the object instance that should receive the invoke "message".
- The method name is specified as a literal.
- The SETUP-DATA method populates the object instance with the data supplied in employee-data.

14.2.2 Transferring Object References

To transfer an object reference from one data item to another you use the SET statement. For example, we could introduce another object reference OBJ-X to the above example - the code below has the same effect:

```
*>   :
 01  OBJ-1       USAGE OBJECT REFERENCE Employee-class.
 01  OBJ-X       USAGE OBJECT REFERENCE Employee-class.
*>   :
 PROCEDURE DIVISION.
*>   :
     INVOKE Employee-class "NEW" RETURNING OBJ-1
     SET OBJ-X TO OBJ-1
     INVOKE OBJ-X "SETUP-DATA" USING employee-data
*>   :
```

Note

- Object references are initialized to NULL. They cannot be initialized using the VALUE clause.
- Do not attempt to alter the contents of object reference data items by any means other than the SET statement. The format of the object reference is defined by the OO COBOL system and incorrect values are likely to cause unpredictable behavior.
- When you specify the object reference data item in the parameter of CALL statement, then the sending and receiving items must be a same USAGE clause.

14.2.3 Object Reference Types

The USAGE OBJECT REFERENCE clause has five formats, defining five types of object reference. The formats are:

1. USAGE OBJECT REFERENCE
2. USAGE OBJECT REFERENCE class-name
3. USAGE OBJECT REFERENCE class-name ONLY
4. USAGE OBJECT REFERENCE FACTORY OF class-name.
5. USAGE OBJECT REFERENCE FACTORY OF class-name ONLY.

The five formats determine the types of object references that can be stored in the object reference data items. The type of object reference influences the conformance checking that is performed at compilation and execution time - see "14.4 Conformance Checking" for full details.

Briefly, the types defined by the above formats are as follows:

- Can store an object reference of any class.
- Can store object references of the specified class or subclasses of that class.
- Can only store object references of the specified class.
- Can store object references to the factory object of the specified class or its subclasses.
- Can only store object references to the factory object of the specified class.

14.2.4 Predefined Object Identifiers

OO COBOL provides you with some pre-defined object identifiers - which you can use in place of object references. The predefined object identifiers are:

NULL

An object reference set to NULL is not pointing at any object.

SUPER

Is used when invoking methods. It indicates that the method should be taken from one of the superclasses, not from the class containing the invocation.

SELF

Is used when invoking methods. It indicates that the method should be taken from the class containing the invocation.

EXCEPTION-OBJECT

Is used in declarative procedures. It references the current exception object.

See the sections "14.6.3 Binding with the Predefined Object Identifier SUPER" and "14.6.4 Binding with the Predefined Object Identifier SELF" for examples on the use of these predefined object identifiers.

14.2.5 Invoking Methods

Once an object instance is created, you communicate with that object by invoking its methods. To do this you use the INVOKE statement. In the INVOKE statement you define:

- The object to be invoked
- The method to be invoked
- The parameters to be passed to and returned from the method

The INVOKE statement has the following format:

```
INVOKE {object-identifier} method-name
       {class-name       }
```

```
[USING  parameter-list]
[RETURNING return-parameter]
```

See the "COBOL Language Reference" for a detailed description of the syntax.

The various parts of this statement are explained in the following paragraphs as well as a technique for invoking methods from within other statements - known as "in-line invocation".

14.2.5.1 Invoking Factory Object Methods

You may recall that each class has one and only one factory object. The OO COBOL system creates the factory objects when the application starts, and deletes them when the application finishes.

You indicate that you want to invoke a factory method by providing the class name in the INVOKE statement. The OO COBOL system translates that into the object reference for the factory object. For example, to invoke the factory method "NEW" in the Employee-class you code:

```
INVOKE Employee-class "NEW" RETURNING OBJ-1
```

14.2.5.2 Invoking Object Methods

To invoke an object method you provide an object identifier - this can be an object reference or one of the pre-defined object identifiers (other than NULL). For example, to invoke the object method "Setup-Data" you can code:

```
INVOKE employee-obj-1 "Setup-Data" USING employee-data
```

Where employee-obj-1 contains the object reference for the object whose Setup-Data method you want to invoke.

See "14.6.3 Binding with the Predefined Object Identifier SUPER" and "14.6.4 Binding with the Predefined Object Identifier SELF" for examples on invoking with these predefined object identifiers.

14.2.5.3 Using Identifiers for the Method Name

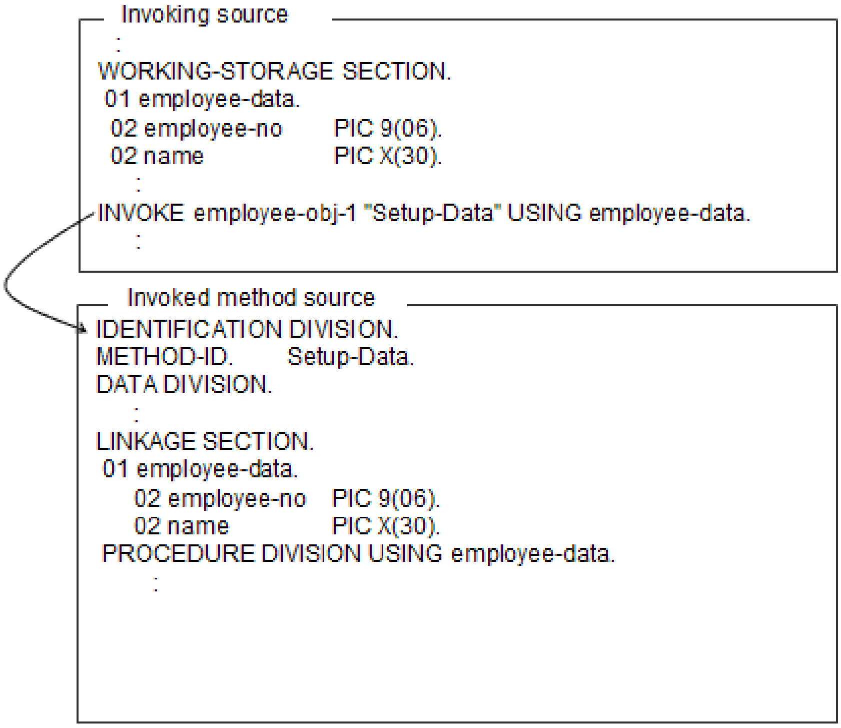
If you use an identifier for the method name in the INVOKE statement, note that the length of the method name string is limited to 30 bytes.

14.2.5.4 USING Parameters

Parameters are passed to and from methods in identical ways to the CALL statement. You define the parameters in the linkage section and procedure division header of the method and pass a matching list of parameters in the INVOKE statement.

For example, consider the Setup-Data method that takes a single parameter "employee-data":

Figure 14.11 Passing a parameter to a method



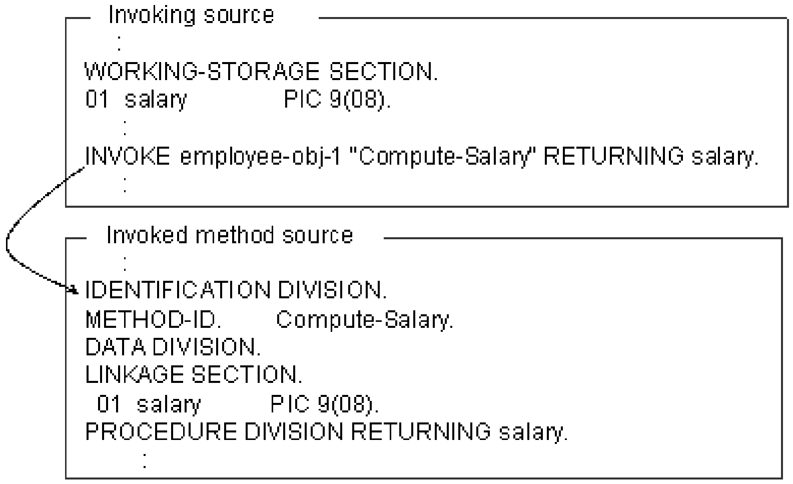
```
 :
WORKING-STORAGE SECTION.
 01 employee-data.
  02 employee-no        PIC 9(06).
  02 name               PIC X(30).
    :
INVOKE employee-obj-1 "Setup-Data" USING employee-data.
    :
```

```
IDENTIFICATION DIVISION.
METHOD-ID.       Setup-Data.
DATA DIVISION.
    :
LINKAGE SECTION.
 01 employee-data.
     02 employee-no    PIC 9(06).
     02 name           PIC X(30).
 PROCEDURE DIVISION USING employee-data.
       :
```

14.2.5.5 RETURNING parameter

The RETURNING clause specifies a single item that can be used to receive a value from the method. For example, consider the Compute-Salary method that returns the resulting salary in the RETURNING item:

Figure 14.12 Using a RETURNING item

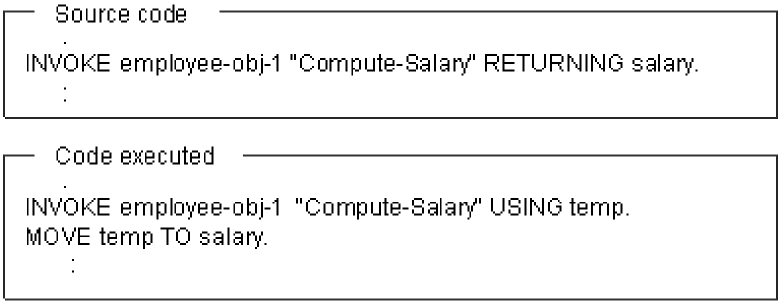


```
   :
WORKING-STORAGE SECTION.
01  salary            PIC 9(08).
    :
INVOKE employee-obj-1 "Compute-Salary" RETURNING salary.
    :
```

```
    :
IDENTIFICATION DIVISION.
METHOD-ID.       Compute-Salary.
DATA DIVISION.
LINKAGE SECTION.
 01  salary         PIC 9(08).
PROCEDURE DIVISION RETURNING salary.
       :
```

The RETURNING item cannot be accessed by the invoked method - its value is only passed back from the invoked method to the invoking code. A helpful way of understanding this, is to assume that the RETURNING clause generates the following code:

Figure 14.13 Equivalent code to INVOKE ... RETURNING


Source code

```
    .
INVOKE employee-obj-1 "Compute-Salary" RETURNING salary.
    :
```

Code executed

```
    .
INVOKE employee-obj-1  "Compute-Salary" USING temp.
MOVE temp TO salary.
     :
```

You can use both USING parameters and RETURNING parameter in the same INVOKE statement.

14.2.5.6 In-line Invocation

You can invoke methods that return a value in the RETURNING clause, using a technique called "in-line invocation". To do this you place the in-line invocation string in place of a data item in a COBOL statement. Any data item that is not a target item can be replaced as long as the RETURNING item is the same type as the item that is replaced. The method is invoked then the statement is executed using the value returned.

The in-line invocation string has the following format:

```
object-identifier :: method-name (parameter-list)
```

For example, suppose you want to invoke the Compute-Salary method and move the returned value to a print report. Using the INVOKE statement, you would code:

```
INVOKE employee-obj-1 "Compute-Salary" RETURNING salary
MOVE salary TO print-salary
```

Using in-line invocation, you would code:

```
MOVE employee-obj-1 :: "Compute-Salary" TO salary
```

In-line invocation is discussed in more detail in "14.8 Using More Advanced Functions".

14.2.6 Object Life

In "Creating and Referring to Object Instances" you saw how to create objects. This section explains when objects cease to be - when they are deleted or removed from memory.

Factory Object Life

Factory objects exist for the lifetime of the application. They cannot be deleted while the application is running.

Object Instance Life

Object instances are deleted when:

- The application ends.

  NOTE: In this case, _FINALIZE method is not called.

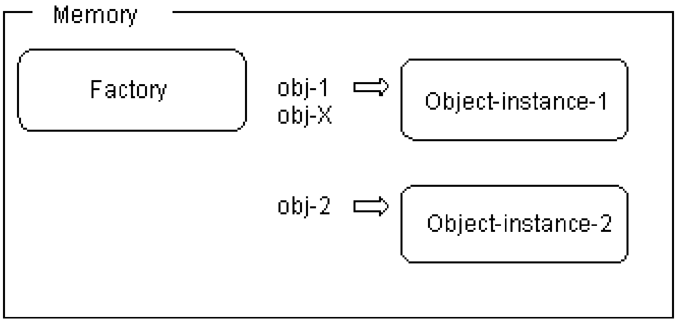
- There are no object reference data items containing references to the object.

For example, consider the following code where one object has a single object reference, and another object reference is contained in two object reference data items:

Example showing object instance life

[invoking source]

```
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  OBJ-1  USAGE OBJECT REFERENCE Employee-class.
  01  OBJ-2  USAGE OBJECT REFERENCE Employee-class.
  01  OBJ-X  USAGE OBJECT REFERENCE Employee-class.
 PROCEDURE DIVISION.
     INVOKE Employee-class "NEW" RETURNING OBJ-1.
     SET OBJ-X TO OBJ-1.
     INVOKE Employee-class "NEW" RETURNING OBJ-2.
```



We now remove the objects by setting the object reference data items to NULL:

```
     SET OBJ-2 TO NULL.         *>[1]
     SET OBJ-1 TO NULL.         *>[2]
     SET OBJ-X TO NULL.         *>[3]
```

The above statements have the following effects:

Object-instance-2, referenced only by the object reference data item obj-2, is deleted from memory.

Although obj-1 is set to NULL, object-instance-1 is not deleted because obj-X still references it.

Object-instance-1 is deleted from memory because obj-X, set to NULL, no longer references it.

Note that although the objects in the above code are logically deleted, in that they can no longer be referenced, the COBOL runtime system may not free up memory immediately. The runtime system uses other criteria to determine when to free up memory (often referred to as garbage collection).

14.3 Defining Inheritance

Purpose of Inheritance

As explained in "Chapter 13 Introduction to Object-Oriented Programming", inheritance is one of the key features in OO programming. It provides the following benefits:

- Existing components can be reused easily
- Changes can be accommodated in a flexible manner
- Standard behaviors can be inherited from system classes.

14.3.1 Inheritance Concept and Implementation

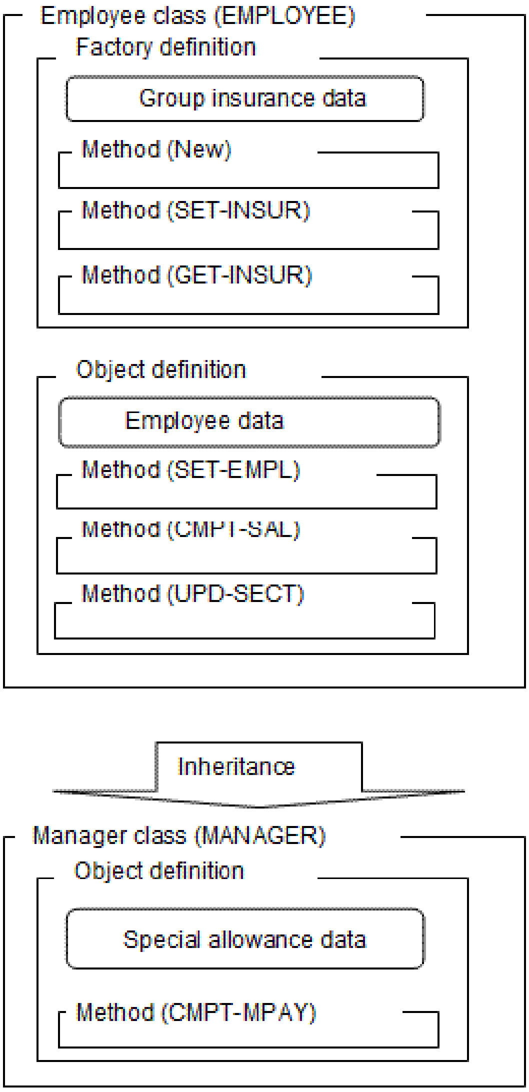
This section briefly provides a brief review of inheritance, which is outlined in "Chapter 13 Introduction to Object-Oriented Programming".

Inheritance concept

In conventional programming techniques, existing source programs were copied, and programming operation was performed, on the basis of the copied programs. This was true when a component containing a function similar to that of an existing component (routine) was created.

In the object-oriented programming technique, similar programming can be implemented only by coding differences with existing components (classes). That is, all functions contained in a class can be easily inherited; this is referred to as inheritance.

The following is an example of a method to create a manager class that inherited an employee class.



[Group insurance data]

Data of group premium to be reduced from salary

[NEW method]

Creates object instances.

[SET-INSUR method]

Sets group insurance data.

[GET-INSUR method]

References (retrieves) group insurance data.

[Employee data]

Data such as employee numbers, names, addresses, and salaries

[SET-EMPL method]

Sets employee data in created object instances.

[CMPT-SAL method]

Computes employee salaries.

[UPD-SECT method]

Changes employee sections.

Describe only differences with the employee class (parent class).

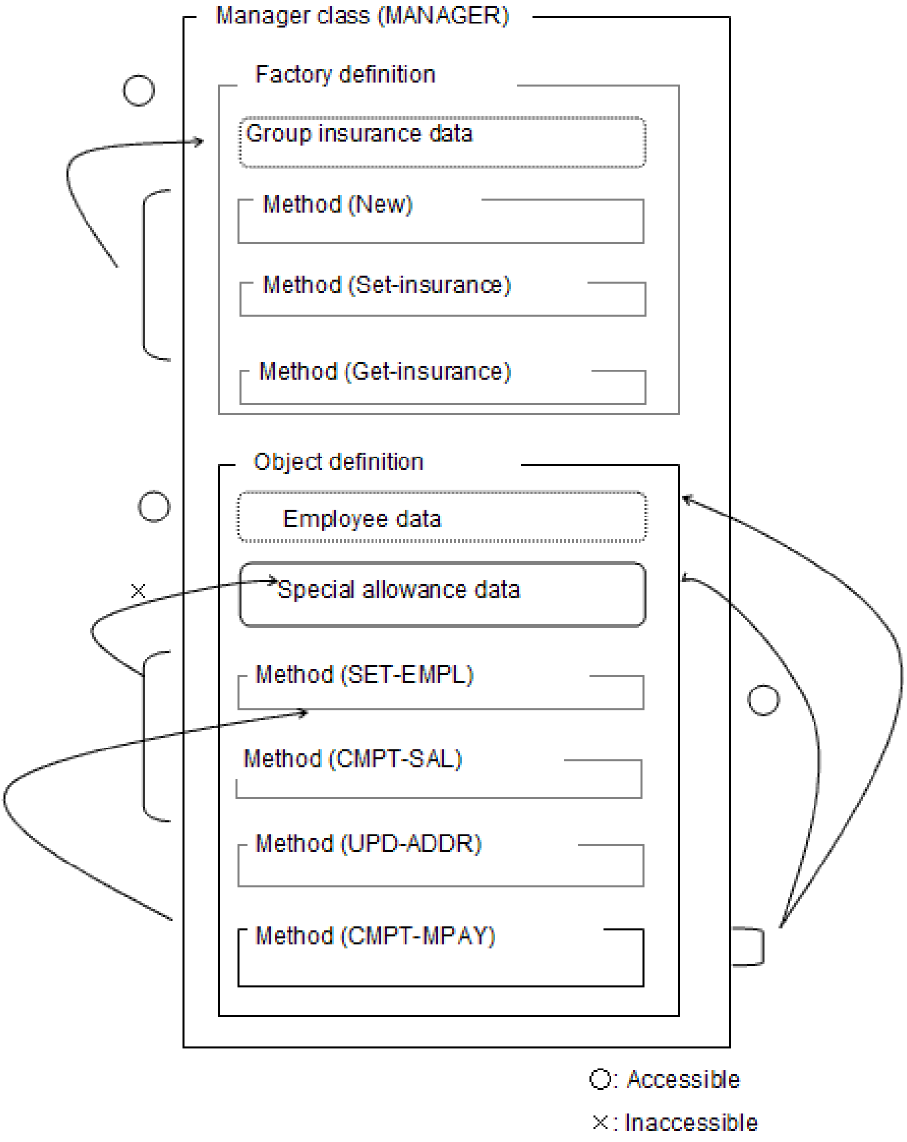
[Special allowance data]

Data of special allowance for managers

[CMPT-MPAY method]

Computes a special allowance.

Shown below is the logical structure of the management class.



Areas enclosed in solid lines in the diagram above indicate data and methods defined explicitly, and those enclosed in solid lines indicate data and methods defined implicitly by inheritance.

The explicit and implicit definitions do not need to be distinguished to invoke each method. Implicitly defined methods can be invoked in the same manner as explicitly defined methods.

Use the inheritance feature freely because no restrictions are placed on the inheritance hierarchy (depth). We recommend designing a hierarchical structure with a moderate depth, because too deep a hierarchy makes it time-consuming to manage resources in that type of structure.

When there are two classes in an inheritance relationship, a class (inherited class) derived from another is referred to as a child class, and the class to be inherited is referred to as a parent class.

In the above chart, the employee class (EMPLOYEE) is a parent, and the manager class (MANAGER) is a child.

Accessing data

The following explains how to access data defined implicitly (group insurance and employee data) and how to access data defined explicitly (special allowance data).

Factory and object data can be accessed only by using methods defined explicitly in its class definition. In the diagram given above, the object data (object instance) includes both employee and special allowance data; however, the special allowance data defined explicitly can be accessed only by using an object method (CMPT-MPAY) defined explicitly. On the other hand, the employee data defined implicitly can be accessed only by using object methods (SET-EMPL, CMPT-SAL, and UPD-SECT) defined implicitly.

Defining Inheritance in the COBOL Source

Your inheritance model - in which classes are parents or children of other classes - is defined by your OO design process. To implement the inheritance model you create the parent classes first, and code INHERITS clauses in the CLASS-ID paragraph of the child classes.

For example, let us look at the code for a manager class that inherits from a general employee class:

```
*>----------------------------
*> CLASS DEFINITION
*>----------------------------
 IDENTIFICATION DIVISION.
 CLASS-ID. Manager-class
           INHERITS Employee-class.  *> Specify the parent class.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
   REPOSITORY.
       CLASS Employee-class.         *> Specify the repository.
*>----------------------------
*> OBJECT DEFINITION
*>----------------------------
 IDENTIFICATION DIVISION.
 OBJECT.
 DATA DIVISION.                      *> Object data definition:
 WORKING-STORAGE SECTION.            *> Only code the data to be added.
 01 W-INCENT-PERCENT PIC 9(02).      *> (Differential coding)
 PROCEDURE DIVISION.
*>----------------------------
*> METHOD DEFINITION
*>----------------------------
 IDENTIFICATION DIVISION.            *> Object method definition:
 METHOD-ID.  Compute-Bonus.          *> Only code the methods to  be added
 *> :                                *> or changed. (Differential coding)
 END METHOD Compute-Bonus.
 END OBJECT.
 END CLASS Manager-class.
```

Notice that, in addition to the INHERITS clause, you need to specify the repository for the parent class in the REPOSITORY paragraph. See "14.5 Using Repositories" more details.

Now, let's define inheritance.

Inheritance can be implemented by specifying a parent class name in the INHERITS clause in the class-name paragraph (CLASS-ID). At this point, declare a parent class in the repository paragraph in the environment division.

Method and Data Accessibility

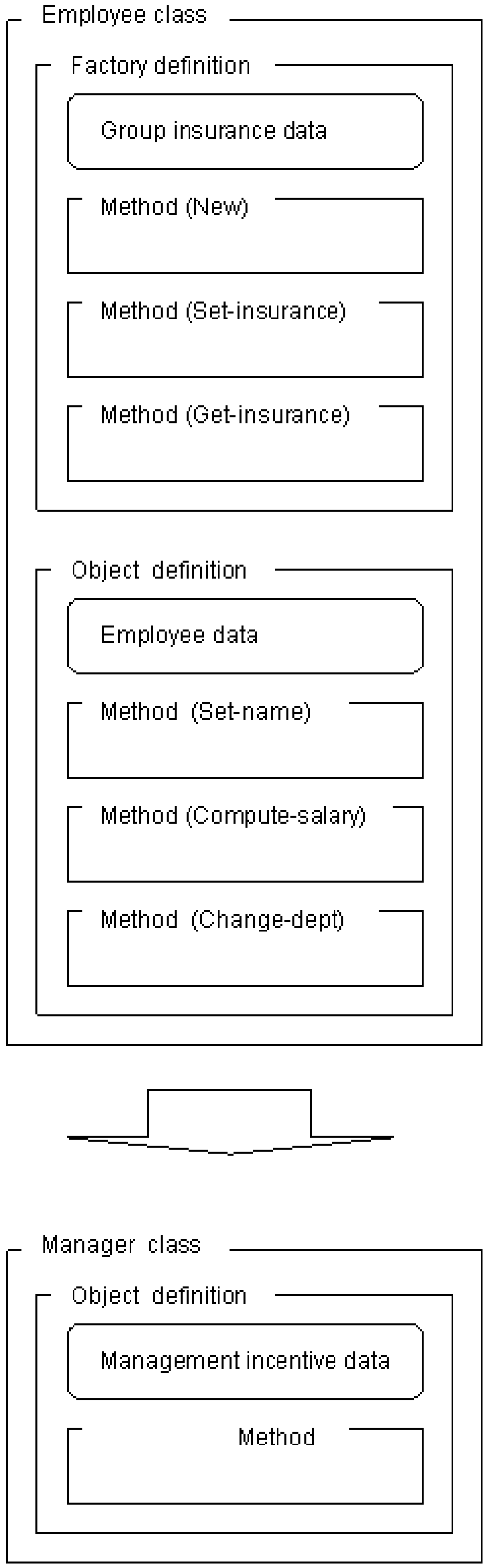
A child class is made up of the data and methods of its parent class(es). The methods in the child class can invoke the methods defined in both the parent class(es) and the child class. The methods of the parent class cannot invoke the methods in the child class, unless invoking a method defined in the parent class but overridden in the child class.

Data, however, can only be accessed by the methods of the class containing the data. Child methods cannot access the data in the parent class and parent methods cannot access the data in the child class. This is a consequence of encapsulating the data within the class.

If a child method requires values of parent data, it must use the parent's methods to access that data.

Consider the example of the manager class inheriting from the employee class. First, here is a schematic view of the code in each class definition:

Figure 14.14 Inheritance adding data and methods



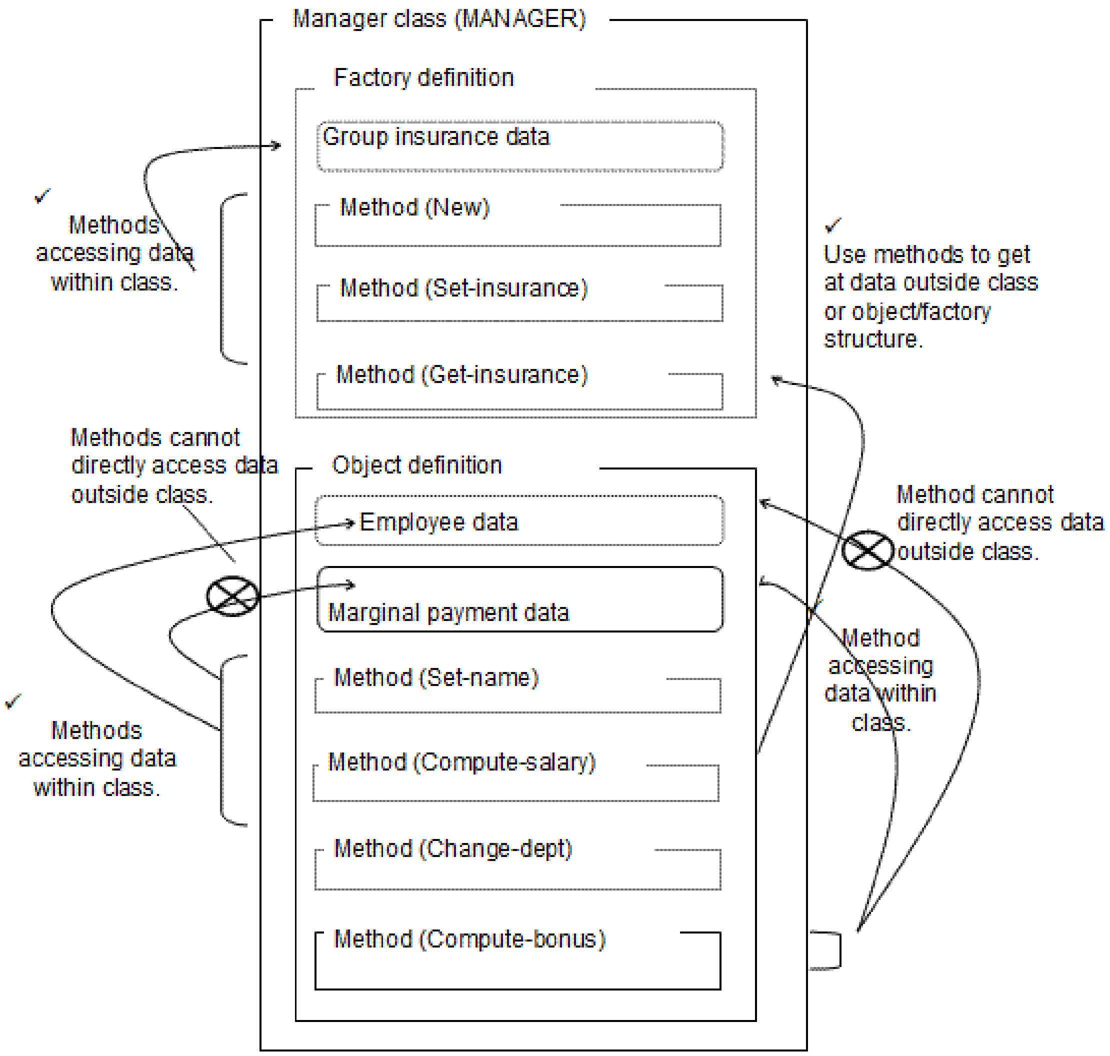
This example shows the factory containing a New method to create the employee objects and data and methods for handling a group insurance scheme – common to all employees.

The object contains employee data such as name, address, salary and methods for accessing the data and calculating this month's pay.

The Manager class adds data and methods to handle a management incentive bonus.

Now consider how the inherited pieces merge with the pieces added in the manager class. The schematic view below shows what is accessible to the various methods. The dotted lines indicate inherited data and methods, the solid lines show data and methods coded in the manager class. The arrows represent attempts to access the data.

Figure 14.15 Data accessibility

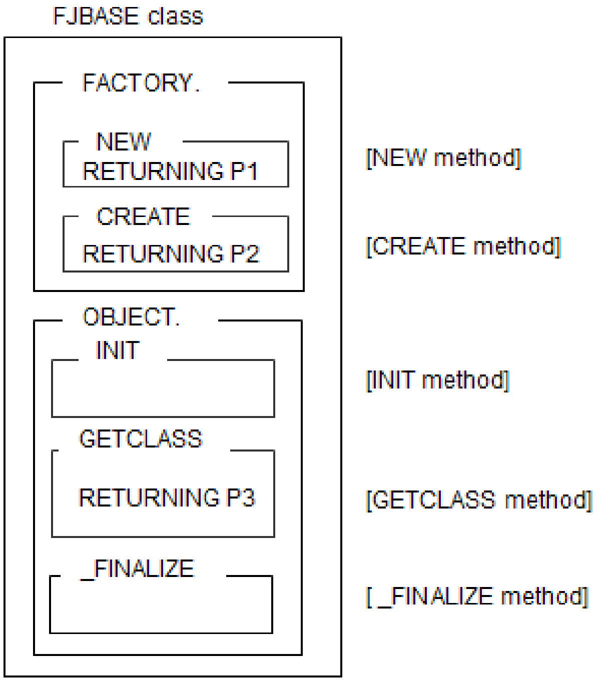

Manager class (MANAGER)
Factory definition
Group insurance data
Method (New)
Method (Set-insurance)
Method (Get-insurance)
Object definition
Employee data
Marginal payment data
Method (Set-name)
Method (Compute-salary)
Method (Change-dept)
Method (Compute-bonus)
✓ Methods accessing data within class.
Methods cannot directly access data outside class.
✓ Methods accessing data within class.
✓ Use methods to get at data outside class or object/factory structure.
Method cannot directly access data outside class.
Method accessing data within class.

14.3.2 The FJBASE Class

The COBOL system provides general-purpose classes to support common functions. A critical class is the FJBASE class that defines the basic methods required by all classes, such as object instance creation. Any class with no (application) parent classes should inherit from FJBASE.

An overview of the FJBASE class is shown below:

Figure 14.16 The FJBASE class



For details of the FJBASE class methods, refer to the "COBOL Language Reference".

See "14.8.6 Initialization and Termination Methods" the INIT method and the _FINALIZE method

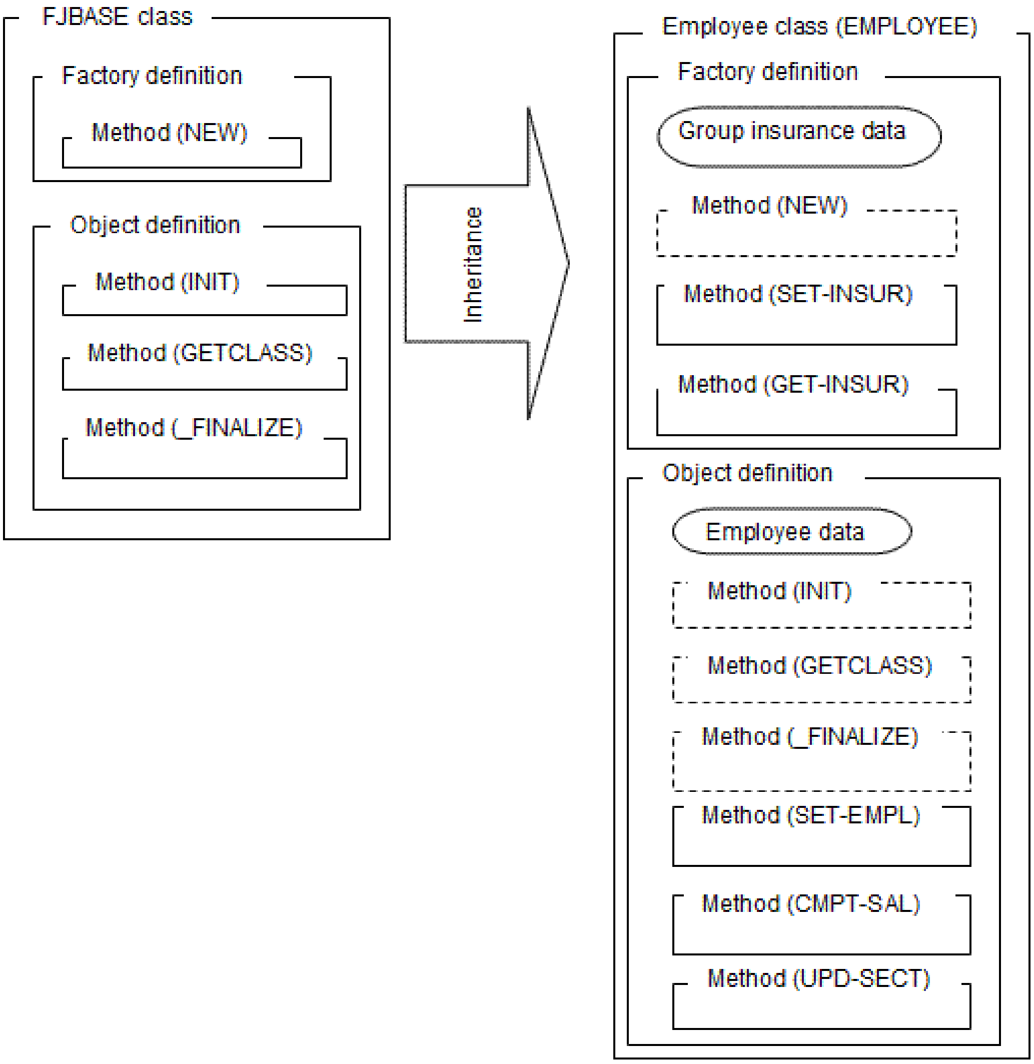
The NEW and GETCLASS methods are the ones most frequently used. Other methods tend to be used in more advanced programming. However, the other methods, such as the CREATE method, the INIT method, the GETCLASS method and the _FINALIZE method, are embedded, even if only the NEW method is required. This is because inheritance is implemented to the class definition. You do not have to be concerned about those methods, because they have no effect unless they are called.

FJBASE has no object data so there is no overhead in inheriting the class with methods that you might not use.

When you define a new class structure, be sure that your root class (the one with no parents) inherits the FJBASE class.

Up to this point, the examples were shown as if the NEW method was defined in the employee class (EMPLOYEE). Actually, however, the NEW method was the one defined implicitly by inheriting the FJBASE class; that is, the inheritance relationship given below was established.

Figure 14.17 FJBASE Inheritance Relationship



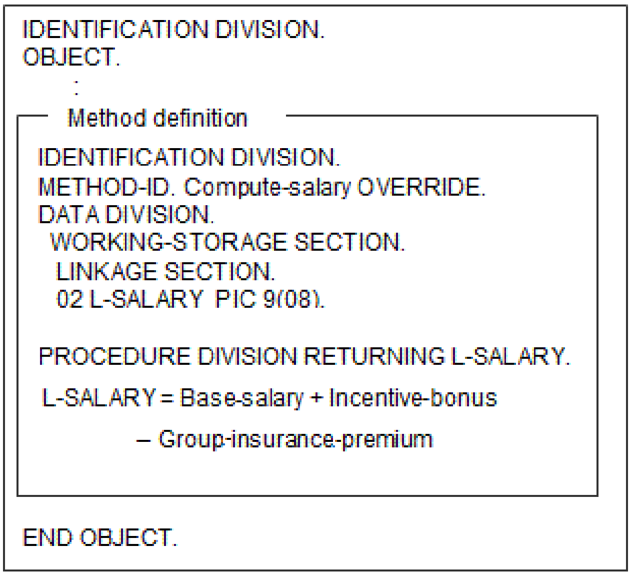
14.3.3 Overriding Methods

In some cases, you may want a child class to have a method that has a slightly different algorithm to an equivalent method in the parent class. Instead of writing a method with a different name or creating a new class, you can write a method with the same name and specify that it overrides the method from the parent class. When an object in the child class invokes the method, control is passed to the overriding method.

You indicate that a child method overrides a parent method by adding the OVERRIDE clause to the METHOD-ID paragraph.

For example, suppose the Compute-salary method in the manager class (of the sample used above) needs to include the manager's incentive bonus. A slightly amended Compute-salary method could be added to the manager class as shown in Figure below:

Figure 14.18 Example of an overriding method



```
IDENTIFICATION DIVISION.
OBJECT.
       :
  Method definition
  IDENTIFICATION DIVISION.
  METHOD-ID. Compute-salary OVERRIDE.
  DATA DIVISION.
   WORKING-STORAGE SECTION.
   LINKAGE SECTION.
   02 L-SALARY  PIC 9(08).

  PROCEDURE DIVISION RETURNING L-SALARY.
  L-SALARY = Base-salary + Incentive-bonus
               – Group-insurance-premium

END OBJECT.
```

(In the above code, the Group insurance premium would likely be obtained by invoking a factory method that returns that premium.)

Notes on Overriding Methods

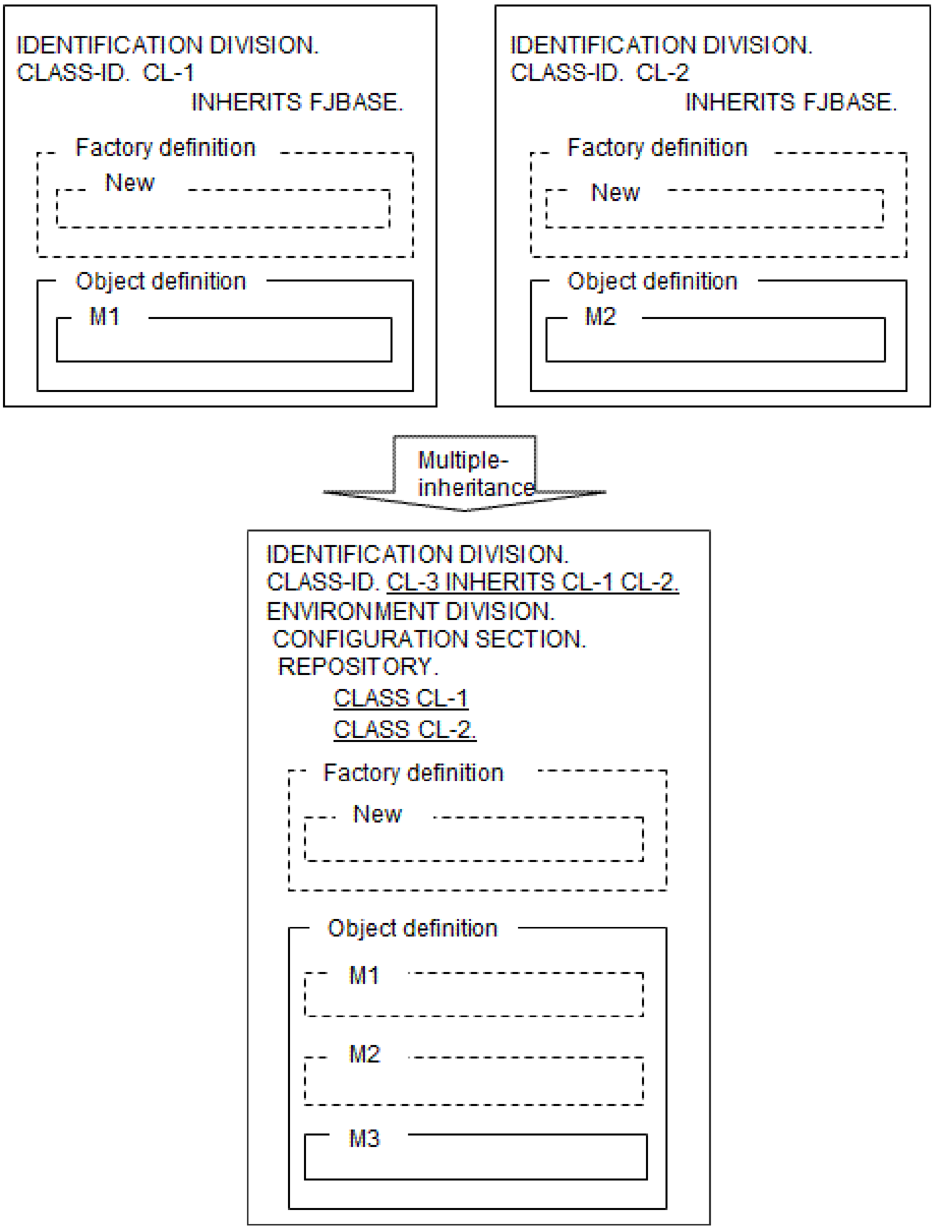
1. The interface for an overriding method must be the same as that for the method being overridden.
2. You can override methods explicitly defined in the parent class or that are implicitly defined in the parent class by inheritance from other classes.

14.3.4 Multiple Inheritance

OO COBOL allows you to code more than one class in the INHERITS clause - so that the child class has more than one parent. This feature is called "multiple inheritance".

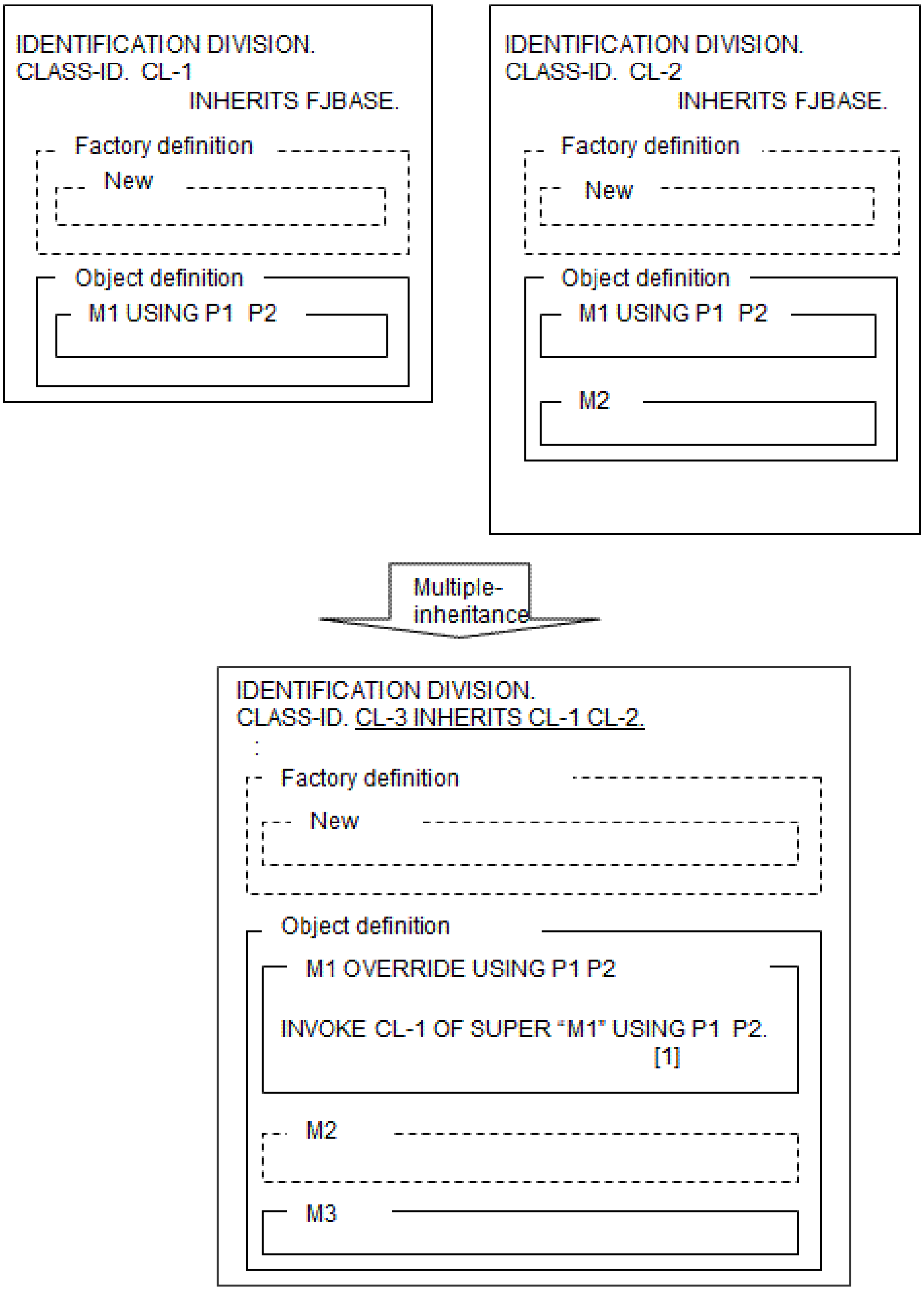
Figure below shows a simple multiple inheritance. Notice that the system can identify that the New methods both come from the same source class (FJBASE) so there is no conflict in the inheriting class.

Figure 14.19 Multiple Inheritance



When two super classes being inherited have the same method name, OO COBOL states that the two methods must have the same interfaces, and an overriding method must be defined. The overriding method's interface must be the same as the two inherited methods - which may not be possible, in which case the inheritance is invalid. (See Figure below).

Figure 14.20 Common inherited methods in multiple inheritance.



In this case, when one of multiple methods defined is being invoked by using the predefined object identifier SUPER, the method must be identified by explicitly specifying the parent class name as shown in [1] above.

14.3.4.1 Caution

Multiple inheritance can increase system complexity and introduce behaviors that are difficult to predict. Because it is possible to achieve the same results without using multiple inheritance, many products and system designers choose not to use this feature. Before committing to a multiple inheritance class structure you should consult literature that discusses the pros and cons of multiple inheritance.

Multiple inheritance is discussed in more detail in "14.8 Using More Advanced Functions".

14.4 Conformance Checking

The concept of conformance is explained briefly in the "13.4.12 Conformance".

Most OO programming systems do conformance checking. The purpose being to detect errors as early as possible - either at compile or execution time before serious damage results. For the COBOL programmer these checks concern two issues:

1. Are object references being handled in a way that ensures that an unsupported, or incompatible, method invocation cannot occur?
2. Do the parameters coded in a method invocation match the parameters the method expects to receive?

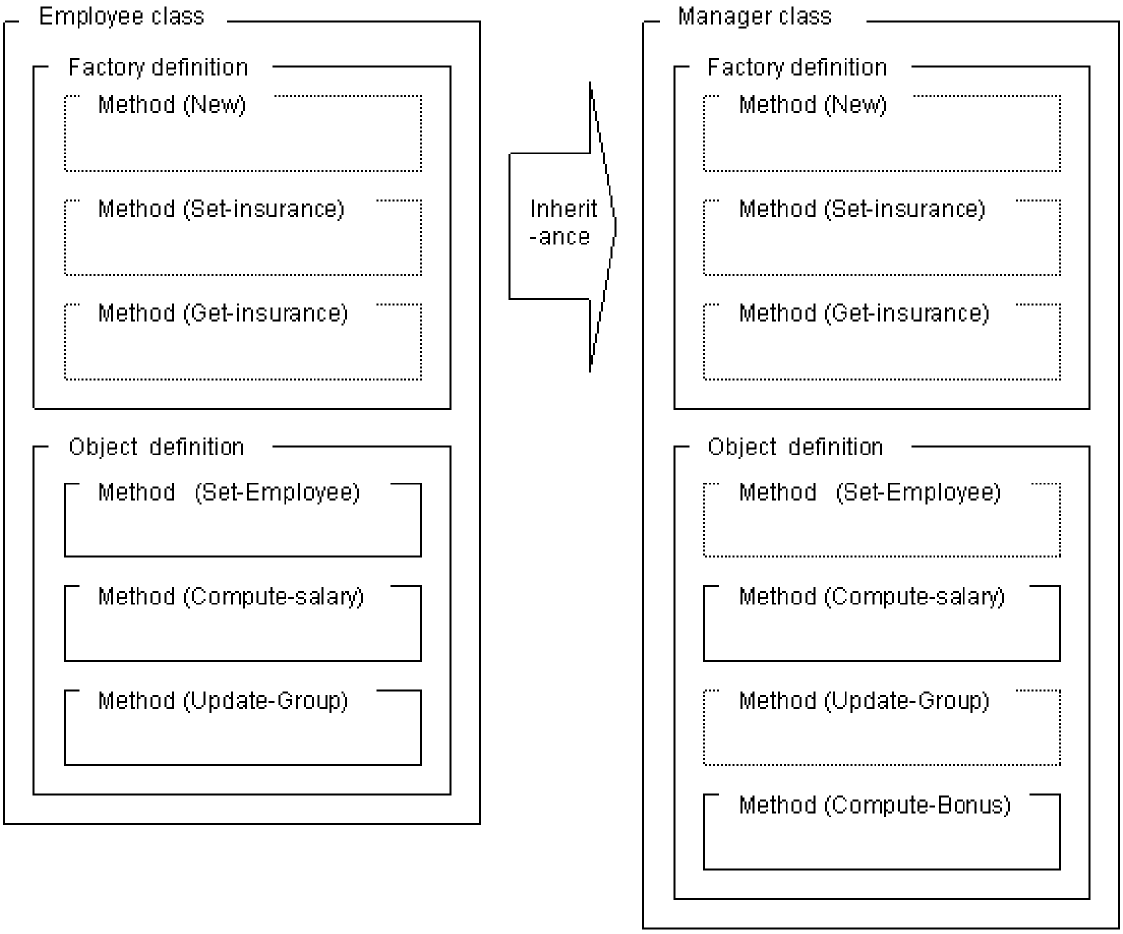
Basically, interfaces are being checked to ensure routines are invoked with the right parameters.

The following sections describe how and when these checks are performed.

14.4.1 Conforming Classes

It first helps to understand when one class conforms to another class and when they don't. Consider one class, the manager class, which inherits from the Employee class:

Figure 14.21 Manager class inheriting from employee class



You can see that the Manager class includes all of the features provided by the Employee class. We can say, "the Manager class conforms to the Employee class" because all the methods of the Employee class can be executed in the Manager class. The reverse is not true because not all the methods of the Manager class can be executed in the Employee class. We can say, "the Employee class does not conform to the Manager class".

In practical terms, this means the system can check that object reference data items contain object references that will match the method invocations in the code. Using the example above a line that has:

```
INVOKE employee-object-reference "Compute-Salary" ...
```

will also be valid if the employee-object-reference item contains a reference to a Manager class object.

The system determines when and how to make these checks by the way the object reference is declared.

14.4.2 Conformance Checking on Setting Object References

The following examples illustrate how conformance checking works when you set object references. See the section "Object Reference Types" above for explanations of the types.

```
 *>    :
   WORKING-STORAGE SECTION.
   01  obj-1       USAGE OBJECT REFERENCE.
   01  obj-2       USAGE OBJECT REFERENCE Employee-class.
   01  obj-3       USAGE OBJECT REFERENCE Employee-class ONLY.
 *>   :
   01  obj-X       USAGE OBJECT REFERENCE.
   01  obj-Y       USAGE OBJECT REFERENCE Manager-class.
   01  obj-Z       USAGE OBJECT REFERENCE Manager-class ONLY.
  *>  :
  PROCEDURE DIVISION.
      *>  :
      SET obj-1 TO obj-X.     *> [1]
      SET obj-1 TO obj-Y.     *> [2]
      SET obj-1 TO obj-Z.     *> [3]
      *> :
      SET obj-2 TO obj-X.     *> [4]
      SET obj-2 TO obj-Y.     *> [5]
      SET obj-2 TO obj-Z.     *> [6]
      *>:
      SET obj-3 TO obj-X.     *> [7]
      SET obj-3 TO obj-Y.     *> [8]
      SET obj-3 TO obj-Z.     *> [9]
      *> :
```

The above code produces the following results:

- Lines 1 - 3: No error.

  obj-1 can store object references of any class.

- Line 4: Compile-time error.

  obj-2 can only store object references of the Employee class and its subclasses, so it cannot be set to obj-X that could contain a reference to any class.

- Lines 5 - 6: No error.

  obj-2 can store object references of the Employee class and its subclasses - in this case the Manager subclass.

- Lines 7 - 9: Compile-time error.

  obj-3 can only store object references of the Employee class, so it cannot be set to the other references that contain, or might contain, references to objects of other classes.

14.4.3 Conformance Checking on Invoking Methods

The following examples illustrate how conformance checking works when you invoke methods:

```
  :
 WORKING-STORAGE SECTION.
 01  obj-1       USAGE OBJECT REFERENCE.
 01  obj-2       USAGE OBJECT REFERENCE Employee.
 01  obj-3       USAGE OBJECT REFERENCE Employee ONLY.
  :
PROCEDURE DIVISION.
     :
    INVOKE obj-1 "Compute-Salary" RETURNING salary.     ... [1]
    INVOKE obj-2 "Compute-Salary" RETURNING salary.     ... [2]
    INVOKE obj-3 "Compute-Salary" RETURNING salary.     ... [3]
     :
```

- Line 1

  The object reference, obj-1, can store a reference to an object of any class. Exactly which reference is stored is not determined until execution time. Therefore, the OO COBOL system can only check for conformance at execution time. If, at execution time, obj-1 contains a reference to an object that does not have a Compute-Salary method, an error will be generated.

- Line 2

- Because obj-2 is limited to storing object references of the Employee class or its subclasses, the compiler can check that a Compute-Salary method exists and that the correct parameters have been supplied. The compiler uses the data in the repository file for the Employee class to check the conformance.

- Line 3

  As with line 2, the class of the object reference is known so conformance is checked at compile-time.

14.4.3.1 Compile-Time and Execution-Time Conformance Checks

A conformance check is divided into two types: compile-time and execution-time conformance checks.

This section explains when a conformance check is carried out.

14.4.3.2 Assignment-Time Conformance Check

This section explains the assignment-time conformance check.

```
 *> :
 WORKING-STORAGE SECTION.
 01  OBJ-1       USAGE OBJECT REFERENCE.
 01  OBJ-2       USAGE OBJECT REFERENCE Employee-class.
 01  OBJ-3       USAGE OBJECT REFERENCE Employee-class ONLY.
 *> :
 01  OBJ-X       USAGE OBJECT REFERENCE.
 01  OBJ-Y       USAGE OBJECT REFERENCE Manager-class.
 01  OBJ-Z       USAGE OBJECT REFERENCE Manager-class ONLY.
 *> :
PROCEDURE DIVISION.
    *> :
    SET OBJ-1 TO OBJ-X.                         *> [1]
    SET OBJ-1 TO OBJ-Y.                         *> [2]
    SET OBJ-1 TO OBJ-Z.                         *> [3]
    *> :
    SET OBJ-2 TO OBJ-X.                         *> [4]
    SET OBJ-2 TO OBJ-Y.                         *> [5]
    SET OBJ-2 TO OBJ-Z.                         *> [6]
    *> :
    SET OBJ-3 TO OBJ-X.                         *> [7]
```

```
      SET OBJ-3 TO OBJ-Y.                         *> [8]
      SET OBJ-3 TO OBJ-Z.                         *> [9]
```

Explanation of diagram

- OBJ-1 is not subject to a conformance check during execution of the SET statement because it can contain references to objects of any class. Therefore, a conformance error does not occur in [1], [2], and [3].
- OBJ-2 can contain references to objects of the employee class and its child class. Therefore, a conformance error (compile time) occurs in [4] but does not occur in [5] and [6].
- OBJ-3 can contain only references to objects of the employee class, so a conformance error occurs in [7], [8], and [9]. A conformance check is carried out during compilation.

14.4.3.3 Conformance Check Carried Out during Method Invocation

This section explains a conformance check carried out during method invocation.

```
    :
   WORKING-STORAGE SECTION.
   01  OBJ-1       USAGE OBJECT REFERENCE.
   01  OBJ-2       USAGE OBJECT REFERENCE Employee-class.
   01  OBJ-3       USAGE OBJECT REFERENCE Employee-class ONLY.
    :
  PROCEDURE DIVISION.
       :
      INVOKE OBJ-1 "CMPT-SAL" RETURNING SALARY.
      INVOKE OBJ-2 "CMPT-SAL" RETURNING SALARY.
      INVOKE OBJ-3 "CMPT-SAL" RETURNING SALARY.
       :
```

Explanation of diagram

- OBJ-1 can contain references to objects of any class, so the method to be invoked cannot be determined until execution time. Therefore, whether an incorrect parameter or method is specified is checked during execution. If data other than references to objects of the employee class or its child class is defined for OBJ-1 during execution in the diagram above, a message indicating that the corresponding method is not found is output.
- OBJ-2 can contain only references to objects of the employee class and its child class. Because information such as method names and parameters can be recognized during compilation (by using repository information described later), a conformance check is accordingly carried out during compilation.
- OBJ-3 can contain only references to objects of the employee class. Therefore, a conformance check is carried out during compilation.

Specify an object reference data item with a class name to carry out a compile-time conformance check. This makes it possible to remove a fault during compilation if the fault is caused by an incompatible parameter during method invocation.

14.4.4 Modifying Object Reference Types

There may be times when you want to override the compiler's conformance checking on object references. You can do this using the object-modifier feature.

For example, consider the following code that assumes the manager class inherits from the all-employee class:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Main.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
   REPOSITORY.
     CLASS Employee-class
     CLASS Manager-class.
 DATA DIVISION.
```

```
  WORKING-STORAGE SECTION.
  01  obj-1   USAGE OBJECT REFERENCE All-Employee-class.
  01  obj-2   USAGE OBJECT REFERENCE Manager-class.
 PROCEDURE DIVISION.
*>    :
     INVOKE Manager-class "NEW" RETURNING obj-2.
*>    :
     SET obj-1 TO obj-2.
     SET obj-2 TO obj-1.  *> error
```

A compilation error occurs in the line indicated because the all-employee class does not conform to the manager class. However, in this particular case the reference in obj-1 is a reference to a manager object so executing the statement would not produce a non-conforming reference.

You can add an object modifier to obj-1 in the SET statement:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. Main.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS All-Employee-class
      CLASS Manager-class.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  obj-1   USAGE OBJECT REFERENCE All-Employee-class.
  01  obj-2   USAGE OBJECT REFERENCE Manager-class.
 PROCEDURE DIVISION.
*>    :
     INVOKE Manager-class "NEW" RETURNING obj-2.
*>    :
     SET obj-1 TO obj-2.
     SET obj-2 TO obj-1 AS Manager-class.
*>    :
```

This tells the compiler to treat obj-1 as if it contains a reference of the manager type (i.e. as if it were defined with USAGE OBJECT REFERENCE Manager-class). The compiler does its conformance check based on this usage. At execution time, the OO system checks to make sure that the object reference in obj-1 conforms to the manager class.

See the "COBOL Language Reference" for more details about object modifiers.

14.4.5 Summary

You can see that limiting object reference data items to specific classes allows conformance checks to be performed at compile time rather than execution time. Errors in invoking the wrong method or specifying the wrong parameters can be discovered at compile time, helping to shorten the development cycle and increasing the quality of your applications.

14.5 Using Repositories

14.5.1 Overview of Repositories

You may have noticed, in the examples for defining class inheritance and separate method definitions, that OO COBOL requires that you specify repositories for any classes referenced in your source.

Repositories are files that contain information about classes, interfaces, functions and properties. In OO applications, you will mostly be concerned with the class repository.

The class repository is a file created when a class is compiled. NetCOBOL creates a file called "class-name.REP", where class-name is the name specified in the CLASS-ID paragraph. This file contains all the necessary information about the class, and its super classes, to support the compilation of other classes or programs that inherit, invoke or reference the class. Therefore, whenever you reference a class in your program you must specify the class repository in the REPOSITORY paragraph in the CONFIGURATION section.

The repository file contains class-related information in a non-text format, which cannot be referenced directly by the user.

The repository file is used in the following ways:

- Input to the compiler to implement inheritance.
- Input to the compiler for conformance checking.

The following explains each operation.

14.5.1.1 Implementing Inheritance

Inheritance is implemented by inputting the repository file of a parent class. For example, when creating the manager class by inheriting the employee class, input the repository file of the employee class during compilation of the manager class.

```
IDENTIFICATION DIVISION.
CLASS-ID. Manager-class INHERITS Employee-class.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
     CLASS Employee-class.
 *> :
END CLASS Manager-class.
```

Note

The parent class (class specified in the INHERITS clause) must be specified in the repository paragraph.

A structure of two or more hierarchies to be inherited can be compiled by inputting the repository file of the parent class directly. For example, when the manager class is compiled, the FJBASE class is inherited indirectly; however, the repository file of the FJBASE class does not need to be input.

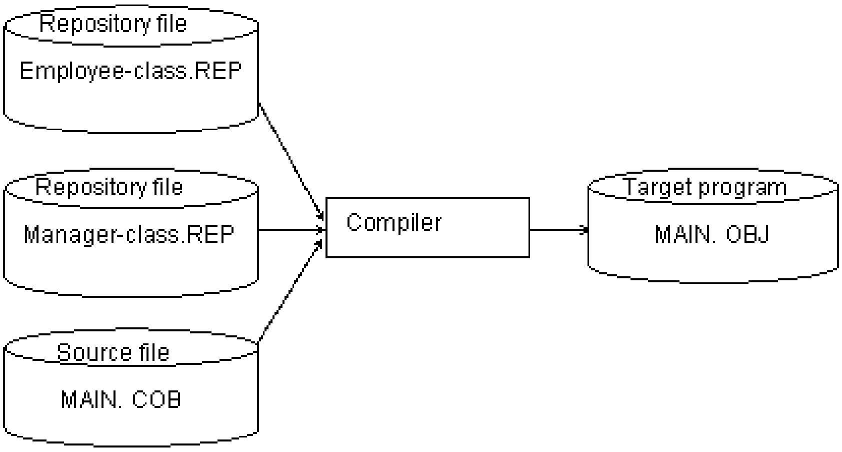
14.5.1.2 Implementing a Conformance Check

A conformance check can be implemented by inputting the repository file of a class to be invoked. For example, when the employee and manager classes are used in an employee management program, the repository files of those classes must be input during compilation of the employee management program.

[Employee management program]

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
     CLASS Employee-class
     CLASS Manager-class.
*> :
DATA DIVISION.
 WORKING-STORAGE SECTION.
01  employee-data.
    02  employee-no  PIC 9(06).
    02  name         PIC X(30).
    02  sect         PIC X(40).
    02  basic-salary PIC 9(08).
```

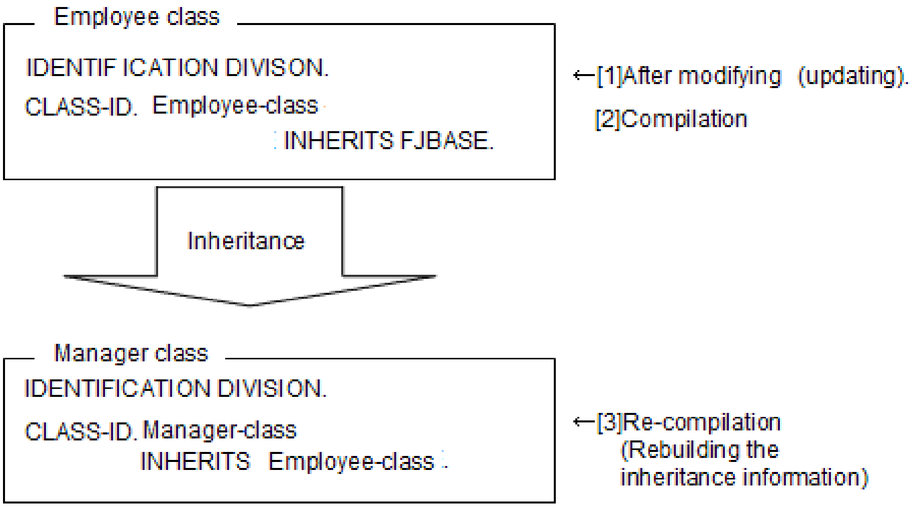
```
*> :
 01  OBJ-1       USAGE OBJECT REFERENCE Employee-class.
 01  OBJ-2       USAGE OBJECT REFERENCE Manager-class.
*> :
PROCEDURE DIVISION.
*>      :
    INVOKE Employee-class "NEW" RETURNING OBJ-1.
    INVOKE OBJ-1 "SETUP-DATA" USING EMPLOYEE-DATA.
*>      :
    INVOKE Manager-class "NEW" RETURNING OBJ-2.
    INVOKE OBJ-2 "SETUP-DATA" USING EMPLOYEE-DATA.
*>      :
```



14.5.2 Effects of Updating Repository Files

If you modify a parent class or an invoked class after compiling the source file that contains the references, you should recompile all sources that reference the modified class. However, if you only want to recompile when it is absolutely necessary, then the guidelines are:

- Recompile subclasses when the inherited structure changes.
- Recompile invoking classes or programs when the method interfaces change.



```
IDENTIF ICATION DIVISON.
CLASS-ID. Employee-class
                 INHERITS FJBASE.
```

```
IDENTIFICATION DIVISION.
CLASS-ID. Manager-class
          INHERITS  Employee-class .
```

As in the figure above, the child class (manager class) of the corrected class (employee class) needs to be recompiled even if it has not been corrected (for restructuring inheritance information).

Also, the program and class that calls the corrected class should be recompiled (for the re-attempting of the conformance check).

[Manager program]

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.                                *> Recompilation
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
     CLASS Employee-class
     CLASS Manager-class.
*> :
DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  OBJ-1 USAGE OBJECT REFERENCE.
 01  OBJ-2 USAGE OBJECT REFERENCE.
*> :
PROCEDURE DIVISION.
*>      :
    INVOKE Employee-class "NEW" RETURNING OBJ-1.  *>--+ For reattempting of
*>      :                                         *>  |  the conformanc check
    INVOKE Manager-class "NEW" RETURNING OBJ-2.   *>--+
```

Users should execute these recompilation. Therefore, pay much attention when you correct the class definition that has been constructed once.

14.6 Method Binding

Binding is the process by which names in code are connected to actual data or procedures. For methods, the OO system has to determine:

- What is the object whose method is to be invoked?
- What is the method name?

There are two types of binding: static binding and dynamic binding. Static binding is when the object and method can be determined at compile time. Dynamic binding is when the object and method can only be determined at execution time. The OO COBOL system determines which form of binding is required.

The sections below explain the type of binding used in different situations. It is important that you understand the binding process as it can effect the behavior and performance of your OO application.

Note that other OO environments, such as C++, use the term "late binding" instead of "dynamic binding".

14.6.1 Static Binding

For static binding, the compiler must be able to determine the object and method at compile time.

The following examples show situations in which static binding will be used:

```
*>
 01  obj-1 USAGE OBJECT REFERENCE Employee-class ONLY.
 01  Salary  PIC 9(08).
PROCEDURE DIVISION.
    *>      :
    INVOKE Employee-class "NEW" RETURNING obj-1.    *>[1]
    *>      :
    INVOKE obj-1 "Compute-Salary" RETURNING Salary. *>[2]
    *>      :
```

```
    INVOKE SUPER "Compute-Salary" RETURNING Salary. *>[3]
    *>         :
```

- [1] Because the class name "Employee" is provided for the object identifier, the factory object is being invoked. There is only one factory object for a class so the target object and method are completely determined at compile time.
- [2] Because obj-1 can only contain references to the Employee class, the Calculate-Salary method must be that defined for the Employee class. The method can be statically bound.
- [3] Because SUPER refers to the parent (or higher) class the method is known at compile time and can be statically bound.

14.6.2 Dynamic Binding

Dynamic binding is used when the object and method to be used cannot be determined until execution time.

The following examples illustrate situations where dynamic binding is necessary:

```
     :
   WORKING-STORAGE SECTION.
   01  obj-1       USAGE OBJECT REFERENCE.
   01  obj-2       USAGE OBJECT REFERENCE Employee.
     :
  PROCEDURE DIVISION.
        :
      INVOKE obj-1 "Calculate-Salary"  RETURNING salary.    ... [1]
        :
      INVOKE obj-2 "Calculate-Salary"  RETURNING salary.    ... [2]
        :
      INVOKE SELF  "Calculate-Salary"  RETURNING salary.    ... [3]
        :
```

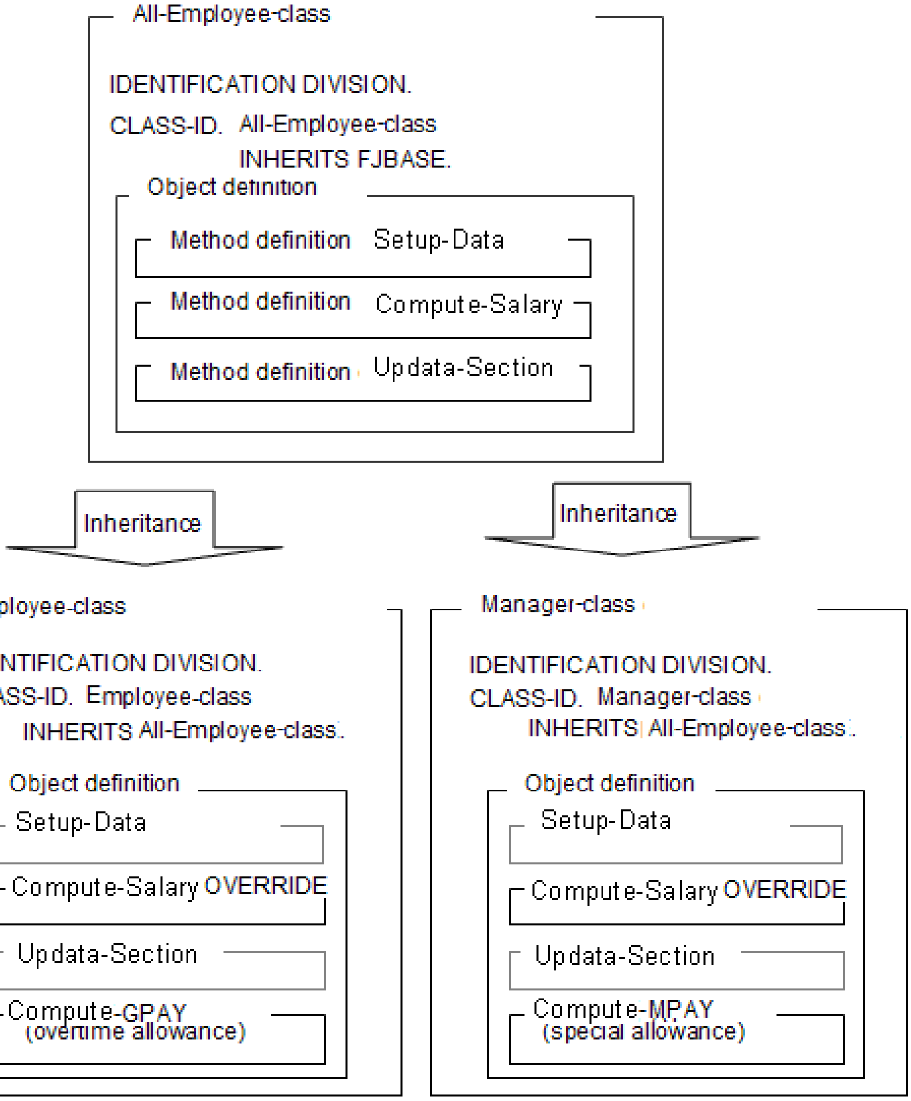
- [1] Because obj-1 can contain a reference to an object of any class, the method to be used cannot be determined until execution time.
- [2] Obj-2 can contain a reference to an object of the employee class, or one of its subclasses. Which of these classes will be referenced is not know until execution time, so dynamic binding must be used.
- [3] SELF refers to the object that is being executed. The method could be executed for an object of the class in which the method was defined, or for an object of a class that inherits from the class that defined the method. Exactly which object is being executed cannot be determined until execution time, so dynamic binding is used.

Binding and Polymorphism

Dynamic binding enables the same code to invoke different routines at execution time.

For example, we could structure our employee and manager class example slightly differently. Suppose an abstract class called all-employees is created to contain the common features of general employees and managers, and that the employee and manager classes both inherit from the all-employees class:

Figure 14.22 Alternative structure for employee/manager class example



[Employee management program(MAIN)]

DATA DIVISION

```
  WORKING-STORAGE SECTION.
  01  OBJ-1   USAGE OBJECT REFERENCE Employee-class.
  01  OBJ-2   USAGE OBJECT REFERENCE Manager-class.
  01  OBJ-X   USAGE OBJECT REFERENCE All-Employee-class.
*>   :
```

PROCEDURE DIVISION

```
*>
     SET OBJ-X TO OBJ-1.
     PERFORM Salary-computation.
*>      :
     SET OBJ-X TO OBJ-2.
     PERFORM Salary-computation.
     EXIT PROGRAM.
*>      :
 Salary-computation SECTION.
     INVOKE OBJ-X "Compute-Salary" RETURNING Salary
*>   :
```

As shown above, you can execute the salary computation without being conscious of objects (general employees or managers). That is, two or more objects defined separately can be operated with one object reference data item; this function is called polymorphism.

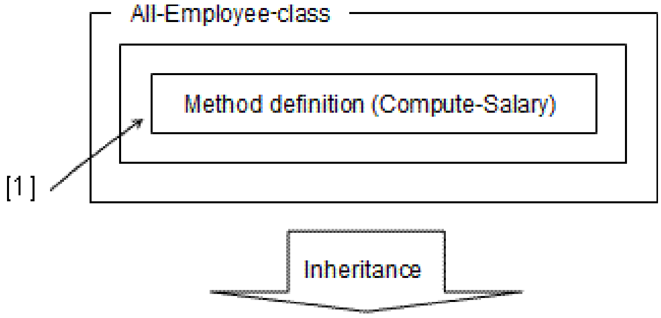
You can see that the code in the SALARY-CALCULATION section is identical for managers and employees but it is actually invoking different methods in the Compute-Salary invocation.

14.6.3 Binding with the Predefined Object Identifier SUPER

The predefined object identifier SUPER can help you ensure that the correct method is bound at execution time.

For example, suppose a manager class inherits from an all-employee class, and that it overrides the Compute-Salary method. However, in this case, the overriding code wants to use the Compute-Salary method of the all-employee class as part of its calculations. To ensure that the parent's method is used the identifier SUPER is coded:

Figure 14.23 Binding with SUPER



[Manager-class]

```
CLASS-ID. Manager-class INHERITS All-Employee-class.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS All-Employee-class.
 FACTORY.
 PROCEDURE DIVISION.
 METHOD-ID. Get-Insur.
 DATA DIVISION.
  LINKAGE SECTION.
  01 INSUR     PIC 9(08).
 PROCEDURE DIVISION RETURNING INSUR.
     MOVE 500 TO INSUR.
 END METHOD Get-Insur.
 END FACTORY.
 OBJECT.
 PROCEDURE DIVISION.
 METHOD-ID. Compute-Salary OVERRIDE.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 BASE-SAL  PIC 9(08).
  01 MPAY      PIC 9(08).
  01 INSUR     PIC 9(08).
  01 OBJ-F     USAGE OBJECT REFERENCE FACTORY OF SELF.
  LINKAGE SECTION.
  01 Salary    PIC 9(08).
 PROCEDURE DIVISION RETURNING SALARY.
     INVOKE SUPER "Compute-Salary"  RETURNING BASE-SAL.  *>--->[1]
     INVOKE SELF  "Compute-MPAY" RETURNING MPAY.
     INVOKE SELF  "GETCLASS"  RETURNING OBJ-F.
     INVOKE OBJ-F "Get-Insur" RETURNING INSUR.
     COMPUTE Salary = BASE-SAL + MPAY - INSUR.
 END METHOD Compute-Salary.
```

14.6.4 Binding with the Predefined Object Identifier SELF

The predefined object identifier SELF can also help you manage the dynamic binding process.

For example, suppose a manager class overrides the Compute-Salary method for the all-employee class, but you want a Compute-Bonus method to work for both manager and employee classes. You could code the following:

[All-Employee-class]

```
CLASS-ID. All-Employee-class INHERITS FJBASE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS FJBASE.
 OBJECT.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
  01  employee-data.
    02  employee-no  PIC 9(06).
    02  name         PIC X(30).
    02  sect         PIC X(40).
    02  basic-salary PIC 9(08).
 PROCEDURE DIVISION.
*>
 METHOD-ID. Compute-Salary. *>[1] Salary Calculation Method
 DATA DIVISION.
  LINKAGE SECTION.
  01  Salary  PIC 9(08).
 PROCEDURE DIVISION RETURNING SALARY.
     MOVE basic-salary TO Salary.
 END METHOD Compute-Salary.
*>
 METHOD-ID. Compute-Bonus.  *>    Bonus Calculation Method
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  Salary  PIC 9(08).
 PROCEDURE DIVISION.
*>      :
     INVOKE SELF "Compute-Salary" RETURNING Salary.  *>-->  [1]
     DISPLAY "SALARY =" SALARY
*>      :
 END METHOD Compute-Bonus.
*>      :
```

Inheritance

[Manager-class]

```
CLASS-ID. Manager-class INHERITS All-Employee-class.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS All-Employee-class.
 OBJECT.
 PROCEDURE DIVISION.
 METHOD-ID. Compute-Salary OVERRIDE.     *>[2]
 DATA DIVISION.
  LINKAGE SECTION.
  01  SALARY  PIC 9(08).
 PROCEDURE DIVISION RETURNING SALARY.
     INVOKE SUPER "Compute-Salary" RETURNING SALARY.
     COMPUTE SALARY = SALARY + 20000.
 END METHOD Compute-Salary.
*>
*> The Compute-Bonus method was the one defined implicitly
*> by inheriting the All-Employee-class.
*>
*> METHOD-ID. Compute-Bonus.
*> DATA DIVISION.
*>  WORKING-STORAGE SECTION.
*>  01  Salary  PIC 9(08).
*> PROCEDURE DIVISION.
*>      :
*>     INVOKE SELF "Compute-Salary" RETURNING Salary.  *>-->[2]
*>     DISPLAY "SALARY =" SALARY
*>      :
*> END METHOD Compute-Bonus.
 END OBJECT.
 END CLASS Manager-class.
```

14.7 Using Properties

OO COBOL provides a convenient technique of getting and setting data within objects, known as the object PROPERTY feature. The technique has two parts:

- Implicit or explicit definition of GET and SET methods
- Using object-property syntax to invoke the GET and SET methods.

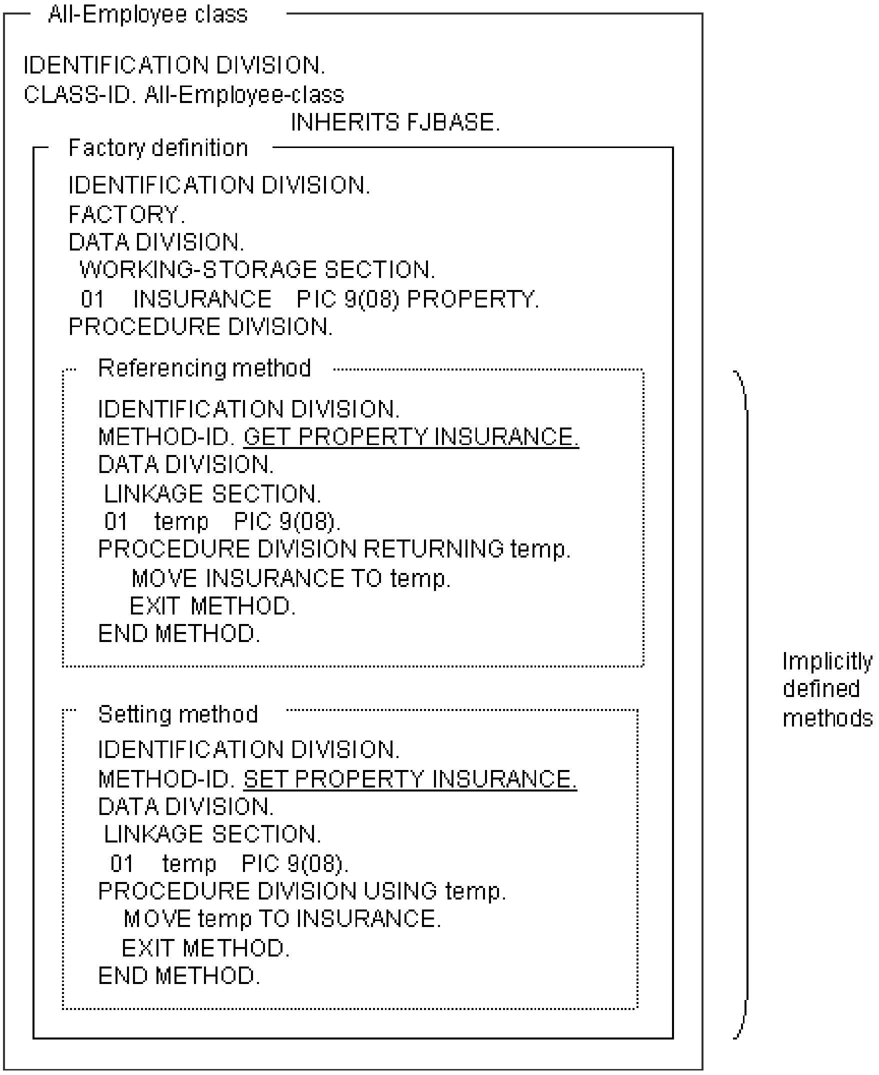
These parts are explained in the following sections:

- Defining Properties
- Accessing Properties
- Coding Explicit Property GET and SET methods

14.7.1 Defining Properties

You indicate that an elementary data item can be accessed by implicitly generated methods by specifying the PROPERTY clause in its definition. When you specify the PROPERTY clause, two methods, "GET PROPERTY data-name" and "SET PROPERTY data-name" are implicitly defined, as shown in this example:

Figure 14.24 Implicitly defined property methods



```
IDENTIFICATION DIVISION.
CLASS-ID. All-Employee-class
                           INHERITS FJBASE.
  IDENTIFICATION DIVISION.
  FACTORY.
  DATA DIVISION.
   WORKING-STORAGE SECTION.
   01    INSURANCE    PIC 9(08) PROPERTY.
  PROCEDURE DIVISION.
    IDENTIFICATION DIVISION.
    METHOD-ID. GET PROPERTY INSURANCE.
    DATA DIVISION.
     LINKAGE SECTION.
     01    temp    PIC 9(08).
    PROCEDURE DIVISION RETURNING temp.
        MOVE INSURANCE TO temp.
        EXIT METHOD.
    END METHOD.

    IDENTIFICATION DIVISION.
    METHOD-ID. SET PROPERTY INSURANCE.
    DATA DIVISION.
     LINKAGE SECTION.
      01    temp    PIC 9(08).
    PROCEDURE DIVISION USING temp.
       MOVE temp TO INSURANCE.
       EXIT METHOD.
    END METHOD.
```

14.7.2 Accessing Properties

You do not access properties by using INVOKE statements, rather you use the object property syntax:

```
Property-name OF object-identifier
```

For example, suppose that both the employee and manager classes inherit from the all-employee class in the code sample above. You would refer to the INSURANCE property as shown below:

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. MAIN.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS All-Employee-class
      CLASS Manager-class.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  insurance-premium  PIC 9(08).
  01  Employee-id        PIC 9(02).
  CONSTANT SECTION.
  01  id-Employee PIC 9(02) VALUE 1.
  01  id-Manage   PIC 9(02) VALUE 2.
 PROCEDURE DIVISION.
*>      :
*> GROUP INSURANCE PREMIUM SETTING PROCESS
*>
     MOVE insurance-premium TO INSURANCE OF All-Employee-class.  *> invokes the setting
     MOVE insurance-premium TO INSURANCE OF Manager-class.       *> method.
*>
*> GROUP INSURANCE PREMIUN RETRIEVAL PROCESS
*>
   Group-insurance-process.
     EVALUATE Employee-id
     WHEN id-Employee
       MOVE INSURANCE OF All-Employee-class TO insurance-premium *> invokes the
     WHEN id-Manage                                              *> referencing method.
       MOVE INSURANCE OF Manager-class TO insurance-premium      *>
     END-EVALUATE.
*>   :
```

The compiler invokes the appropriate GET or SET method depending on whether the object property is specified for the source or target data of the statement.

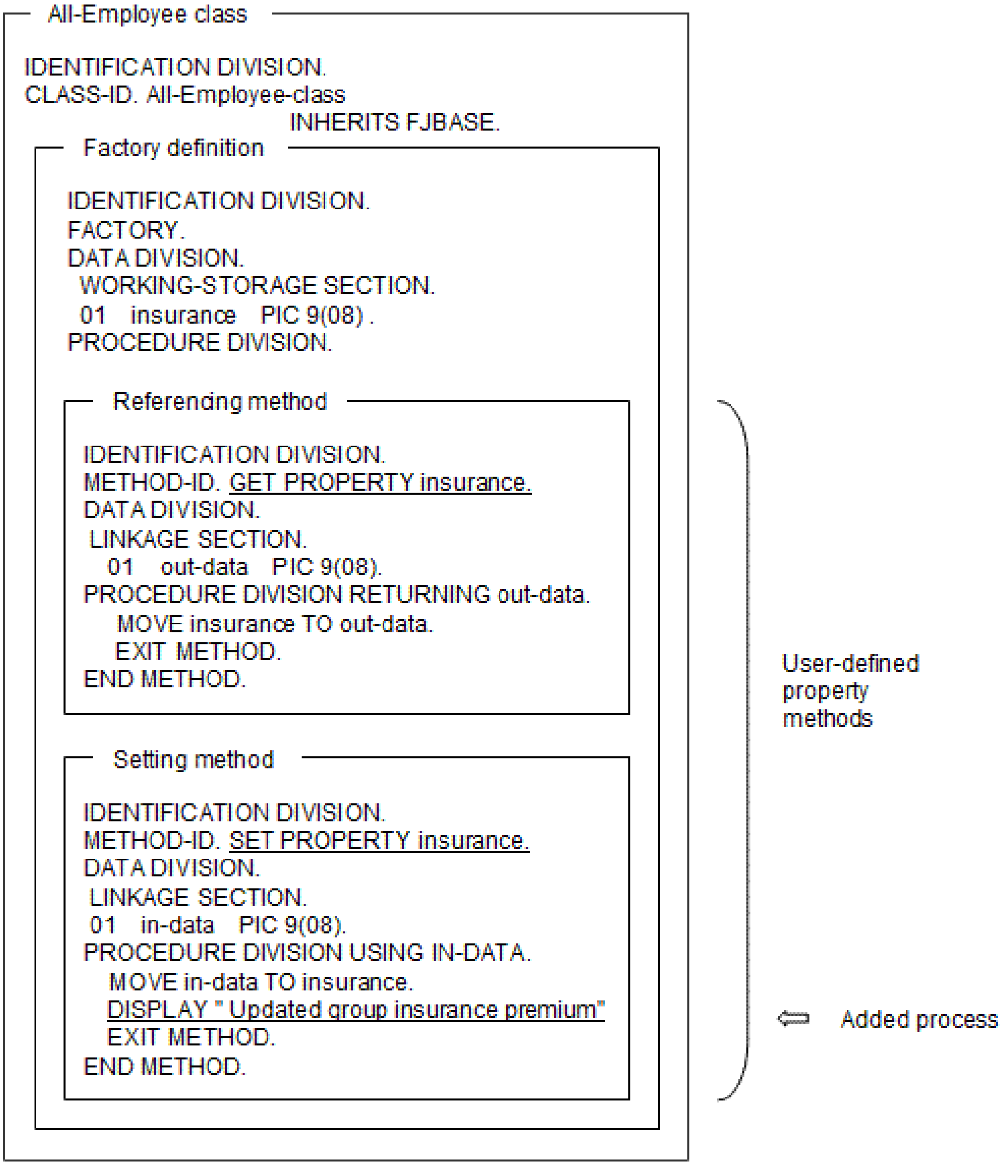
Note that the object-identifiers in the above examples are the class names because the property is in the factory data.

14.7.3 Coding Explicit Property GET and SET methods

You may want to use the object property syntax (property-name OF object-identifier) but also have some special processing when the data is referenced or set. You can do this by omitting the PROPERTY clause from the data definition and explicitly coding the "GET PROPERTY data-name" and "SET PROPERTY data-name" methods.

The code below shows how you could add a simple message when the INSURANCE property is updated. The code shown in the Accessing Properties section above would remain the same.

Figure 14.25 User defined property methods



```
IDENTIFICATION DIVISION.
CLASS-ID. All-Employee-class
                          INHERITS FJBASE.
   IDENTIFICATION DIVISION.
   FACTORY.
   DATA DIVISION.
    WORKING-STORAGE SECTION.
    01   insurance   PIC 9(08) .
   PROCEDURE DIVISION.

     IDENTIFICATION DIVISION.
     METHOD-ID. GET PROPERTY insurance.
     DATA DIVISION.
      LINKAGE SECTION.
        01   out-data   PIC 9(08).
     PROCEDURE DIVISION RETURNING out-data.
         MOVE insurance TO out-data.
         EXIT METHOD.
     END METHOD.

     IDENTIFICATION DIVISION.
     METHOD-ID. SET PROPERTY insurance.
     DATA DIVISION.
      LINKAGE SECTION.
      01   in-data   PIC 9(08).
     PROCEDURE DIVISION USING IN-DATA.
        MOVE in-data TO insurance.
        DISPLAY " Updated group insurance premium"
        EXIT METHOD.
     END METHOD.
```

Note that if you specify one of the GET or SET methods you must specify both, because you cannot use the PROPERTY clause on the data item.

The PROPERTY clause is discussed in more detail in "14.8 Using More Advanced Functions".

14.8 Using More Advanced Functions

The basic concepts and functions of object-oriented programming were explained in the preceding sections. Object-oriented programming can be implemented using these functions. However, there are many other, more advanced, functions and those make it easy to use the basic functions.

This section explains how to use the extended functions.

14.8.1 PROTOTYPE Declaration of a Method

Allows a method definition described in a class definition to be stored physically in another file.

Describe a method name and interface in the class definition and a method procedure and data in another compilation unit. The method described in the class definition is referred to as a PROTOTYPE method, and the one described in the compilation unit is referred to as a separate method.

All-Employee-class

```
IDENTIFICATION DIVISION.
  CLASS-ID. All-Employee-class INHERITS FJBASE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS FJBASE.
 OBJECT.
 PROCEDURE DIVISION.
 METHOD-ID. UPD-SECT PROTOTYPE.          *>---+ PROTOTYPE method
 DATA DIVISION.                          *>    | Describe only interface.
 LINKAGE SECTION.                        *>    |
 01  NEW-SECT     PIC X(40).             *>    |
 PROCEDURE DIVISION  USING NEW-SECT.     *>    |
 END METHOD UPD-SECT.                    *>---+
*>   :
```

UPD-SECT(Separate method)

```
IDENTIFICATION DIVISION.
 METHOD-ID. UPD-SECT OF All-Employee-class. *>   [1]
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS All-Employee-class.             *>   [2]
 DATA DIVISION.                             *>--+
  WORKING-STORAGE SECTION.                  *>  |[3]
*>  :                                           |
  LINKAGE SECTION.                          *>  |
  01  NEW-SECT     PIC X(40).               *>--+
 PROCEDURE DIVISION  USING NEW-SECT.        *>--+
     DISPLAY NEW-SECT.                      *>  |[4]
*>  :                                        ---+
 END METHOD UPD-SECT.
```

- [1],[2] Specify the name of a class logically containing the method.
- [3] Describe data to be used in the method,
- [4] Describe a method procedure.

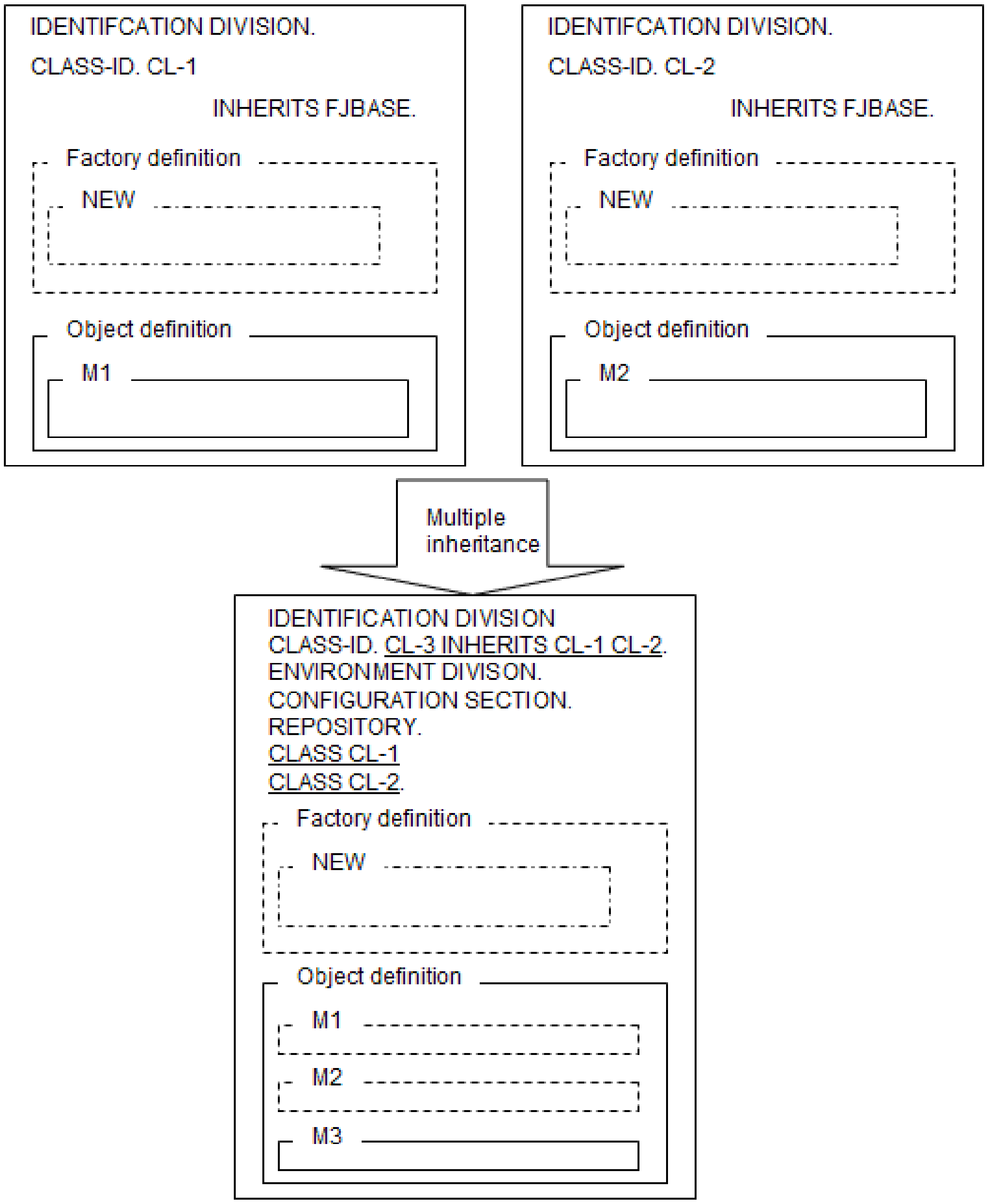
When two or more persons are developing a class definition, describe its method definition in separate compilation units as shown above.

When compiling the separate method, input the repository file of a class logically containing that method. Therefore, the class definition must be compiled before the method is. For details, see "Chapter 15 Developing OO COBOL Applications".

14.8.2 Multiple Inheritance

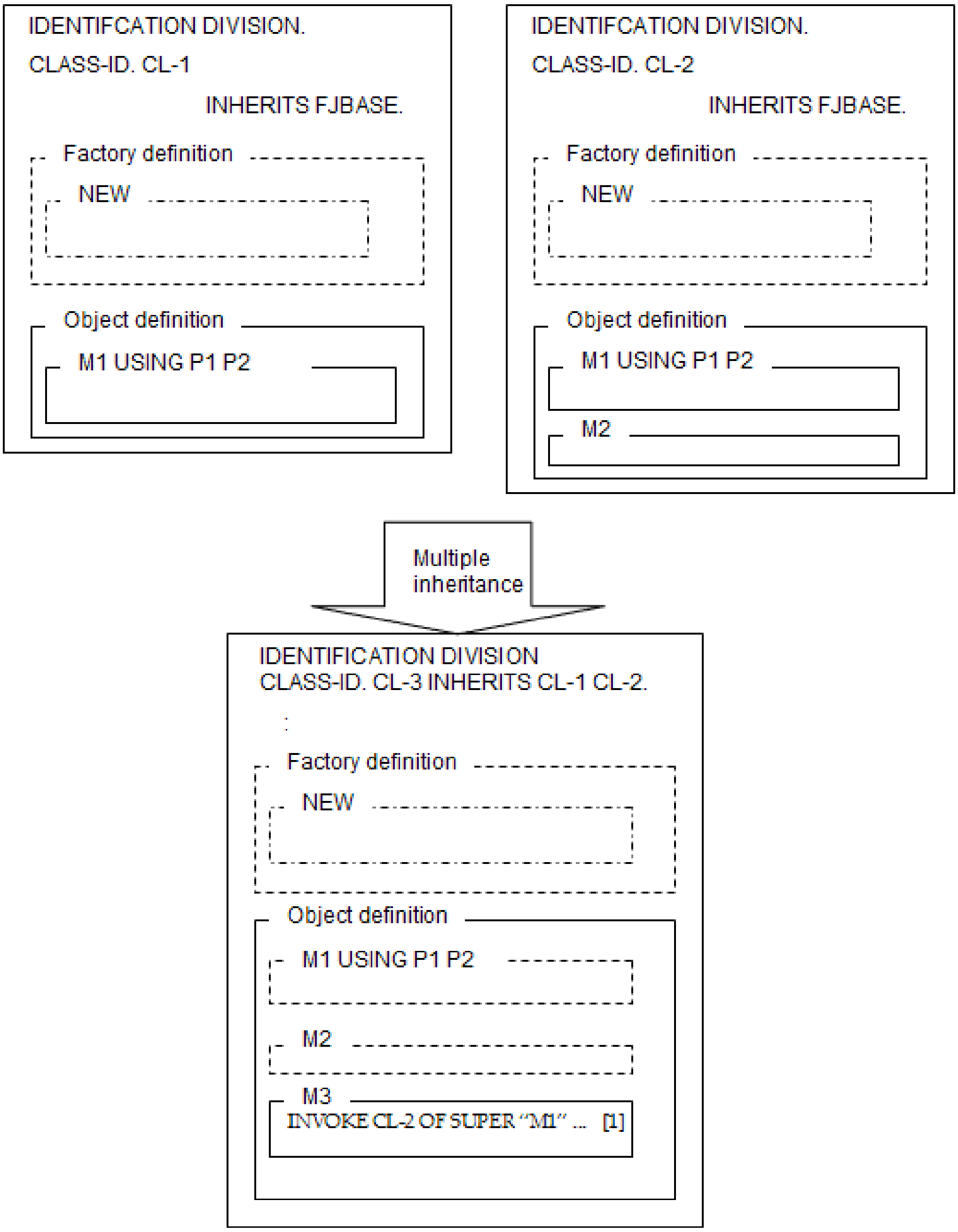
Although inheritance was described in the preceding section, two or more classes can be inherited at the same time. This function is referred to as multiple inheritance.

Figure 14.26 Multiple inheritance



The logic of multiple inheritance is the same as that of single-class inheritance. However, observe the rule "When two or more methods of the same name are defined in two or more parent classes, they must have the same interface" (see the diagrams below).

Figure 14.27 Common inherited methods in multiple inheritance



When two or more parent classes contain two or more methods of the same name, the following operation is performed unless the user explicitly overrides these methods:

A list of class names specified in the INHERITS clause is searched from the left until a class containing the pertinent method is found, then the method of the class found first is inherited.

In the diagram shown above, method M1 is inherited from classes CL-1 and CL-2 to CL-3. M1 is determined to be that of CL-1 as a result of searching from the left the class names specified in the INHERITS clause of CL3. Therefore, when invoking M1 of class CL-2 from CL-3, explicitly specify the class name with the predefined object identifier SUPER as shown in [1] in the diagram shown above.

14.8.3 In-line Invocation

Usually, methods are invoked using the INVOKE statement; however, a method (writing style) that does not use the INVOKE statement is available. This method is referred to as an in-line method invocation.

The in-line invocation is used to reference only a return value (item value specified with RETURNING) from a method. Therefore, it can be used only for the methods that have a return item.

```
IDENTIFICATION DIVISION.
 CLASS-ID. CL-1 INHERITS FJBASE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS FJBASE.
 OBJECT.
 PROCEDURE DIVISION.
 METHOD-ID. M1.
 DATA DIVISION.
  LINKAGE SECTION.
  01  P1     PIC X(10).
  01  P2     PIC X(10).
  01  P3     PIC S9(4) BINARY.
 PROCEDURE DIVISION  USING P1 P2 RETURNING P3.
 *> :
```

When invoking method M1 and referencing return value P3, the definition is described as shown below if the INVOKE statement is used.

```
*>     :
   01  OBJ-1 USAGE OBJECT REFERENCE CL-1.
  PROCEDURE DIVISION.
*>        :
      INVOKE CL-1  "NEW" RETURNING OBJ-1
      INVOKE OBJ-1 "M1" USING P1 P2 RETURNING P3.
      IF P3 = 0 THEN
*>        :
```

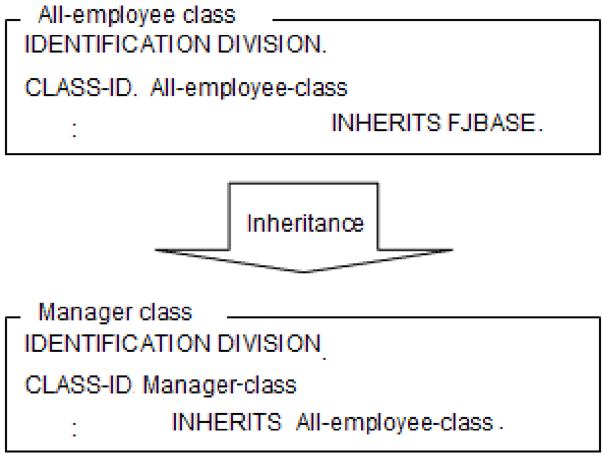
The definition can be described as shown below if in-line invocation is used.

```
*>  :
   01  OBJ-1 USAGE OBJECT REFERENCE CL-1.
 PROCEDURE DIVISION.
*>        :
      INVOKE CL-1  "NEW" RETURNING OBJ-1
      IF OBJ-1 :: "M1"(P1 P2) = 0 THEN
*>    :
```

14.8.4 Object Specifiers

The conformance check was explained in the preceding section; however, it may be possible to execute processing normally even if a conformance rule is infringed upon. In this case, a conformance check made by the compiler can be avoided by using an object specifier.

Figure 14.28 Inheritance example



```
IDENTIFICATION DIVISION.
CLASS-ID.  All-employee-class
    :                            INHERITS FJBASE.
```

```
IDENTIFICATION DIVISION.
CLASS-ID. Manager-class
    :           INHERITS  All-employee-class.
```

When the inheritance relationship shown above exists

Employee management program:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. MAIN.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS All-Employee-class
      CLASS Manager-class.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  OBJ-1   USAGE OBJECT REFERENCE All-Employee-class.
  01  OBJ-2   USAGE OBJECT REFERENCE Manager-class.
 PROCEDURE DIVISION.
*>    :
     INVOKE Manager-class "NEW" RETURNING OBJ-2.
*>    :
     SET OBJ-1 TO OBJ-2.  *>[1]
     SET OBJ-2 TO OBJ-1.  *>[2] error
*>    :
```

The assignment can be performed normally in [1] because the manager class is the child class of the all-employee class. However, if an attempt is made to assign or return a value to the original item in [2], an error occurs because the all-employee class is not the child class of the manager class. That is, the assignment is disabled by an interclass inheritance relationship even if it is normal from the point of view of the stored data.

Use object specifiers to enable the assignment shown above.

Employee management program:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. MAIN.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS All-Employee-class
      CLASS Manager-class.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  OBJ-1   USAGE OBJECT REFERENCE All-Employee-class.
  01  OBJ-2   USAGE OBJECT REFERENCE Manager-class.
 PROCEDURE DIVISION.
*>     :
     INVOKE Manager-class "NEW" RETURNING OBJ-2.
*>     :
     SET OBJ-1 TO OBJ-2.
     SET OBJ-2 TO OBJ-1 AS Manager-class.
*>     :
```

When the definition is described as shown above, the assignment is enabled because a conformance check is made, assuming that MANAGER is specified in the USAGE OBJECT REFERENCE clause for OBJ-1.

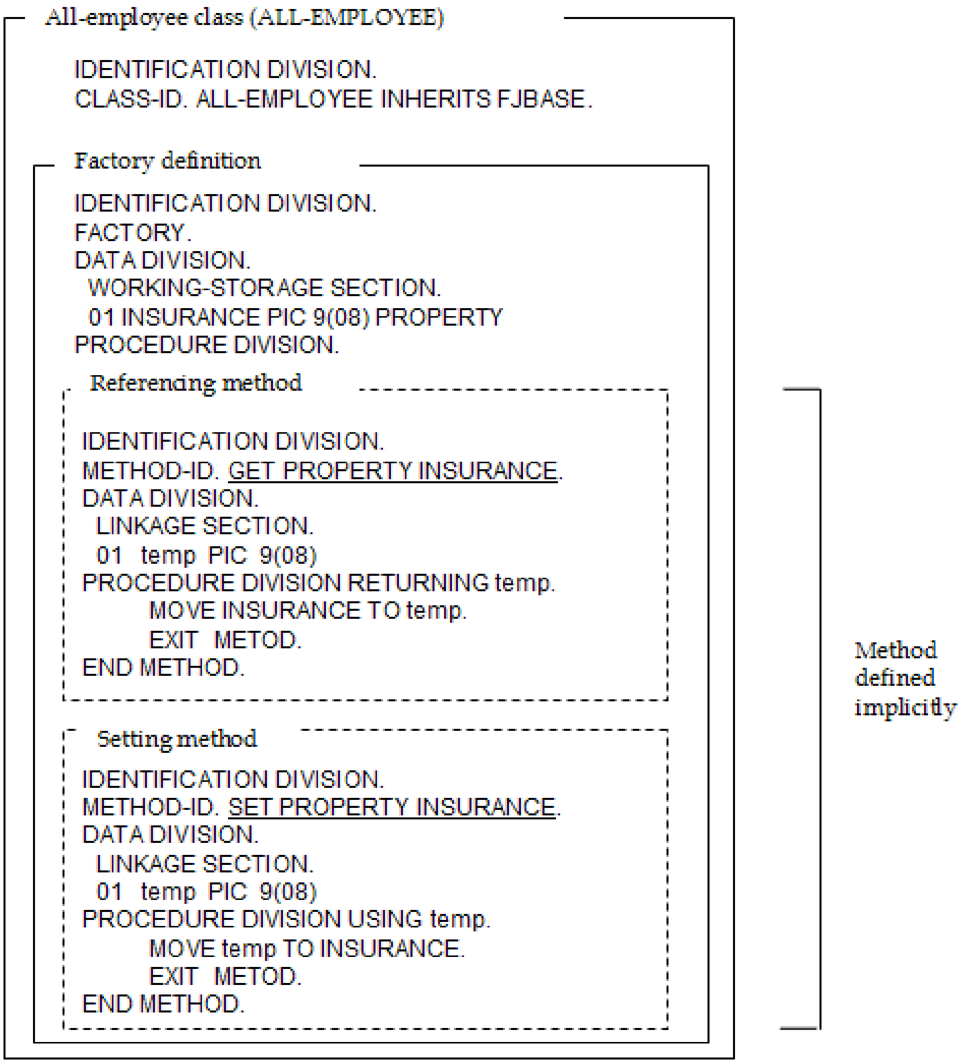
When an object specifier is used, a conformance check is made with a specified class name during compilation. During execution, however, the conformance check is made with an object reference actually stored. That is, an error is assumed at execution time if a conformance rule is infringed upon.

14.8.5 PROPERTY Clause

Factory and object data can be referenced and set with ease by using the PROPERTY clause.

The method in which data is referenced and set is created automatically by specifying the PROPERTY clause.

In the example of the all-employee class, a method was created in which a group insurance premium (factory data) is referenced and set. However, an implicit method (method having no source description but existing logically) is created automatically as shown below by specifying the PROPERTY clause as a data declaration.



```
IDENTIFICATION DIVISION.
CLASS-ID. ALL-EMPLOYEE INHERITS FJBASE.
```

```
IDENTIFICATION DIVISION.
FACTORY.
DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 INSURANCE PIC 9(08) PROPERTY
PROCEDURE DIVISION.
```

```
IDENTIFICATION DIVISION.
METHOD-ID. GET PROPERTY INSURANCE.
DATA DIVISION.
 LINKAGE SECTION.
 01  temp  PIC  9(08)
PROCEDURE DIVISION RETURNING temp.
       MOVE INSURANCE TO temp.
       EXIT  METOD.
END METHOD.
```

```
IDENTIFICATION DIVISION.
METHOD-ID. SET PROPERTY INSURANCE.
DATA DIVISION.
 LINKAGE SECTION.
 01  temp  PIC  9(08)
PROCEDURE DIVISION USING temp.
       MOVE temp TO INSURANCE.
       EXIT  METOD.
END METHOD.
```

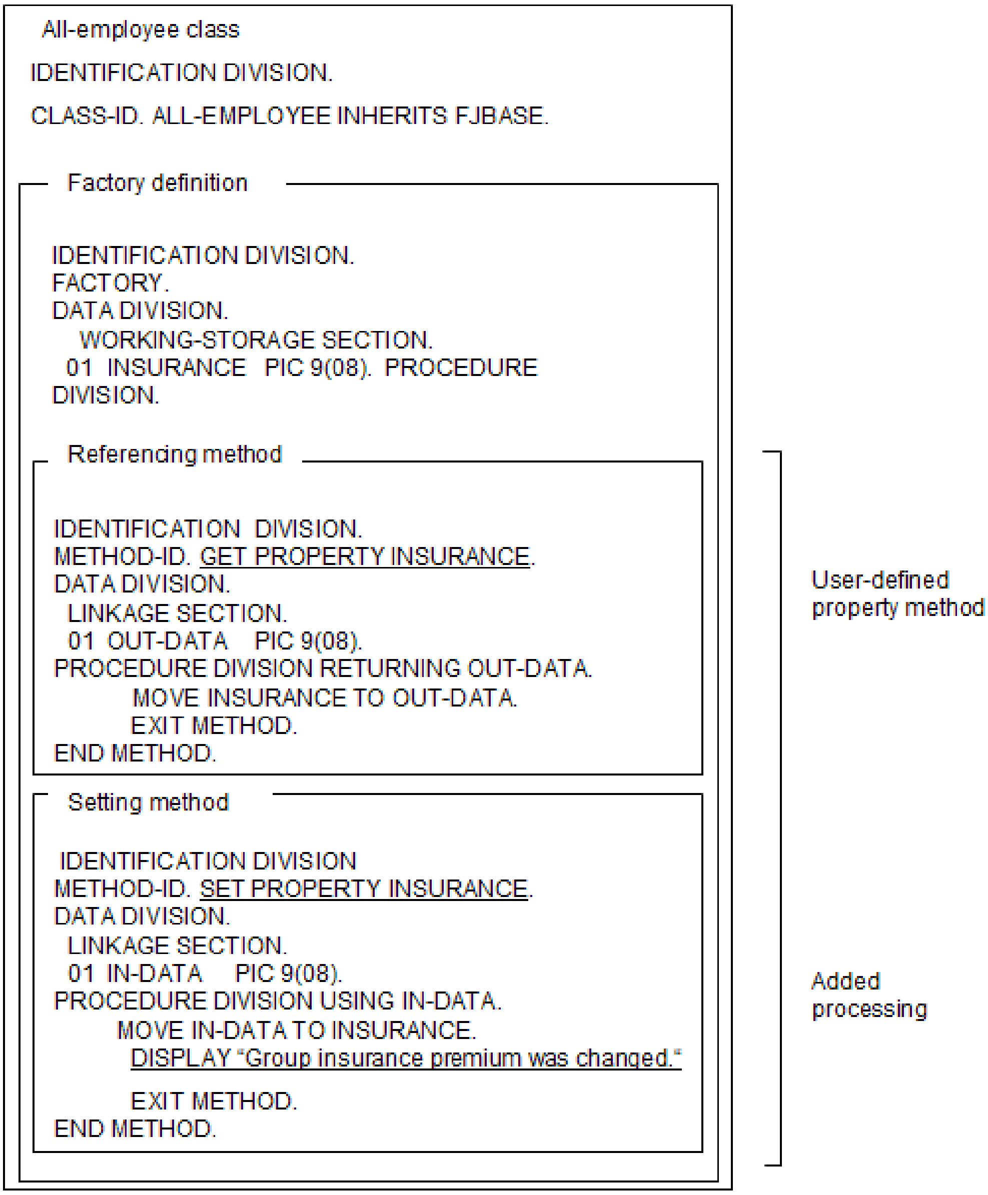
The method defined implicitly in the PROPERTY clause as shown above is referred to as a property method.

The property method cannot be invoked using the INVOKE statement. Instead, use a unique reference, referred to as an object property, as shown below.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. MAIN.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS All-Employee-class
      CLASS Manager-class.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  insurance-premium  PIC 9(08).
  01  Employee-id        PIC 9(02).
  CONSTANT SECTION.
  01  id-Employee PIC 9(02) VALUE 1.
  01  id-Manage   PIC 9(02) VALUE 2.
 PROCEDURE DIVISION.
*>      :
*> GROUP INSURANCE PREMIUM SETTING PROCESS
*>
     MOVE insurance-premium TO INSURANCE OF All-Employee-class.  *> invokes the setting
     MOVE insurance-premium TO INSURANCE OF Manager-class.       *> method.
*>
*> GROUP INSURANCE PREMIUN RETRIEVAL PROCESS
*>
   Group-insurance-process.
     EVALUATE Employee-id
     WHEN id-Employee
       MOVE INSURANCE OF All-Employee-class TO insurance-premium *> invokes the
     WHEN id-Manage                                              *> referencing method.
       MOVE INSURANCE OF Manager-class TO insurance-premium      *>
     END-EVALUATE.
*>   :
```

Whether to invoke the referencing or setting method depends on whether the object property is specified on the sending or receiving side.

When you want the property method to have additional processing, you can define it explicitly. In this case, you do not need to specify the PROPERTY clause in the data.



```
IDENTIFICATION DIVISION.
CLASS-ID. ALL-EMPLOYEE INHERITS FJBASE.

  IDENTIFICATION DIVISION.
  FACTORY.
  DATA DIVISION.
    WORKING-STORAGE SECTION.
   01  INSURANCE   PIC 9(08).  PROCEDURE
  DIVISION.

  IDENTIFICATION  DIVISION.
  METHOD-ID. GET PROPERTY INSURANCE.
  DATA DIVISION.
   LINKAGE SECTION.
   01  OUT-DATA    PIC 9(08).
  PROCEDURE DIVISION RETURNING OUT-DATA.
         MOVE INSURANCE TO OUT-DATA.
         EXIT METHOD.
  END METHOD.

   IDENTIFICATION DIVISION
  METHOD-ID. SET PROPERTY INSURANCE.
  DATA DIVISION.
   LINKAGE SECTION.
   01  IN-DATA     PIC 9(08).
  PROCEDURE DIVISION USING IN-DATA.
        MOVE IN-DATA TO INSURANCE.
         DISPLAY "Group insurance premium was changed."

         EXIT METHOD.
  END METHOD.
```

In the example shown above, the data name (INSURANCE) having the same name as the property name cannot be specified with the PROPERTY clause. When both setting and referencing methods are required, they must be specified explicitly.

14.8.6 Initialization and Termination Methods

The FJBASE class contains the following as object methods: methods to be invoked immediately after the object instance is generated, and methods to be invoked immediately before the object instance is deleted.

Invoking the NEW method creates the object instance. On the other hand, the object instance is deleted when the COBOL system determines automatically that its useful life has terminated (see "14.2.6 Object Life"). The former is called the INIT method. It is used when initialization, which cannot be executed by the VALUE clause, needs to be processed. The latter is called the _FINALIZE method. It is used in any necessary termination processing when the object instance is deleted.

The user does not need to invoke these methods directly with the INVOKE statement. These methods are invoked by overriding them (see "14.2.5 Invoking Methods") or by describing the process. Though the method of the FJBASE class is invoked even if you do not override it, nothing is processed.

Program definition:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. SAMPLE.
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
   REPOSITORY.
     CLASS I-F-SAMPLE.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
   01 OBJREF USAGE OBJECT REFERENCE I-F-SAMPLE.
  PROCEDURE DIVISION.
      INVOKE I-F-SAMPLE "NEW" RETURNING OBJREF.  *> [1]
      INVOKE OBJREF "XXX".                       *> [2]
      SET OBJREF TO NULL.                        *> [3]
 END PROGRAM SAMPLE.
```

- [1] Displays "OBJECT INSTANCE IS GENERATED".
- [2] Displays "XXX IS INVOKED".
- [3] Displays "OBJECT INSTANCE IS TERMINATED".

Class definition:

```
IDENTIFICATION DIVISION.
  CLASS-ID. I-F-SAMPLE INHERITS FJBASE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
   REPOSITORY.
     CLASS FJBASE.
*>  :
  IDENTIFICATION DIVISION.
  OBJECT.
*>   :
  PROCEDURE DIVISION.
   IDENTIFICATION DIVISION.
   METHOD-ID. INIT OVERRIDE.
   DATA DIVISION.
   PROCEDURE DIVISION.
     DISPLAY "OBJECT INSTANCE IS GENERATED".
   END METHOD INIT.
*>
   IDENTIFICATION DIVISION.
   METHOD-ID. _FINALIZE OVERRIDE.
   DATA DIVISION.
   PROCEDURE DIVISION.
     DISPLAY "OBJECT INSTANCE IS TERMINATED".
   END METHOD _FINALIZE.
*>
   IDENTIFICATION DIVISION.
   METHOD-ID. XXX.
   DATA DIVISION.
   PROCEDURE DIVISION.
     DISPLAY "XXX IS INVOKED".
   END METHOD XXX.
  END OBJECT.
 END CLASS I-F-SAMPLE.
```

Note

You can compose an application as usual by recompiling part of the source unless you use _FINALIZE.

When using _FINALIZE, do not use the STOP RUN statement in the _FINALIZE method. Also, do not use the STOP RUN statement in the programs and methods to be invoked from _FINALIZE.

14.8.7 Indirect Reference Classes

When the return value of a method invoked is an object reference data item, a class not appearing in the source, that is, a class referenced implicitly may be required. The implicitly referenced class is called an indirect reference class.

This section explains how to use the indirect reference class, using an in-line invocation nest as an example in which the class often appears.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. P.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
     CLASS A
     CLASS B.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 OBJ-A USAGE OBJECT REFERENCE A.
  01 RET   PIC X(4).
 PROCEDURE DIVISION.
*>      :
     MOVE OBJ-A::"M-A"::"M-B" TO RET. *> ->[1]->[2]
*>      :
```

```
IDENTIFICATION DIVISION.
 CLASS-ID. A.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
     CLASS B.
  OBJECT.
  PROCEDURE DIVISION.
*>
   METHOD-ID. M-A.   *>[1]
   DATA DIVISION.
    LINKAGE SECTION.
    01 OBJ-B USAGE OBJECT REFERENCE B.
   PROCEDURE DIVISION RETURNING OBJ-B.
   END METHOD M-A.
  END OBJECT.
 END CLASS A.
```

```
IDENTIFICATION DIVISION.
 CLASS-ID. B.
  OBJECT.
  PROCEDURE DIVISION.
*>
   METHOD-ID. M-B.   *>[2]
   DATA DIVISION.
    LINKAGE SECTION.
    01 RET PIC X(4).
   PROCEDURE DIVISION RETURNING RET.
   END METHOD M-B.
  END OBJECT.
 END CLASS B.
```

In figure above, the in-line invocation nest described in the program P is disassembled as shown below.

```
  WORKING-STORAGE SECTION.
  01 OBJ-A USAGE OBJECT REFERENCE A.
  01 RET   PIC X(4).
  01 temp  USAGE OBJECT REFERENCE B.    *> A temporary area is created internally.
 PROCEDURE DIVISION.
*>   :
*>   MOVE OBJ-A :: "M-A" :: "M-B" TO RET.
*>   :
     SET  temp TO OBJ-A::"M-A"  *> --+ The nest is expanded using the created
     MOVE temp::"M-B" TO RET.   *> --+ temporary area.
*>   :
```

The internally created temporary area (temp in the diagram shown above) is defined as the object reference data item of class B because it takes the same attribute as the return value of method M-A. That is, class B is referenced from the internally created temporary area (data item defined implicitly). This type of class is referred to as the indirect reference class, which must be declared in the REPOSITORY paragraph in the same manner as for a class referenced explicitly. That is, class B must be declared in the REPOSITORY paragraph of program P.

The definition shown above is explained using the in-line invocation nest as an example. However, the indirect reference class may also need to be declared in the following cases:

- Nested object properties
- When invoking a method in which the object reference data item of another class (having a conformance relationship) is specified as a return value.

When the return value invokes the method of the object reference data item, including the in-line invocation and object property, describe the definition, being conscious of the cases described above.

When compilation is performed without the indirect reference class being declared in the REPOSITORY paragraph, an error message is output. In this case, correct the source in accordance with the message.

14.8.8 Cross Reference Classes

You may wish to link multiple object instances, that is, to define object reference data items in the object data during execution. In such case, the cross-reference relationship may be established either directly or indirectly. The classes having the cross-reference relationship are called as cross-reference classes, which require some techniques for execution format creation.

This section explains some patterns in which the cross reference classes are established, and the operations required for the creating execute form.

14.8.8.1 Cross Reference Patterns

The cross reference patterns are classified into the following three types:

- Cross reference by the self class
- Direct cross reference with the other class
- Indirect cross reference with the other class

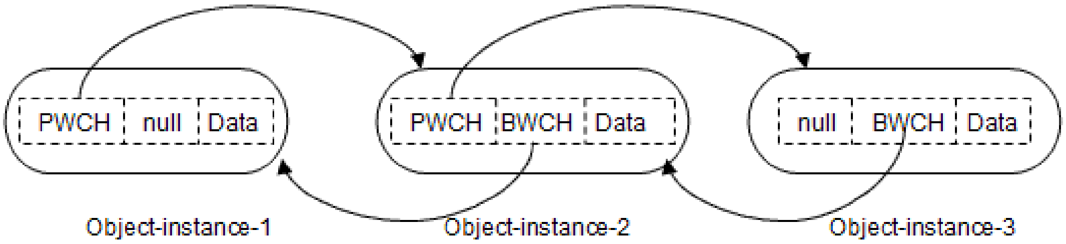
Each pattern is explained specifically below.

Cross reference by the self class

When managing the object instances of the self class in a list structure, declare the object reference data items that contain the self class in the object data.

```
   IDENTIFICATION DIVISION.
   CLASS-ID. A INHERITS FJBASE.
   ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    REPOSITORY.
        CLASS FJBASE.
   OBJECT.
   DATA DIVISION.
    WORKING-STORAGE SECTION.
    01 FWCH     OBJECT REFERENCE A.
    01 BWCH     OBJECT REFERENCE A.
    01 OBJ-DATA PIC X(20).
   PROCEDURE DIVISION.
*>     :
```

Image of execution-time object instances



All of the created object instances can be processed in the forward and backward directions, and referenced with ease by linking them as shown above.

Note

Usually, the classes to be referenced in the class definition must be declared in the REPOSITORY paragraph. However, the self class does not need to be declared; otherwise, a compile-time error occurs.

Direct cross reference with the other class

When an object instance has a close relationship with that of the other class, that is, when it can be followed from another, the direct cross relationship is established. This type of definition is explained below using NAME and address ADDR classes as examples.

Name class

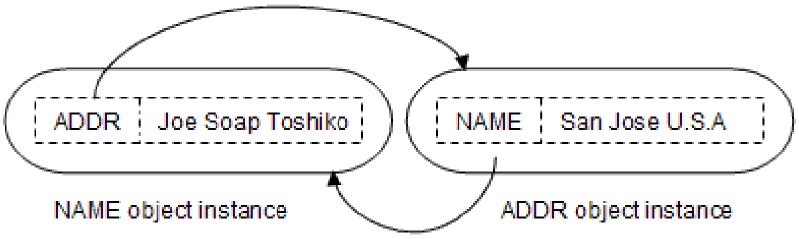
```
IDENTIFICATION DIVISION.
 CLASS-ID. NAME INHERITS FJBASE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
     CLASS FJBASE
     CLASS ADDR.
 OBJECT.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
   01 OBJ-ADDR OBJECT REFERENCE ADDR.
   01 D-NAME   PIC X(20).
  PROCEDURE DIVISION.
*>     :
```

Address class

```
IDENTIFICATION DIVISION.
 CLASS-ID. ADDR INHERITS FJBASE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
     CLASS FJBASE
     CLASS NAME.
 OBJECT.
 DATA DIVISION.
```

```
  WORKING-STORAGE SECTION.
   01 OBJ-NAME OBJECT REFERENCE NAME.
   01 D-ADDR   PIC X(80).
  PROCEDURE DIVISION.
*>     :
```

Image of execution-time object instances



The address can be retrieved from the name, and the name can be retrieved from the address, by linking their object instances mutually as shown above.

Indirect cross reference with the other class

When an object instance has a close relationship with that of the other class, an indirect cross reference relationship may be established. This type of definition is explained below using the following instances as examples:

- Name class (NAME) containing the object instance of the address class
- Address class (ADDR) containing the object instance of the section class
- Section class (SECT) containing the object instance of the name class
-

Name class

```
IDENTIFICATION DIVISION.
 CLASS-ID. NAME INHERITS FJBASE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
     CLASS FJBASE
     CLASS ADDR.
 OBJECT.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
   01 OBJ-ADDR OBJECT REFERENCE ADDR.
   01 D-NAME   PIC X(20).
  PROCEDURE DIVISION.
*>     :
```

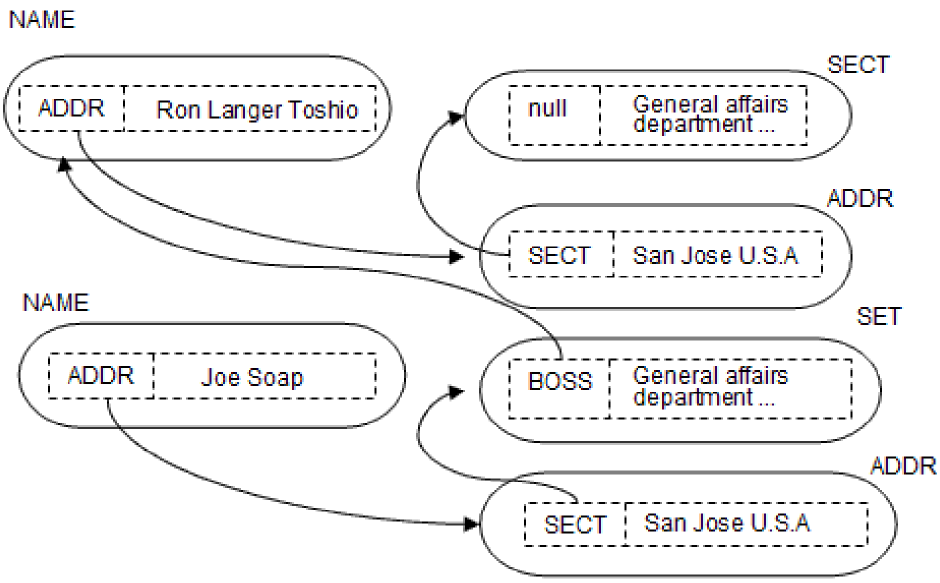
Address class

```
IDENTIFICATION DIVISION.
 CLASS-ID. ADDR INHERITS FJBASE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
     CLASS FJBASE
     CLASS SECT.
 OBJECT.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
   01 OBJ-SECT OBJECT REFERENCE SECT.
   01 D-ADDR   PIC X(80).
  PROCEDURE DIVISION.
*>     :
```

Section class

```
IDENTIFICATION DIVISION.
 CLASS-ID. SECT INHERITS FJBASE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
     CLASS FJBASE
     CLASS NAME.
 OBJECT.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
   01 OBJ-BOSS OBJECT REFERENCE NAME.
   01 D-SECT   PIC X(40).
  PROCEDURE DIVISION.
*>     :
```

Image of execution-time object instances



When object instances are linked in a complicated manner as shown above, an indirect cross reference relationship is established with ease.

14.8.8.2 Compiling Cross Reference Classes

As described above, the cross reference classes are roughly classified into three types. For the cross reference of the self class, however, no special considerations are required during compilation and linkage. When creating the execute form, perform compilation and linkage in the same manner as for the ordinary classes. For the cross reference with the other class, however, the necessary repository file must be prepared before compilation because it is not prepared during compilation.

An example of direct cross reference (NAME and ADDR classes) is explained below.

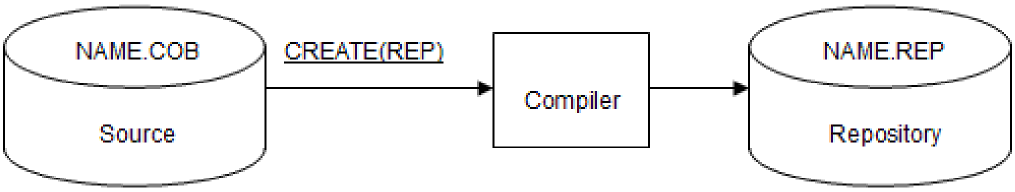
The repository file of the address class is required to compile the name class. The address class must be compiled to create the repository file of that class. At this point, however, the repository file of the name class is required. So, it is thrown into the state of dead lock.

Compiler option CREATE (REP) has been prepared to ensure that the problem described above can be avoided. When CREATE (REP) is specified during compilation, the compiler creates only the repository file. The name and address classes are compiled below using this option.

Step 1: Creating the repository file of the name class

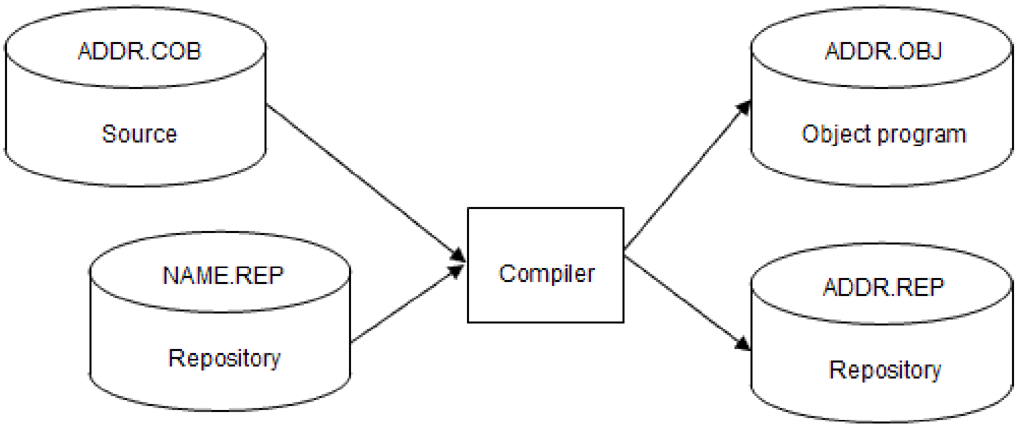
Specify the compiler option CREATE (REP) and compile the name class.

The purpose of this step is to create the repository file, so the repository file of the class (ADDR) referenced does not need to be input. However, the repository file of the parent class is required (input of FJBASE is omitted in the diagram below). Also, input any existing libraries.



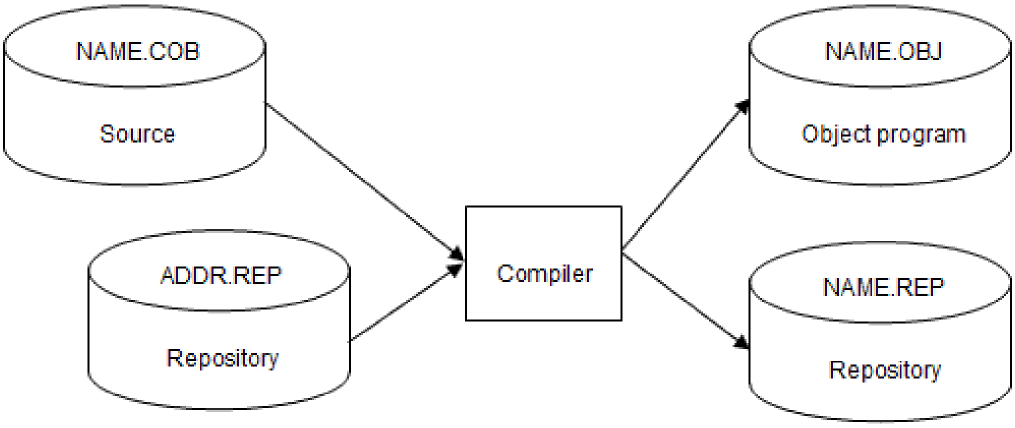
Step 2: Compiling the address class using the created repository

Create the object program of the address class using the repository (NAME.REP) of the name class created in Step 1. In this case, validate compiler option CREATE (OBJ) (default value).



Step 3: Compiling the name class using the repository of the address class

Create the object program of the name class using the repository (ADDR.REP) of the address class created in Step 2. Validate the compiler option CREATE (OBJ) as was done in Step 2.



When creating the object programs, create the repository of a certain class and perform the usual compilation consecutively as described above. This is true for the indirect cross reference; that is, create the repository of a certain class and create the object programs in turn.

The repository file, created by specifying the compiler option CREATE (REP), is called a temporary repository. It is a temporary file used until the regular repository is created. It can be used only to implement a cross reference class. Note that the following restrictions are placed on the temporary repository:

- The temporary repository cannot be used as the repository file of a class declared by the PROTOTYPE clause in the separated method.

Note

When the CREATE (REP) option is specified, the compiler does not analyze the procedure division. Therefore, even if the procedure division contains an error, no message is output. This error is detected later during creation of the object program. Modify the procedure division as required at that time.

14.8.8.3 Linking Cross Reference Classes

When creating the execute form in a static link or dynamic program structure, perform linkage in the same manner as that for the ordinary class definitions. However, when creating the program object in a dynamic linkage structure and making a cross reference with the other class, special considerations are required. (If classes in the cross reference relationship are composed by the same DLL, no special considerations are required. When they are composed by each DLL, special considerations are required.)

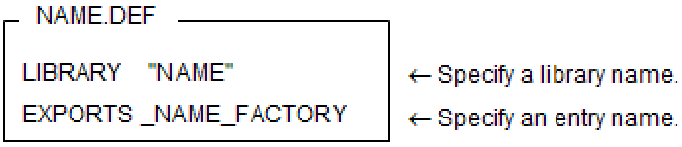
The following explains an example of the direct cross reference (name and address classes).

The import library of the address class is required to link the name class. To create the import library of the address class, the address class must be linked. In this case, however, the import library of the name class is required. That is thrown into the state of dead lock as in the case of compile time.

To avoid the problem described above, create the import library of a certain class. Even though the import library can also be created during linkage, create it from a module definition file.

Step 1: Creating the module definition file of the name class

Create the module definition file of the name class (NAME.DEF) using the editor. The module definition file has the format given below.



```
NAME.DEF

LIBRARY    "NAME"               ← Specify a library name.
EXPORTS _NAME_FACTORY           ← Specify an entry name.
```

Specify the library name for LIBRARY and the entry name for EXPORTS.

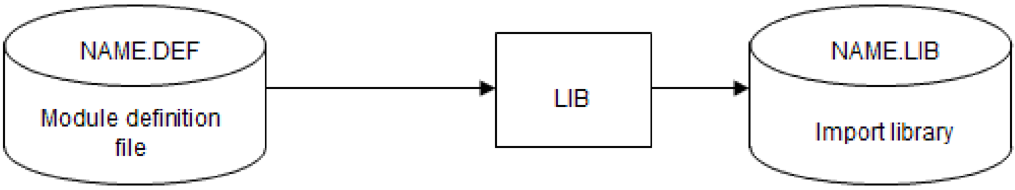
The entry name has the following rules:

- Enter a underscore in the first one byte (fixed character).
- Then, enter a class name (variable character).
- Then, enter a underscore (fixed character).
- Lastly, enter FACTORY (fixed character).

Step 2: Creating the import library of the name class

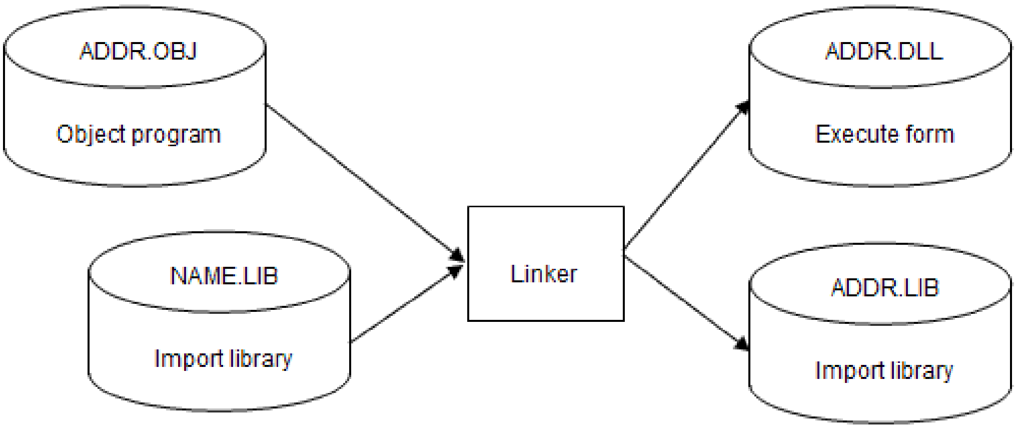
Create the import library of the name class using the module definition file created in Step 1.

LIB /OUT:NAME.LIB /DEF:NAME.DEF /MACHINE:x64



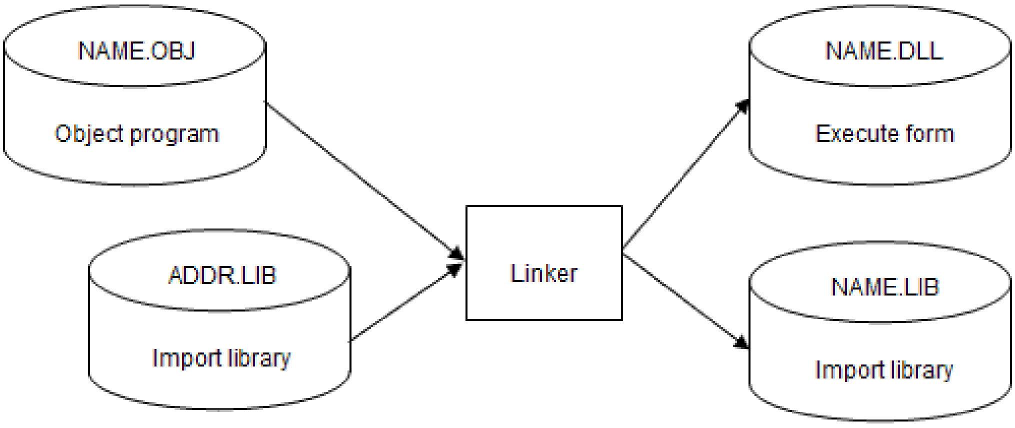
Step 3: Creating the execute form of the address class

Create the execution form (DLL) of the address class using the import library of the name class created in Step 2. At this point, the import library of the address class is created.



Step 4: Creating the execution form of the name class

Create the execution form of the name class using the import library of the address class created in Step 3. At this point, override the import library of the name class.



The creation method described above is the same as that at compile time. Create the import library of a certain class and create the execution forms by linking each library. This is true for the indirect cross reference; that is, create the import library of a certain class and create the execution forms in turn.

Note

Exercise caution at the time of linkage in the following case:

- The factory method of the self class is invoked from a separated method (INVOKE self-class-name "factory method name"),
- And the class definition and separated method are composed by each DLL,

- And a dynamic linkage structure is used.

In the above case, object-oriented programming can be implemented by preparing an import library in advance. It is generally recommended, however, to compose both class definition and separated method by a single DLL.

14.8.8.4 Executing a Cross Reference Class

When the cross reference class is executed, no special considerations are required.

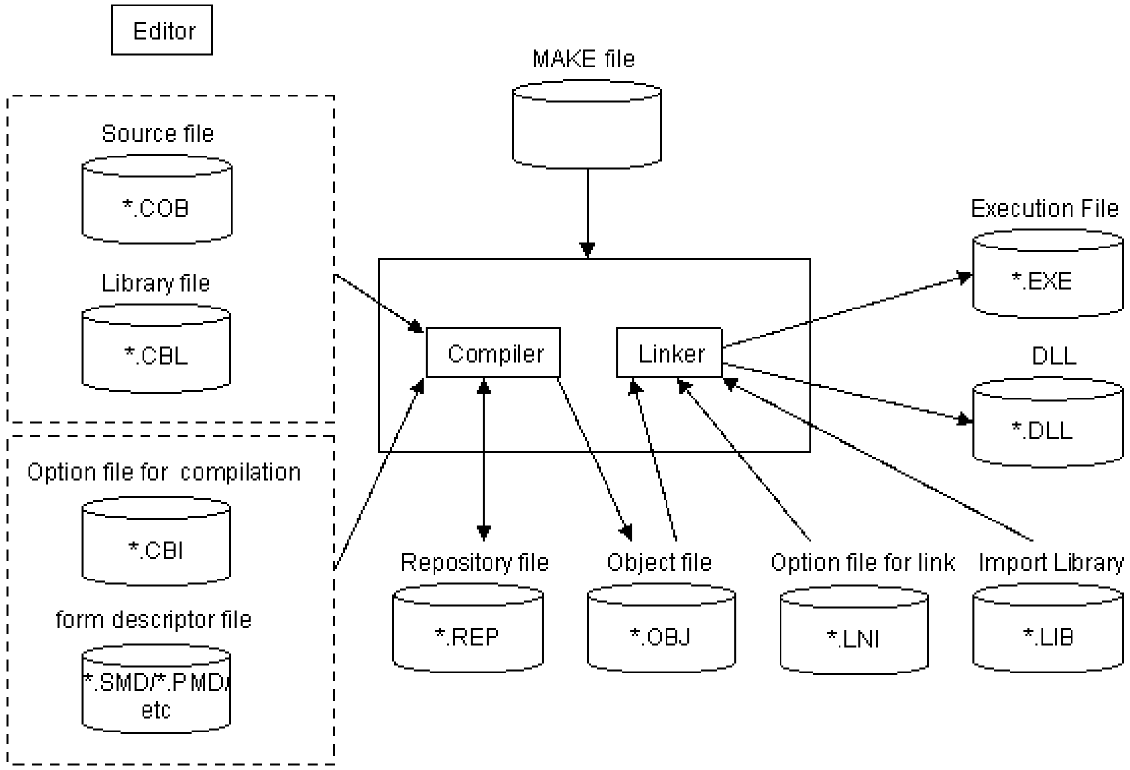
The cross reference class can be executed in the same manner as that for the usual class definitions.

Chapter 15 Developing OO COBOL Applications

This chapter explains how to use the tools supplied with Fujitsu OO COBOL to develop OO COBOL Applications.

15.1 Resources Used by OO COBOL Programming

The diagram in Figure below illustrates the various files that are used and created in OO COBOL.



File contents	File-name format	Input-output	Operation or creation condition
Class information	class-external-name.REP	Input	Used when COBOL source files are compiled or class information databases are created.
		Output	Created when class definition COBOL source files are compiled.

15.2 The OO COBOL Development Process

Perform the following steps to develop Object-Oriented programming:

1. Design the classes. Review existing classes for code that can be reused.
2. Use an editor to create the source files.
3. Compile and link the classes. It is recommend to use the NMAKE command because dependent relationships of the resource may often become complex when compiling and linking the classes. Please refer to the online help of Microsoft Corporation for the NMAKE command.
4. Execute/debug the application.
5. Make the classes available to other developers.

The following section explains items that need special attention in developing OO COBOL programs.

15.3 Designing Classes

When you design new classes, note the following points which are necessary to take full advantage of "Making to components", a feature of object-oriented programs:

- Allowing the features to be general-purpose.
- Allowing the function to be analogized from class-name and method-name.
- Moreover, when class-name and method-name are decided, the following points should be noted:
- Class-name that are used or inherited during the processes of the applications should be unique in the application.
- Method-name that starts with an underscore (_) cannot be used as a method- name created by users.

15.4 Selecting Classes

OO COBOL programming provides the following advantages:

- The definitions can be separated into components with ease.
- Existing components can be appropriated with ease.

No matter how easy it is to create separate components, other people will not be able to use them unless the information covering their use is transmitted. Even if existing components can be appropriated with ease, they cannot be used unless the information covering their use is obtained.

When appropriating existing components in object-oriented programming, the information you have to obtain includes:

- Class names and functions
- Method names and functions of each class
- Interfaces of each method

15.4.1 Class names and functions

You perform programming operations to achieve a purpose.

The reused classes must be those that can be used for this purpose.

Example:

When employee objects are prepared to create the employee management program of a certain company, it is useless to appropriate classes that have no relationship with the purpose (for example, a class in which the health of cows is managed in a farm).Before reusing the classes to create a program, you have to obtain the information containing the functions and names of those classes.

15.4.2 Method names and functions of a class

When creating objects from a class and operating those objects in object-oriented programming, you have to invoke methods in that class.

When you want to reuse existing classes, if you have no methods that execute the processing that suits the purpose, or you do not know the name of any available methods, you cannot use the classes.

Example:

When you create a program that computes an average employee salary using employee objects, assume the following:

If the employee objects include a method in which the user can check employee salaries, but the user does not know the name of that method, the user cannot use it.

As explained above, the user has to obtain the information containing the name and function of the methods defined in the class to be used.

15.4.3 Interfaces of each method

Even if you found the class and method that meet the purpose of the program you want to create, if you do not know the value to be transferred and its format, and the value to be returned and its format, you cannot obtain the desired results.

Example:

Assume that you have an office object in which employee objects in a certain office are collected, and a retrieval object for retrieving each employee object.

When retrieving the object of a certain person from the office object, if you do not know interface information, such as what information to pass to the retrieval method (for example, names and employee numbers) and what information is returned, you cannot use the method.

You must understand the interface of the method to be used.

15.5 Program structures

This section explains the program structure.

15.5.1 Compilation Units and Linkage Units

In OO COBOL programming, the following definitions are considered to be a unit of one compilation:

- Class definition
- Program definition
- Method definition separated by PROTOTYPE declaration

Also, a link unit means the unit for which one executable file or shared object file is created by a link processing. It is also possible that one executable file or shared object file is composed of more than one object file. Therefore, compilation units and link units might not correspond. The figures below show the relationship between compilation unit and link unit: When a compilation unit corresponds to a link unit:

Figure 15.1 When a compilation unit is corresponds to a linkage unit

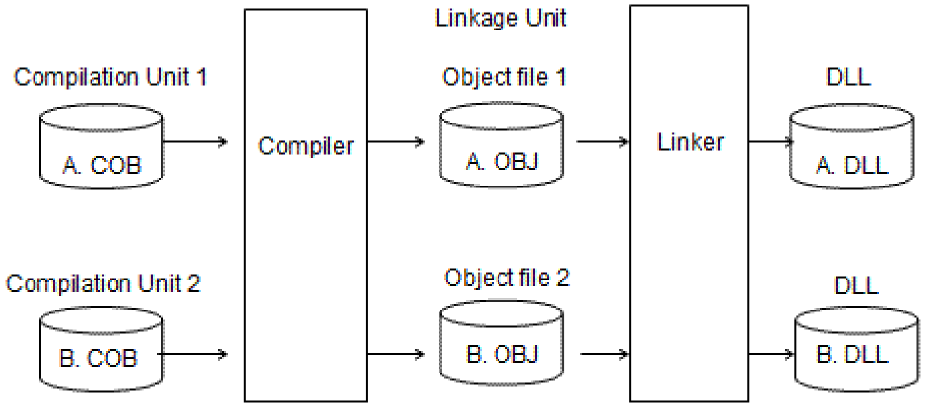
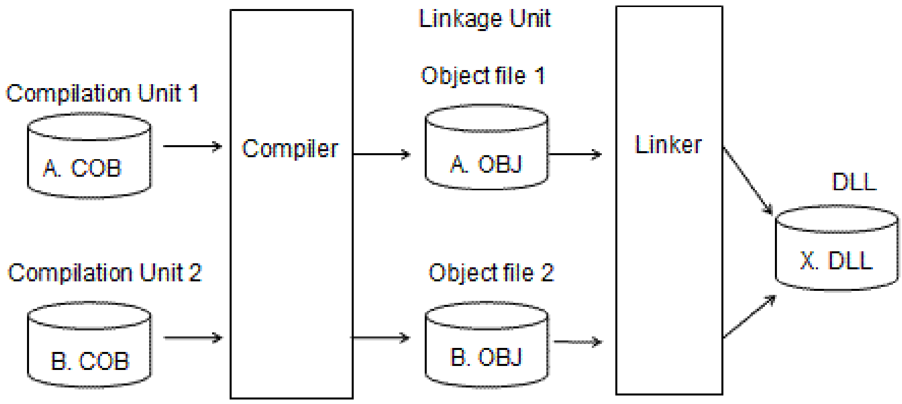


Figure 15.2 When a compilation unit is not correspond to a linkage unit



15.5.2 Overview of Program structure

The following linkages are supported as the linked program structure:

- Static linkage
- Dynamic linkage

The following section explains the case where OO COBOL programs are linked by a static linkage or a dynamic linkage.

Static Linkage

In static link structure, there is a simple structure.

When linking, the object file is needed so that static link structure may solve the link relation in the same library though the import library of linked Websites is not necessary.

Because static link structure is composed of two or more object files, the size of the executable file or DLL grows.

Also, when an object file of a general purpose program or class is created, it should be embedded in all executable files or DLL that use it.

Therefore, a static linkage is used in the following case where the link specified in a few patterns:

- Class definition to declare PROTOTYPE and separated method definition
- Program used only by specific program or class and the caller

Dynamic Link Structure

In dynamic structure, there are dynamic link structure and dynamic program structure.

Here, it explains dynamic link structure. Please refer to "Chapter 4 Linking Programs" for dynamic program structure.

When linking, the import library is needed because dynamic link structure is solved in shape that the link relation extended over two or more libraries (executable file or DLL) though the object file is not necessary.

The size of the executable file or DLL is small and ends because dynamic structure is composed of single or small number of object files.

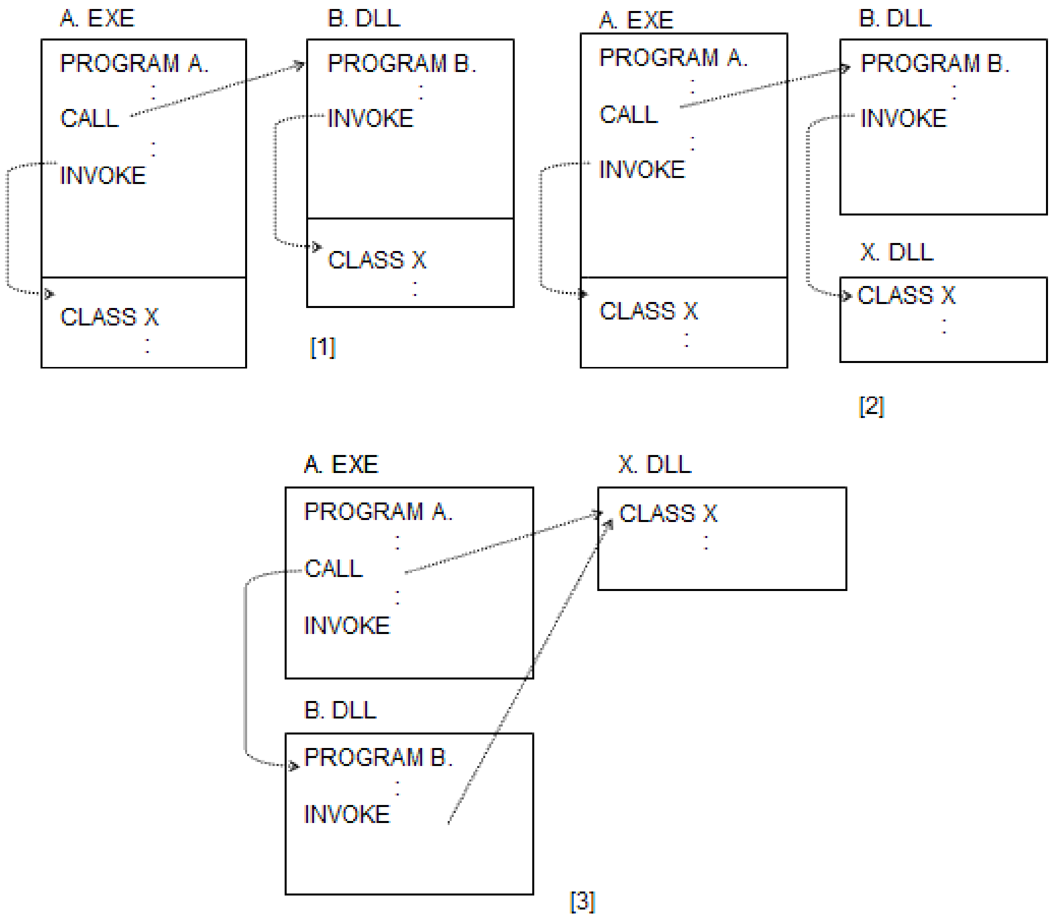
In addition, it ends only because only DLL of the object is linked when the source file is corrected unlike static link structure and it translates again (It doesn't influence other libraries).

Therefore, it is suitable in the following cases because they are general-purpose and link relationship is diversified:

- Class definition and classes that use the class/Program/Method definition.
- High general-purpose program and the caller.

Note

The diagram below shows to scenarios, [1] and [2], in which two copies of Class X executable code would be loaded into the same execution unit. The correct way to structure the application is shown in diagram [3].

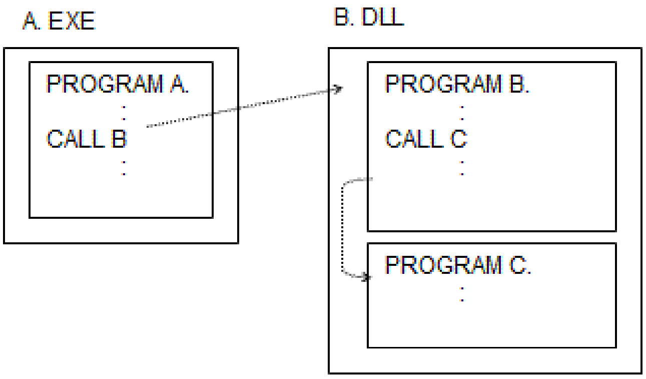


The program structures used in the above diagram are:

- [1] : Class X is statically linked with program A and statically linked with program B.
- [2] : Class X is statically linked with program A, and invoked dynamically from program B.
- [3] : Class X is invoked dynamically from both program A and program B. (Structure could be either dynamic link or dynamic program.)

Information

The static linkage explained here indicates the link relationship between object files of the calling definition and object file of the called definition are decided statically. Therefore, if the object files are included in the same shared object file, the relationship between these two is a static linkage. For instance, in the following case, the relationship between program A and program B is a dynamic linkage. However, the relationship between program B and program C is a static linkage:



15.6 Compile Process

This section explains the following two cases that should be specially considered in the compilation processing to develop object-oriented programs:

Repository files and the compilation procedure

For information about general compilation processing, refer to "Chapter 3 Compiling Programs" and "Chapter 4 Linking Programs".

15.6.1 Repository File

15.6.1.1 Outline

Repository file stores information on classes generated when the class definition is compiled. Repository files are used to notify compilers of information on the classes to be reused when compiling. The file name of repository file is "Class-name (externalized name).REP".

Figure 15.3 Outputting Repository files

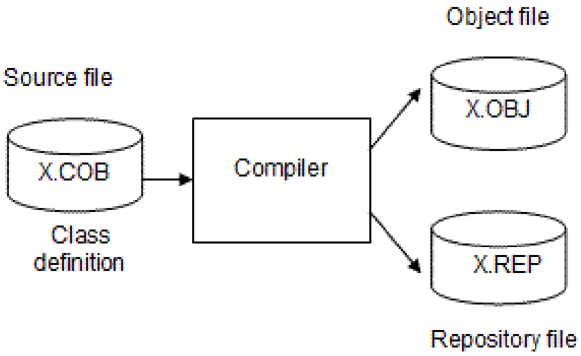
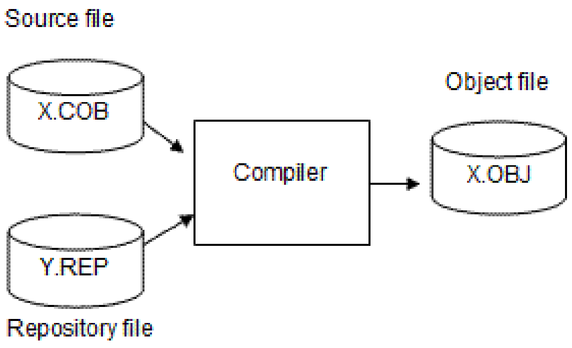


Figure 15.4 Inputting Repository files



Compilers input to only repository files that class-name have been described in the repository paragraph when compiling.

Repository files (class-name) that should be described in the REPOSITORY paragraph are as follows:

- Direct parent classes when inheritance is used
- Classes specified in object reference data items
- A class which includes method prototype definition for separate method definition

Direct Parent Classes when Inheritance is Used

```
[MEMBER.COB]
       :
  CLASS-ID. Member-class INHERITS AllMember-class.
  ENVIRONMENT DIVISION.                 [a]
   CONFIGURATION SECTION.
    REPOSITORY.
      CLASS AllMember-class.
       :          [b]
```

In this program, [a] is a direct parent class-name to be inherited. The repository file "AllMember-class.REP" is needed when compiling because this program inherits the AllMember-class class. Therefore, the class name should be described in the REPOSITORY paragraph. (See [b] in the program) The compiler retrieves corresponding repository files based on this class name.

Class Specified in the Object Reference Variable

```
 [ALLMEM.COB]
       :
  CLASS-ID. AllMember-class INHERITS FJBASE.
       :
  ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
    REPOSITORY.
       :
      CLASS Address-class.
       :                    [b]
    OBJECT.
    DATA DIVISION.
     WORKING-STORAGE SECTION.
       :
     01  waddress      OBJECT REFERENCE Address-class ...
       :                    [a]
```

In this program, [a] is a class-name specified by the object reference variable. The repository file "Address-class.REP" is needed when compiling because this program refers to the Address-class class. The class-name should be described in the REPOSITORY paragraph so that the compiler may retrieve repository files in this case as well. (See [b] in the program)

Class in which a Method is Defined when a Separated Method is Created

```
[SALA_MEM.COB]
  METHOD-ID. Salary-method  OF  Member-class.
  ENVIRONMENT DIVISION.    [a]
   CONFIGURATION SECTION.
    REPOSITORY.
      CLASS Member-class.
       :                   [b]
```

In this program, [a] is a class-name that the method belongs. The repository file "Member-class.REP" is needed when compiling because this program refers to the information on the Member-class class. In this case, to retrieve repository files needed for the compilers, the class names should be described in the REPOSITORY paragraph. (See [b] in the program)

15.6.1.2 Compilation Procedure

When COBOL source file is compiled, the compiler retrieves the repository file (repository file of the class described in the repository paragraph) that should be referred. At this time, if a necessary repository file doesn't exist, it becomes a compilation error.

Moreover, when the class definition is re-compiled, the repository file is updated. In that case, it is necessary to compile the source file input when the updated repository file is re-compiled.

Thus, the restriction is caused by the relation of the input of the repository file in the compilational order.

```
[MEMBER.COB]
       :
  CLASS-ID. Member-class INHERITS AllMember-class.
       :
    REPOSITORY.
      CLASS AllMember-class.
       :
```

It is understood that the repository file of AllMember-class class is necessary for compilation.

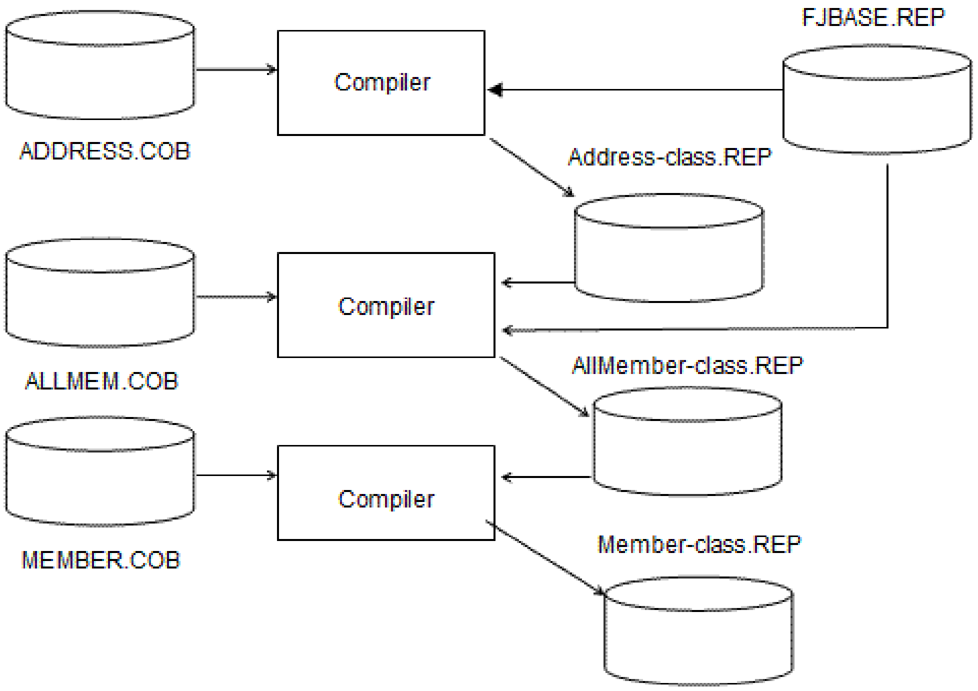
```
[ALLMEM.COB]
       :
  CLASS-ID. AllMember-class INHERITS FJBASE.
       :
    REPOSITORY.
      CLASS FJBASE
      CLASS Address-class.
       :
```

It is understood that the repository file of FJBASE class and Address-class class is necessary for compilation.

```
[ADDRESS.COB]
       :
  CLASS-ID. Address-class INHERITS FJBASE.
       :
    REPOSITORY.
      CLASS FJBASE.
       :
```

It is understood that the repository file of FJBASE class is necessary for compilation.

Then the compilation order and repository flow is as shown in the figure below:



The compilation procedure should be executed in the following order as understood from above Figure:

1. ADDRESS.COB
2. ALLMEM.COB
3. MEMBER.COB

15.6.1.3 Target Repository File

The repository file created by compiling a source program is referred to as the target repository file. The target repository file of "MEMBER.COB" is "Member-class.REP"

15.6.1.3.1 Target Repository File Folder

The folder in which target repository fi compiled is decided by specifying the compilation command option -dr as shown in the table below.

-dr option	Filename
Valid	Folder specified by the -dr operand
Invalid	Folder containing the COBOL source file

15.6.1.4 Dependent Repository File

A repository file needed when the source file is compiled is referred to as a dependent repository file. The dependent repository file of "MEMBER.COB" is "Allmenber-class.REP".

Dependent Repository File Search Folders

When the compiler searches a repository, it uses the folder name specified in the following table:

-R option	dr option	Order of search Folders
Valid	Invalid	1. Folder specified by the -R option 2. Folder specified by the -dr option

-R option	dr option	Order of search Folders
		3. Current path
	Invalid	1. Folder specified by the -R option 2. Current path
Invalid	Valid	1. Folder specified by the -dr option 2. Current path
	Invalid	1. Current path

Example

Example of Compiling by cobol Commands

The following example shows the compilation of ALLMEM.COB by using the cobol command. For information about the input format of the COBOL command, refer to "3.5.1 COBOL Command".

```
COBOL -c -R C:\COBOL\SAMPLES\SAMPLE19 -dr C:\COBOL\SAMPLES\SAMPLE19
```

- Input
    - ALLMEM.COB (Source file)
    - Address-class.REP (Repository file: C:\COBOL\SAMPLES\SAMPLE19)
    - Membermaster-class.REP (Repository file: C:\COBOL\SAMPLES\SAMPLE19)
- Output
    - allmem.o (Object file)
    - AllMember-class.REP (Repository file: C:\COBOL\SAMPLES\SAMPLE19)
- Option
    - -c (Specification of executing only compilation)
    - -R (Folder of input destination of repository file)
    - -dr (Folder of input and output destination of repository file)

15.6.1.5 Compiling Cross Reference Classes

Techniques are needed to create dependent repository files to compile cross reference classes. Refer to "14.8.8.2 Compiling Cross Reference Classes" for details.

15.7 Linking Process

This section explains linking process for development of OO COBOL programs:

15.7.1 Required Import Libraries

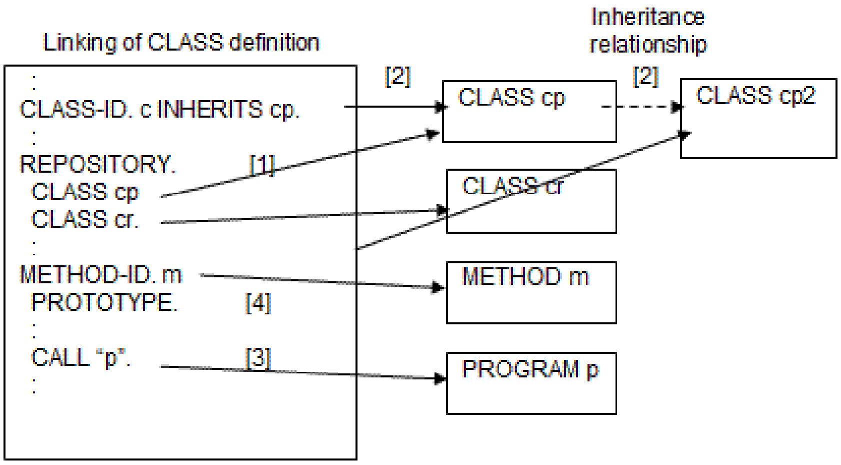
Only programs in the call (CALL statement) relationship needed linkage statically or dynamically in the range of the language specification of COBOL85. However, there are more patterns where the linkages between definitions of class/program/method occur in OO COBOL programming.

Table below shows the cases where linkage is needed in OO COBOL programming:

Definition name		Link to		
		Class definition	Class definition	Class definition
Link from	Class definition	[1] Class described in REPOSITORY paragraph. [2] Parent class	[3] Program called by CALL statement.	[4] Method separated by PROTOTYPE declaration.
	Program definition	[5] Class described in REPOSITORY paragraph.	[6] Program called by CALL statement.	--
	Method definition	[7] Class described in REPOSITORY paragraph (except for classes including PROTOTYPE declaration of its own method). [8] Parent class including PROTOTYPE declaration of its own method. (See Note below)	[9] Program called by CALL statement.	--

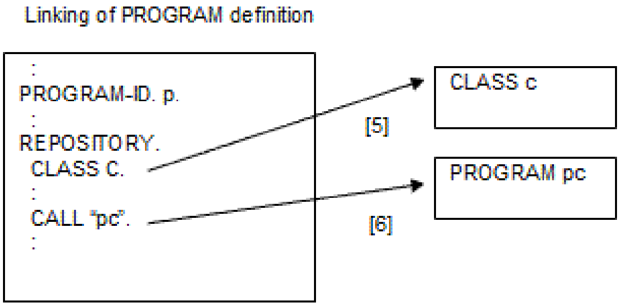
* : A high-ranking class in the inheritance relation where a direct parent class is included is indicated.

The link in each definition is shown below:



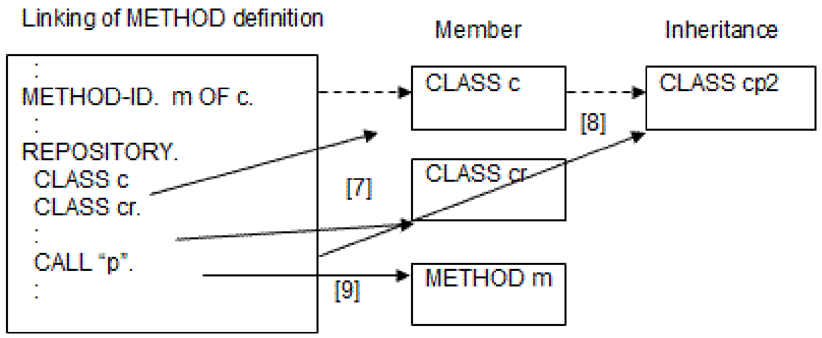
Linking of CLASS definition

```
 :
CLASS-ID. c INHERITS cp.
 :
REPOSITORY.
 CLASS cp
 CLASS cr.
 :
METHOD-ID. m
 PROTOTYPE.
 :
 CALL "p".
 :
```



Linking of PROGRAM definition

```
 :
PROGRAM-ID. p.
 :
REPOSITORY.
 CLASS c.
 :
 CALL "pc".
 :
```



The solid line arrows in the above figure show links.

To solve the link relationship shown by the solid line arrows with dynamic link structures, DLL are needed.

15.7.2 Procedure of Link

When executable files or DLL are created using a linker, DLL to be linked are needed. Therefore, when two or more executable files or DLL are created, the restriction occurs in the link processing order.

The files "MEMBER.COB", "ALLMEM.COB" and "ADDRESS.COB" in the sample program are used as exercise programs.

```
[MEMBER.COB]
     :
CLASS-ID. Member-class INHERITS AllMember-class.
     :
  REPOSITORY.
    CLASS AllMember-class.
     :
```

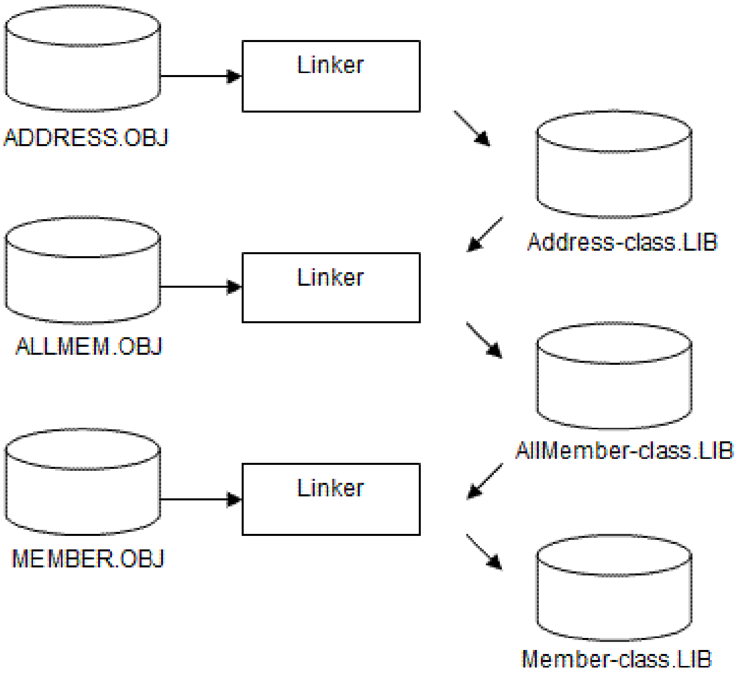
```
[ALLMEM.COB]
     :
CLASS-ID. AllMember-class ...
     :
  REPOSITORY.
     :
    CLASS Address-class.
     :
```

```
[ADDRESS.COB]
     :
CLASS-ID. Address-class ...
     :
```

When separately compiled program and the unit of the link are made to the same, a necessary import library and the generated import library are shown in the table below because of each link processing.

Source file / Object file	Necessary import library	Generated import library
MEMBER.COB / MEMBER.OBJ	Member-class.LIB	AllMember-class.LIB
ALLMEM.COB / ALLMEM.OBJ	AllMember-class.LIB	Address-class.LIB
ADDRESS.COB / ADDRESS.OBJ	Address-class.LIB	------

This relationship is shown in Figure below.



Information

Shared object files of the FJBASE class are automatically linked without users' specification.

The link procedure should be executed in the following order as understood from above Figure:

1. ADDRESS.OBJ
2. ALLMEM.OBJ
3. MEMBER.OBJ

15.8 Disclosing Classes

When development and testing are completed for a class, you generally want to make that class available to other developers. This process if called "class disclosure".

The diagram below illustrates the disclosure process.

Figure 15.5 below explains the resources you need to disclose to share classes with other developers.

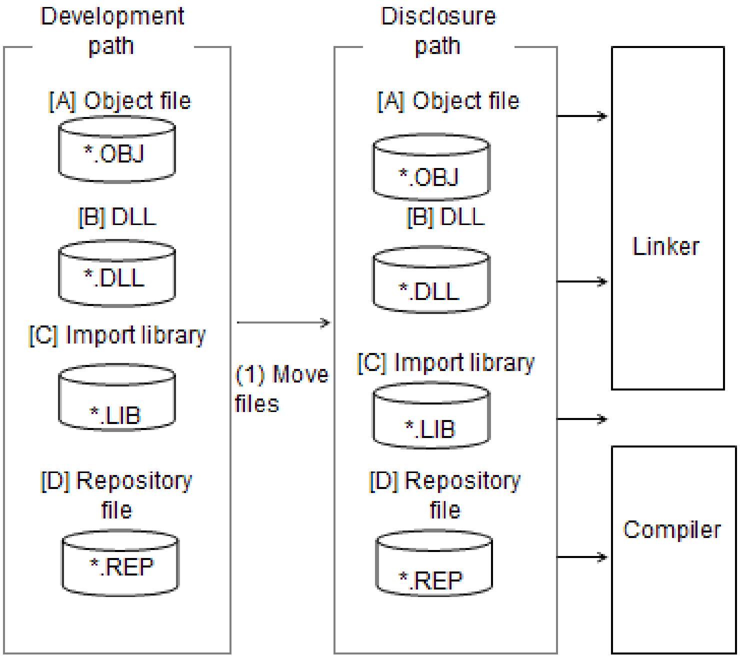


Table 15.1 Resources to Disclose

	Resource name	Use
[A]	Object file	Required when classes or programs are to be linked in a static structure.
[B]	DLL	Required when classes or programs are to be linked with either a dynamic linkage structure or dynamic program structure.
[C]	Import library	Required when classes or programs are to be linked with a dynamic linkage structure or dynamic program structure.
[D]	Repository	Required if the classes or programs are to be compiled. (*)
-	Document	Required to inform a function of the class, an interface (method name, parameter, and property name, etc.), and necessary resources (repository file name and DLL name, etc.) to the user of the class.

(*) If classes or programs are still being developed the system is not ready for full disclosure as sources must also be made available for update. Consider using a staged disclosure with limited availability of classes to a particular development group.

The disclosure process usually involves two steps:

Move the files listed above from the development folders (paths) to the disclosure folders (paths). The development paths are only accessible to a particular development team. The disclosure paths are accessible to many development teams.

Populate the class information database with the newly disclosed class(es) so that others can view information about the class(es) using the Class Browser. (See Creating the Class Information Database above).

Note

It is necessary to describe the following content in the document when the class is disclosed :

- DLL name
- Class name succeeded to including class and the outline of functions
- All method names and the outlines of functions

- All property names and the outlines of functions
- Interface of method or property (parameter and resetting value)
- Repository file name
- Other notes

15.9 MAKE file

The file that describes the rule of arranging and the execution of the command to the management of two or more resources and those resources is called a MAKE file.

15.9.1 Usage

The MAKE file has the following advantages.

- The MAKE file is a text file. Therefore, it is possible to change in the content with a text editor.
- As for the NMAKE command, the construction processing under batch environment becomes possible because it is operable as the batch.

The user becomes from the above-mentioned advantage and the following.. work becomes possible by using the MAKE file.

- The execution of other commands becomes possible by writing it directly in the MAKE file.
- It is opened from an interactive operation by operating under batch environment.

15.9.2 Example

```
       :
COBOL_PATH=C:\COBOL                                          ...[1]
       :
SAMP1.OBJ : SAMP1.COB SAMP2.REP                              ...[2]
     $(COBOL_PATH)\COBOL -M SAMP1.COB                        ...[3]
       :
SAMP1.EXE : SAMP1.OBJ ... SAMP2.LIB                          ...[4]
     $( COBOL_PATH)\LINK ...                                 ...[5]
     /OUT:SAMP1.EXE
     SAMP1.OBJ  SAMP2.LIB F4AGCIMP.LIB LIBCMT.LIB
       :
```

[1] Folder in which compiler is installed

[2] Dependence for compilation

[3] Compilation command

[4] Dependence for link

[5] Link command

Chapter 16 Advanced Features of OO COBOL

This chapter provides details of how to use some of the more advanced features of OO COBOL. These are:

- Defining exception processes
- Using dynamic program structure
- Tuning memory use
- Combining OO COBOL with Visual C++
- Making objects persistent
- Special classes
- Programming using the ANY LENGTH clause

16.1 Defining Exception Processes

16.1.1 Overview

OO COBOL extends the standard COBOL exception process handling provided in the USE statement of the DECLARATIVES section.

In the DECLARATIVES section, you can code a "USE AFTER EXCEPTION class-name" statement.

You can generate an exception condition, called an exception object, in your procedure code by using the RAISE statement. In the RAISE statement, you specify an object identifier. This object identifier becomes the exception object.

When an exception object is raised, the system searches the appropriate DECLARATIVES sections for a USE procedure defined with the class of the exception object, or a superclass of the exception object. If a matching procedure is found, that procedure is executed.

The intention is that you can define error routines to handle errors that may occur in many places. When an error occurs you "raise" the exception object, which causes the appropriate error routine to be invoked.

There are two ways of raising exception objects: in the RAISE statement, and in the EXIT statement with the RAISING clause. These are explained below.

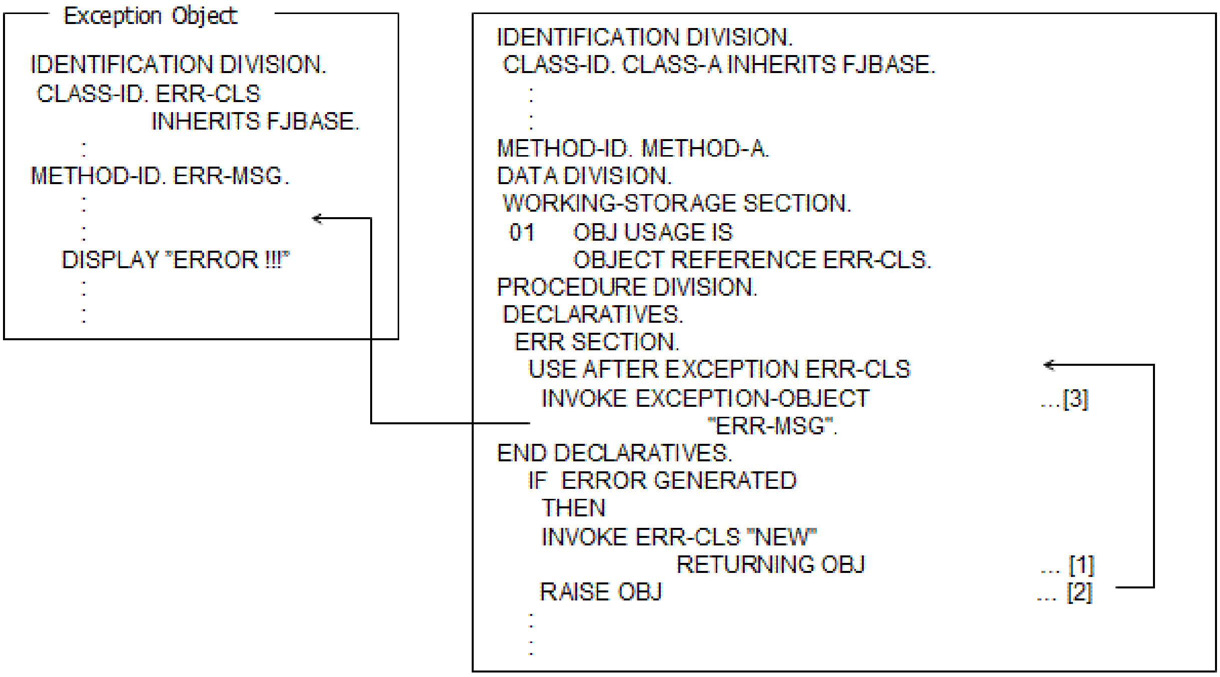
16.1.2 EXCEPTION-OBJECT Predefined Object Reference

OO COBOL provides a predefined object reference to use in the USE AFTER EXCEPTION procedure. It is the EXCEPTION-OBJECT reference and contains the object reference of the exception object that caused the USE procedure to be activated.

16.1.3 The RAISE Statement

When you recognize an error condition in your program and want to invoke the general error handling routine through exception processing, you code the RAISE statement. You provide an object reference in your RAISE statement for an object of your error handling class.

The processing is illustrated in the diagram below:



Exception Object

```
IDENTIFICATION DIVISION.
 CLASS-ID. ERR-CLS
                 INHERITS FJBASE.
       :
METHOD-ID. ERR-MSG.
       :
       :
    DISPLAY "ERROR !!!"
       :
       :
```

```
IDENTIFICATION DIVISION.
 CLASS-ID. CLASS-A INHERITS FJBASE.
     :
     :
METHOD-ID. METHOD-A.
DATA DIVISION.
 WORKING-STORAGE SECTION.
  01    OBJ USAGE IS
        OBJECT REFERENCE ERR-CLS.
PROCEDURE DIVISION.
 DECLARATIVES.
  ERR SECTION.
   USE AFTER EXCEPTION ERR-CLS
    INVOKE EXCEPTION-OBJECT                        ...[3]
                    "ERR-MSG".
END DECLARATIVES.
   IF  ERROR GENERATED
    THEN
    INVOKE ERR-CLS "NEW"
                RETURNING OBJ                      ... [1]
    RAISE OBJ                                      ... [2]
     :
     :
```

The steps in the above illustration are:

- [1] The error condition is recognized and a new instance of the error handling class (ERR-CLS) is created. (The instance could be created earlier in the code).
- [2] The object reference for the instance is use in the RAISE statement. This makes the object the exception object.
- [3] Because the exception object is of the class ERR-CLS, the declaratives code USE AFTER EXCEPTION ERR-CLS is executed. It contains a line that invokes the appropriate method in the exception object.

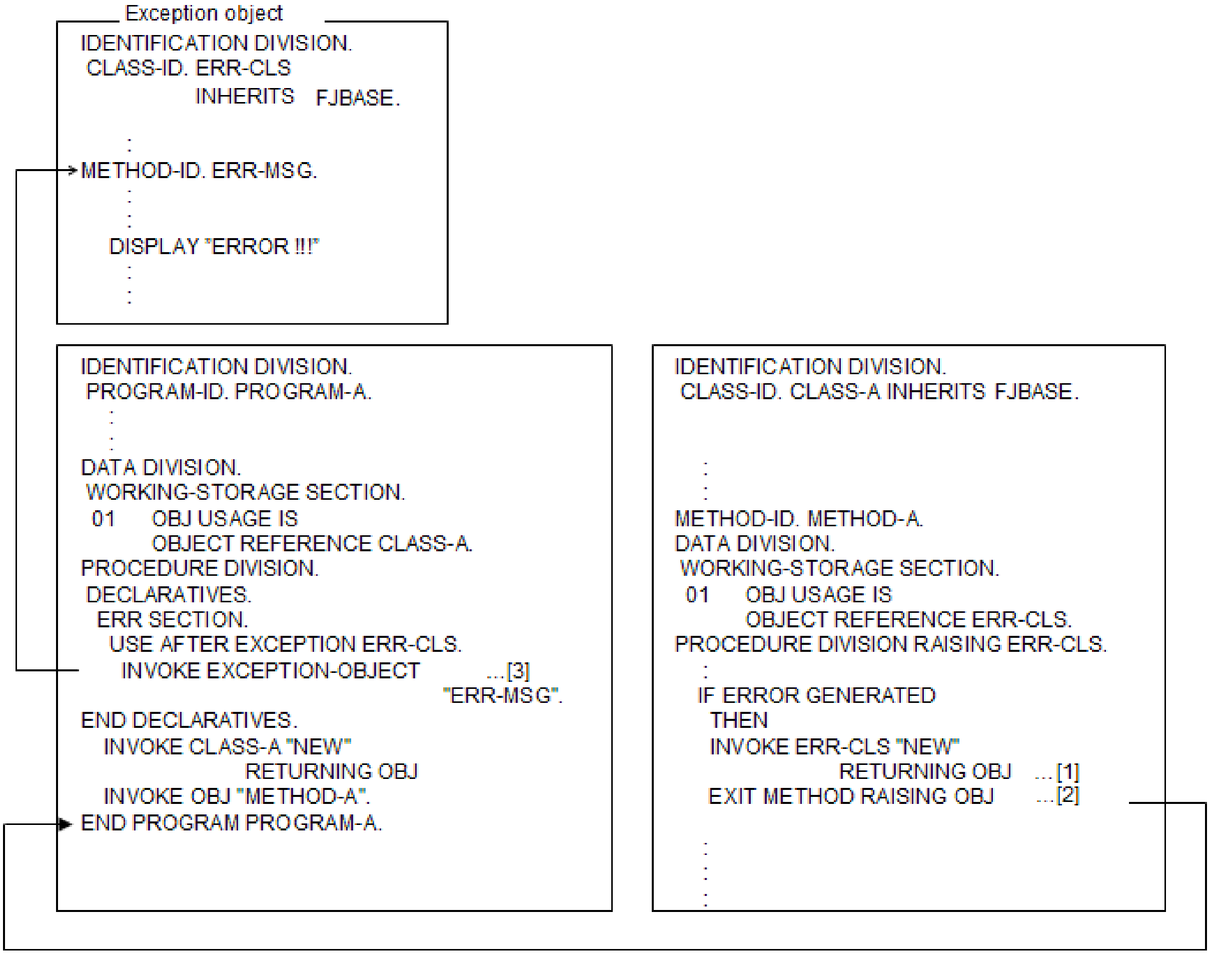
No Exception Procedure or NULL Object

If the RAISE statement is executed and there is no exception procedure, or the exception object is a NULL object, then control moves to the statement immediately after the RAISE statement.

16.1.4 The EXIT Statement with RAISING Clause

Another way to raise the exception object is to use the EXIT statement with a RAISING clause. This has the effect of exiting the program or method, returning to the invoking code, then generating the exception condition. The system looks for the exception procedure in the invoking code. As with the RAISE statement you provide an object reference to an object of your error handling class.

This is illustrated in the diagram below:



```
IDENTIFICATION DIVISION.
 CLASS-ID. ERR-CLS
           INHERITS  FJBASE.

     :
METHOD-ID. ERR-MSG.
     :
     :
   DISPLAY "ERROR !!!"
     :
     :
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. PROGRAM-A.
   :
   :
DATA DIVISION.
 WORKING-STORAGE SECTION.
  01     OBJ USAGE IS
         OBJECT REFERENCE CLASS-A.
PROCEDURE DIVISION.
 DECLARATIVES.
  ERR SECTION.
   USE AFTER EXCEPTION ERR-CLS.
    INVOKE EXCEPTION-OBJECT        ...[3]
                              "ERR-MSG".
END DECLARATIVES.
   INVOKE CLASS-A "NEW"
                RETURNING OBJ
   INVOKE OBJ "METHOD-A".
END PROGRAM PROGRAM-A.
```

```
IDENTIFICATION DIVISION.
 CLASS-ID. CLASS-A INHERITS FJBASE.


   :
   :
METHOD-ID. METHOD-A.
DATA DIVISION.
 WORKING-STORAGE SECTION.
  01     OBJ USAGE IS
         OBJECT REFERENCE ERR-CLS.
PROCEDURE DIVISION RAISING ERR-CLS.
   :
  IF ERROR GENERATED
   THEN
   INVOKE ERR-CLS "NEW"
                RETURNING OBJ    ...[1]
   EXIT METHOD RAISING OBJ       ...[2]

   :
   :
   :
```

The steps in the above diagram are:

- [1] The error condition is recognized and a new instance of the error handling class (ERR-CLS) is created. (The instance could be created earlier in the code).
- [2] The object reference for the instance is use in the EXIT METHOD RAISING statement. This makes the object the exception object. Control is first returned to the invoking program before the exception is generated.
- [3] Because the exception object is of the class ERR-CLS, the declaratives code USE AFTER EXCEPTION ERR-CLS is executed. It contains a line that invokes the appropriate method in the exception object.

Note

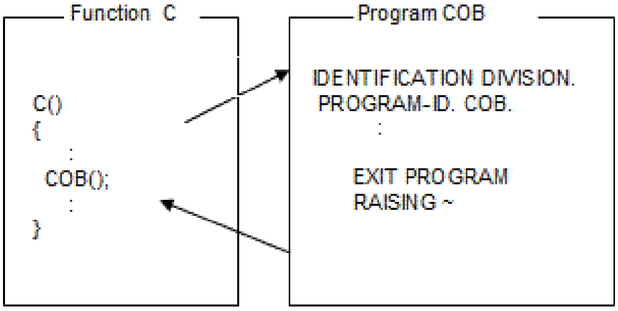
EXIT ... RAISING cannot be used from a main program because there is no invoking program in which an exception condition can be generated.

Other-Language Programs

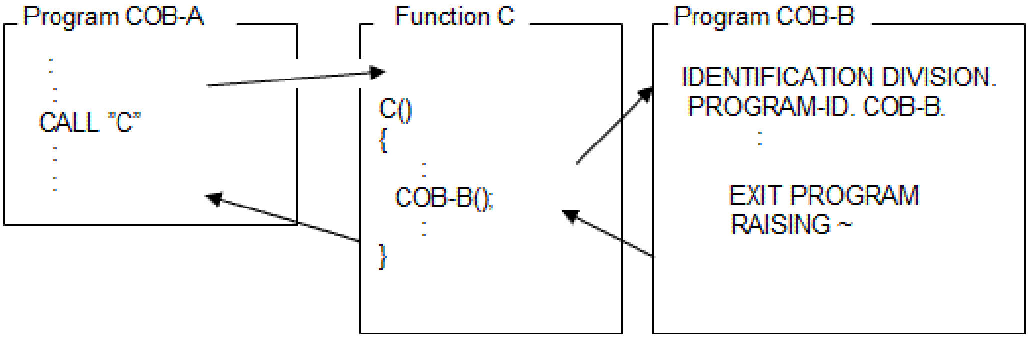
If a COBOL program is invoked from a program written in another language then an exception condition cannot be generated by the EXIT ... RAISING statement - unless that other-language program was itself invoked by a COBOL program.

For example, suppose a C function calls a COBOL program, then:

1. This code does not produce an exception condition:



2. Whereas this code does produce an exception condition when control returns to COBOL program A:



No Exception Procedure or NULL Object

If the EXIT statement with RAISING clause is executed and there is no exception procedure, or the exception object is a NULL object, then the program or method ends abnormally.

16.1.5 Multiple USE Class-name Statements

If the declaratives sections contain multiple USE statements for the exception object then the first matching procedure is executed.

For example, suppose that declaratives contain the following code:

```
DECLARATIVES.
  ERR-1 SECTION.
    USE AFTER EXCEPTION CLASS-C.
    DISPLAY "ERR CLASS-C".              *>[1]
  ERR-2 SECTION.
    USE AFTER EXCEPTION CLASS-B.
    DISPLAY "ERR CLASS-B".
  ERR-3 SECTION.
    USE AFTER EXCEPTION CLASS-A.
    DISPLAY "ERR CLASS-A".
END DECLARATIVES.
```

Also suppose that:

- CLASS-A inherits CLASS-B and CLASS-C.
- An object of CLASS-A is specified as the exception object.

Then the exception procedure labeled [1] is executed because CLASS-C is a superclass of CLASS-A and that is the first procedure that matches.

16.2 Using Dynamic Program Structure

Dynamic program structure is explained in "9.1.3 Dynamic Program Structure". This section discusses the details of dynamic program structure as it relates to OO COBOL programs. Specifically it covers the following points:

Brief review of the dynamic program structure

- Dynamic program structure loading mechanism
- Compiling for dynamic program structure
- Linking considerations
- DLL file naming conventions
- Entry information

You should also check the "9.1.3.3 Cautions" for a list of points you need to be aware of when using a dynamic program structure.

16.2.1 Review of Dynamic Program Structure

Dynamic program structure is a Fujitsu term used to describe an application structure in which DLL files are loaded under the control of the COBOL runtime system rather than by the operating system.

Because the DLL files are going to be loaded by the COBOL runtime system, you either have to follow a standard naming convention for the DLL files or specify program entry information at the start-up of the runtime system. If you don't take either of these actions, the runtime system will not find the DLL files. Both actions are explained in topics below.

Note on mixing program structures

Although you can mix dynamic link structure with dynamic program structure, you may cause unpredictable or unexpected behavior. For example, using CANCEL may produce different effects in canceling subprograms and closing files when a mix of structures is used rather than a dynamic program structure throughout.

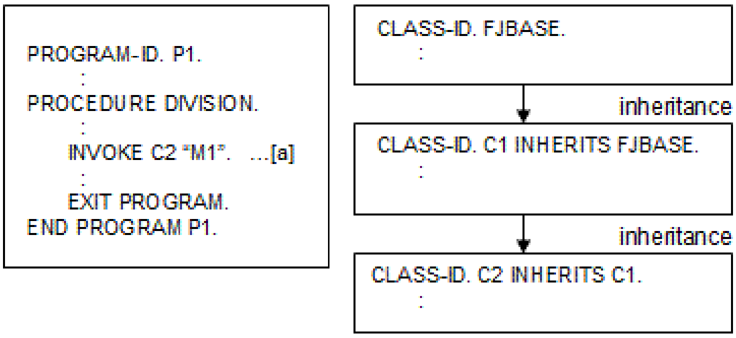
This section therefore assumes a uniform structure throughout the application.

16.2.2 Dynamic Program Structure Loading Mechanism

It may help your understanding of dynamic program structure to consider the processes that occur when classes and methods are loaded. We shall do this by considering what classes are loaded when a method is invoked and how a separately compiled method is loaded.

Class Loading

Consider the situation in which a program (P1) invokes a method (M1) contained in a class (C2). That class (C2) inherits from a class (C1) that in turn inherits from the FJBASE class:



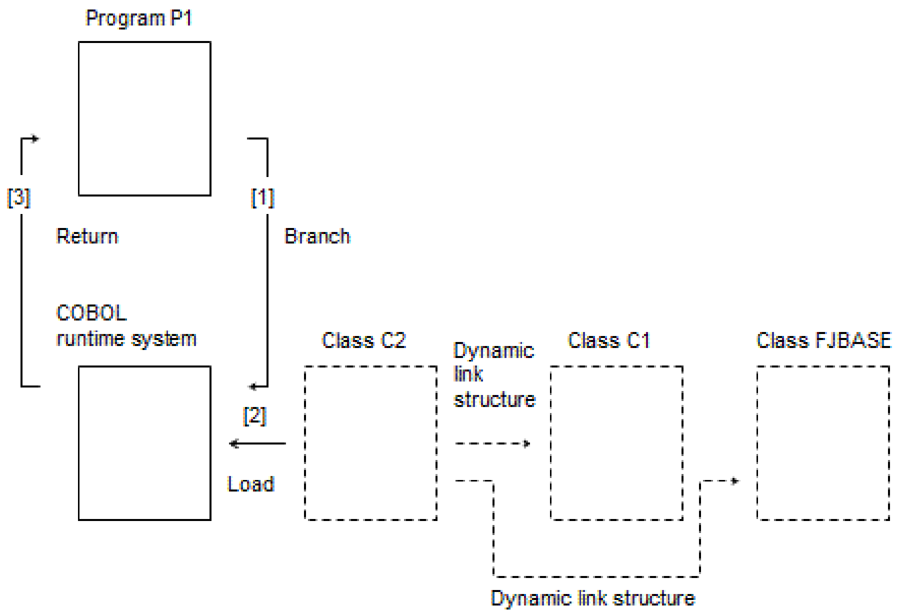
```
PROGRAM-ID. P1.
    :
PROCEDURE DIVISION.
    :
    INVOKE C2 "M1".   ...[a]
    :
    EXIT PROGRAM.
END PROGRAM P1.
```

```
CLASS-ID. FJBASE.
    :
```

```
CLASS-ID. C1 INHERITS FJBASE.
    :
```

```
CLASS-ID. C2 INHERITS C1.
    :
```

Assuming all these program are compiled with the DLOAD option, the load processes are shown in the diagram below:

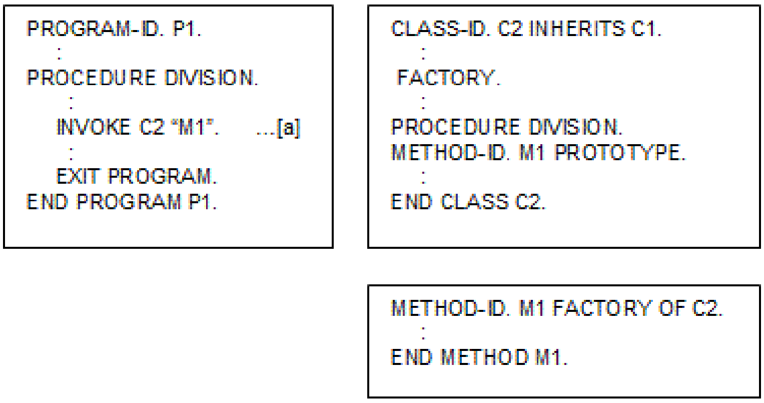


The load steps are:

- [1] Because class C2 is referenced in the INVOKE statement the COBOL runtime system is called to load class C2.
- [2] The COBOL runtime system loads class C2. Inherited classes are always loaded using the dynamic link structure so classes C1 and FJBASE are loaded at the same time as C2.
- [3] The COBOL runtime system returns control to program P1 so that P1 can invoke the method.

16.2.3 Method Loading

In the same example described above let us assume that method M1 is specified as a prototype within class C2, and compiled and linked separately in a DLL:

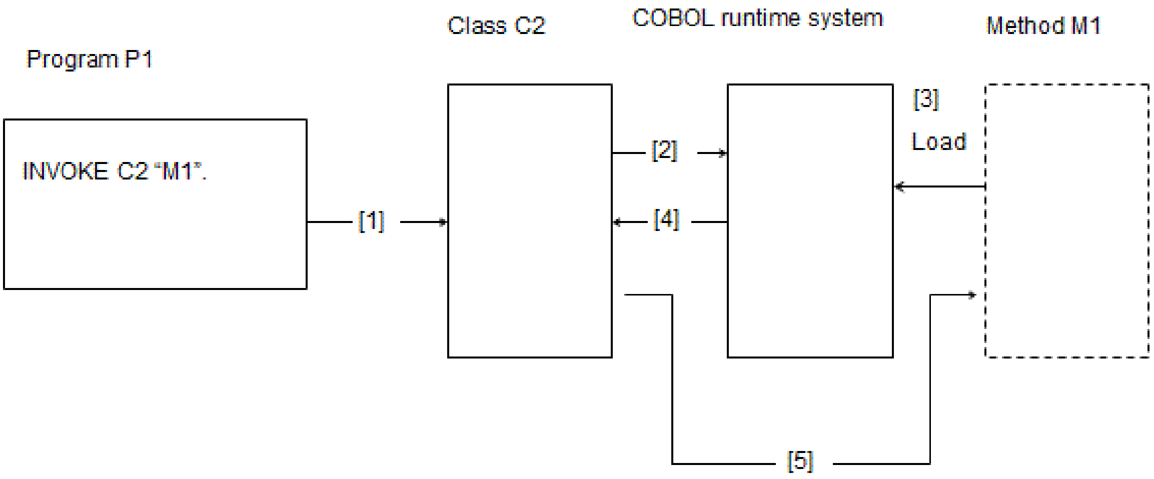


```
PROGRAM-ID. P1.
    :
PROCEDURE DIVISION.
    :
    INVOKE C2 "M1".      ...[a]
    :
    EXIT PROGRAM.
END PROGRAM P1.
```

```
CLASS-ID. C2 INHERITS C1.
    :
 FACTORY.
    :
PROCEDURE DIVISION.
METHOD-ID. M1 PROTOTYPE.
    :
END CLASS C2.
```

```
METHOD-ID. M1 FACTORY OF C2.
    :
END METHOD M1.
```

The diagram below shows the load processes when the INVOKE statement (a) is executed:



The load steps are:

- [1] Control passes to class C2 when method M1 is invoked.
- [2] Because M1 is not contained in class C2 the COBOL runtime system is invoked.
- [3] The COBOL runtime system loads the separate DLL containing M1.
- [4] Control returns to class C2.
- [5] Class C2 invokes M1.

16.2.4 Creation and Execution

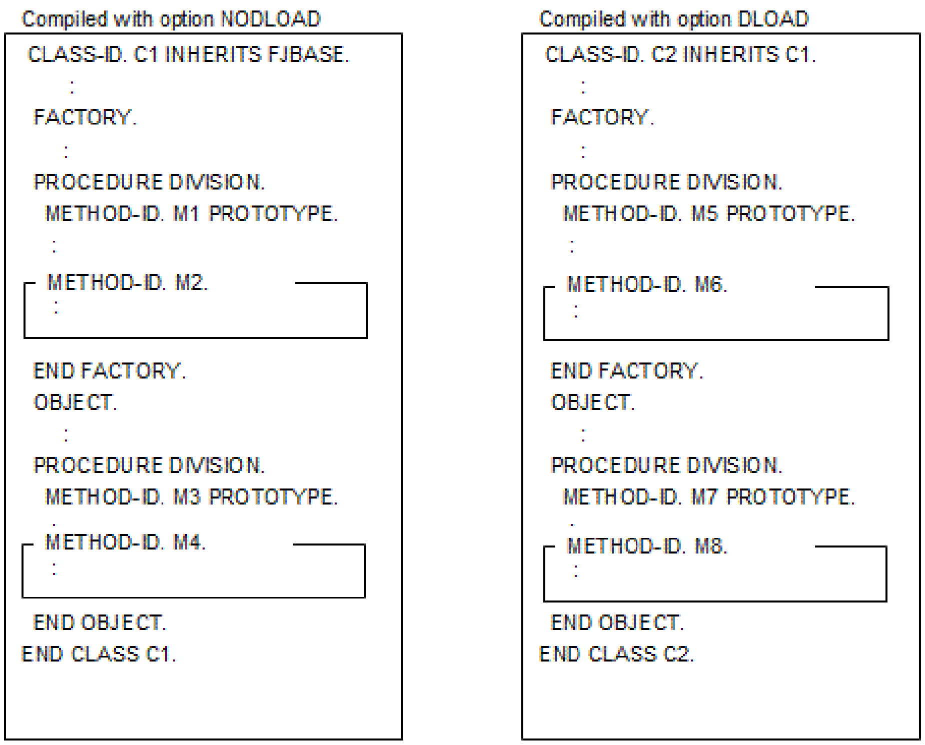
16.2.4.1 Compiling for Dynamic Program Structure

As mentioned above the way to create an application with the dynamic program structure is to compile the programs with the DLOAD compiler option. ("A.2.11 DLOAD(program structure specification)")

Prototype and Non-Prototype Methods

Only prototype methods can be separated from the class by the dynamic program structure. Methods whose code is defined within the class structure are compiled and linked in the same executable files.

Consider the two examples below:



Compiled with option NODLOAD

```
CLASS-ID. C1 INHERITS FJBASE.
    :
 FACTORY.
    :
 PROCEDURE DIVISION.
  METHOD-ID. M1 PROTOTYPE.
  :
  METHOD-ID. M2.
  :

 END FACTORY.
 OBJECT.
    :
 PROCEDURE DIVISION.
  METHOD-ID. M3 PROTOTYPE.
  .
  METHOD-ID. M4.
  :

 END OBJECT.
END CLASS C1.
```

Compiled with option DLOAD

```
CLASS-ID. C2 INHERITS C1.
    :
 FACTORY.
    :
 PROCEDURE DIVISION.
  METHOD-ID. M5 PROTOTYPE.
  :
  METHOD-ID. M6.
  :

 END FACTORY.
 OBJECT.
    :
 PROCEDURE DIVISION.
  METHOD-ID. M7 PROTOTYPE.
  .
  METHOD-ID. M8.
  :

 END OBJECT.
END CLASS C2.
```

The program structures used are as follows:

- Methods M5 and M7 are invoked using the dynamic program structure as they are separately defined prototypes invoked from a program compiled with the DLOAD option.
- Methods M6 and M8 are not invoked with the dynamic program structure because they are not separately defined prototypes.
- Methods M1, M2, M3, M4 are not invoked with the dynamic program structure because C1 is compiled with the NODLOAD option.

16.2.4.2 Linking Considerations

When using dynamic program structure import library requirements are slightly different than when using dynamic link structure. This is because the COBOL runtime system is resolving references that would otherwise be resolved by the linker.

Table 16.1 libraries for dynamic program structure

Target executable	Import libraries required
.EXE or .DLL for standard COBOL program	None
.DLL for a class	Import libraries for all directly or indirectly inherited classes that are built in separate DLLs.
.DLL for separate method	Import libraries for all directly or indirectly inherited classes in which the method is defined.

16.2.4.3 DLL File Naming Conventions

The COBOL runtime system searches for particular filenames when it loads DLL files. Table below shows these names and the expected file organization. If you do not use these names or organizations you need to specify entry information as described in the next topic.

Table 16.2 DLL file organization and names

Item searched for	File organization	File name
Class	Single DLL per class	Class-name.DLL
Method	Single DLL per method	Class-name_method-name.DLL

Item searched for	File organization	File name
GET property method	Single DLL per method	Class-name__GET_property-name.DLL (*)
SET property method	Single DLL per method	Class-name__SET_property-name.DLL (*)

(*) There are two underscores between the class-name and GET/SET (and only one underscore after GET/SET).

16.2.4.4 Entry Information

If you do not use the standard naming conventions for DLL files described above, you need to specify program, class and method entry information to the COBOL runtime system. To do this you create a file, called the entry information file, and provide the name of the file in the @CBR_ENTRYFILE environment variable. See "5.4.1.20 @CBR_ENTRYFILE(Set the Entry Information File)"

See "5.3 Setting Runtime Environment Information" for details of the entry information for subprograms. Note that although the entry information for subprograms can be placed in the runtime initialization file (COBOL85.CBR), the entry information for classes and methods must be supplied in an entry information file.

Entry Information File Format

The entry information file is a text file with sections containing information about subprogram, class and method file names. The format for class and method sections is given in table below:

Table 16.3 information file line formats

Item	Format	Description
Class section header	[CLASS]	Fixed text "CLASS" enclosed in brackets. An entry information file should have only one class header.
Class entry information	Class-name = DLL-filename	Class-name is the name of the class as defined in the CLASS-ID paragraph. DLL-filename is the name of the DLL file containing this class.
Method section header	[class-name.METHOD]	Class-name is the name of the class to which the methods in this section belong. ".METHOD" is fixed text appended to class-name.
Method entry information	Method-name = DLL-filename	Method-name is the name of the method as defined in the METHOD-ID paragraph. DLL-filename is the name of the DLL file containing this method.
GET property entry information	_GET_property-name = DLL-filename (specify on a single line)	"_GET_" is fixed text prefixed to property-name. property-name is the name of the property. DLL-filename is the name of the DLL file containing the GET property method.
SET property entry information	_SET_property-name = DLL-filename (specify on a single line)	"_SET_" is fixed text prefixed to property-name. property-name is the name of the property. DLL-filename is the name of the DLL file containing the SET property method.

Note

If the source programs containing classes or methods are compiled with the compiler option NOALPHAL class and method names will be case sensitive. See "A.2.1 ALPHAL(lowercase handling (in the program))".

Required Entry Information

Table below indicates when you need to specify entry information in the entry information file - if you do not use the standard file names listed in the "DLL File Naming Conventions" section above.

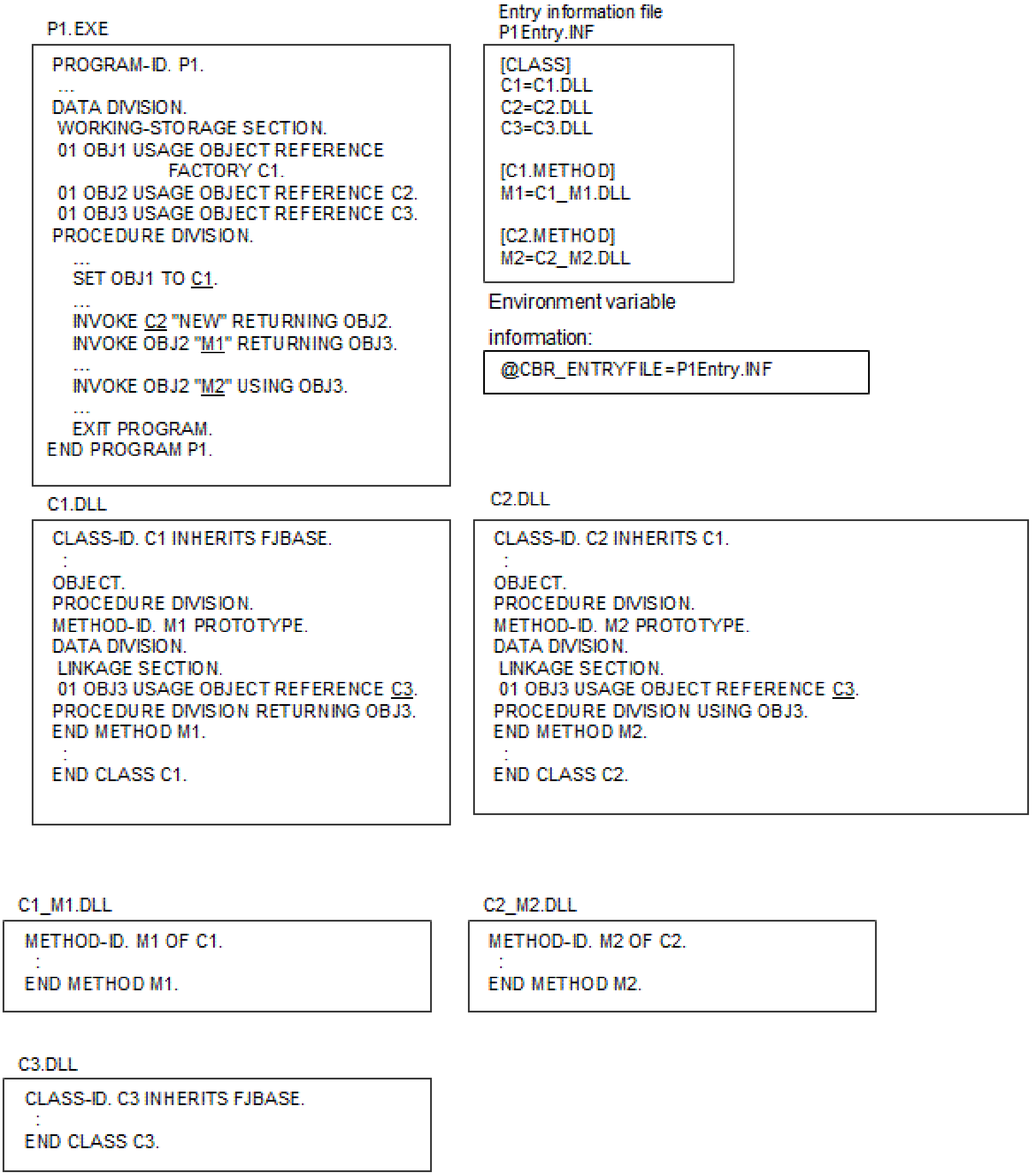
Table 16.4 Required entry information

	Class entry information	Method entry information
Program	- Classes referenced in the procedure division	- Any methods accessed with the dynamic program structure
Method		
Class	- Classes referenced in the procedure division - Classes specified in USAGE clauses of object references in the linkage sections of class methods (*) - when the object references are returned from the methods.	

* : Entry information needed for this class when conformance checking at execution time is required. See "14.4 Conformance Checking" and the description of the CHECK(ICONF) in "A.2.5 CHECK(whether the CHECK function should be used)" for more information.

Entry Information File Example

The code below shows what an entry information file would look like for the illustrated classes and methods. The underlined text indicates the class and method references that require entries in the entry information file.



P1.EXE

```
PROGRAM-ID. P1.
 ...
DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 OBJ1 USAGE OBJECT REFERENCE
                FACTORY C1.
 01 OBJ2 USAGE OBJECT REFERENCE C2.
 01 OBJ3 USAGE OBJECT REFERENCE C3.
PROCEDURE DIVISION.
    ...
    SET OBJ1 TO C1.
    ...
    INVOKE C2 "NEW" RETURNING OBJ2.
    INVOKE OBJ2 "M1" RETURNING OBJ3.
    ...
    INVOKE OBJ2 "M2" USING OBJ3.
    ...
    EXIT PROGRAM.
END PROGRAM P1.
```

Entry information file
P1Entry.INF

```
[CLASS]
C1=C1.DLL
C2=C2.DLL
C3=C3.DLL

[C1.METHOD]
M1=C1_M1.DLL

[C2.METHOD]
M2=C2_M2.DLL
```

Environment variable information:

```
@CBR_ENTRYFILE=P1Entry.INF
```

C1.DLL

```
CLASS-ID. C1 INHERITS FJBASE.
  :
OBJECT.
PROCEDURE DIVISION.
METHOD-ID. M1 PROTOTYPE.
DATA DIVISION.
 LINKAGE SECTION.
 01 OBJ3 USAGE OBJECT REFERENCE C3.
PROCEDURE DIVISION RETURNING OBJ3.
END METHOD M1.
  :
END CLASS C1.
```

C2.DLL

```
CLASS-ID. C2 INHERITS C1.
  :
OBJECT.
PROCEDURE DIVISION.
METHOD-ID. M2 PROTOTYPE.
DATA DIVISION.
 LINKAGE SECTION.
 01 OBJ3 USAGE OBJECT REFERENCE C3.
PROCEDURE DIVISION USING OBJ3.
END METHOD M2.
  :
END CLASS C2.
```

C1_M1.DLL

```
METHOD-ID. M1 OF C1.
  :
END METHOD M1.
```

C2_M2.DLL

```
METHOD-ID. M2 OF C2.
  :
END METHOD M2.
```

C3.DLL

```
CLASS-ID. C3 INHERITS FJBASE.
  :
END CLASS C3.
```

Note

In the example above all the DLL filenames were in the standard formats, so they could all be omitted from the entry information file.

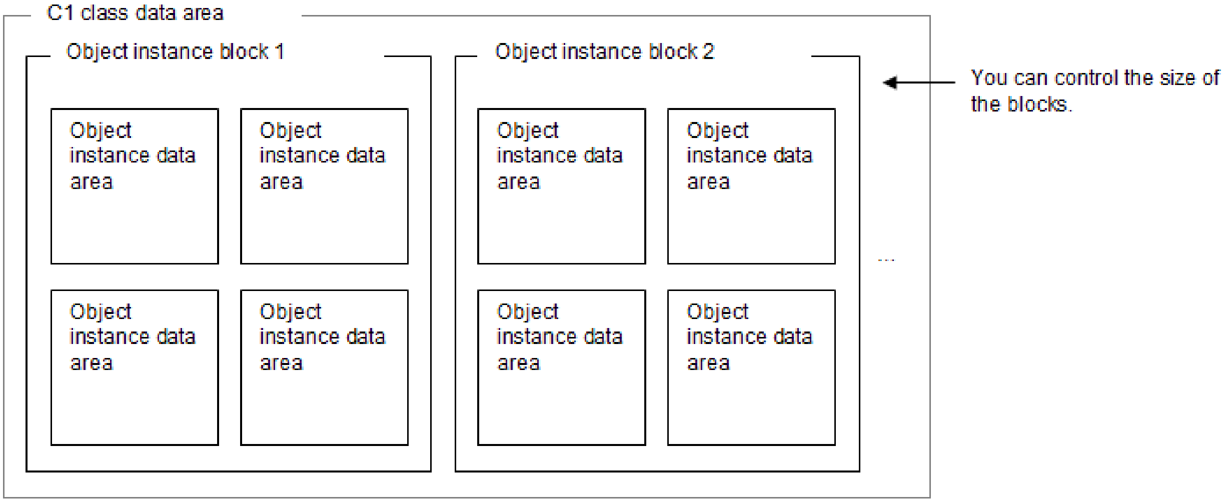
16.3 Tuning Memory Use

When using Fujitsu OO COBOL you can specify how memory is acquired when new objects are created. You can tune the memory acquisition process for optimal use of memory, best run time performance, or a combination of both. This section explains how the OO COBOL system acquires memory and what options are available to you to control the memory acquisition parameters.

16.3.1 Memory Acquired for New Objects

When a new object is created (by using the NEW method) an area of memory is acquired to hold the data for the new object instance. The size of the object instance data area depends on the amount of data defined for that object and the inheritance relationships of the object's class.

The runtime system usually acquires memory in blocks. Each block may be able to hold the data for several object instances. Once a block of memory has been acquired, subsequent new object instances are given data areas within this block until the block is fully used. When the block is full, the next new instance causes another block of memory to be acquired.


C1 class data area
Object instance block 1
Object instance block 2
Object instance data area
Object instance data area
Object instance data area
Object instance data area
Object instance data area
Object instance data area
Object instance data area
Object instance data area
...
You can control the size of the blocks.

16.3.2 Minimizing Memory Use

If you need to minimize the amount of memory used by your application, you should turn off blocking or specify the minimum initial block sizes your application requires with an incremental block size of one.

To turn off blocking use @CBR_InstanceBlock=UNUSE with no class information file.

To specify the minimum initial block size with an incremental block size of one, enter lines of the following form in your class information file:

```
Class-name=minimum-block-size
```

16.3.3 Improving Execution Performance

Each time a block of memory is acquired there is an execution overhead. If your application creates a large number of object instances, you may be able to improve performance by specifying a larger block size.

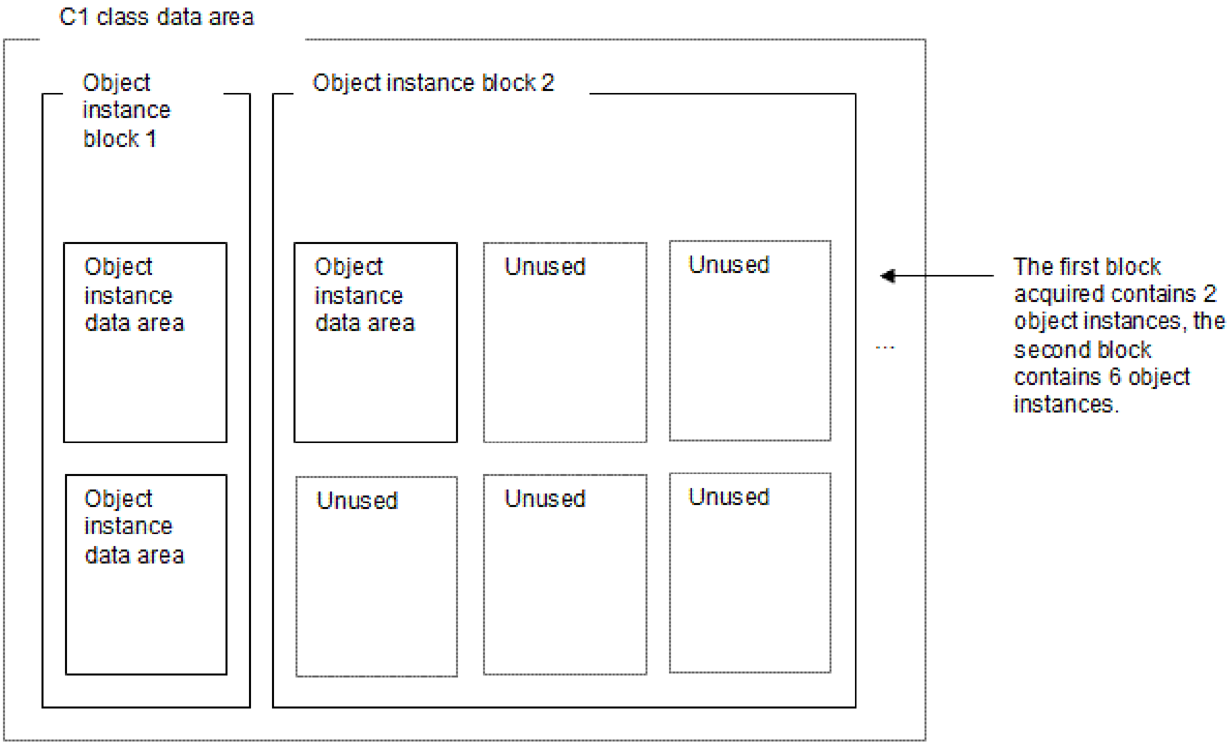
Specifying too large a block size may cause excessive paging of memory. Finding the optimum block size is a matter of experimentation or experience, based on the amount of memory on the application machines and other execution environment factors.

Example

If you provide the following class information file:

```
[InstanceBlock]
C1=2,6
```

And you create three object instances of class C1, the memory picture will look like this:


C1 class data area
Object instance block 1
Object instance block 2
Object instance data area
Object instance data area
Object instance data area
Unused
Unused
Unused
Unused
Unused
...
The first block acquired contains 2 object instances, the second block contains 6 object instances.

16.3.4 Controlling the Memory Acquired

Fujitsu OO COBOL provides the means to control, for each class, the initial block size, the size of subsequent blocks, and even whether blocking is used.

The control is provided by means of two environment variables and a text file called the class information file.

16.3.4.1 Environment variable

The environment variables are:

@CBR_ClassInfFile

points to the class information file that specifies blocking information by class. Any settings in the class information file override the default set by the @CBR_InstanceBlock variable.

@CBR_InstanceBlock

determines the default behavior for all classes: whether blocking is used or not

16.3.4.1.1 @CBR_ClassInfFile Environment Variable

The @CBR_ClassInfFile environment variable gives the name of a text file that specifies blocking factors for classes. It has the following format:

```
@CBR_ClassInfFile=class-information-filename
```

Class-information-filename is an absolute or relative pathname of the file containing the class information. If a relative pathname is specified, it is assumed to be relative to the folder containing the executable file.

The format of the class information file is explained in the next topic.

16.3.4.1.2 @CBR_InstanceBlock Environment Variable

The @CBR_InstanceBlock environment variable specifies the default blocking behavior. It has the following format:

```
                       ┌ USE      ┐
@CBR_InstanceBlock=[   ┤          ├ ]
                       └ UNUSE    ┘
```

Setting the variable to UNUSE tells the COBOL runtime system to acquire only sufficient memory to hold the data for one object instance. Every time a new object instance is created, a new block of memory is acquired.

Setting the variable to USE tells the COBOL runtime system to use blocking.

Either setting of @CBR_InstanceBlock can be overridden for a class by the values specified in the class information file.

16.3.4.2 Class Information File

The class information file is a text file that contains runtime configuration information for OO COBOL classes.

Currently it has a single section of information - the InstanceBlock section.

16.3.4.2.1 InstanceBlock section

The format of this section is:

```
[InstanceBlock]
class-name = initial-number [, incremental-number]
```

[InstanceBlock]

is fixed text and is the header of the section. Note that "[" and "]" are parts of the header.

class-name

is the name of the class whose blocking you wish to specify. Use the class name as specified in the CLASS-ID paragraph. You can specify as many class-name lines as there are classes.

initial-number

is the size of the first block of memory acquired, expressed as the number of object instance data areas the block can contain. The initial-number should be 1 or greater.

incremental-number

is the size of subsequent blocks of memory, expressed as the number of object instance data areas the block can contain. The incremental-number should be 1 or greater. If incremental-number is omitted, the value 1 is assumed.

16.4 Using Visual C++ with OO COBOL

This section explains how to use Microsoft Visual C++ (referred to simply as Visual C++) programs with Fujitsu OO COBOL.

16.4.1 Overview

You can work with Visual C++ objects in two ways:

1. When only the returned result is required.
2. When you want to use Visual C++ objects as if they were COBOL objects. A COBOL INVOKE statement accesses the Visual C++ member functions, and member variables can be referenced or set with COBOL property referencing and setting syntax.

You can achieve requirement 1 by accessing C or Visual C++ routines as external programs.

Requirement 2 takes more effort to achieve, requiring special code in both COBOL and C++ environments. This section therefore focuses on explaining how to achieve the more complex requirement. We shall call this closer cooperation with Visual C++ "collaborating with Visual C++".

See "Chapter 9 Calling Subprograms (Inter-Program Communication)" for information on accessing the routines as external programs. This section assumes that you are familiar with the material in "Chapter 9 Calling Subprograms (Inter-Program Communication)".

16.4.2 Collaborating with Visual C++

16.4.2.1 Collaborating Overview

The goal of collaborating with Visual C++ is to achieve a correspondence of classes and objects as shown in table below:

Table 16.5 correspondence between COBOL and Visual C++

Concept	Visual C++	COBOL
Class	Class	Class
Object	Object	Object (instance)
Object data	Member variable declared as public	Property
Method	Member function declared as public	Method

To achieve this level of collaboration, work has to be done in both the COBOL and Visual C++ environments.

COBOL Environment:

An interface class is created that lets COBOL programs and methods access the Visual C++ class as a COBOL class.

Visual C++ Environment:

Interface functions are created to access the Visual C++ class public functions and data. Because Visual C++ classes are usually defined in header files, the interface programs can be created by inspecting the header files.

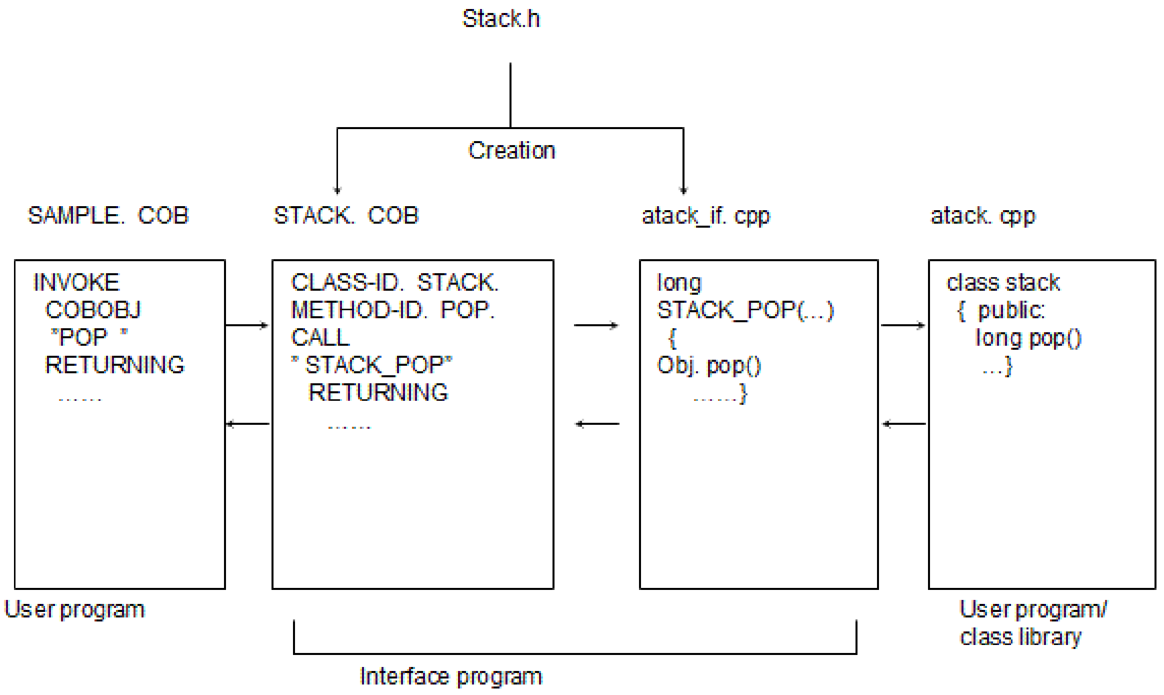
Overview Example

Collaborating with Visual C++ will be explained by working through a simple example of OO COBOL working with Visual C++. The example is a stack-handling program with push and pop functions. The diagram below gives an overview of the collaboration structure and the classes/programs involved.

"atack.cpp" is the existing Visual C++ class.

"SAMPLE.COB" is a COBOL program illustrating how we wish to access the Visual C++ functionality.

"STACK.COB" and "atack_if.cpp" are the COBOL and Visual C++ programs that we need to create to achieve the collaboration. Both are based on the header file "stack.h".



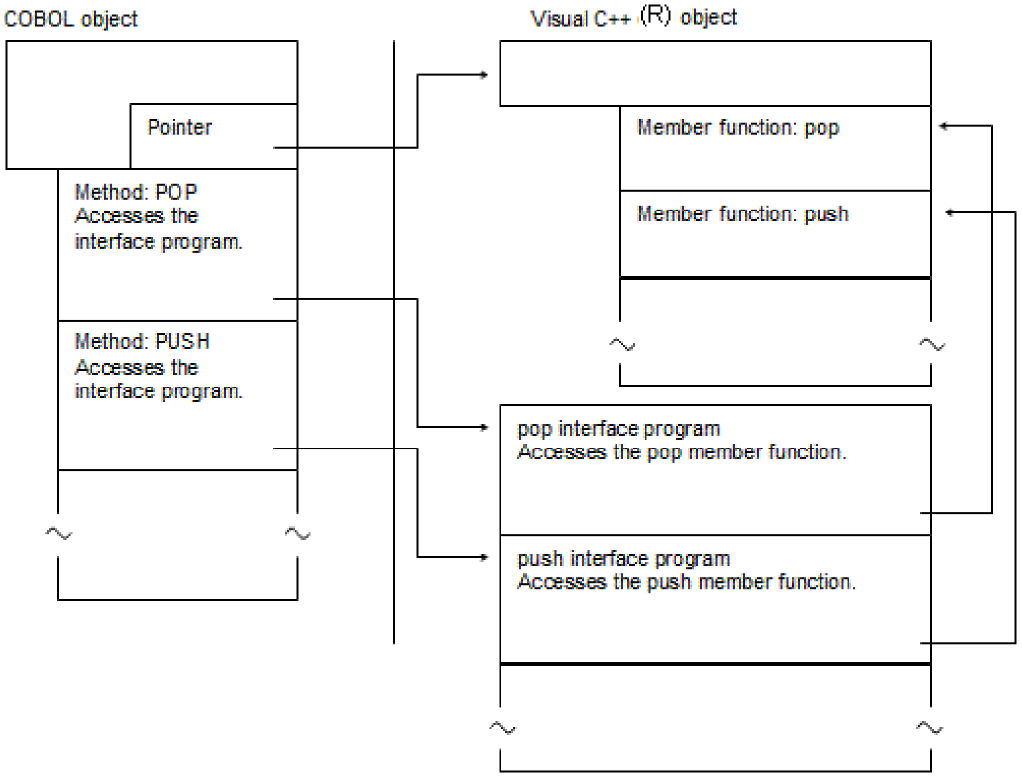
16.4.2.2 Collaborating Program Structure

Before looking at the code that needs to be created, we will take a closer look at how the interface programs are structured.

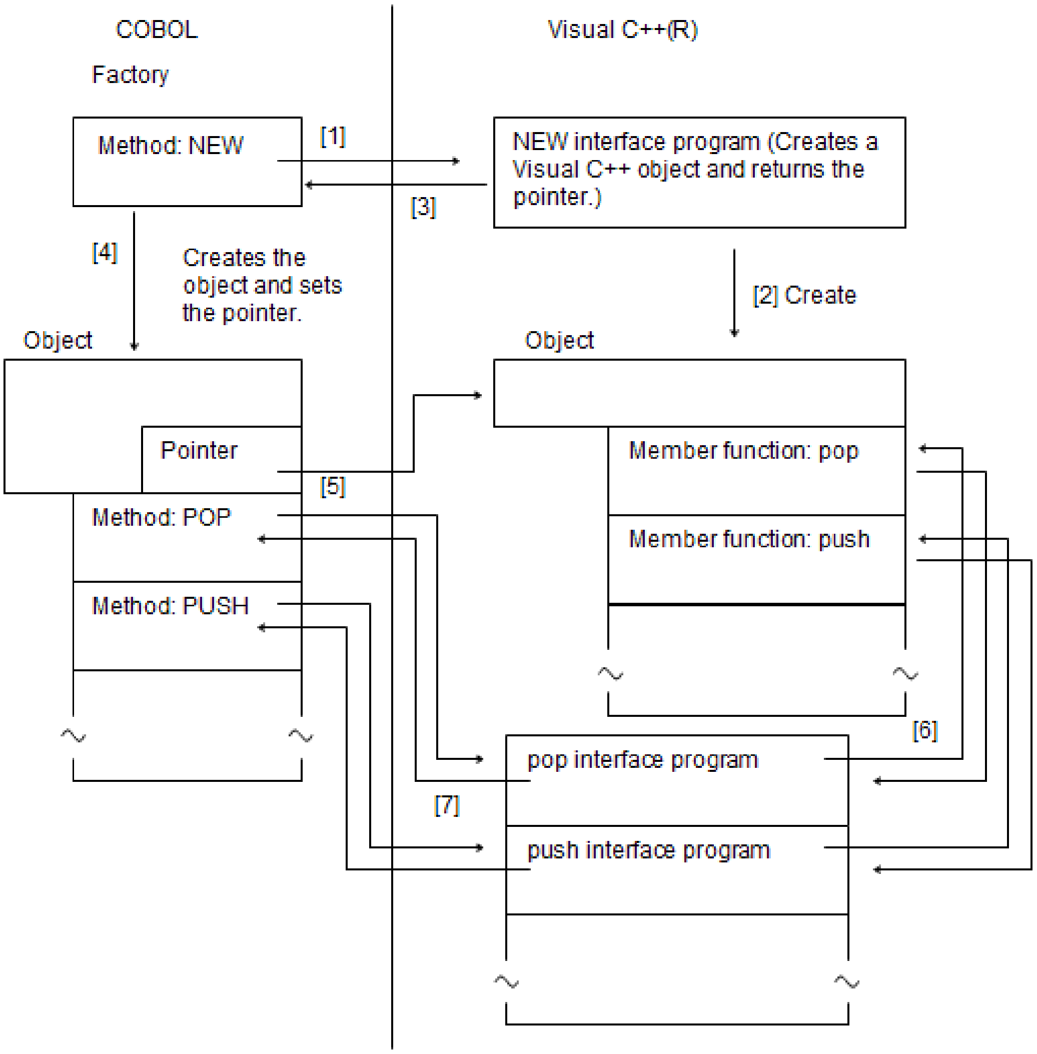
The key points for structuring the interface programs are:

- Create a COBOL class with the same structure as the target Visual C++ class.
- In the COBOL object data include an item to hold a pointer to the Visual C++ object. This pointer is passed to the Visual C++ interface program along with other required arguments.
- The Visual C++ interface programs call the member functions in the corresponding objects.

This structure is illustrated in the following diagram:



The next diagram shows the control flow for this structure:



The control flow shown in the above diagram is:

1. When the NEW method in the COBOL interface class is invoked, it invokes the NEW function of the Visual C++ interface program.
2. The Visual C++ interface program creates the Visual C++ object.
3. The Visual C++ interface program returns the pointer to the newly created object.
4. The NEW method in the COBOL interface class creates the COBOL interface object and stores the object pointer in the new COBOL object.
5. When the COBOL POP method is invoked, it calls the pop interface program.
6. The pop interface program calls the pop member function.
7. The pop interface program returns any results to the COBOL POP method, which in turn returns the results to the invoking program.

16.4.3 Collaborating Programming Procedure

To create the interface programs you go through the following steps:

1. Review the Visual C++ class definition
2. Define the COBOL class
3. Define the Visual C++ interface programs

These steps are detailed below based on this Visual C++ class definition:

```
class stack {
public:
  unsigned long pntr;
  long pop();
  void push(long val);
private:
  long data[100];
} ;
```

16.4.3.1 Review the Visual C++ Class Definition

From the Visual C++ class definition select the member functions and member variables that you wish to make available to the COBOL environment.

For our example, we want to make all the member functions and variables available. These are:

- Member functions
    - pop
    - push
- Member variables
    - pntr

16.4.3.2 Define the COBOL class

To ease understanding it is recommended that you use equivalent names in the COBOL class as are used in the Visual C++ class. If names conflict with COBOL reserved words or use non-COBOL characters, alternative names should be found.

Table below describes the class elements to be defined.

Table 16.6 COBOL Interface Class Elements

Class element	Comments	Example items/names
Class name	The class name will be used in COBOL to reference the class.	STACK
Properties	Code the GET and SET methods for properties yourself. They should call the Visual C++ member variable referencing and setting interfaces. Also define a property to retain the Visual C++ object pointer.	PNTR CPP-OBJ-POINTER
Factory methods	You need to redefine the NEW factory method to: Create a new object of the COBOL class Call the Visual C++ object creation interface program and acquire the object pointer. Store the pointer in the pointer storage area.	NEW
Object methods	Create methods to call the corresponding member function interface programs. Also, create a method to call an object-deleting interface program.	POP PUSH DELETE-OBJ

16.4.3.3 Define the Visual C++ interface programs

Table below describes the interface programs that you create.

Table 16.7 Visual C++ Interface Programs

Interface Program	Comments	Example Name
Object creation	The object creation program is called by the COBOL NEW method. It creates the Visual C++ object and returns the object pointer.	CPP_STACK_NEW
Member function access	Define one access program for each member function. Each takes the arguments required by the member function along with the object pointer, and returns the member function value.	CPP_STACK_POP CPP_STACK_PUSH
Object deletion	The object deletion program is called by the COBOL DELETE-OBJ method. It deletes the object created in Visual C++.	CPP_STACK_DELETE_OBJ
Property access	Define SET and GET programs for each member variable.	CPP_STACK_GET_PNTR CPP_STACK_SET_PNTR

Table below summarizes the interface programs created for accessing the example Visual C++ class.

Table 16.8 Summary of Interface Programs in Example

	COBOL class definition	Visual C++ interface program	Visual C++ class definition
Class	STACK	None	stack
Method	NEW	CPP_STACK_NEW	new
	POP	CPP_STACK_POP	pop
	PUSH	CPP_STACK_PUSH	push
	DELETE-OBJ	CPP_STACK_DELETE_OBJ	delete
Property	PNTR GET	CPP_STACK_GET_PNTR	pntr (member variable)
	PNTR SET	CPP_STACK_SET_PNTR	

16.4.4 Using the Collaborating Class

The code below shows how the new class would be invoked from a COBOL program.

```
 WORKING-STORAGE SECTION.
 01  STACKOBJ USAGE OBJECT REFERENCE STACK.
 01  POP-VALUE PIC S9(9) COMP-5.
 PROCEDURE DIVISION.
     INVOKE STACK "NEW" RETURNING STACKOBJ.
     MOVE 0 TO PNTR OF STACKOBJ.
     INVOKE STACKOBJ "PUSH" USING 10.
     INVOKE STACKOBJ "PUSH" USING 20.
     INVOKE STACKOBJ "PUSH" USING 30.
     INVOKE STACKOBJ "POP" RETURNING POP-VALUE.
     INVOKE STACKOBJ "DELETE-OBJ".
     SET STACKOBJ TO NULL.
*>      :
```

16.4.5 The Sample Program

The code samples below show the COBOL interface class in detail and an outline of the Visual C++ code.

COBOL class definition for Visual C++ collaboration

```
*>
*> DEFINE CLASS STACK
*>
 IDENTIFICATION DIVISION.
 CLASS-ID.   STACK INHERITS FJBASE.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
     CLASS FJBASE.
*>
*>  FACTORY DEFINITION
*>    RE-DEFINE NEW METHOD
*>
 IDENTIFICATION DIVISION.
 FACTORY.
 PROCEDURE DIVISION.
 IDENTIFICATION DIVISION.
 METHOD-ID. NEW OVERRIDE.
 DATA                 DIVISION.
 WORKING-STORAGE      SECTION.
  01 CPP-STACK USAGE IS POINTER.
 LINKAGE SECTION.
  01 STACKOBJ USAGE OBJECT REFERENCE SELF.
 PROCEDURE DIVISION RETURNING STACKOBJ.
      INVOKE SUPER "NEW" RETURNING STACKOBJ.
      CALL "CPP_STACK_NEW" RETURNING CPP-STACK.
      INVOKE STACKOBJ "SET-CPP-OBJ-POINTER" USING CPP-STACK.
      EXIT METHOD.
 END METHOD NEW.
 END FACTORY.
*>
*>  OBJECT DEFINITION
*>    DATA RETAINING C++ OBJECT AS OBJECT DATA
*>    (CPP-OBJ-POINTER)DEFINE
*>
 IDENTIFICATION DIVISION.
 OBJECT.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
  01 CPP-OBJ-POINTER USAGE IS POINTER.
 PROCEDURE DIVISION.
*>
*> SET-CPP-OBJ-POINTER METHOD DEFINITION
*>
 IDENTIFICATION DIVISION.
 METHOD-ID. SET-CPP-OBJ-POINTER.
 DATA DIVISION.
 LINKAGE SECTION.
  01 USE-VALUE USAGE IS POINTER.
 PROCEDURE DIVISION USING USE-VALUE.
     MOVE USE-VALUE TO CPP-OBJ-POINTER.
     EXIT METHOD.
 END METHOD SET-CPP-OBJ-POINTER.
*>
*>  POP METHOD DEFINITION
*>
 IDENTIFICATION DIVISION.
 METHOD-ID. POP.
 DATA DIVISION.
 LINKAGE SECTION.
```

```
  01 RET-VALUE PIC S9(9) COMP-5.
 PROCEDURE DIVISION RETURNING RET-VALUE.
     CALL "CPP_STACK_POP" USING CPP-OBJ-POINTER
                                RETURNING RET-VALUE.
     EXIT METHOD.
 END METHOD POP.
*>
*>  PUSH METHOD DEFINITION
*>
 IDENTIFICATION DIVISION.
 METHOD-ID. PUSH.
 DATA DIVISION.
 LINKAGE SECTION.
  01 SET-VALUE PIC S9(9) COMP-5.
 PROCEDURE DIVISION USING SET-VALUE.
     CALL "CPP_STACK_PUSH" USING CPP-OBJ-POINTER SET-VALUE.
     EXIT METHOD.
 END METHOD PUSH.
*>
*>  DEFINITION OF METHOD REFERENCING PNTR
*>
 IDENTIFICATION DIVISION.
 METHOD-ID. GET PROPERTY PNTR.
 DATA DIVISION.
 LINKAGE SECTION.
  01 RET-VALUE PIC S9(9) COMP-5.
 PROCEDURE DIVISION RETURNING RET-VALUE.
     CALL "CPP_STACK_GET_PNTR" USING CPP-OBJ-POINTER RETURNING
     RET-VALUE.
     EXIT METHOD.
 END METHOD.
*>
*>  PNTR SETTING METHOD DEFINITION
*>
 IDENTIFICATION DIVISION.
 METHOD-ID. SET PROPERTY PNTR.
 DATA DIVISION.
 LINKAGE SECTION.
  01 SET-VALUE PIC S9(9) COMP-5.
 PROCEDURE DIVISION USING SET-VALUE.
     CALL "CPP_STACK_SET_PNTR" USING CPP-OBJ-POINTER SET-VALUE.
     EXIT METHOD.
 END METHOD.
*>
*>  DELETE-OBJ METHOD DEFINITION
*>
 IDENTIFICATION DIVISION.
 METHOD-ID. DELETE-OBJ.
 DATA DIVISION.
 LINKAGE SECTION.
 PROCEDURE DIVISION.
     CALL "CPP_STACK_DELETE_OBJ" USING CPP-OBJ-POINTER.
     EXIT METHOD.
 END METHOD DELETE-OBJ.
 END OBJECT.
 END CLASS STACK.
```

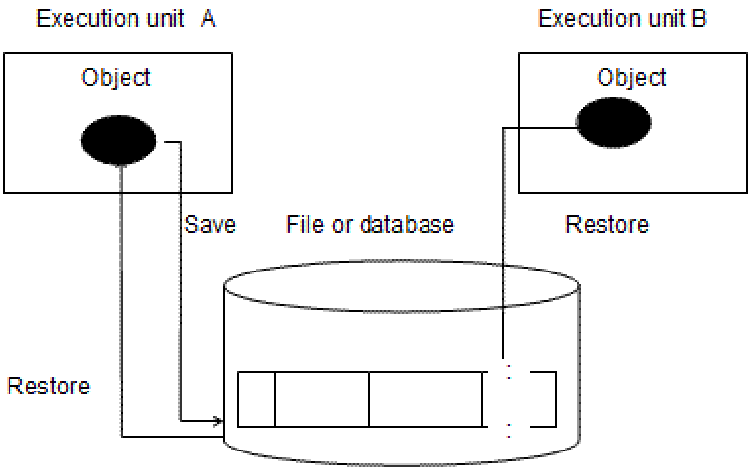
Visual C++ interface program Visual C++ collaboration

```
#include "stack.h"

extern "C" stacK* CPP_STACK_NEW() {
  return new stack;
}

extern "C" long int CPP_STACK_POP(stack** stk) {
  return (*stk)->pop();
}

extern "C" void CPP_STACK_PUSH(stack** stk, long int* value){
  (*stk)->push(*value);
}

extern "C" long int CPP_STACK_GET_PNTR(stack** stk){
  return (*stk)->pntr;
}

extern"C" void CPP_STACK_SET_PNTR(stack** stk, long int* value){
  (*stk)->pntr = *value;
}

extern "C" void CPP_STACK_DELETE_OBJ(stack** stk) {
  delte *stk;
}
```

16.5 Making Objects Persistent

Objects are intended to model real-life objects such as employees and bank accounts. The real-life objects are usually long-lived, i.e. they can exist for many years, certainly longer than a normal application's execution time. OO COBOL objects exist in the computer's memory and therefore cease to exist when the application terminates or if the computer is switched off. For software objects to model real-life objects, they need to exist for the same length of time as the real-life objects. In OO programming, long-lived objects are called "persistent objects" and the general long-life feature is referred to as "persistency".

OO COBOL does not have persistency built into it, however, you can make objects persistent by using the COBOL file system - in the same way you keep a permanent, or long-lived, record of application data in standard COBOL applications. You save and restore objects just as you save and read application data:



It is actually the object data that is saved and restored, but we will generally talk about saving and restoring objects.

This section discusses the issues you need to address when making your objects persistent. It uses a sample class structure to help explain the issues. The topics are:

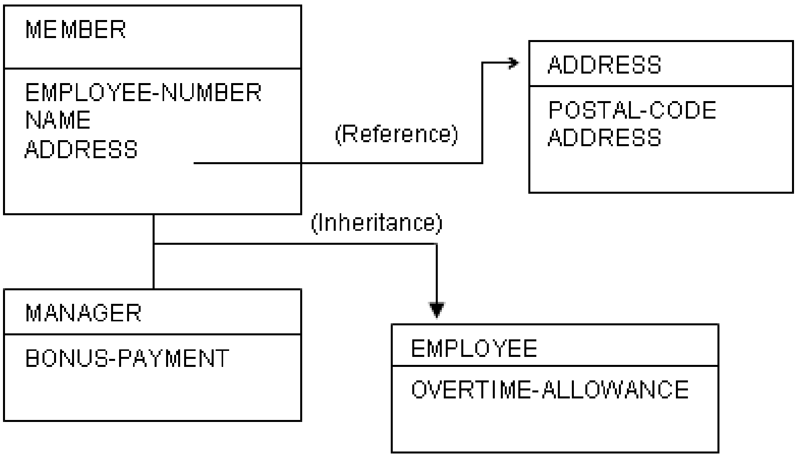
- The sample class structure
- Possible class to file mappings
- A suggested index file structure

- Class and method additions for persistency
- Saving and restoring processes

When making objects persistent using a database, read "database table" for "file. "See "17.2.6 Accessing a Database Using Object-Oriented Programming Functions".

16.5.1 The Persistency Sample Class Structure

The topics on persistency use the class structure shown below to illustrate the various issues. Methods are not shown, as these are not relevant to persistency. Each class contains only the minimum amount of data required to illustrate the principles.



The class structure contains the elements:

- MEMBER class: represents all employees and contains the employee number, name and address as object data. The address is associated with the ADDRESS object.
- MANAGER class: represents managers and contains a bonus payment as object data. It inherits from the MEMBER class.
- EMPLOYEE class: represents general employees and contains an overtime allowance as object data. It also inherits from the MEMBER class.
- ADDRESS class: contains address and postal code as object data. The ADDRESS class has no inheritance relationship with other classes.

16.5.2 Mapping of index file and object

16.5.2.1 Possible Class to File Mappings for Persistency

When designing persistency for your system you need to decide which classes are saved to which files. The model you choose will depend on your inheritance structure, how simple you want your file management or data restructuring procedures to be.

We will look the following class to file mappings:

- One file for each class
- One file for each elementary class and its parents
- One file for classes with common parents

One file for each class

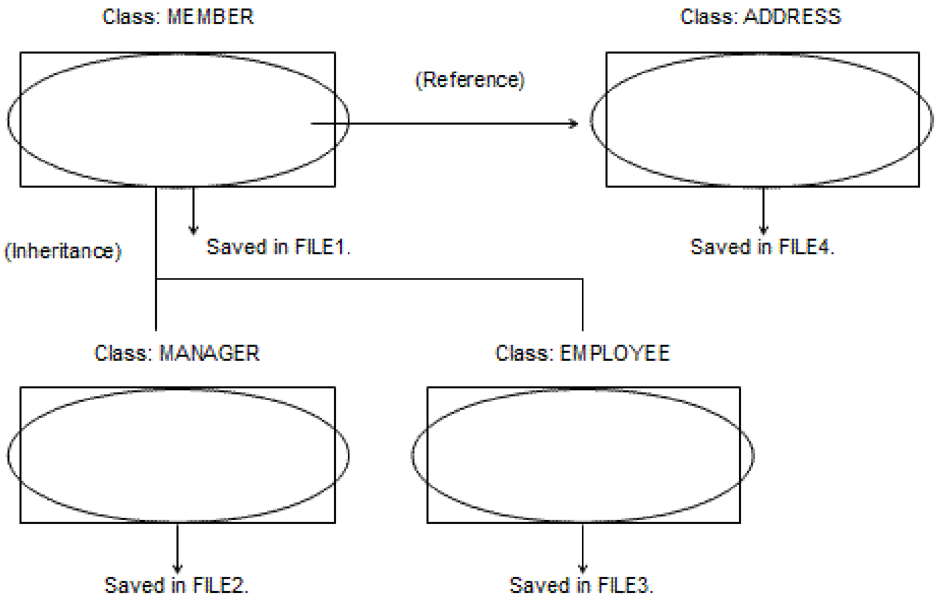
In this model, every class is saved to a separate file. For our sample class structure:

MEMBER class would be saved to FILE1

MANAGER class would be saved to FILE2

EMPLOYEE class would be saved to FILE3

ADDRESS class would be saved to FILE4.



In this model data that is unique to the MANAGER class is saved to FILE2, and the data unique to the MEMBER class is saved to FILE1.

Restoring the objects starts with reading MEMBER data from FILE1. If the data is manager data, then the corresponding data is read from FILE2. The combined data is then used to create the MANAGER object.

Benefits

- Because there is one file for each class, changing the data structures in a class only affects one file. File maintenance is therefore straightforward.

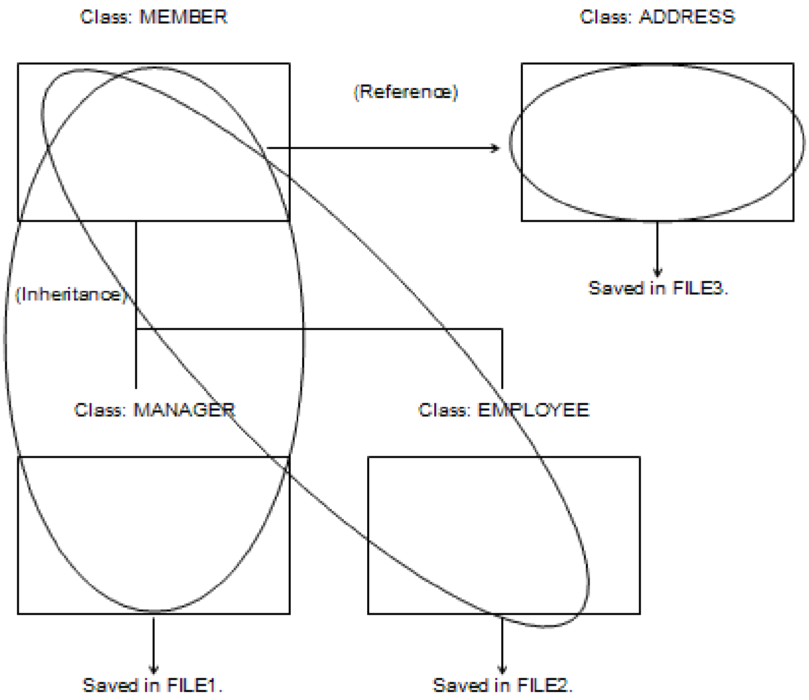
Disadvantages

- Because there are many files involved in saving the structure, managing the files can be complex.

One file for each elementary class and its parents

For this discussion, the term "elementary class" is used to describe a class that is not inherited by any other classes.

If the MEMBER class is defined as an abstract class, there are no independent MEMBER class objects. In this case processing can be simplified by saving the object data of the MEMBER class in the same file as the object data of the MANAGER class. Likewise, the EMPLOYEE class is saved by combining its object data with the object data of the MANAGER class.



To restore the MANAGER objects you read the object data from FILE1 and create a MANAGER object.

Benefits

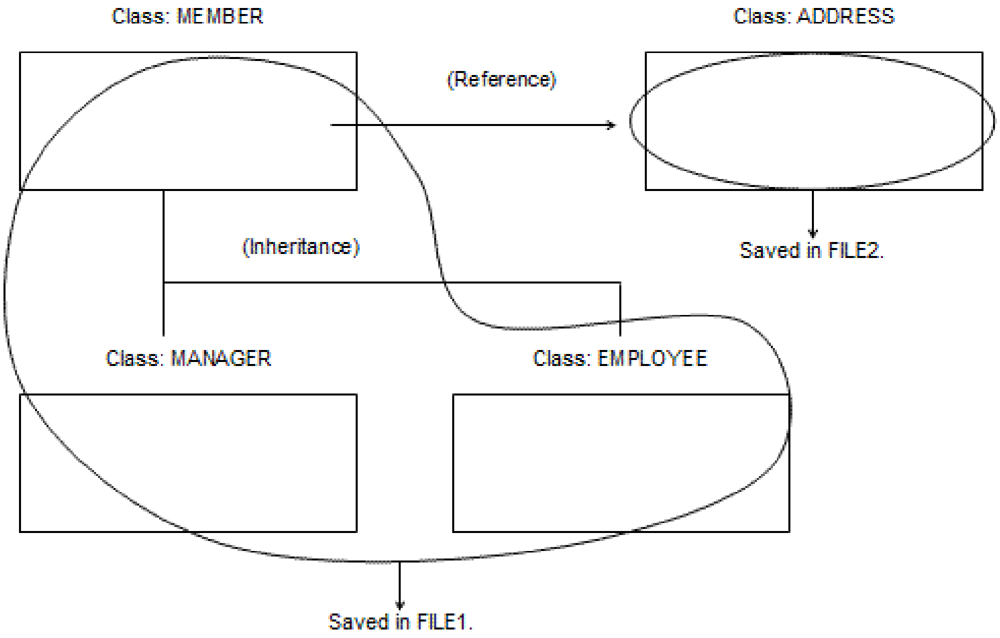
- Fewer files are required than the one file to one class model.
- All the inherited data for an elementary class is in the one file.

Disadvantages

- If the structure of a superclass' data changes all the files that save classes that inherit from that superclass have to be changed.
- If you want to restore the object for a particular employee number, you do not know which file the employee object is stored in. The employee number does not indicate whether the employee is a manager or a general employee.

One file for classes with common parents

If restoring objects from a particular ID, such as employee number, is a major concern, then a better file structure is to store the elementary classes with common parents in a single file.



Benefits

- Uses fewer files than either of the other two models.
- Supports restoring objects based on a particular ID.
- All the inherited data for an elementary class is in the one file.

Disadvantages

- If the structure of a superclass' data changes all the files that save classes that inherit from that superclass have to be changed.

The remaining topics on persistency use this model.

16.5.2.2 A Suggested Index File Structure for Persistency

When you construct files for saving objects you need to ensure that you can recreate the objects accurately and that you can access each object's data uniquely.

If you use the file structure of one file for classes with common parentage then, for our sample class structure, you will have two files: one for the MEMBER, MANAGER, EMPLOYEE classes, and one for the ADDRESS class. The employee number already uniquely identifies a record so that is use to identify the objects - it becomes the prime key for the indexed file. Another field is added to distinguish MANAGER objects from EMPLOYEE objects, called the class identification area in the diagram below. The example uses 1 for MANAGER objects, 2 for EMPLOYEE objects.

Similarly, you assign a unique identifier for the ADDRESS objects. In this example, because ADDRESS is only referenced by the MEMBER class, you could use the employee number as the ADDRESS ID (and remove the Address ID field from File 1).

File 1:

	Prime key	Class identification area			
MANAGER	Employee number	1	Name	Address ID	Bonus payment
EMPLOYEE	Employee number	2	Name	Address ID	Overtime allowance

File 2:

ADDRESS	Address ID	Postal code	Address

16.5.3 Class and Method Additions for Persistency

To support persistency in your application it is best to create a class to handle each file and add methods to the classes whose objects are to be saved.

16.5.3.1 Indexed File Handling Class

The indexed file handling class should contain the object methods shown in table below.

Table 16.9 Indexed file handling class methods

Object method name	Description
OPEN-DATA-FILE	Opens the indexed file.
CLOSE-DATA-FILE	Closes the indexed file.
SAVE	Is passed an object reference as an argument, identifies the class of the object, retrieves the appropriate data from the object, and saves that data to the file.
RETRIEVE	Receives an object identifier as an argument, reads the file to retrieve the data, and returns the data.

You invoke the OPEN-DATA-FILE method at the beginning of your application and the CLOSE-DATA-FILE method at the end of your application. You invoke the SAVE and RETRIEVE functions from methods of objects to be saved or restored as described below.

16.5.3.2 Methods Added to Persistent Classes

To make a class persistent you add SAVE and RETRIEVE methods to that class. The RETRIEVE method is a factory method because its function is to recreate an object. Table below describes the SAVE and RETRIEVE methods. You only need to define these methods in the parent class, they do not need to be defined for each class.

Table 16.10 Methods added for persistency

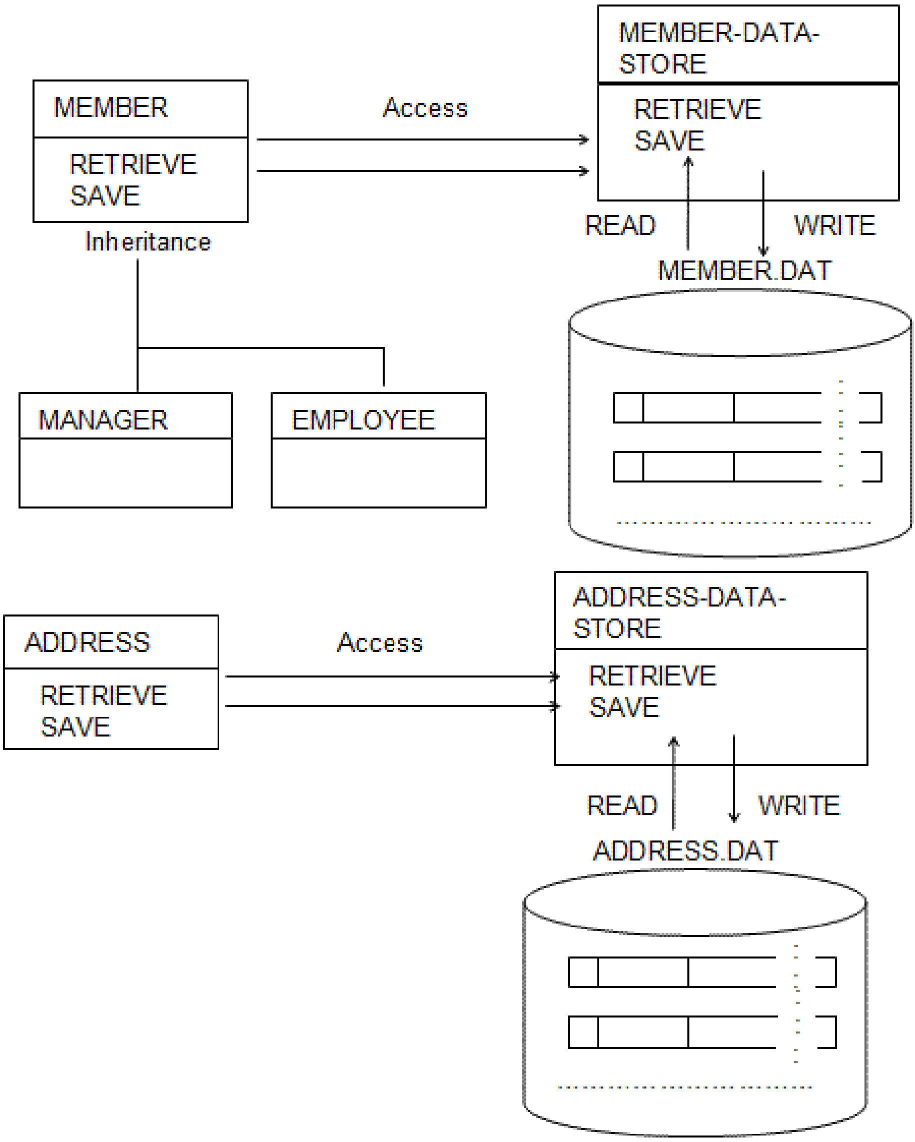
Factory/Object method	Method name	Arguments	Description
FACTORY	RETRIEVE	USING: Object-identifier RETURNING: Object-reference	Receives the object-identifier of the object to be retrieved. Uses the data file RETRIEVE method to get the data. Creates a new object populated with the data, and returns that object reference.
OBJECT	SAVE	USING: None RETURNING: Success-code	Invokes the data file SAVE method using itself as the object reference. It is best to include a success code and logic to cope with problems in saving the data.

Table below shows the classes and files that you would set up for our example, and the following figure illustrates the relationship between the methods, classes and files.

Table 16.11 Saved class/indexed file/indexed file operating class

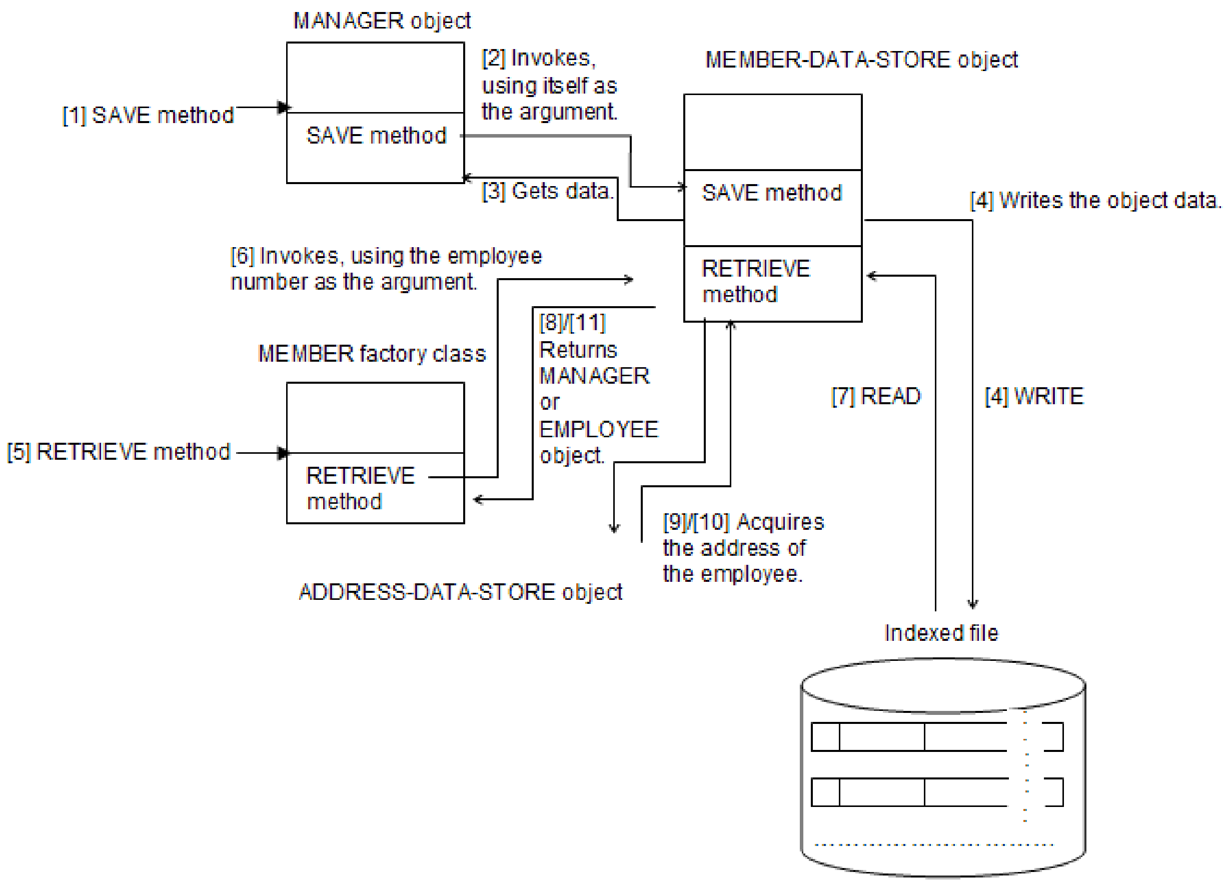
Class to be saved	Parent class	Indexed filename	Indexed file operating class
MANAGER	MEMBER	MEMBER.DAT	MEMBER-DATA-STORE
EMPLOYEE			
ADDRESS	None	ADDRESS.DAT	ADDRESS-DATA-STORE

Figure 16.1 Relationships between persistent classes and supporting file handling classes



16.5.3.3 The Persistency Saving and Restoring Processes

The diagram below illustrates the process for saving and restoring objects.


MANAGER object
[1] SAVE method
SAVE method
[2] Invokes, using itself as the argument.
MEMBER-DATA-STORE object
[3] Gets data.
SAVE method
[4] Writes the object data.
RETRIEVE method
[6] Invokes, using the employee number as the argument.
[8]/[11] Returns MANAGER or EMPLOYEE object.
MEMBER factory class
[5] RETRIEVE method
RETRIEVE method
[7] READ
[4] WRITE
[9]/[10] Acquires the address of the employee.
ADDRESS-DATA-STORE object
Indexed file

Saving:

1. A MANAGER object's SAVE method is invoked.
2. The SAVE method invokes the SAVE method of the MEMBER-DATA-STORE object passing a reference to itself (the MANAGER object).
3. The MEMBER-DATA-STORE SAVE method checks the class of the argument, and obtains the required data from the MANAGER object.
4. The MEMBER-DATA-STORE SAVE method saves the data to the indexed file.

Restoring:

1. The MANAGER class's factory RETRIEVE method is invoked.
2. The RETRIEVE method invokes the RETRIEVE method of the MEMBER-DATA-STORE, providing the employee number of the object to be restored.
3. The MEMBER-DATA-STORE RETRIEVE method reads the indexed file using the employee number as the prime key.
4. The MEMBER-DATA-STORE RETRIEVE method checks the class identifier (MANAGER or EMPLOYEE) and creates an object of the appropriate class.
5. The MEMBER-DATA-STORE RETRIEVE method invokes the RETRIEVE method of the ADDRESS-DATA-STORE object so that there is a matching ADDRESS object.
6. The MEMBER-DATA-STORE RETRIEVE method uses data from the ADDRESS object to populate the MANAGER or EMPLOYEE object.
7. The MEMBER-DATA-STORE RETRIEVE method returns the MANAGER or EMPLOYEE object to the RETRIEVE method of the MEMBER class.

16.6 Programming Using the ANY LENGTH Clause

16.6.1 Class that Handles Character Strings

When a class that handles various lengths of character strings is created in an object-oriented program, the maximum length of character strings to be handled must be determined in advance, because there is no way of declaring COBOL character strings without determining the maximum length. Also, there is a "matching rule" to prevent interface errors between object-oriented functions. This means that if the calling side wants to pass character strings of different lengths, it must invoke a method by storing a character string in the variable declared with a maximum length suited for the called method.

In this case, if you want to change the maximum character string length defined in a class, you must change the maximum length of the class in which the called method is defined, as well as the programs and classes that reference that class. Then you must recompile all of the changed classes or programs. If the maximum length is declared with a margin sufficient for future changes, ordinary operational performance using short length character strings is adversely affected. To resolve this, a process is required that passes the actual length of a character string to the called method, which performs reference modification using the length.

For instance, when a class (method) used to authenticate a name and password as shown in the example below is to be created, the maximum length of character strings must be determined in advance. The example defines the maximum length of the name as ten alphanumeric characters, and the password as eight alphanumeric characters.

Before character strings are passed:

```
PROGRAM-ID.   INFORMATION-CHANGE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
 REPOSITORY.
     CLASS  AUTHENTICATION-CLASS.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  NAME-STRING.
  02  NAME                 PIC X(20).
01  PASSWORD-DATA          PIC X(8).
01  AUTHENTICATION-RESULT  PIC X(2).
01  AUTHENTICATED-OBJECT  USAGE OBJECT REFERENCE  AUTHENTICATION-CLASS.
PROCEDURE DIVISION.
    INVOKE  AUTHENTICATION-CLASS "NEW" RETURNING AUTHENTICATED-OBJECT.
    DISPLAY  "ENTER THE NAME AND PASSWORD".
    ACCEPT  NAME-STRING.
    ACCEPT  PASSWORD-DATA.
    INVOKE AUTHENTICATED-OBJECT "AUTHENTICATION-METHOD"
                                          USING NAME  PASSWORD-DATA
                                          RETURNING   AUTHENTICATION-RESULT.
    IF  AUTHENTICATION-RESULT = "OK" THEN
      CALL  "INFORMATION-CHANGE-PROCESS"
    ELSE
       DISPLAY "YOU ARE NOT AUTHORIZED TO CHANGE."
    END-IF.
END PROGRAM INFORMATION-CHANGE.
```

General-purpose class that handles passed character strings:

```
CLASS-ID.   AUTHENTICATION-CLASS INHERITS FJBASE.
 :
OBJECT.
PROCEDURE DIVISION.
METHOD-ID.  AUTHENTICATION-METHOD.
DATA DIVISION.
LINKAGE SECTION.
01  NAME                   PIC X(20).
01  PASSWORD-DATA          PIC X(8).
01  AUTHENTICATION-RESULT  PIC X(2).
PROCEDURE DIVISION USING  NAME  PASSWORD-DATA
                   RETURNING  AUTHENTICATION-RESULT.
```

```
       EVALUATE NAME             ALSO  PASSWORD-DATA
       WHEN   "JOHN TRAVOLTA"  ALSO "A1"
       WHEN   "PAUL SEAGULL"   ALSO "XXXXYYY2"
         MOVE  "OK" TO  AUTHENTICATION-RESULT
       WHEN OTHER
         MOVE  "NG" TO  AUTHENTICATION-RESULT
       END-EVALUATE
  END METHOD  AUTHENTICATION-METHOD.
  END OBJECT.
  END CLASS   AUTHENTICATION-CLASS.
```

If another process using the authentication class is generated under the above conditions, and the maximum length of the password is to be changed, the authentication class must first be modified, thereby triggering the need to modify the information change program. Also, after changing the interface, the class inheriting the authentication class must be recompiled.

16.6.2 Using the ANY LENGTH Clause

In consideration of the above, COBOL supports the ANY LENGTH clause. The ANY LENGTH clause can be specified for an alphanumeric data item in a method's LINKAGE SECTION. When the method is invoked, the length of the item is automatically evaluated as the length of the item specified by the invoking side. Therefore, coding as shown below enables the creation of a class that can handle items of any length. The LENGTH function can be used to determine the number of characters making up the item for which the ANY LENGTH clause is specified. Also, the LENGTH function can be used to determine the length (the number of bytes). The ANY LENGTH clause can also be specified for the return item, to return character strings with the length of the return item defined by the invoking side.

Sample program using the ANY LENGTH clause:

```
  CLASS-ID.   AUTHENTICATION-CLASS INHERITS FJBASE.
*>  :
  OBJECT.
  PROCEDURE DIVISION.
  METHOD-ID.  AUTHENTICATION-METHOD.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  NAME-LENGTH            PIC 9(4) COMP-5.
  01  PASSWORD-LENGTH        PIC 9(4) COMP-5.
  LINKAGE SECTION.
  01  NAME                   PIC X  ANY LENGTH.
  01  PASSWORD-DATA          PIC X  ANY LENGTH.
  01  AUTHENTICATION-RESULT  PIC X  ANY LENGTH.
  PROCEDURE DIVISION USING  NAME PASSWORD-DATA
                     RETURNING  AUTHENTICATION-RESULT.
      COMPUTE NAME-LENGTH      = FUNCTION LENGTH(NAME).
      COMPUTE PASSWORD-LENGTH  = FUNCTION LENG(PASSWORD-DATA).
*>  :
  END METHOD  AUTHENTICATION-METHOD.
  END OBJECT.
  END CLASS   AUTHENTICATION-CLASS.
```

When a general-purpose class is created, because abstract classes have a greater effect on interface changes, it is important to be able to create a general-purpose class without recognizing the maximum length.

Chapter 17 Database (SQL)

This chapter covers remote database access (ODBC). It explains how to write embedded SQL in a COBOL program and access a database with an ODBC driver.

The database (SQL) function accesses a database on a server from a PC client using embedded SQL. Embedded SQL is a database manipulation language written in a COBOL source program. The database function enables distributed development of a variety of application types.

There are two methods of database access from a COBOL application using a SQL statement:

- Pre-compiler in the database offer origin

- ODBC interface

In the database from which the precompiler is being offered, the precompiler that can use the function of SQL statement, performance, etc. is recommended.

The database from which the pre-compiler is not being offered uses the ODBC interface.

ODBC (Open DataBase Connectivity), proposed by Microsoft Corporation, is an application program interface for database access.

Topics covered are:

- ODBC Outline

- Connection

- Manipulating Data

- Advanced Data Manipulation

- Calling a Stored Procedure

- Accessing a Database Using Object-Oriented Programming Functions

- Deadlock Exits

- Compiling/Linking the Program

- Executing the Program

- Embedded SQL Keyword List

- Correspondence Between ODBC-Handled Data and COBOL-Handled Data

- SQLSTATE, SQLCODE, and SQLMSG

- Notes on Using the ODBC Driver

17.1 Access that uses precompiler

For database for which a precompiler can be used, refer to "SOFTWARE RELEASAE GUIDE". The use of a precompiler is different for each database. For details, refer to the precompiler manuals of each database.

17.2 Access with an ODBC

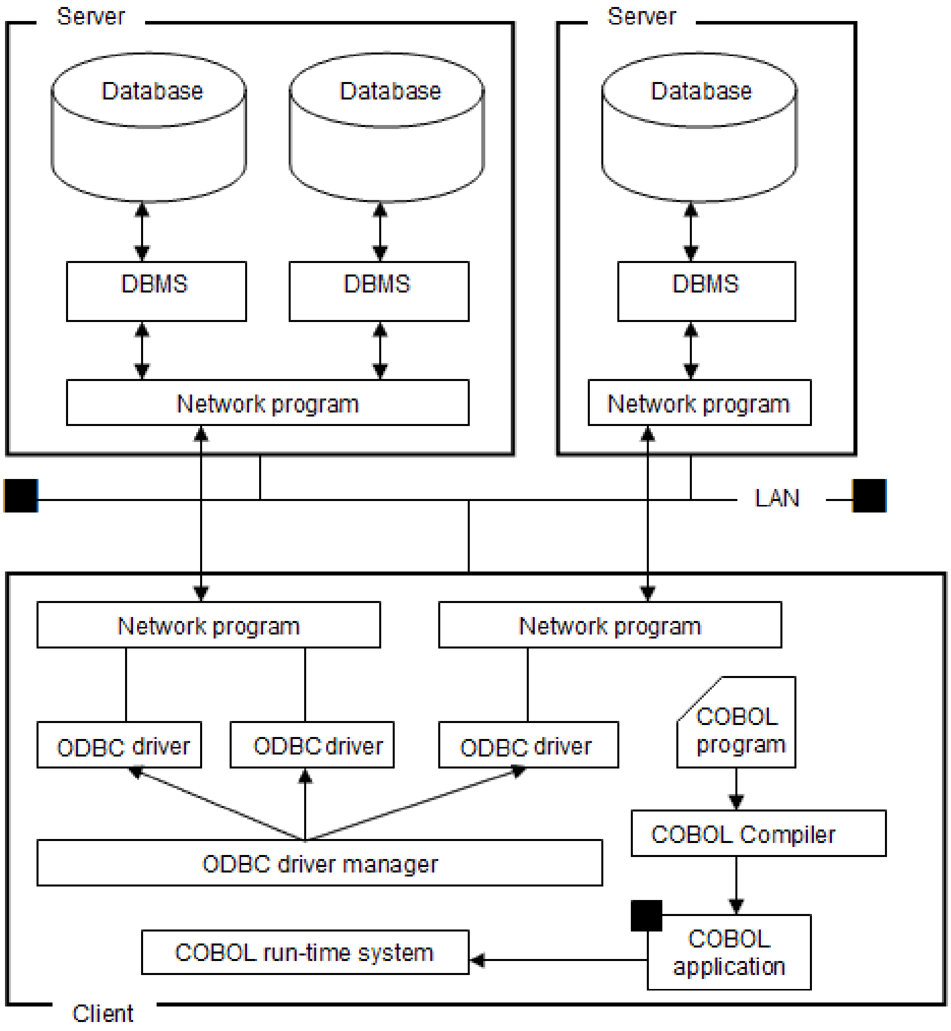
This section describes database access with an ODBC driver from a COBOL application to which embedded SQL is written.

17.2.1 ODBC Outline

This section outlines database access with an ODBC driver from a COBOL application. For an example of using the ODBC driver, refer to Sample10 supplied with NetCOBOL.

The ODBC driver enables access to one or more databases.

Figure 17.1 Outline of database access using the ODBC driver



17.2.1.1 Configuration of a COBOL Program with SQL

"Configuration of a COBOL program" shows the configuration of a COBOL program with SQL:

Configuration of a COBOL program

```
 IDENTIFICATION  DIVISION.
*>        :
 ENVIRONMENT DIVISION.                              *>...(1)
*>        :
 DATA DIVISION.
*>        :
 WORKING-STORAGE SECTION.
*>        :
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.       *>...(2)
 01  SQLSTATE    PIC X(5).
*>        :
     EXEC SQL END DECLARE SECTION END-EXEC.
*>        :
 PROCEDURE DIVISION.
*>        :
     EXEC SQL CONNECT ... END-EXEC.                 *>...(3)
*>        :
     EXEC SQL DECLARE CUR1 ... END-EXEC.            *>...(4)
*>        :
     EXEC SQL OPEN CUR1 END-EXEC.                    *>...(5)
*>        :
     EXEC SQL FETCH CUR1 ... END-EXEC.
*>        :
     EXEC SQL CLOSE CUR1 END-EXEC.                   *>...(6)
*>        :
     EXEC SQL ROLLBACK WORK END-EXEC.
*>        :
     EXEC SQL DISCONNECT ... END-EXEC.               *>...(7)
*>        :
     STOP RUN.
```

Description of Figure

- (1) No SQL-specific description.
- (2) Declare SQLSTATE in the DECLARE section. Define a host variable if necessary.
- (3) Connect to the server.
- (4) Declare the cursor, if using one.
- (5) Open the cursor.
- (6) Close the cursor.
- (7) Disconnect from the server.

Write SQL in a COBOL program as shown in figure above. When describing each SQL statement in a COBOL program, each SQL statement must begin with EXEC SQL (SQL prefix) and end with END-EXEC (SQL terminator). The database is processed using the SQL statements written in the PROCEDURE DIVISION.

17.2.1.2 Operations Using Embedded SQL

The following operations can be executed with embedded SQL statements:

Connection

A client connects to a server to access a database. The connection enables execution of SQL statements for accessing a server database from the client. To connect a client and server, use the CONNECT statement. See "17.2.2.1 Connecting to a Server".

Selecting a connection

To select connections, use the SET CONNECTION statement. See "17.2.2.3 Selecting a Connection".

Manipulating data

Data manipulation means the setup of application program data in the database and the reference to the database data. A single line of data or multiple lines of data of the database can be manipulated.

To manipulate data, use the SELECT, INSERT, UPDATE, and DELETE statements. If the cursor is defined, the FETCH statement can be used to fetch data. These statements can be executed dynamically. See "17.2.3 Manipulating Data", "17.2.4 Advanced Data Manipulation".

Note

For correspondence between ODBC-handled data and COBOL-handled data, see "17.2.11 Correspondence Between ODBC-Handled Data and COBOL-Handled Data".

Calling a stored procedure

To call a stored procedure, use the CALL statement.

The stored procedure to be called must have been registered on the database.

See "17.2.5 Calling a Stored Procedure".

Note

For correspondence between ODBC-handled data and COBOL-handled data, see "17.2.11 Correspondence Between ODBC-Handled Data and COBOL-Handled Data".

Transaction processing

Consistency of database data manipulation is ensured within transaction units. A transaction starts when the first SQL statement is executed, and terminates when the COMMIT or ROLLBACK statement is executed.

Disconnection

To disconnect a program from the server, use the DISCONNECT statement.

Before executing the DISCONNECT statement, terminate the transaction. See "17.2.2.2 Disconnecting from a Server."

The following sections provide examples of the above operations.

17.2.2 Connection

This section explains methods for connecting, disconnecting, and selecting a server (connection).

17.2.2.1 Connecting to a Server

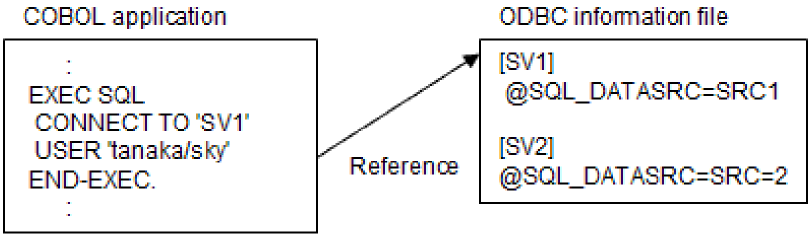
To connect a client to a server, use the CONNECT statement. The following steps should be used for Client/Server connection:

1. Define server information.
2. Connect to the server by either of the following methods:
   - Specify the server name
   - Specify DEFAULT

Connecting by Specifying the Server Name

Before executing the program, define the server information in the ODBC information file. For details on the information to be defined and how to define the information, see "17.2.8 Executing the Program".

Specify the server name in the CONNECT statement, and execute the statement. This will result in the ODBC information file being searched for the specified server information. If a section is found having the same name, the server information is referenced in definition order to establish a connection to the server.



```
     :
EXEC SQL
 CONNECT TO 'SV1'
 USER 'tanaka/sky'
END-EXEC.
     :
```

```
[SV1]
 @SQL_DATASRC=SRC1

[SV2]
@SQL_DATASRC=SRC=2
```

Description of Figure

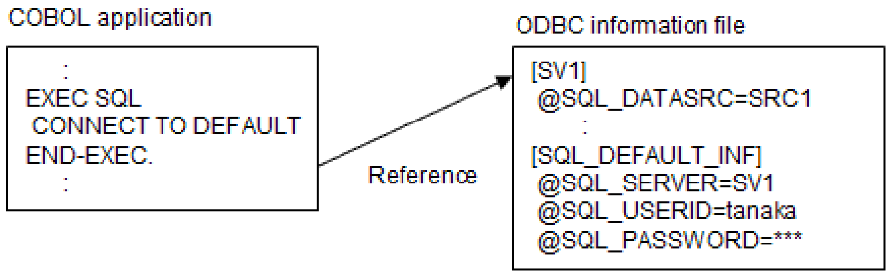
The server information in the ODBC information file is referenced at execution of the CONNECT statement when connecting to the server.

Connecting by Specifying DEFAULT

Before executing the program, define the default connection information in the ODBC information file. For details on definition information, and how to define it, see "17.2.8 Executing the Program".

Specify DEFAULT in the CONNECT statement and execute the statement. This will result in the default connection information file for ODBC to be referenced. The server information is defined in the default connection information. The ODBC information file is searched for the server information. If the server information is found, it is referenced when connecting to the server.

The default connection information is referenced at execution of the CONNECT statement.



```
    :
EXEC SQL
 CONNECT TO DEFAULT
END-EXEC.
    :
```

```
[SV1]
 @SQL_DATASRC=SRC1
      :
[SQL_DEFAULT_INF]
 @SQL_SERVER=SV1
 @SQL_USERID=tanaka
 @SQL_PASSWORD=***
```

Description of Figure

The SV1 information defined in the default connection information is referenced to connect the server.

17.2.2.2 Disconnecting from a Server

To disconnect a client from a server, use the DISCONNECT statement.

Specify the connection to be terminated in the DISCONNECT statement. The connection name, DEFAULT, CURRENT, or ALL can be specified in the DISCONNECT statement. For each use, see "Example of connecting/ selecting/ disconnecting multiple servers".

17.2.2.3 Selecting a Connection

To select the available connections, use the SET CONNECTION statement.

More than one server can be connected by executing the CONNECT statement for each database to be connected. If an application connects more than one server, each server for which SQL statements are to be executed must be defined.

If more than one CONNECT statement is executed, the server connected with the last CONNECT statement will be used as the current server (current connection).

SQL statements are executed for the current connection. To execute SQL statements for another connection, use the SET CONNECTION statement to select the appropriate connection.

"Example of connecting/ selecting/ disconnecting multiple servers" shows a COBOL program that illustrates the connection to multiple servers. It shows the selection of a connection, connecting and disconnecting of these multiple servers. The example has a default connection, and connection to SQL servers SV1, SV2, SV3, and SV4 as defined in the ODBC information file.

Example of connecting/ selecting/ disconnecting multiple servers

```
*>         :
     EXEC SQL CONNECT TO DEFAULT END-EXEC.           *>...(1)
     EXEC SQL
       CONNECT TO 'SV1' AS 'CNN1' USER 'tanaka/sky'  *>...(2)
     END-EXEC.
     EXEC SQL
       CONNECT TO 'SV2' AS 'CNN2'  USER 'tanaka/sky' *>...(3)
     END-EXEC.
     EXEC SQL
       CONNECT TO 'SV3' AS 'CNN3' USER 'tanaka/sky'  *>...(4)
     END-EXEC.
     EXEC SQL
       CONNECT TO 'SV4' AS 'CNN4' USER 'tanaka/sky'  *>...(5)
     END-EXEC.
     EXEC SQL DISCONNECT 'CNN4' END-EXEC.            *>...(6)
     EXEC SQL SET CONNECTION 'CNN1' END-EXEC.        *>...(7)
*>         :
     EXEC SQL ROLLBACK WORK END-EXEC.                *>...(8)
     EXEC SQL DISCONNECT CURRENT END-EXEC.           *>...(9)
     EXEC SQL SET CONNECTION DEFAULT END-EXEC.       *>...(10)
*>          :
     EXEC SQL COMMIT WORK END-EXEC.                  *>...(11)
     EXEC SQL DISCONNECT DEFAULT END-EXEC.           *>...(12)
     EXEC SQL SET CONNECTION 'CNN2' END-EXEC.
*>          :
     EXEC SQL ROLLBACK WORK END-EXEC.
     EXEC SQL DISCONNECT ALL END-EXEC.               *>...(13)
```

Description of Figure

- (1) The default connection is enabled.
- (2) The server SV1 is connected. This connection name is CNN1.
- (3) The server SV2 is connected. This connection name is CNN2.
- (4) The server SV3 is connected. This connection name is CNN3.
- (5) The server SV4 is connected. This connection name is CNN4. The last-connected CNN4 is the current connection.
- (6) The connection name CNN4 is disconnected.
- (7) The connection name CNN1 is selected. CNN1 becomes the current connection.
- (8) Any changes made in the CNN1 are canceled.
- (9) The current connection is disconnected. CNN1 is now the current connection in the example, so CNN1 is disconnected.
- (10) Default connection is selected. The default connection becomes the current connection.
- (11) Any changes made in the default connection are saved.
- (12) The default connection is disconnected.
- (13) All available connections are disconnected.

Note

Generally, up to 128 servers can be connected using the CONNECT statement at the same time. The number can vary depending on the ODBC driver and the related environments.

17.2.3 Manipulating Data

This section explains the following data manipulations:

- Retrieving data
- Updating data
- Deleting data
- Inserting data
- Using dynamic SQL
- Using variable length character strings
- Operating the cursor with more than one connection

Also, see "17.2.11 Correspondence Between ODBC-Handled Data and COBOL-Handled Data".

17.2.3.1 Sample Database

The following three sample database tables are used in the program examples that follow:

STOCK table

Shows product numbers (GNO), product names (GOODS), quantity in stock (QOH), and warehouse numbers (WHNO).

GNO	GOODS	QOH	WHNO
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
137	RADIO	150	2
138	RADIO	200	2
140	CASSETTE DECK	120	2
141	CASSETTE DECK	80	2
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
215	VIDEO	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2
380	SHAVER	870	3
390	DRIER	540	3

ORDERS table

Shows order numbers (ORDERID), company numbers (COMPANYNO), product trade numbers (GOODSNO), purchase prices (PRICE), and order quantities (OOH).

ORDERID	COMPANY NO.	GOODSNO	PRICE	OOH
1	61	123	48000	60
2	61	124	64000	40
3	61	138	6400	180
4	61	140	9000	80
5	61	215	240000	10
6	61	240	80000	20
7	62	110	37500	120
8	62	226	112500	20
9	62	351	375	800
10	63	111	57400	80
11	63	200	123000	60

ORDERID	COMPANY NO.	GOODSNO	PRICE	OOH
12	63	201	164000	50
13	63	212	205000	30
14	63	215	246000	10
15	71	140	7800	50
16	71	351	390	600
17	72	137	3500	120
18	72	140	7000	70
19	72	215	210000	10
20	72	226	105000	20
21	72	243	84000	10
22	72	351	350	1000
23	73	141	16000	60
24	73	380	2400	250
25	73	390	2400	150
26	74	110	39000	120
27	74	111	54000	120
28	74	226	117000	20
29	74	227	140400	10
30	74	351	390	700

COMPANY table

Shows company numbers (CNO), company names (NAME), telephone numbers (PHONE), and address (ADDRESS).

CNO	NAME	PHONE	ADDRESS
61	ADAM LTD.	731-1111	SANTA CLARA CA USA
62	IDEA INC.	423-222	LONDON W.C.2 ENGLAND
63	MOON CO.	143-3333	FIFTH AVENUE NY USA
71	RIVER CO.	344-1212	PARIS FRANCE
72	DRAGON CO.	373-7777	SAN FRANCISCO CA USA
73	BIG INC.	391-0808	DALLAS TX USA
74	FIRST CO.	255-9944	SYDNEY AUSTRALIA

17.2.3.2 Retrieving Data

This section explains methods for retrieving data from a database.

17.2.3.2.1 Retrieving Data from All Table Rows

How to retrieve data from all table rows in the database is explained below.

"Retrieving data from all rows" shows a sample COBOL program that retrieves data from all rows of the STOCK table in the sample database.

Retrieving data from all rows

```
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.          *>-+
 01  STOCK-LIST.                                        *> |
   02  PRODUCT-NUMBER    PIC S9(4) COMP-5.              *> |
   02  PRODUCT-NAME      PIC X(20).                     *> |(1)
   02  QUANTITY-IN-STOCK PIC S9(9) COMP-5.              *> |
   02  WAREHOUSE-NUMBER  PIC S9(4) COMP-5.              *> |
 01  SQLSTATE            PIC X(5).                      *> |
     EXEC SQL END DECLARE SECTION END-EXEC.             *>-+
 PROCEDURE DIVISION.
     EXEC SQL WHENEVER NOT FOUND GO TO :P-END END-EXEC. *> (2)
     EXEC SQL
       DECLARE CUR1 CURSOR FOR SELECT * FROM STOCK      *> (3)
     END-EXEC.
 P-START.
     EXEC SQL CONNECT TO DEFAULT END-EXEC.              *> (4)
     EXEC SQL OPEN CUR1 END-EXEC.                       *> (5)
 P-LOOP.
     EXEC SQL
       FETCH CUR1 INTO :STOCK-LIST                      *> (6)
     END-EXEC.
*>         :
     GO TO P-LOOP.
 P-END.
     EXEC SQL CLOSE CUR1 END-EXEC.                      *> (7)
     EXEC SQL ROLLBACK WORK END-EXEC.                   *> (8)
     EXEC SQL DISCONNECT DEFAULT END-EXEC.              *> (9)
     STOP RUN.
```

Description of Figure

- (1) Specifies an embedded SQL DECLARE section in the WORKING-STORAGE section, and defines the data input areas (all columns of the STOCK table) as host variables. Refer to the "COBOL Language Reference" for the host variable declaration rules.
- (2) Operation for an exception event can be specified by specifying the embedded SQL exception declaration. The example specifies NOT FOUND as a condition. Therefore, the exception declaration is effective if "no data" is shown as the SQLSTATE value. The example specifies executing P-END procedure when there is no row fetched by the FETCH statement in (6). Operation for an exception event can also be specified by checking SQLSTATE with a COBOL IF statement.
- (3) Declares a cursor to define the cursor name for referring to the STOCK table. The example neither selects any specific columns from the STOCK table nor specifies any search conditions. Thus, a table derived from the query expression is identical to the original STOCK table.
- (4) Executes the CONNECT statement to connect the server.
- (5) Executes the OPEN statement to enable the specified cursor.
- (6) Executes the FETCH statement to fetch data row by row from the table, and sets the values of each column into the corresponding host variable area.
- (7) Executes the CLOSE statement to disable the specified cursor.
- (8) Executes the ROLLBACK statement to terminate the transaction.
- (9) Executes the DISCONNECT statement to disconnect the server.

If an asterisk (*) is specified in the select list of the query expression, the columns are selected in the order specified when the table was defined.

In the example of data retrieval from all rows (shown in "Retrieving data from all rows"), the host variable of multiple columns specified is used. This variable must be defined as a slave entry to a single group item of host variables corresponding to each column of the database. It can be referred to as the group item name when referred to by the embedded SQL statement. This is one of the abbreviated format specifications and has the same results as the following host variable definition or reference.

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*>       :
  01 PRODUCT-NUMBER    PIC S9(4) COMP-5.
  01 PRODUCT-NAME      PIC X(20).
  01 QUANTITY-NAME     PIC S9(9) COMP-5.
  01 WAREHOUSE-NUMBER  PIC S9(4) COMP-5.
  01 SQLSTATE          PIC X(5).
*>       :
     EXEC SQL END DECLARE SECTION END-EXEC.
 PROCEDURE DIVISION.
*>       :
     EXEC SQL
       FETCH  CUR1
         INTO :PRODUCT-NUMBER, :PRODUCT-NAME,
              :QUANTITY-NAME, :WAREHOUSE-NUMBER
     END-EXEC.
*>       :
```

17.2.3.2.2 Retrieving Data with Conditions Specified

The following illustrates retrieving data only for rows that meet specified conditions. In "Retrieving data with conditions specified", the condition of the quantity in stock (QOH) is tested to be less than 51. It retrieves the data (product number, product name, and quantity in stock) of products that meet the condition.

Retrieving data with conditions specified

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.            *>-+
01  STOCK-LIST.                                          *> |
   02  PRODUCT-NUMBER    PIC S9(4) COMP-5.               *> |
   02  PRODUCT-NAME      PIC X(20).                      *> |(1)
   02  QUANTITY-IN-STOCK PIC S9(9) COMP-5.               *> |
 01  SQLSTATE PIC X(5).                                  *> |
     EXEC SQL END DECLARE SECTION END-EXEC.              *>-+
 PROCEDURE DIVISION.
     EXEC SQL WHENEVER NOT FOUND GO TO :P-END END-EXEC.
     EXEC SQL
       DECLARE CUR2 CURSOR FOR
       SELECT GNO, GOODS, QOH FROM STOCK
         WHERE QOH <= 50
     END-EXEC.                                           *>(2)
 P-START.
     EXEC SQL CONNECT TO DEFAULT END-EXEC.               *>(3)
     EXEC SQL OPEN CUR2 END-EXEC.                        *>(4)
 P-LOOP.
     EXEC SQL
       FETCH CUR2 INTO :STOCK-LIST                       *>(5)
     END-EXEC.
*>         :
     GO TO P-LOOP.
 P-END.
     EXEC SQL CLOSE CUR2 END-EXEC.                       *>(6)
     EXEC SQL ROLLBACK WORK END-EXEC.                    *>(7)
     EXEC SQL DISCONNECT DEFAULT END-EXEC.               *>(8)
     STOP RUN.
```

Description of Figure

- (1) The embedded SQL DECLARE section defines all host variables to be specified in embedded SQL statements.
- (2) The table is not opened by declaring a cursor. The table is made accessible when the OPEN statement in (4) is executed.
- (3) Executes the CONNECT statement to connect the server.

- (4) Executes the OPEN statement to create a virtual table including rows that meet the condition specified during cursor declaration in (2). The virtual table is created and rows showing 50 or less in column QOH are extracted from the STOCK table, then the data of columns GNO, GOODS, and QOH are retrieved from the extracted rows. Generally, the row order of a virtual table is undefined.
- (5) Executes the FETCH statement to fetch data row by row from the beginning of the virtual table created in (4), then writes the values of each column to the corresponding host variable area. To retrieve the values of a column in ascending or descending order, specify the ORDER BY clause at the end of the query expression in (2).
- (6) Executes the CLOSE statement to disable the virtual table created in (4). The virtual table can no longer be referenced by SQL statements unless the OPEN statement is executed again.
- (7) Executes the ROLLBACK statement to terminate the transaction.
- (8) Executes the DISCONNECT statement to disconnect the server.

17.2.3.2.3 Retrieving Data from a Single Row

Sample programs in "Retrieving data from all rows" and "Retrieving data with conditions specified" assume that data from multiple table rows is retrieved.

Use the SELECT statement if retrieving the data for only one row. No cursor is used in this case, and cursor declaration and processing using the OPEN, FETCH, and CLOSE statements is unnecessary.

The following is the SELECT statement used to retrieve the product name where the quantity in stock for the product is 200 as noted in the STOCK table:

```
    EXEC SQL
      SELECT GOODS, QOH INTO :PRODUCT-NAME,:QUANTITY-IN-STOCK
        FROM  STOCK
        WHERE GNO = 200
    END-EXEC.
```

If a set function is specified as the select list value expression, the SELECT statement obtains the radix of the table (the number of rows) and the maximum, minimum, average, and summation (total) of the specified value expression.

The following is the SELECT statement used to retrieve the maximum, minimum, and average of the purchase prices in the ORDERS table:

```
    EXEC SQL
      SELECT MAX(PRICE), MIN(PRICE), AVG(PRICE)
        INTO :MAX-VALUE, :MIN-VALUE, :AVG-VALUE FROM ORDERS
    END-EXEC.
```

The results are set in the host variables MAX-VALUE, MIN-VALUE, and AVG-VALUE.

17.2.3.2.4 Retrieving Data from Related Tables

Retrieving Data from a Table Created by Relating Different Tables

Data can be retrieved from a table created by relating different tables. The tables are related according to their column values.

The following example relates three tables of the sample database in order to retrieve data. The example specifies TELEVISION as a target product name, and retrieves the names of companies that deal with the specified product and the order quantity of each company.

- The STOCK and ORDERS tables are related according to the product numbers (column GNO) and product trade numbers (column GOODSNO).
- The ORDERS and COMPANY tables are related according to the company numbers (column COMPANYNO) and company numbers (column CNO).

The following shows the query expression:

```
    SELECT NAME, OOH
      FROM STOCK, ORDERS, COMPANY
      WHERE GOODS = 'TELEVISION' AND
```

```
             GNO = GOODSNO        AND
             COMPANYNO = CNO
```

Rows that meet the search conditions are derived from the table created by relating three tables:

GNO	...	WHNO	COMPANYNO	...	OOH	CNO	...	ADDRESS
110	...	2	61	...	60	61	...	SANTA CLARA CA USA
110	...	2	61	...	60	62	...	LONDON W.C.2 ENGLAND
110	...	2	61	...	60	63	...	FIFTH AVENUE NY USA
~	~	~	~	~	~	~	~	~
110	...	2	61	...	40	61	...	SANTA CLARA CA USA
110	...	2	61	...	40	62	...	LONDON W.C.2 ENGLAND
110	...	2	61	...	40	63	...	FIFTH AVENUE NY USA
~	~	~	~	~	~	~	~	~
111	...	2	61	...	60	61	...	SANTA CLARA CA USA
111	...	2	61	...	60	62	...	LONDON W.C.2 ENGLAND
111	...	2	61	...	60	63	...	FIFTH AVENUE NY USA
~	~	~	~	~	~	~	~	~
390	...	3	74	...	700	72	...	SAN FRANCISCO CA USA
390	...	3	74	...	700	73	...	DALLAS TX USA
390	...	3	74	...	700	74	...	SYDNEY AUSTRALIA

Yielding the results of the query expression:

NAME	OOH
IDEA INC.	120
MOON CO.	80
FIRST CO.	120
FIRST CO.	120

Retrieving Data from a Table Where Rows are Related

Data can be retrieved from a table where rows are related in the same manner as when different tables are related.

The following example retrieves the names of products that are in the warehouse where televisions (TELEVISION) are stored. Two different aliases (correlation names) are given to the STOCK table. They are treated as if they were different tables.

"Retrieving data from a table where rows are related" shows a sample COBOL program that relates rows in a table and retrieves the name of the product stocked in the same warehouse as stocks product name TELEVISION.

Retrieving data from a table where rows are related

```
*>       :
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PRODUCT-NAME   PIC X(20).
01  SQLSTATE       PIC X(5).
    EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
    EXEC SQL WHENEVER NOT FOUND GO TO :P-END END-EXEC.
    EXEC SQL
      DECLARE CUR4  CURSOR FOR
```

```
         SELECT DISTINCT X2.GOODS
         FROM STOCK X1,STOCK X2                          *>(1)
         WHERE X1.GOODS = 'TELEVISION' AND
               X1.WHNO  = X2.WHNO
     END-EXEC.
 P-START.
     EXEC SQL CONNECT TO DEFAULT END-EXEC.
     EXEC SQL OPEN CUR4 END-EXEC.
 P-LOOP.
     EXEC SQL
       FETCH CUR4 INTO :PRODUCT-NAME
     END-EXEC.
*>       :
     GO TO P-LOOP.
 P-END.
     EXEC SQL CLOSE CUR4 END-EXEC.
     EXEC SQL ROLLBACK WORK END-EXEC.
     EXEC SQL DISCONNECT DEFAULT END-EXEC.
     STOP RUN.
```

Description of Figure

- (1) Specifies correlation names (X1 and X2) for the STOCK table in the FROM clause of the cursor declaration statement. X1 and X2 are treated as if they are different tables. The example specifies search conditions in the WHERE clause. The column name is qualified with the correlation names. The rows that result show products that are in the warehouse storing televisions (TELEVISION).

17.2.3.3 Updating Data

Use the UPDATE statement to update table data. The following example specifies the UPDATE statement to decrement each quantity in stock (column QOH) of the STOCK table by 10%:

```
EXEC SQL
  UPDATE STOCK SET QOH = QOH * 0.9
END-EXEC.
```

This UPDATE statement multiplies each value in column QOH of the STOCK table by 0.9, and replaces the original value with the result.

To update values in only rows that meet a specified condition, specify a search condition in the WHERE clause of the UPDATE statement.

The following example changes the above UPDATE statement to decrement only the number of televisions in stock by 10%:

```
EXEC SQL
  UPDATE STOCK SET QOH = QOH * 0.9
    WHERE GOODS = 'TELEVISION'
END-EXEC.
```

Note

This method cannot update data in a table created by relating multiple tables.

17.2.3.4 Deleting Data

Use the DELETE statement to delete data. The following example specifies the DELETE statement to delete all rows showing CASSETTE DECK from the STOCK table:

```
EXEC SQL
  DELETE FROM STOCK WHERE GOODS = 'CASSETTE DECK'
END-EXEC.
```

Note

This method cannot delete data from a table created by relating multiple tables.

17.2.3.5 Inserting Data

Use the INSERT statement to insert data. Select either of the following methods to insert data:

- Inserting the data in only one row
- Inserting the data in a set of rows extracted from another table based on search conditions

Inserting a Single Row

The following example specifies the INSERT statement to add a row showing product number 301 to the STOCK table. The product name is WASHER, the quantity in stock is 50, and the warehouse number is 1:

```
EXEC SQL
  INSERT INTO STOCK (GNO,GOODS, QOH, WHNO)
    VALUES (301, 'WASHER', 50, 1)
END-EXEC.
```

Inserting Multiple Rows from Another Table

The following example assumes that the database of the STOCK table contains another stock table (table name: SUBSTOCK, column names and attributes: same as those of the STOCK table).

The example specifies the INSERT statement to insert rows showing product name MICROWAVE OVEN in the SUBSTOCK table into the STOCK table.

The example also sets the warehouse number of each new row to 2:

```
EXEC SQL
  INSERT INTO STOCK (GNO, GOODS, QOH, WHNO)
    SELECT GNO, GOODS, QOH, 2 FROM SUBSTOCK
    WHERE GOODS = 'MICROWAVE OVEN'
END-EXEC.
```

Note

If a data manipulation without using a cursor is executed for a table opened with a cursor, an error may occur. The error message depends on the database and the configuration. If the message is displayed, confirm the specification of the database.

17.2.3.6 Using Dynamic SQL

To generate SQL statements during program execution and to execute the statements, use dynamic SQL.

Determining Search Conditions Dynamically

The previous examples use host variables to set search conditions at execution. The following explains a method of directly determining search conditions at execution.

"Specifying search conditions at execution" shows a sample COBOL program that dynamically executes SQL statements that have been entered using an ACCEPT statement.

```
SELECT GNO, GOODS, QOH FROM STOCK
  WHERE GOODS = 'REFRIGERATOR' AND QOH < 10
```

The ACCEPT statement is used to read the query expression at execution.

Specifying search conditions at execution

```
 *>       :
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  STOCK-LIST.
   02  PRODUCT-NUMBER     PIC S9(4) COMP-5.
   02  PRODUCT-NAME       PIC X(20).
   02  QUANTITY-IN-STOCK  PIC S9(9) COMP-5.
 01  STMVAR             PIC X(254).                            *>(1)
 01  SQLSTATE           PIC X(5).
     EXEC SQL END DECLARE SECTION END-EXEC.
 PROCEDURE DIVISION.
     EXEC SQL WHENEVER NOT FOUND GO TO :P-END END-EXEC.
     EXEC SQL
       DECLARE CUR8 CURSOR FOR STMIDT                          *>(2)
     END-EXEC.
     ACCEPT STMVAR FROM CONSOLE.                               *>(3)
 P-START.
     EXEC SQL CONNECT TO DEFAULT END-EXEC.
     EXEC SQL PREPARE STMIDT FROM :STMVAR END-EXEC.            *>(4)
     EXEC SQL OPEN CUR8 END-EXEC.                              *>(5)
 P-LOOP.
     EXEC SQL
       FETCH CUR8 INTO :STOCK-LIST                             *>(6)
     END-EXEC.
*>        :
     GO TO P-LOOP.
 P-END.
     EXEC SQL CLOSE CUR8 END-EXEC.                             *>(7)
     EXEC SQL ROLLBACK WORK END-EXEC.
     EXEC SQL DISCONNECT DEFAULT END-EXEC.
     STOP RUN.
```

Description of Figure

- (1) The SQL statement variable STMVAR is referred to when the PREPARE statement in (4) is executed.
- (2) The SQL statement identifier STMIDT is corresponding to the SQL statement variable STMVAR when the PREPARE statement in (4) is executed.
- (3) Executes the ACCEPT statement to read the query expression, and sets the read data into the SQL statement variable STMVAR.
- (4) Executes the PREPARE statement corresponding to the statement (dynamic SELECT statement in the example) and set in the SQL statement variable STMVAR of the SQL statement identifier STMIDT.
- (5) Executes the dynamic OPEN statement to extract, from the STOCK table, rows showing a value less than 10 as the number of refrigerators in stock, then creates a table including the data of columns GNO (product number), GOODS (product name), and QOH (quantity in stock) that is retrieved from the extracted rows.
- (6) Executes the dynamic FETCH statement to fetch data row by row from the table, and sets the values of each column into the corresponding host variable area.
- (7) Executes the dynamic CLOSE statement to disable the specified cursor and the table corresponding to the cursor.

Determining SQL Statements Dynamically

Dynamic SQL not only determines search conditions in a query expression through cursor declaration, but also dynamically determines the SQL statements to be executed there. The following explains methods of determining the SQL statements using the EXECUTE statement.

"Dynamically determining SQL statements (1)" is a COBOL program used to dynamically execute SQL statements input with the ACCEPT statement. The example inputs the UPDATE statement at execution:

```
     UPDATE STOCK SET QOH = 0 WHERE GOODS = 'TELEVISION'
```

Dynamically determining SQL statements (1)

```
*>        :
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  STMVAR      PIC X(254).
 01  SQLSTATE    PIC X(5).
     EXEC SQL END DECLARE SECTION END-EXEC.
 PROCEDURE DIVISION.
     ACCEPT STMVAR FROM CONSOLE.                    *>(1)
     EXEC SQL CONNECT TO DEFAULT END-EXEC.
     EXEC SQL PREPARE STMIDT FROM :STMVAR END-EXEC. *>(2)
     EXEC SQL EXECUTE STMIDT END-EXEC.              *>(3)
*>        :
     EXEC SQL ROLLBACK WORK END-EXEC.
     EXEC SQL DISCONNECT DEFAULT END-EXEC.
     STOP RUN.
```

Description of Figure

- (1) Executes the ACCEPT statement to read the UPDATE statement, and sets the read data into the SQL statement variable STMVAR.
- (2) Executes the PREPARE statement corresponding to the statement (the UPDATE statement in the example) set in the SQL statement variable STMVAR of the SQL statement identifier STMIDT.
- (3) The EXECUTE statement executes the prepared statement that is associated with the specified SQL statement identifier.

If parameter specification is not required for dynamically determining SQL statements, the EXECUTE IMMEDIATE statement can be used instead of the EXECUTE statement.

"Dynamically determining SQL statements (2)" shows a sample COBOL program that uses the EXECUTE IMMEDIATE statement to perform the same process as in "Dynamically determining SQL statements (1)". Upon execution, the same UPDATE statement as in "Dynamically determining SQL statements (1)" is input.

Dynamically determining SQL statements (2)

```
*>        :
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  STMVAR      PIC X(254).
 01  SQLSTATE    PIC X(5).
     EXEC SQL END DECLARE SECTION END-EXEC.
 PROCEDURE DIVISION.
     ACCEPT STMVAR FROM CONSOLE.                    *>(1)
     EXEC SQL CONNECT TO DEFAULT END-EXEC.
     EXEC SQL EXECUTE IMMEDIATE :STMVAR END-EXEC.   *>(2)
*>        :
     EXEC SQL ROLLBACK WORK END-EXEC.
     EXEC SQL DISCONNECT DEFAULT END-EXEC.
     STOP RUN.
```

Description of Figure

- (1) Executes the ACCEPT statement to read the UPDATE statement, then writes the read data to SQL statement variable STMVAR.
- (2) The EXECUTE IMMEDIATE statement directly executes the SQL statements written to the SQL statement variable.

Specifying Dynamic Parameters

The following COBOL program in "Specifying dynamic parameters" illustrates specifying dynamic parameters and retrieving data from the STOCK table. The example processes the SELECT statement at execution:

```
     SELECT GNO, GOODS FROM STOCK WHERE WHNO = ?
```

Specifying dynamic parameters

```
        :
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  PRODUCT-NUMBER      PIC S9(4) COMP-5.
```

```
01  PRODUCT-NAME        PIC X(20).
01  QUANTITY-IN-STOCK   PIC S9(9) COMP-5.
01  WAREHOUSE-NUMBER    PIC S9(4) COMP-5.
01  STMVAR              PIC X(254).
01  SQLSTATE            PIC X(5).
01  SQLMSG              PIC X(254).
    EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
    EXEC SQL WHENEVER NOT FOUND GO TO :P-END END-EXEC.
    EXEC SQL
      DECLARE CUR11 CURSOR FOR STMIDT               *>(1)
    END-EXEC.
    ACCEPT STMVAR FROM CONSOLE.                     *>(2)
P-START.
    EXEC SQL CONNECT TO DEFAULT END-EXEC.
    EXEC SQL PREPARE STMIDT FROM :STMVAR END-EXEC. *>(3)
    ACCEPT WAREHOUSE-NUMBER FROM CONSOLE.           *>(4)
    EXEC SQL OPEN CUR11 USING :WAREHOUSE-NUMBER
      END-EXEC.                                     *>(5)

 P-LOOP.
    EXEC SQL
      FETCH CUR11 INTO :PRODUCT-NUMBER,
                       :PRODUCT-NAME                *>(6)
    END-EXEC.
       :
    GO TO P-LOOP.
P-END.
    EXEC SQL CLOSE CUR11 END-EXEC.                  *>(7)
    EXEC SQL ROLLBACK WORK END-EXEC.
    EXEC SQL DISCONNECT DEFAULT END-EXEC.
    STOP RUN.
```

Description of Figure

- (1) Defines CUR11 through dynamic cursor declaration.
- (2) Executes the ACCEPT statement to read the dynamic SELECT statement.
- (3) Executes the PREPARE statement to correspond the statement set in the SQL statement variable STMVAR to the SQL statement identifier STMIDT.
- (4) Reads the search condition values corresponding to dynamic parameters.
- (5) Executes the dynamic OPEN statement to extract rows matching the search conditions from the table. The values specified in the USING clause are referred to as the values of the dynamic parameters in the prepared statement. The values specified in the USING clause and the dynamic parameters in the prepared statement are associated in the order they appear.
- (6) Executes the dynamic FETCH statement to fetch data row by row from the table, and then writes the values of each column to the host variables specified in the INTO clause.
- (7) Executes the dynamic CLOSE statement to disable the specified cursor and the table corresponding to the cursor.

Note

When using host variables with multiple columns specified in a dynamic parameter, prepare the SQL statement that can use this variable as the prepared statement.

If the variable is specified in the SQL statement that cannot support it, the operation is unreliable.

17.2.3.7 Using Variable Length Character Strings

This section explains how to use variable length character strings as the host variables of a COBOL program.

To operate variable length character string data, the length of the character string is needed. Host variables of a variable length character string type are defined as the following items:

- Signed binary data item for storing the character string length information.
- Group item (an alphanumeric data item or national data item) for storing character strings.

"Operating variable length character string data" Illustrates retrieving an address from the COMPANY table. The host variable used as the search condition for storing are variable length character string data.

Operating variable length character string data

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.             *>-+
01  COMPANY-NAME              PIC X(20).                 *> |
01  TELEPHONE-NUMBER.                                    *> |
  49 TELEPHONE-NUMBER-LENGTH  PIC S9(4) COMP-5.          *> |(1)
  49 TELEPHONE-NUMBER-STRING  PIC X(20).                 *> |
01  ADDR.                                                *> |
  49 ADDRESS-LENGTH           PIC S9(9) COMP-5.          *> |
  49 ADDRESS-STRING           PIC X(30).                 *> |
01  SQLSTATE                  PIC X(5).                  *> |
    EXEC SQL END DECLARE SECTION END-EXEC.               *>-+
PROCEDURE DIVISION.
    DISPLAY "Retrieves company according to telephone number."
    DISPLAY "Input telephone number. ->" WITH NO ADVANCING.
    ACCEPT TELEPHONE-NUMBER-STRING FROM CONSOLE.         *>(2)
    INSPECT TELEPHONE-NUMBER-STRING                      *>(3)
      TALLYING TELEPHONE-NUMBER-LENGTH
      FOR CHARACTERS BEFORE SPACE.
    EXEC SQL CONNECT TO DEFAULT END-EXEC.
    EXEC SQL
      SELECT NAME,ADDRESS INTO :COMPANY-NAME,
                               :ADDR
             FROM COMPANY
             WHERE PHONE = :TELEPHONE-NUMBER
    END-EXEC.                                            *>(4)
*>       :
    EXEC SQL ROLLBACK WORK END-EXEC.
    EXEC SQL DISCONNECT DEFAULT END-EXEC.
    STOP RUN.
```

Description of Figure

- (1) Declares variable length character strings as host variables.
- (2) Executes the ACCEPT statement to read the value of the host variable used as the search condition, and sets the value as TELEPHONE-NUMBER-STRING.
- (3) Sets the length of the read value as TELEPHONE-NUMBER-LENGTH.
- (4) Executes the SELECT statement (single row) to retrieve the row meeting the search condition from column ADDRESS. The length and value of the retrieved character string are set into the host variable "ADDRESS".

If the data of a host variable used as a search condition is a character string type or a national character string type, define the length of the host variable to be equal to or shorter than the length of the character string.

17.2.3.8 Operating the Cursor with More than One Connection

This section explains how to operate a cursor with more than one connection. "Operating a cursor with more than one connection " assumes there are two SQL servers, SV1 and SV2, and that tables with the same table name and format exist on each server.

Operating a cursor with more than one connection

```
*>       :
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  PRODUCT-NUMBER      PIC S9(4) COMP-5.
 01  PRODUCT-NAME        PIC X(20).
 01  QUANTITY-IN-STOCK   PIC S9(9) COMP-5.
 01  WAREHOUSE-NUMBER    PIC S9(4) COMP-5.
 01  SQLSTATE            PIC X(5).
     EXEC SQL END DECLARE SECTION END-EXEC.
 01  NEXTFLAG            PIC X(4) VALUE SPACE.
 PROCEDURE DIVISION.
     EXEC SQL DECLARE CUR9 CURSOR FOR                         *>(1)
                SELECT * FROM STOCK
     END-EXEC.
     EXEC SQL  WHENEVER NOT FOUND
               GO TO :P-NEXT END-EXEC.                        *>(2)
 P-START.
     EXEC SQL
       CONNECT TO 'SV1' AS 'CNN1' USER 'summer/w43'           *>(3)
     END-EXEC.
     EXEC SQL
       CONNECT TO 'SV2' AS 'CNN2' USER 'tanaka/sky'           *>(4)
     END-EXEC.
 P-CNN2-1.
     EXEC SQL OPEN CUR9 END-EXEC.                             *>(5)
     GO TO P-LOOP.
 P-NEXT.
     IF NEXTFLAG = "NEXT" THEN
       GO TO P-CNN1-2
     END-IF.
P-CNN1-1.
     EXEC SQL SET CONNECTION 'CNN1' END-EXEC.                 *>(7)
     EXEC SQL
       INSERT INTO STOCK
         VALUES(:PRODUCT-NUMBER, :PRODUCT-NAME,
                :QUANTITY-IN-STOCK, :WAREHOUSE-NUMBER)
     END-EXEC.                                                *>(8)
     EXEC SQL OPEN CUR9 END-EXEC.                             *>(9)
     MOVE "NEXT" TO NEXTFLAG.
     GO TO P-LOOP.
 P-CNN1-2.
     EXEC SQL CLOSE CUR9 END-EXEC.                            *>(10)
     EXEC SQL ROLLBACK WORK END-EXEC.
     EXEC SQL DISCONNECT 'CNN1' END-EXEC.
 P-CNN2-2.
     EXEC SQL SET CONNECTION 'CNN2' END-EXEC.                 *>(11)
     EXEC SQL CLOSE CUR9 END-EXEC.                            *>(12)
     EXEC SQL ROLLBACK WORK END-EXEC.
     EXEC SQL DISCONNECT CURRENT END-EXEC.
 P-END.
     STOP RUN.
 P-LOOP.
     EXEC SQL
       FETCH CUR9                                                *>(6)
         INTO :PRODUCT-NUMBER, :PRODUCT-NAME,
              :QUANTITY-IN-STOCK, :WAREHOUSE-NUMBER
     END-EXEC.
*>       :
     GO TO P-LOOP.
```

Description of Figure

- (1) Defines the cursor for retrieving data from the table.
- (2) Specifies branching to the procedure name P-NEXT if there is no row to be retrieved when the embedded SQL exception declaration is specified.
- (3) The server SV1 is connected. This connection name is CNN1.

- (4) The server SV2 is connected. This connection name is CNN2. CNN2 is the current connection.
- (5) Executes the OPEN statement in the server with CNN2 to enable the cursor CUR9.
- (6) Executes the FETCH statement to fetch data row by row from the table, then writes the values of each column to the corresponding host variable area.
- (7) Select the current connection to CNN1.
- (8) Executes the INSERT statement to insert a row into the STOCK table of the server with CNN1.
- (9) Executes the OPEN statement in the server with CNN1 to enable the cursor CUR9. The cursor CUR9 is treated as another cursor different from the cursor that was opened in the server with CNN2.
- (10) Executes the CLOSE statement in the server with CNN1 to disable the cursor CUR9.
- (11) Select the current connection to CNN2. Note that the cursor CUR9 is still enabled in the server with CNN2.
- (12) Executes the CLOSE statement in the server with CNN2 to disable the cursor CUR9.

17.2.4 Advanced Data Manipulation

This section explains the host variables that enable advanced data manipulation.

The explanation uses the STOCK table in "17.2.3.1 Sample Database", as a COBOL program sample.

17.2.4.1 Host variables that enable advanced data manipulation

As explained in the foregoing sections about data manipulation using embedded SQL statements, execution of one embedded SQL statement basically manipulates one row of data in the database table. However, an embedded SQL statement specifying one of the following host variables can manipulate multiple rows of data in the database table at a time.

- Host variable with multiple rows specified
- Host variable with a table specified

Using these variables can improve the performance of application programs.

Note

This function is dedicated to remote database access using ODBC. It must be disabled for development of applications that run on platforms other than Windows, such as for distributed development. Specify compiler option NOSQLGRP for compilation. See "A.2.47 SQLGRP(SQL host variable definition expansion)".

17.2.4.1.1 Host variable with multiple rows specified

Function

A host variable with multiple rows specified can manipulate multiple rows of data for one column in the table.

Definition

This variable is a basic item defined as a repetition item having an OCCURS clause. Refer to the *COBOL Reference Manual* for more information.

Usage

Retrieving data

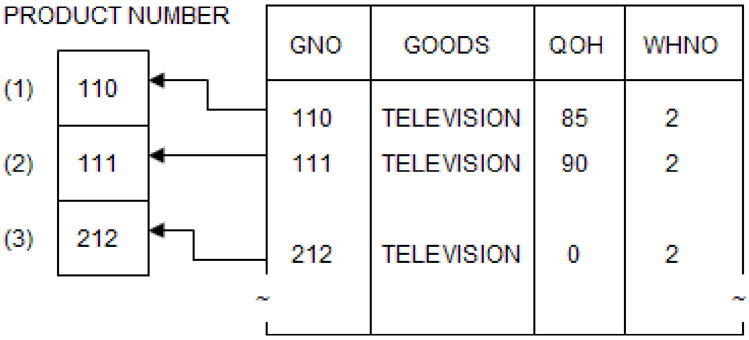
When a host variable with multiple rows specified is used for target specification, as many data items as the repetition count of the host variable with multiple rows specified can be retrieved.

The following COBOL program sample fetches three product numbers (GNO column) on the rows with product name TELEVISION from the STOCK table.

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  STOCK-LIST.
  02 PRODUCT-NUMBER  PIC S9(4) COMP-5 OCCURS 3 TIMES.    *>[1]
01 SQLSTATE          PIC X(5).
    EXEC SQL END  DECLARE SECTION END-EXEC.
*>       :
PROCEDURE DIVISION.
*>       :
    EXEC SQL
      SELECT GNO FROM STOCK
      INTO :PRODUCT-NUMBER
      WHERE GOODS = 'TELEVISION'
    END-EXEC.                                            *>[2]
*>       :
```

- [1] PRODUCT-NUMBER, which is a host variable with multiple rows specified, is defined by declaring the repetition count of the OCCURS clause.

- [2] When data is fetched to PRODUCT-NUMBER by the SELECT statement, 110 is stored in PRODUCT-NUMBER (1), 111 in PRODUCT-NUMBER (2), and 212 in PRODUCT-NUMBER (3).



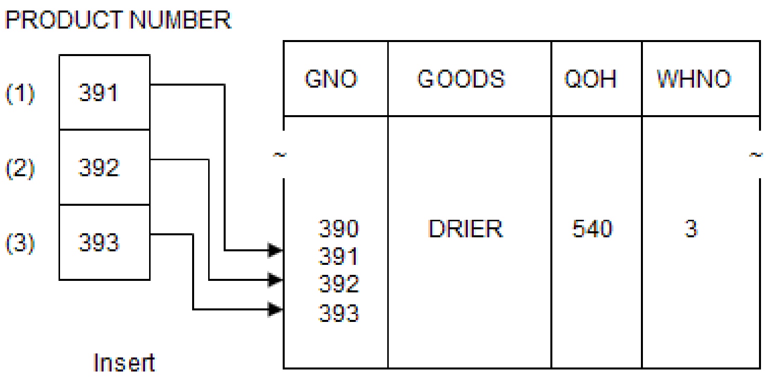
GNO	GOODS	QOH	WHNO
110	TELEVISION	85	2
111	TELEVISION	90	2
212	TELEVISION	0	2

Inserting data

When a host variable with multiple rows specified is used for value specification, as many data items as the repetition count of the host variable with multiple rows specified can be inserted.

The following COBOL program sample inserts three product numbers to the GNO column in the STOCK table.

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  STOCK-LIST.
   02 PRODUCT-NUMBER  PIC S9(4) COMP-5 OCCURS 3 TIMES.
 01 SQLSTATE  PIC X(5).
     EXEC SQL END  DECLARE SECTION END-EXEC.
*>       :
 PROCEDURE DIVISION.
       :
     MOVE 391 TO PRODUCT-NUMBER(1).
     MOVE 392 TO PRODUCT-NUMBER(2).
     MOVE 393 TO PRODUCT-NUMBER(3).
     EXEC SQL
       INSERT INTO STOCK(GNO)
       VALUES (:PRODUCT-NUMBER)
     END-EXEC.
*>       :
```



The INSERT statement is executed as many times as the repetition count of the host variable with multiple rows specified and the values stored in the host variable are inserted in order from the beginning to the table. The above sample program thus works the same as a program written as shown below:

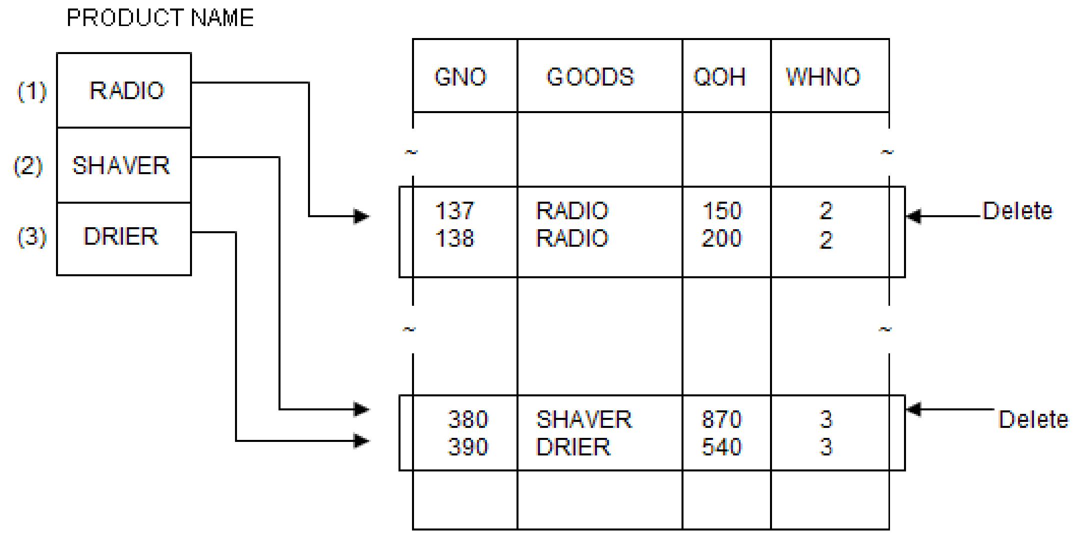
```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  PRODUCT-NUMBER  PIC S9(4) COMP-5.
 01  SQLSTATE        PIC X(5).
     EXEC SQL END  DECLARE SECTION END-EXEC.
*>       :
 PROCEDURE DIVISION.
*>       :
     MOVE 391 TO PRODUCT-NUMBER.
     EXEC SQL
       INSERT INTO STOCK(GNO) VALUES(:PRODUCT-NUMBER)
     END-EXEC.
     MOVE 392 TO PRODUCT-NUMBER.
     EXEC SQL
       INSERT INTO STOCK(GNO) VALUES(:PRODUCT-NUMBER)
     END-EXEC.
     MOVE 393 TO PRODUCT-NUMBER.
     EXEC SQL
       INSERT INTO STOCK(GNO) VALUES(:PRODUCT-NUMBER)
     END-EXEC.
*>       :
```

Deleting data

When a host variable with multiple rows specified is used for inquiry specification, data that meets multiple conditions can be deleted at a time.

The following COBOL program sample deletes data that meets three conditions from the STOCK table.

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  STOCK-LIST.
   02 PRODUCT-NAME  PIC X(20) OCCURS 3 TIMES.
 01  SQLSTATE       PIC X(5).
     EXEC SQL END  DECLARE SECTION END-EXEC.
*>       :
 PROCEDURE DIVISION.
*>       :
     MOVE "RADIO"  TO PRODUCT-NAME(1).
     MOVE "SHAVER" TO PRODUCT-NAME(2).
     MOVE "DRIER"  TO PRODUCT-NAME(3).
     EXEC SQL
       DELETE FROM STOCK WHERE GOODS = :PRODUCT-NAME
     END-EXEC.
*>       :
```



The above sample program works the same as a program written as shown below:

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  PRODUCT-NAME  PIC X(20).
 01  SQLSTATE      PIC X(5).
     EXEC SQL END   DECLARE SECTION END-EXEC.
*>        :
 PROCEDURE DIVISION.
*>        :
     MOVE "RADIO" TO PRODUCT-NAME
     EXEC SQL
       DELETE FROM STOCK WHERE GOODS = :PRODUCT-NAME
     END-EXEC.
     MOVE "SHAVER" TO PRODUCT-NAME
     EXEC SQL
       DELETE FROM STOCK WHERE GOODS = :PRODUCT-NAME
     END-EXEC.
     MOVE "DRIER" TO PRODUCT-NAME
     EXEC SQL
       DELETE FROM STOCK WHERE GOODS = :PRODUCT-NAME
     END-EXEC.
*>        :
```

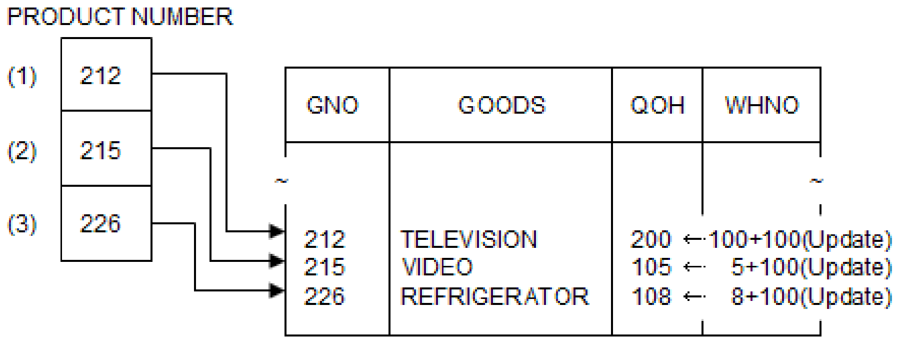
Updating data

When a host variable with multiple rows specified is used for inquiry specification, data that meets multiple conditions can be updated at a time.

The following COBOL program sample updates data of the column that meets the specified conditions while manipulating the data as specified.

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  STOCK-LIST.
   02  PRODUCT-NUMBER  PIC S9(4) COMP-5 OCCURS 3 TIMES.
 01  SQLSTATE          PIC X(5).
     EXEC SQL END  DECLARE SECTION END-EXEC.
*>        :
 PROCEDURE DIVISION.
*>        :
     MOVE 212 TO PRODUCT-NUMBER(1).
     MOVE 215 TO PRODUCT-NUMBER(2).
     MOVE 226 TO PRODUCT-NUMBER(3).
     EXEC SQL
       UPDATE STOCK SET QOH = QOH + 100
```

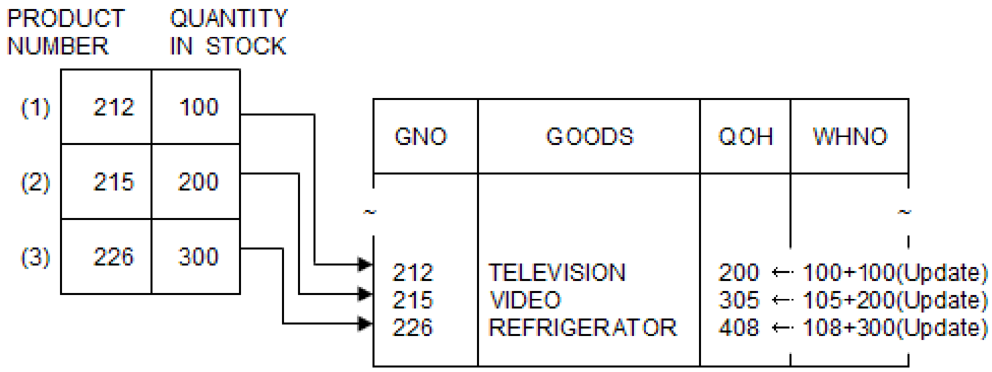
```
         WHERE GNO = :PRODUCT-NUMBER
     END-EXEC.
*>       :
```



When host variables with multiple rows specified are used for both inquiry specification and value specification of the SET clause, data that meet multiple conditions can be updated in order.

The following example uses host variables with multiple rows specified for the SET clause and inquiry specification. The values stored in these variables are used in order from the beginning of the arrays for the corresponding columns (GNO and QOH).

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  STOCK-LIST.
   02 PRODUCT-NUMBER     PIC S9(4) COMP-5 OCCURS 3 TIMES.
   02 QUANTITY-IN-STOCK  PIC S9(9) COMP-5 OCCURS 3 TIMES.
 01  SQLSTATE            PIC X(5).
     EXEC SQL END  DECLARE SECTION END-EXEC.
*>       :
 PROCEDURE DIVISION.
*>       :
     MOVE 212 TO PRODUCT-NUMBER(1).
     MOVE 215 TO PRODUCT-NUMBER(2).
     MOVE 226 TO PRODUCT-NUMBER(3).
     MOVE 100 TO QUANTITY-IN-STOCK(1).
     MOVE 200 TO QUANTITY-IN-STOCK(2).
     MOVE 300 TO QUANTITY-IN-STOCK(3).
     EXEC SQL
       UPDATE STOCK SET QOH = QOH + :QUANTITY-IN-STOCK
         WHERE GNO = :PRODUCT-NUMBER
     END-EXEC.
*>       :
```



Note

Multiple lines cannot be specified for some drivers. Check the specification of each driver.

When two or more host variables with multiple rows specified are used for one SQL statement, specify the same repetition count for both host variables. If different repetition counts are used, use the FOR clause. See "17.2.4.4 Using the FOR clause to control the number of rows to be processed".

If host variables with multiple rows specified have different repetition counts and no FOR clause is specified, the minimum repetition count specified for the host variables is used.

In the following COBOL program sample, host variables with multiple rows specified in one SQL statement uses different repetition counts. One host variable specifies repetition count 8 and one specifies 5. In this case, minimum value 5 is valid for the UPDATE statement.

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  STOCK-LIST.
   02 PRODUCT-NUMBER     PIC S9(4) COMP-5 OCCURS 8 TIMES.
   02 QUANTITY-IN-STOCK  PIC S9(9) COMP-5 OCCURS 5 TIMES.
 01  SQLSTATE            PIC X(5).
     EXEC SQL END  DECLARE SECTION END-EXEC.
*>        :
 PROCEDURE DIVISION.
*>        :
     EXEC SQL
       UPDATE STOCK SET QOH = QOH + :QUANTITY-IN-STOCK
         WHERE GNO = :PRODUCT-NUMBER
     END-EXEC.
*>        :
```

17.2.4.1.2 Host variable with a table specified

Function

A host variable with a table specified can manipulate data extending to multiple rows and columns at a time.

Definition

A host variable with a table specified defines host variables with multiple rows specified corresponding to individual columns in the database table as a group item. Refer to *the COBOL Reference Manual* for more information.

Usage

Retrieving data

A host variable with a table specified can be used to retrieve as many data items as the repetition count of the host variables with multiple rows specified, which are the subordinate items of the host variable with a table specified.

The following COBOL program sample fetches three sets of data (PRODUCT-NUMBER, PRODUCT-NAME, QUANTITY-IN-STOCK, and WAREHOUSE-NUMBER) on the rows with product name TELEVISION from the STOCK table.

```
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.                  *>-+
 01  STOCK-LIST.                                           *> |
   02 PRODUCT-NUMBER     PIC S9(4) COMP-5 OCCURS 3 TIMES.  *> |
   02 PRODUCT-NAME       PIC X(20) OCCURS 3 TIMES.         *> | [1]
   02 QUANTITY-IN-STOCK  PIC S9(9) COMP-5 OCCURS 3 TIMES.  *> |
   02 WAREHOUSE-NUMBER   PIC S9(4) COMP-5 OCCURS 3 TIMES.  *> |
 01  SQLSTATE            PIC X(5).                         *> |
 EXEC SQL END  DECLARE SECTION END-EXEC.                   *>-+
*>     :
 PROCEDURE DIVISION.
*>     :
     EXEC SQL
       SELECT *
       INTO :STOCK-LIST  FROM STOCK
       WHERE GOODS = 'TELEVISION'
     END-EXEC.                                             *>   [2]
*>    :
```

- [1] Host variable STOCK-LIST with a table specified is defined.
- [2] When data is fetched by the SELECT statement, the following values are stored in host variable STOCK-LIST:

Table 17.1 STOCK TABLE

PRODUCT NUMBER	PRODUCT NAME	QUANTITY IN STOCK	WAREHOUSE NUMBER
110	TELEVISION	85	2
111	TELEVISION	90	2
212	TELEVISION	0	2

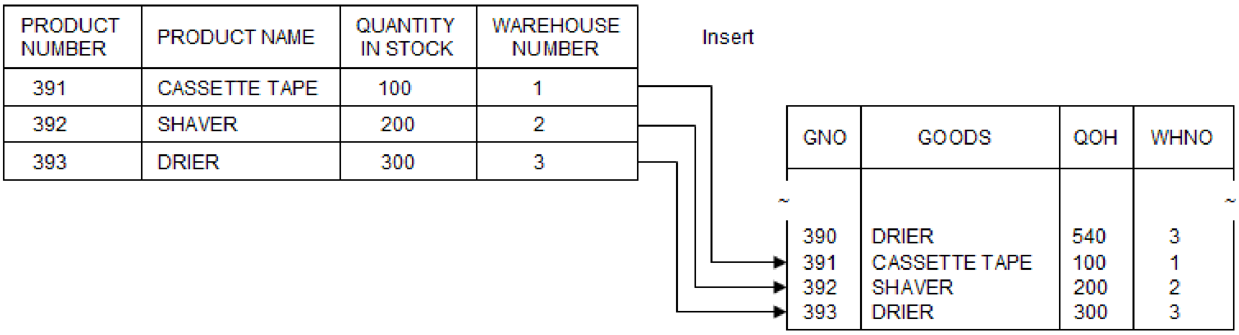
Inserting data

A host variable with a table specified can be used to insert as many sets of data as the repetition count of the host variables with multiple rows specified, which are the subordinate items of the host variable with a table specified.

The following COBOL program sample inserts three sets of data into the STOCK table.

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  STOCK-LIST.
   02 PRODUCT-NUMBER        PIC S9(4) COMP-5 OCCURS 3 TIMES.
   02 PRODUCT-NAME          PIC X(20) OCCURS 3 TIMES.
   02 QUANTITY-IN-STOCK     PIC S9(9) COMP-5 OCCURS 3 TIMES.
   02 WAREHOUSE-NUMBER      PIC S9(4) COMP-5 OCCURS 3 TIMES.
 01  SQLSTATE               PIC X(5).
     EXEC SQL END  DECLARE SECTION END-EXEC.
*>      :
 PROCEDURE DIVISION.
*>      :
     MOVE 391 TO PRODUCT-NUMBER(1)
     MOVE 392 TO PRODUCT-NUMBER(2)
     MOVE 393 TO PRODUCT-NUMBER(3)
     MOVE "CASSETTE TAPE" TO PRODUCT-NAME(1)
     MOVE "SHAVER"        TO PRODUCT-NAME(2)
     MOVE "DRIER"         TO PRODUCT-NAME(3)
     MOVE 100 TO QUANTITY-IN-STOCK(1)
     MOVE 200 TO QUANTITY-IN-STOCK(2)
     MOVE 300 TO QUANTITY-IN-STOCK(3)
     MOVE 1   TO WAREHOUSE-NUMBER(1)
     MOVE 2   TO WAREHOUSE-NUMBER(2)
     MOVE 3   TO WAREHOUSE-NUMBER(3)
     EXEC SQL
       INSERT INTO STOCK(GNO, GOODS, GOH, WHNO)
        VALUES(:STOCK-LIST)
     END-EXEC.
*>      :
```

Figure 17.2 STOCK TABLE



PRODUCT NUMBER	PRODUCT NAME	QUANTITY IN STOCK	WAREHOUSE NUMBER
391	CASSETTE TAPE	100	1
392	SHAVER	200	2
393	DRIER	300	3

GNO	GOODS	QOH	WHNO
390	DRIER	540	3
391	CASSETTE TAPE	100	1
392	SHAVER	200	2
393	DRIER	300	3

Note

Multiple lines cannot be specified for some drivers. Check the specification of each driver.

17.2.4.2 Using host variables in a dynamic SQL statement

A host variable with multiple rows specified or a host variable with a table specified can be used in a dynamic SQL statement in the same manner as conventional host variables. However, the types of SQL statements in which these host variables can be used are limited. If these host variables are used in inappropriate SQL statements, operation is not guaranteed. See "17.2.10 Available Host Variable in Embedded SQL Statements", for more information.

17.2.4.3 Using SQLERRD to check the number of rows processed

When data is manipulated using a host variable with multiple rows specified or a host variable with a table specified, SQLERRD (3) can be retrieved to check the number of rows of data fetched or processed.

Examples

- When multiple rows of data are fetched by the SELECT or FETCH statement, the number of data items fetched is stored in SQLERRD (3). For instance, if the number of data items that can be fetched is 50 when a host variable with multiple rows specified is used with repetition count 100, 50 is stored in SQLERRD (3).
- When multiple rows of data are manipulated by the INSERT, UPDATE (search), or DELETE (search) statement, the number of rows processed by data manipulation of the SQL statement is stored in SQLERRD (3).

The following COBOL program sample updates QOH twice in accordance with the conditions of WHNO in the STOCK table. For WHNO = 1, six QOH data items are updated. For WHNO = 2, eleven QOH data items are updated. Total number 17 is stored in SQLERRD (3).

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  STOCK-LIST.
   02 QUANTITY-IN-STOCK   PIC S9(9) COMP-5 OCCURS 2 TIMES.
   02 WAREHOUSE-NUMBER    PIC S9(4) COMP-5 OCCURS 2 TIMES.
 01  SQLINFOA.
   02 SQLERRD             PIC S9(9) COMP-5 OCCURS 6 TIMES.
 01  SQLSTATE           PIC X(5).
     EXEC SQL END  DECLARE SECTION END-EXEC.
*>     :
 PROCEDURE DIVISION.
*>     :
     MOVE 100 TO QUANTITY-IN-STOCK(1)
     MOVE 200 TO QUANTITY-IN-STOCK(2)
     MOVE 1   TO WAREHOUSE-NUMBER(1)
     MOVE 2   TO WAREHOUSE-NUMBER(2)
     EXEC SQL
       UPDATE STOCK SET QOH = GOH + :QUANTITY-IN-STOCK
        WHERE WHNO = :WAREHOUSE-NUMBER
     END-EXEC.
     DISPLAY SQLERRD(3)
*>     :
```

Note

"No data" is not indicated in SQLSTATE if the number of data items to be fetched is either greater or smaller than the repetition count. "No data" is indicated in SQLSTATE only when no row of data is fetched.

The changed lines of a stored procedure called by the CALL statement are not guaranteed for SQLERRD(3).

17.2.4.4 Using the FOR clause to control the number of rows to be processed

When a host variable with multiple rows specified or a host variable with a table specified is used to manipulate data, the FOR clause can be specified to control the number of rows to be processed or the number of processing times.

Usage

Retrieving data

The following COBOL program sample retrieves five sets of data from the beginning of the STOCK table.

Data is fetched by the FETCH statement using host variable STOCK-LIST with a table specified, which has subordinate host variables with multiple rows specified using repetition count 10. In this case, even if repetition count 10 is specified with each host variable with multiple rows specified, five sets of data are stored in the host variable with a table specified because 5 is specified in the FOR clause of the FETCH statement.

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  STOCK-LIST.
   02 PRODUCT-NUMBER          PIC S9(4) COMP-5 OCCURS 10 TIMES.
   02 PRODUCT-NAME            PIC X(20) OCCURS 10 TIMES.
   02 QUANTITY-IN-STOCK       PIC S9(9) COMP-5 OCCURS 10 TIMES.
   02 WAREHOUSE-NUMBER        PIC S9(4) COMP-5 OCCURS 10 TIMES.
 01  SQLSTATE               PIC X(5).
     EXEC SQL END  DECLARE SECTION END-EXEC.
*>     :
 PROCEDURE DIVISION.
*>     :
     EXEC SQL
       DECLARE CUR1 CURSOR FOR SELECT * FROM STOCK
     END-EXEC.
*>     :
     EXEC SQL
       OPEN CUR1
     END-EXEC.
*>     :
     EXEC SQL
       FOR 5
       FETCH CUR1 INTO :STOCK-LIST
     END-EXEC.
*>     :
```

The sample program for retrieving an entire table shown in "17.2.3.2.1 Retrieving Data from All Table Rows", can also be written as follows:

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.               *>-+
 01  STOCK-LIST.                                           *> |
   02 PRODUCT-NUMBER    PIC S9(4) COMP-5 OCCURS 10 TIMES.  *> |
   02 PRODUCT-NAME      PIC X(20) OCCURS 10 TIMES.         *> | [1]
   02 QUANTITY-IN-STOCK PIC S9(9) COMP-5 OCCURS 10 TIMES.  *> |
   02 WAREHOUSE-NUMBER  PIC S9(4) COMP-5 OCCURS 10 TIMES.  *> |
 01  NUMBER-OF-LINES    PIC S9(9) COMP-5.                  *> |
 01  SQLSTATE           PIC X(5).                          *>-+
    EXEC SQL END  DECLARE SECTION END-EXEC.
*>       :
PROCEDURE DIVISION.
*>       :
    EXEC SQL
      DECLARE CUR1 CURSOR FOR SELECT * FROM STOCK
    END-EXEC.
*>       :
    EXEC SQL
      SELECT COUNT(*) INTO :NUMBER-OF-LINES FROM STOCK
    END-EXEC.                                              *>   [2]
*>       :
    EXEC SQL OPEN CUR1 END-EXEC.
*>         :
    EXEC SQL
      FOR :NUMBER-OF-LINES
      FETCH CUR1 INTO :STOCK-LIST
    END-EXEC.                                              *>    [3]
*>         :
```

- [1] Host variable STOCK-LIST with a table specified is defined.
- [2] Set function COUNT (*) is used to determine the number of table rows.
- [3] Data is retrieved from all table rows into STOCK-LIST.

Inserting data

The following COBOL program sample inserts three rows of data into the STOCK table.

Data is inserted by the INSERT statement using host variable STOCK-LIST with a table specified, which has subordinate host variables with multiple rows specified using repetition count 10. In this case, even if repetition count 10 is specified with each host variable with multiple rows specified, three sets of data are inserted into the STOCK table because 3 is specified in the FOR clause of the INSERT statement.

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  STOCK-LIST.
   02 PRODUCT-NUMBER          PIC S9(4) COMP-5 OCCURS 10 TIMES.
   02 PRODUCT-NAME            PIC X(20) OCCURS 10 TIMES.
   02 QUANTITY-IN-STOCK       PIC S9(9) COMP-5 OCCURS 10 TIMES.
   02 WAREHOUSE-NUMBER        PIC S9(4) COMP-5 OCCURS 10 TIMES.
 01  SQLSTATE               PIC X(5).
     EXEC SQL END  DECLARE SECTION END-EXEC.
*>       :
 PROCEDURE DIVISION.
*>       :
     MOVE 391 TO PRODUCT-NUMBER(1)
     MOVE 392 TO PRODUCT-NUMBER(2)
     MOVE 393 TO PRODUCT-NUMBER(3)
     MOVE "CASSETTE TAPE" TO PRODUCT-NAME(1)
     MOVE "SHAVER"        TO PRODUCT-NAME(2)
     MOVE "DRIER"         TO PRODUCT-NAME(3)
     MOVE 100 TO QUANTITY-IN-STOCK(1)
     MOVE 200 TO QUANTITY-IN-STOCK(2)
     MOVE 300 TO QUANTITY-IN-STOCK(3)
     MOVE 1   TO WAREHOUSE-NUMBER(1)
     MOVE 2   TO WAREHOUSE-NUMBER(2)
     MOVE 3   TO WAREHOUSE-NUMBER(3)
     EXEC SQL
       FOR 3
       INSERT INTO STOCK(GNO, GOODS, GOH, WHNO)
       VALUES(:STOCK-LIST)
     END-EXEC.
*>       :
```

Note

The value specified in the FOR clause must be equal to or smaller than the repetition count of a host variable with multiple rows specified. Otherwise, an error message is output.

17.2.4.5 Data acquisition using a scrollable cursor

A scrollable cursor is one that can access the results of a cursor OPEN statement in a non-sequential, directed order. The following NetCOBOL FETCH statements are used in the scrollable cursor function.

Table 17.2 FETCH statement that uses scrollable cursor

FETCH statement	Operation Description
FETCH PRIOR	Retrieves data from the row prior to the most recently fetched row.
FETCH FIRST	Retrieves data from the first row.
FETCH LAST	Retrieves data from the row prior to the last row.

Usage

Options required to use a scrollable cursor

The options shown below must be set in order to use a scrollable cursor. A scrollable cursor cannot be used with the forward-only default cursor setting. Refer to "17.2.8.2 Using the ODBC Information Setup Tool" for details of the option.

Option	Setup Value
@SQL_CURSOR_TYPE	KEYSET DRIVEN or STATIC or DYNAMIC

The @SQL_CURSOR_TYPE FORWARD_ONLY option cannot be used with a scrollable cursor. If no option is specified for @SQL_CURSOR_TYPE, the FORWARD_ONLY cursor setting will default.

Using FETCH PRIOR

In the example below, the first three rows are fetched from the STOCK table using FETCH NEXT; then FETCH PRIOR is used to fetch the second and first rows, returning the cursor to the top of the table.

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 STOCK-LIST.                                               --+
   02 PRODUCT-NUMBER      PIC S9(4) COMP-5.                     |
   02 PRODUCT-NAME        PIC X(20).                            | [1]
   02 QUANTITY-IN-STOCK   PIC S9(9).                            |
   02 WAREHOUSE-NUMBER    PIC S9(4).                          --+
 01 SQLSTATE     PIC X(5).
       EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
       EXEC SQL
         DECLARE CUR1 CURSOR FOR SELECT * FROM STOCK
       END-EXEC.
       EXEC SQL OPEN CUR1 END-EXEC.                           ... [2]
       EXEC SQL FETCH NEXT CUR1 INTO :STOCK-LIST END-EXEC.    ... [3]
       EXEC SQL FETCH NEXT CUR1 INTO :STOCK-LIST END-EXEC.    ... [4]
       EXEC SQL FETCH NEXT CUR1 INTO :STOCK-LIST END-EXEC.    ... [5]
       EXEC SQL FETCH PRIOR CUR1 INTO :STOCK-LIST END-EXEC.   ... [6]
       EXEC SQL FETCH PRIOR CUR1 INTO :STOCK-LIST END-EXEC.   ... [7]
       EXEC SQL CLOSE CUR1 END-EXEC.                          ... [8]
       EXEC SQL ROLLBACK WORK END-EXEC.                       ... [9]
       EXEC SQL DISCONNECT DEFAULT END-EXEC.                  ... [10]
       STOP RUN.
```

Descriptions of the figure

- [1] Defines the "STOCK-LIST" host variable.
- [2] Opens cursor "CUR1".
- [3] Fetches the first row of Stock table data into the "STOCK-LIST" host variable.
- [4] Fetches the second row of Stock table data into the "STOCK-LIST" host variable.
- [5] Fetches the third row of Stock table data into the "STOCK-LIST" host variable.
- [6] Fetches the second row of Stock table data into the "STOCK-LIST" host variable.
- [7] Fetches the first row of Stock table data into the "STOCK-LIST" host variable.

- [8] Closes cursor "CUR1".
- [9] Executes the ROLLBACK statement, and the transaction is ended.
- [10] Executes the DISCONNECT statement, and the server is disconnected.

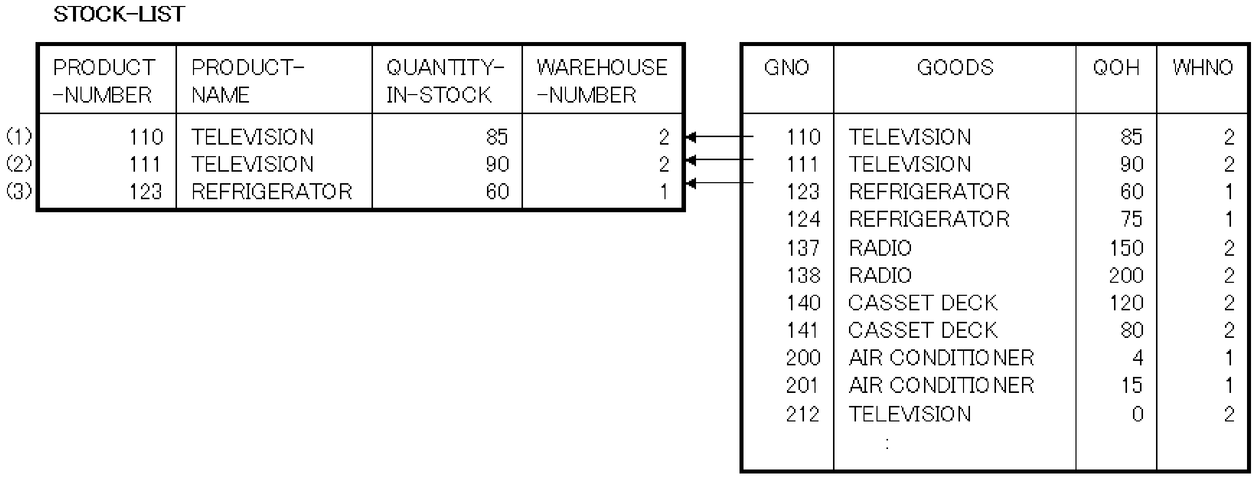
Using FETCH PRIOR for multiple rows

In the example below, nine rows of data are fetched (each fetch returns three rows of data) from the "STOCK" table using FETCH NEXT; then FETCH PRIOR is used twice, returning the cursor to the top of the table.

```
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 STOCK-LIST.                                                  --+
    02 PRODUCT-NUMBER      PIC S9(4) COMP-5.                        |
    02 PRODUCT-NAME        PIC X(20).                               | [1]
    02 QUANTITY-IN-STOCK   PIC S9(9).                               |
    02 WAREHOUSE-NUMBER    PIC S9(4).                             --+
  01 SQLSTATE     PIC X(5).
        EXEC SQL END DECLARE SECTION END-EXEC.
 PROCEDURE DIVISION.
        EXEC SQL
          DECLARE CUR1 CURSOR FOR SELECT * FROM STOCK
        END-EXEC.
        EXEC SQL OPEN CUR1 END-EXEC.                              ... [2]
        EXEC SQL FOR 3 FETCH NEXT CUR1 INTO :STOCK-LIST END-EXEC.   ... [3]
        EXEC SQL FOR 3 FETCH NEXT CUR1 INTO :STOCK-LIST END-EXEC.   ... [4]
        EXEC SQL FOR 3 FETCH NEXT CUR1 INTO :STOCK-LIST END-EXEC.   ... [5]
        EXEC SQL FOR 3 FETCH PRIOR CUR1 INTO :STOCK-LIST END-EXEC.  ... [6]
        EXEC SQL FOR 3 FETCH PRIOR CUR1 INTO :STOCK-LIST END-EXEC.  ... [7]
        EXEC SQL CLOSE CUR1 END-EXEC.                             ... [8]
        EXEC SQL ROLLBACK WORK END-EXEC.                          ... [9]
        EXEC SQL DISCONNECT DEFAULT END-EXEC.                     ... [10]
        STOP RUN.
```

Descriptions of the figure

- [1] Defines the "STOCK-LIST" host variable.
- [2] Opens cursor "CUR1".
- [3] Fetches Stock table rows 1, 2, and 3 into the "STOCK-LIST" host variable.

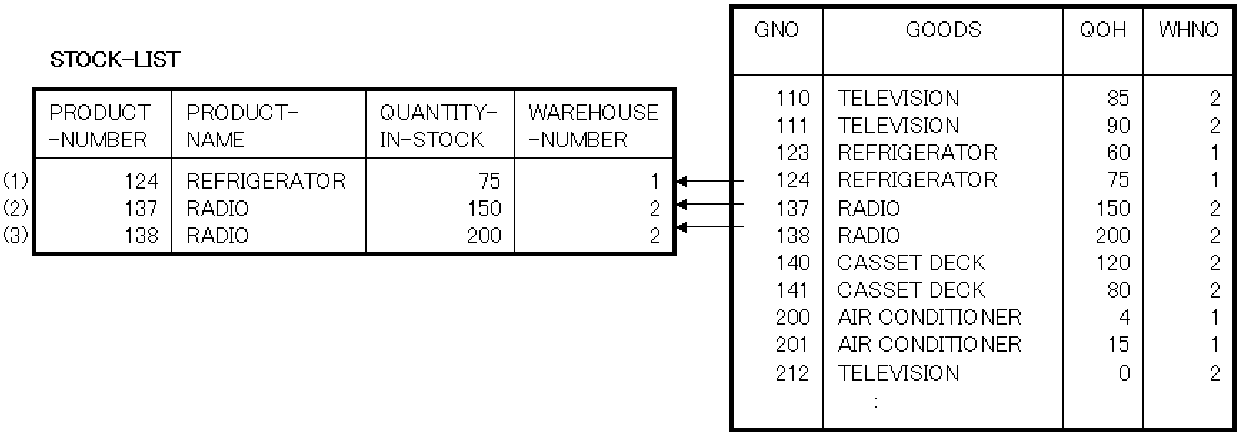


STOCK-LIST

	PRODUCT -NUMBER	PRODUCT- NAME	QUANTITY- IN-STOCK	WAREHOUSE -NUMBER
(1)	110	TELEVISION	85	2
(2)	111	TELEVISION	90	2
(3)	123	REFRIGERATOR	60	1

GNO	GOODS	QOH	WHNO
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
137	RADIO	150	2
138	RADIO	200	2
140	CASSET DECK	120	2
141	CASSET DECK	80	2
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
	:		

- [4] Fetches Stock table rows 4, 5, and 6 into the "STOCK-LIST" host variable.

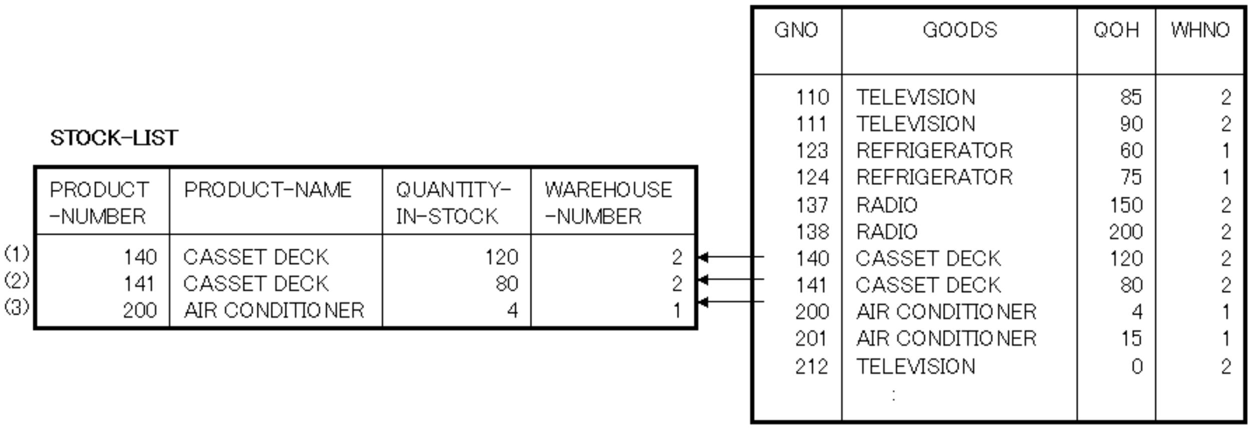


STOCK-LIST

	PRODUCT -NUMBER	PRODUCT-NAME	QUANTITY-IN-STOCK	WAREHOUSE -NUMBER
(1)	124	REFRIGERATOR	75	1
(2)	137	RADIO	150	2
(3)	138	RADIO	200	2

GNO	GOODS	QOH	WHNO
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
137	RADIO	150	2
138	RADIO	200	2
140	CASSET DECK	120	2
141	CASSET DECK	80	2
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
	:		

- [5] Fetches Stock table rows 7, 8, and 9 into the "STOCK-LIST" host variable.

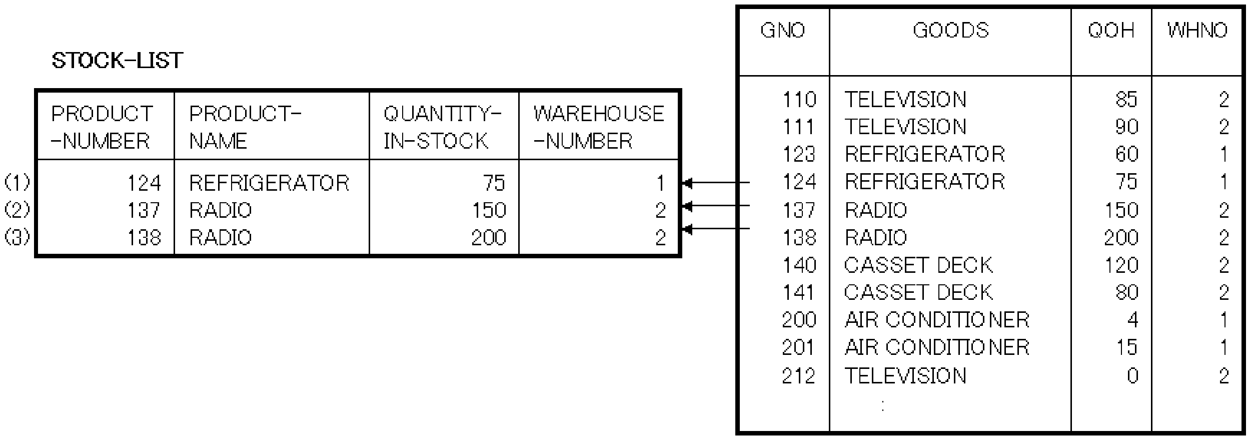


STOCK-LIST

	PRODUCT -NUMBER	PRODUCT-NAME	QUANTITY-IN-STOCK	WAREHOUSE -NUMBER
(1)	140	CASSET DECK	120	2
(2)	141	CASSET DECK	80	2
(3)	200	AIR CONDITIONER	4	1

GNO	GOODS	QOH	WHNO
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
137	RADIO	150	2
138	RADIO	200	2
140	CASSET DECK	120	2
141	CASSET DECK	80	2
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
	:		

- [6] Fetches Stock table rows 4, 5, and 6 into the "STOCK-LIST" host variable, starting with Stock table row 4.

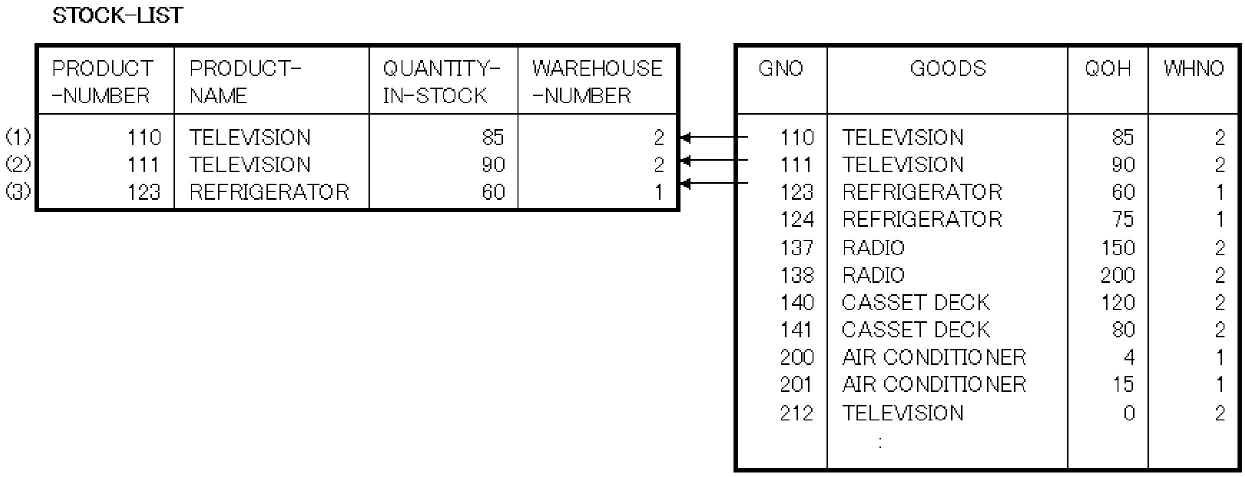


STOCK-LIST

	PRODUCT -NUMBER	PRODUCT-NAME	QUANTITY-IN-STOCK	WAREHOUSE -NUMBER
(1)	124	REFRIGERATOR	75	1
(2)	137	RADIO	150	2
(3)	138	RADIO	200	2

GNO	GOODS	QOH	WHNO
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
137	RADIO	150	2
138	RADIO	200	2
140	CASSET DECK	120	2
141	CASSET DECK	80	2
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
	:		

- [7] Fetches Stock table rows 1, 2, and 3 into the "STOCK-LIST" host variable, starting with Stock table row 1.



STOCK-LIST

	PRODUCT -NUMBER	PRODUCT- NAME	QUANTITY- IN-STOCK	WAREHOUSE -NUMBER
(1)	110	TELEVISION	85	2
(2)	111	TELEVISION	90	2
(3)	123	REFRIGERATOR	60	1

GNO	GOODS	QOH	WHNO
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
137	RADIO	150	2
138	RADIO	200	2
140	CASSET DECK	120	2
141	CASSET DECK	80	2
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
	:		

- [8] Closes cursor "CUR1".
- [9] Executes the ROLLBACK statement, and the transaction is ended.
- [10] Executes the DISCONNECT statement, and the server is disconnected.

Using FETCH FIRST

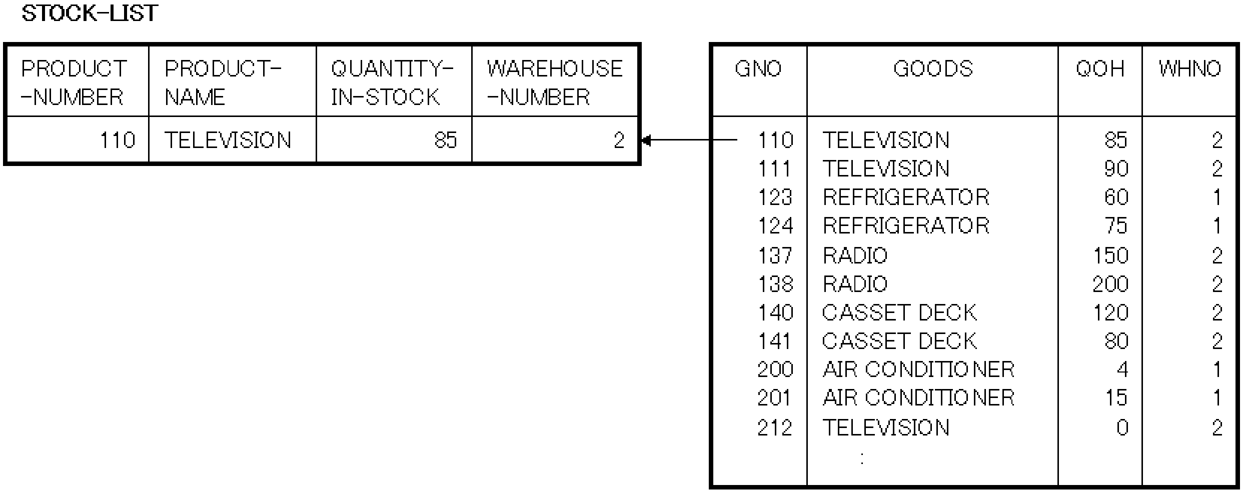
In the example below, the first two rows are fetched from the Stock table using FETCH NEXT; then FETCH FIRST is used to return the cursor to the top of the table and fetch the first row.

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 STOCK-LIST.                                                  --+
   02 PRODUCT-NUMBER      PIC S9(4) COMP-5.                        |
   02 PRODUCT-NAME        PIC X(20).                               | [1]
   02 QUANTITY-IN-STOCK   PIC S9(9).                               |
   02 WAREHOUSE-NUMBER    PIC S9(4).                             --+
 01 SQLSTATE     PIC X(5).
       EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
       EXEC SQL
         DECLARE CUR1 CURSOR FOR SELECT * FROM STOCK
       END-EXEC.
       EXEC SQL OPEN CUR1 END-EXEC.                              ... [2]
       EXEC SQL FETCH NEXT CUR1 INTO :STOCK-LIST END-EXEC.       ... [3]
       EXEC SQL FETCH NEXT CUR1 INTO :STOCK-LIST END-EXEC.       ... [4]
       EXEC SQL FETCH FIRST CUR1 INTO :STOCK-LIST END-EXEC.      ... [5]
       EXEC SQL CLOSE CUR1 END-EXEC.                             ... [6]
       EXEC SQL ROLLBACK WORK END-EXEC.                          ... [7]
       EXEC SQL DISCONNECT DEFAULT END-EXEC.                     ... [8]
       STOP RUN.
```

Descriptions of the figure

- [1] Defines the "STOCK-LIST" host variable.
- [2] Opens cursor "CUR1".

- [3] Fetches the "GNO=110" row of the Stock table into the "STOCK-LIST" host variable.

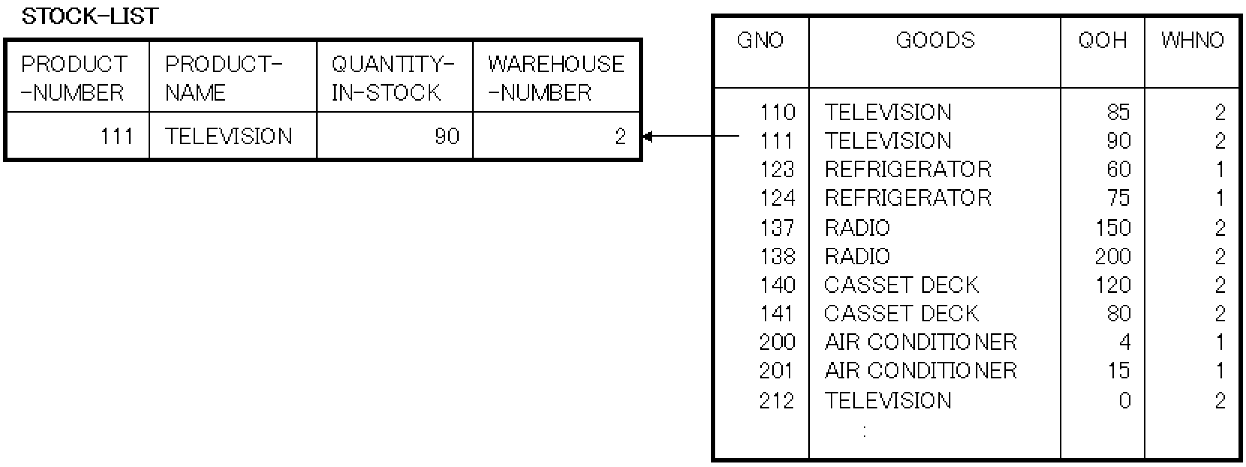


STOCK-LIST

PRODUCT -NUMBER	PRODUCT- NAME	QUANTITY- IN-STOCK	WAREHOUSE -NUMBER
110	TELEVISION	85	2

GNO	GOODS	QOH	WHNO
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
137	RADIO	150	2
138	RADIO	200	2
140	CASSET DECK	120	2
141	CASSET DECK	80	2
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
	:		

- [4] Fetches the "GNO=111" row of the Stock table into the "STOCK-LIST" host variable.

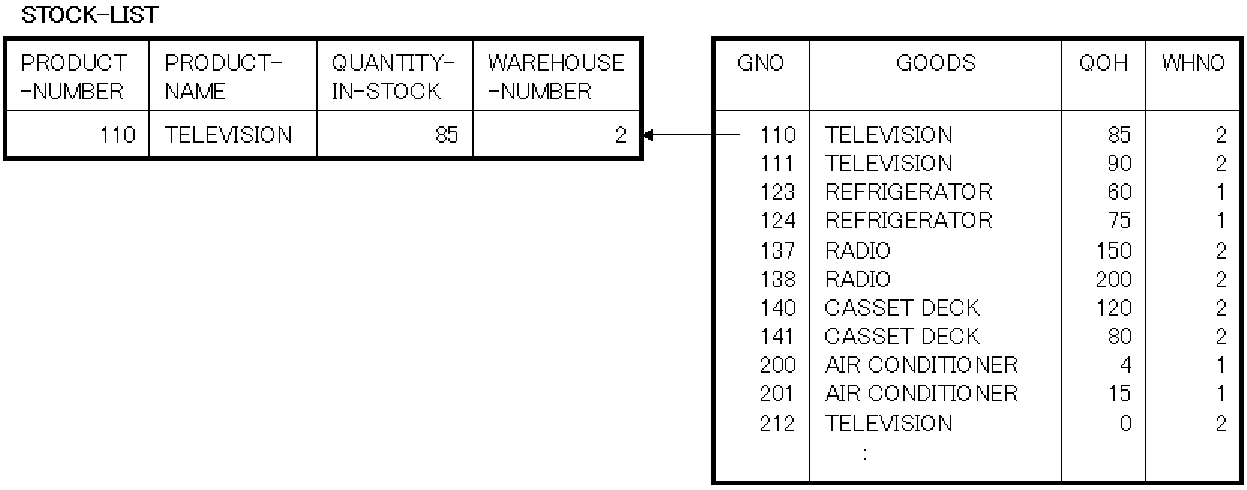


STOCK-LIST

PRODUCT -NUMBER	PRODUCT- NAME	QUANTITY- IN-STOCK	WAREHOUSE -NUMBER
111	TELEVISION	90	2

GNO	GOODS	QOH	WHNO
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
137	RADIO	150	2
138	RADIO	200	2
140	CASSET DECK	120	2
141	CASSET DECK	80	2
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
	:		

- [5] Fetches the first row of "GNO=110" of the Stock table into the "STOCK-LIST" host variable.



STOCK-LIST

PRODUCT -NUMBER	PRODUCT- NAME	QUANTITY- IN-STOCK	WAREHOUSE -NUMBER
110	TELEVISION	85	2

GNO	GOODS	QOH	WHNO
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
137	RADIO	150	2
138	RADIO	200	2
140	CASSET DECK	120	2
141	CASSET DECK	80	2
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
	:		

- [6] Closes cursor "CUR1".
- [7] Executes the ROLLBACK statement, and the transaction is ended.
- [8] Executes the DISCONNECT statement, and the server is disconnected.

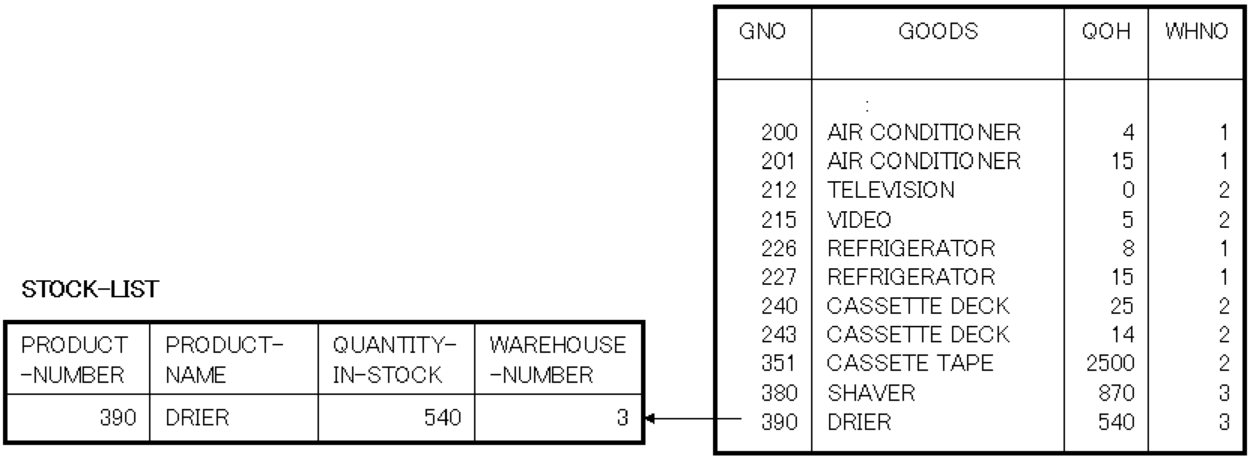
Using FETCH LAST

In the example below, the last row is fetched from the Stock table using FETCH LAST; then FETCH PRIOR is used to fetch the prior rows.

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 STOCK-LIST.                                                       +
   02 PRODUCT-NUMBER      PIC S9(4) COMP-5.                           |
   02 PRODUCT-NAME        PIC X(20).                                  | [1]
   02 QUANTITY-IN-STOCK   PIC S9(9).                                  |
   02 WAREHOUSE-NUMBER    PIC S9(4).                                  +
 01 SQLSTATE     PIC X(5).
       EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
       EXEC SQL
         DECLARE CUR1 CURSOR FOR SELECT * FROM STOCK
       END-EXEC.
       EXEC SQL OPEN CUR1 END-EXEC.                                 ... [2]
       EXEC SQL FETCH LAST CUR1 INTO :STOCK-LIST END-EXEC.          ... [3]
       EXEC SQL FETCH PRIOR CUR1 INTO :STOCK-LIST END-EXEC.         ... [4]
       EXEC SQL FETCH PRIOR CUR1 INTO :STOCK-LIST END-EXEC.         ... [5]
       EXEC SQL CLOSE CUR1 END-EXEC.                                ... [6]
       EXEC SQL ROLLBACK WORK END-EXEC.                             ... [7]
       EXEC SQL DISCONNECT DEFAULT END-EXEC.                        ... [8]
       STOP RUN.
```

Descriptions of the figure

- [1] Defines the "STOCK-LIST" host variable.
- [2] Opens cursor "CUR1".
- [3] Fetches the last row of "GNO=390" of the Stock table into the "STOCK-LIST" host variable.



STOCK-LIST

PRODUCT -NUMBER	PRODUCT- NAME	QUANTITY- IN-STOCK	WAREHOUSE -NUMBER
390	DRIER	540	3

GNO	GOODS	QOH	WHNO
	:		
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
215	VIDEO	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETE TAPE	2500	2
380	SHAVER	870	3
390	DRIER	540	3

- [4] Fetches row "GNO=380" of the Stock table into the "STOCK-LIST" host variable.

GNO	GOODS	QOH	WHNO
	:		
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
215	VIDEO	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETE TAPE	2500	2
380	SHAVER	870	3
390	DRIER	540	3

STOCK-LIST

PRODUCT -NUMBER	PRODUCT-NAME	QUANTITY-IN-STOCK	WAREHOUSE -NUMBER
380	SHAVER	870	3

- [5] Fetches row "GNO=351" of the Stock table into the "STOCK-LIST" host variable.

GNO	GOODS	QOH	WHNO
	:		
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
215	VIDEO	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETE TAPE	2500	2
380	SHAVER	870	3
390	DRIER	540	3

STOCK-LIST

PRODUCT -NUMBER	PRODUCT-NAME	QUANTITY-IN-STOCK	WAREHOUSE -NUMBER
351	CASSETTE TAPE	2500	2

- [6] Closes cursor "CUR1".
- [7] Executes the ROLLBACK statement, and the transaction is ended.
- [8] Executes the DISCONNECT statement, and the server is disconnected.

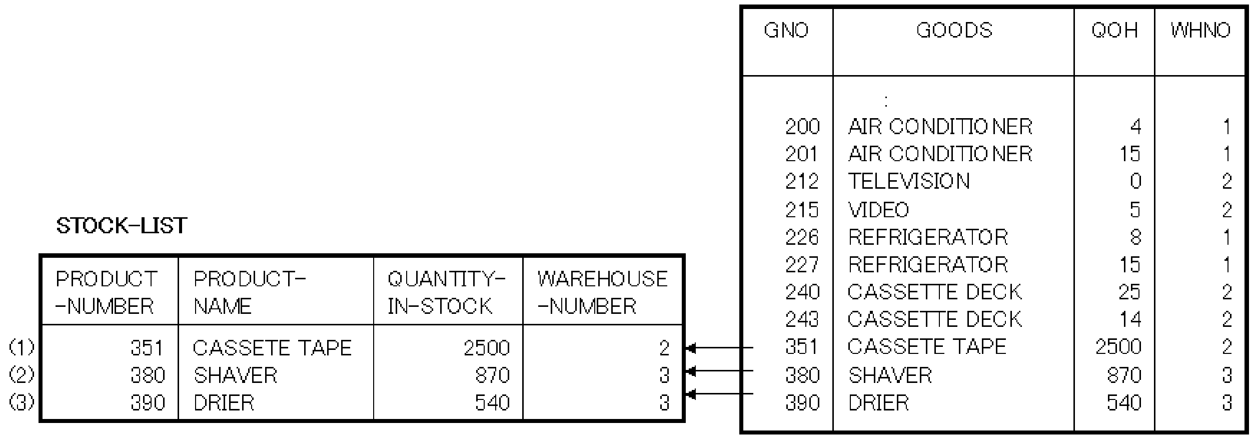
Using FETCH LAST for multiple rows

In the example below, the last three rows are fetched from the Stock table using FETCH LAST; FETCH PRIOR is used to fetch the prior rows every three rows.

```
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 STOCK-LIST.                                                --+
    02 PRODUCT-NUMBER      PIC S9(4) COMP-5.                      |
    02 PRODUCT-NAME        PIC X(20).                             | [1]
    02 QUANTITY-IN-STOCK   PIC S9(9).                             |
    02 WAREHOUSE-NUMBER    PIC S9(4).                           --+
  01 SQLSTATE     PIC X(5).
        EXEC SQL END DECLARE SECTION END-EXEC.
 PROCEDURE DIVISION.
        EXEC SQL
          DECLARE CUR1 CURSOR FOR SELECT * FROM STOCK
        END-EXEC.
        EXEC SQL OPEN CUR1 END-EXEC.                            ... [2]
        EXEC SQL FOR 3 FETCH LAST CUR1 INTO :STOCK-LIST END-EXEC.   ... [3]
        EXEC SQL FOR 3 FETCH PRIOR CUR1 INTO :STOCK-LIST END-EXEC.  ... [4]
        EXEC SQL FOR 3 FETCH PRIOR CUR1 INTO :STOCK-LIST END-EXEC.  ... [5]
        EXEC SQL CLOSE CUR1 END-EXEC.                           ... [6]
        EXEC SQL ROLLBACK WORK END-EXEC.                          ... [7]
        EXEC SQL DISCONNECT DEFAULT END-EXEC.                     ... [8]
        STOP RUN.
```

Descriptions of the figure

- [1] Defines the "STOCK-LIST" host variable.
- [2] Opens cursor "CUR1".
- [3] Fetches rows "GNO=351 to 390" of the Stock table into the "STOCK-LIST" host variable.

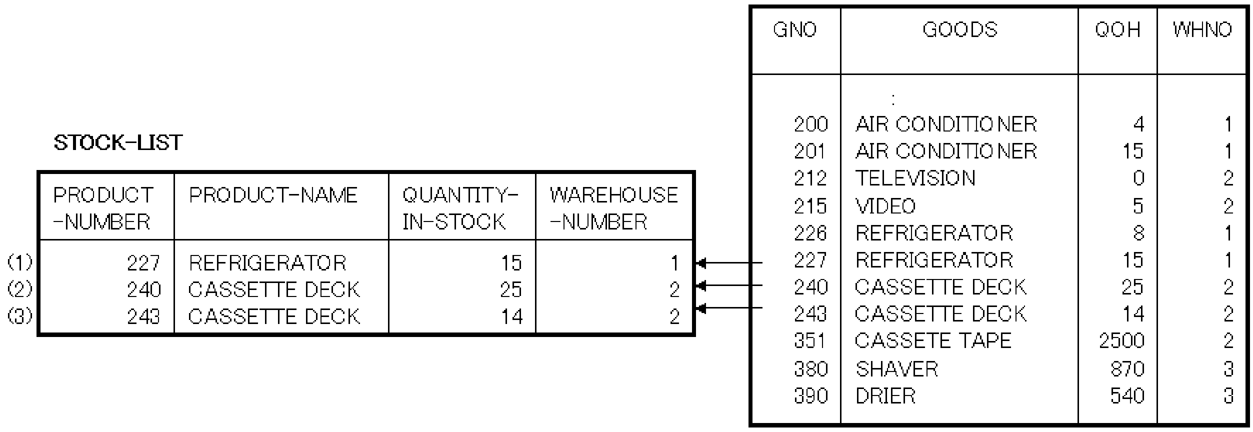


STOCK-LIST

	PRODUCT -NUMBER	PRODUCT-NAME	QUANTITY-IN-STOCK	WAREHOUSE -NUMBER
(1)	351	CASSETE TAPE	2500	2
(2)	380	SHAVER	870	3
(3)	390	DRIER	540	3

GNO	GOODS	QOH	WHNO
	:		
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
215	VIDEO	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETE TAPE	2500	2
380	SHAVER	870	3
390	DRIER	540	3

- [4] Fetches rows "GNO=227 to 243" of the Stock table into the "STOCK-LIST" host variable.

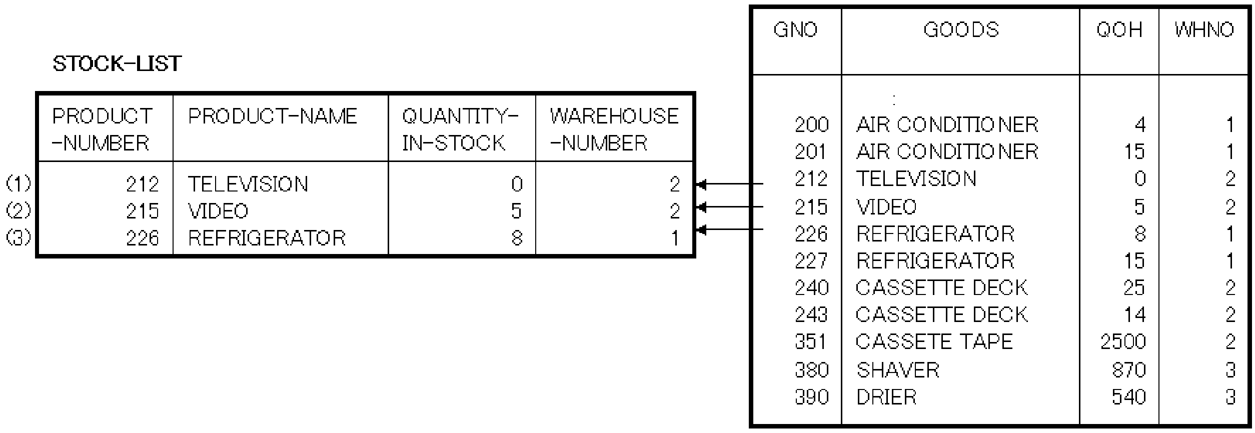


STOCK-LIST

	PRODUCT -NUMBER	PRODUCT-NAME	QUANTITY-IN-STOCK	WAREHOUSE -NUMBER
(1)	227	REFRIGERATOR	15	1
(2)	240	CASSETTE DECK	25	2
(3)	243	CASSETTE DECK	14	2

GNO	GOODS	QOH	WHNO
	:		
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
215	VIDEO	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETE TAPE	2500	2
380	SHAVER	870	3
390	DRIER	540	3

- [5] Fetches rows "GNO=212 to 226" of the Stock table into the "STOCK-LIST" host variable.



STOCK-LIST

	PRODUCT -NUMBER	PRODUCT-NAME	QUANTITY-IN-STOCK	WAREHOUSE -NUMBER
(1)	212	TELEVISION	0	2
(2)	215	VIDEO	5	2
(3)	226	REFRIGERATOR	8	1

GNO	GOODS	QOH	WHNO
	:		
200	AIR CONDITIONER	4	1
201	AIR CONDITIONER	15	1
212	TELEVISION	0	2
215	VIDEO	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETE TAPE	2500	2
380	SHAVER	870	3
390	DRIER	540	3

- [6] Closes cursor "CUR1".

- [7] Executes the ROLLBACK statement, and the transaction is ended.

- [8] Executes the DISCONNECT statement, and the server is disconnected.

17.2.5 Calling a Stored Procedure

17.2.5.1 What is a Stored Procedure?

A stored procedure is a processing procedure registered on the server. This stored procedure is called from the client and executed on the server. Although the advantages of using a stored procedure depend on each database, common advantages are as follows:

- Improvement of the processing procedure execution speed

- Reduction of the communication load between the client and server

- Improvement of development/maintenance productivity

- Security improvement

For details on a stored procedure and how to create it, refer to the manual of each database management system.

Note

Calling a stored procedure can impact performance, and is not recommended if minimal processing will occur within the stored procedure.

17.2.5.2 An Example of Calling a Stored Procedure

The following figure is a COBOL program that calls a stored procedure. In this example, the stored procedure PROC is called. The return value of the stored procedure can be retrieved with SQLERRD(1).

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  INPUT-VARIABLE    PIC S9(4) COMP-5.                     *> [1]
 01  OUTPUT-VARIABLE   PIC S9(4) COMP-5.                     *> [1]
 01  SQLINFOA.
   02  SQLERRD           PIC S9(9) COMP-5 OCCURS 6 TIMES.
     EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
     EXEC SQL CONNECT TO DEFAULT END-EXEC.                   *> [2]
     MOVE 100 TO INPUT-VARIABLE.                             *> [3]
     EXEC SQL
       CALL PROC (:INPUT-VARIABLE, :OUTPUT-VARIABLE)         *> [4]
     END-EXEC.
     DISPLAY SQLERRD(1).                                     *> [5]
     EXEC SQL ROLLBACK WORK END-EXEC.                        *> [6]
     EXEC SQL DISCONNECT DEFAULT END-EXEC.                   *> [7]
     STOP RUN.
```

Descriptions of the figure

- [1] Describes an embedded SQL declare section in the working-storage section and defines all of the arguments for the stored procedure to be called as host variables. For rules on host variable declaration, refer to "COBOL Language Reference".

- [2] Executes the CONNECT statement and establishes server connection.

- [3] Sets the input parameter for the stored procedure as the host variable.

- [4] Executes the CALL statement and calls the PROC stored procedure registered on the server. After this procedure is called, the output parameter from the stored procedure is set as the host variable.

- [5] The return value of the stored procedure is returned.

- [6] Executes the ROLLBACK statement and ends the transaction.

- [7] Executes the DISCONNECT statement and disconnects server connection.

Note

The maximum number of the arguments for a stored procedure call is 700.

The stored procedure's ability to return a return value depends on the database. Check the manual for the database used.

17.2.6 Accessing a Database Using Object-Oriented Programming Functions

This section explains how to access a database using the object-oriented programming functions.

For details, see "Chapter 14 Basic Features of OO COBOL", "Chapter 15 Developing OO COBOL Applications" and "Chapter 16 Advanced Features of OO COBOL".

17.2.6.1 Sample Databases

The class and program definition examples in this section use the STOCK table (inventory table) in "17.2.3.1 Sample Database".

17.2.6.2 Extracting and Editing Data from the Table in the Class Definition

This section explains how to extract data from the table and edit it in the class definition.

"Example definitions of a database access class" shows the COBOL class definitions including those of an inventory management method. The inventory management method extracts data subject to inventory management, computes an inventory quantity for stocking and shipping (input and output), and stores it again in the table.

"Program example of calling the method of the database access class" shows the COBOL program that invokes the inventory management method.

Example definitions of a database access class

```
CLASS-ID. DBACCESS-CLASS  INHERITS FJBASE.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
    CLASS FJBASE.
 OBJECT.
 DATA DIVISION.
 PROCEDURE DIVISION.
 METHOD-ID.  MANAGE-STOCK.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.  *>-+
 01  SQLSTATE           PIC  X(5).            *> | [1]
 01  INVENTORY-QUANTITY PIC  S9(9) COMP-5.    *> |
    EXEC SQL END   DECLARE SECTION END-EXEC.  *>-+
 LINKAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.  *>-+
 01  PRODUCT-NUMBER     PIC S9(4) COMP-5.     *> | [2]
    EXEC SQL END  DECLARE SECTION END-EXEC.   *> |
 01  INPUT-OUTPUT-ID       PIC S9(4) COMP-5.  *> |
 01  INPUT-OUTPUT-QUANTITY PIC S9(9) COMP-5.  *>-+
 PROCEDURE   DIVISION                         *>   [3]
             USING PRODUCT-NUMBER  INPUT-OUTPUT-ID  INPUT-OUTPUT-QUANTITY.
     EXEC SQL
       SELECT QOH INTO  :INVENTORY-QUANTITY FROM STOCK WHERE GNO = :PRODUCT-NUMBER
     END-EXEC.
     IF INPUT-OUTPUT-ID = 1 THEN
       COMPUTE INVENTORY-QUANTITY = INVENTORY-QUANTITY + INPUT-OUTPUT-QUANTITY
     ELSE
       COMPUTE INVENTORY-QUANTITY = INVENTORY-QUANTITY - INPUT-OUTPUT-QUANTITY
     END-IF.
     EXEC SQL
```

```
       UPDATE STOCK SET QOH = :INVENTORY-QUANTITY WHERE GNO = :PRODUCT-NUMBER
     END-EXEC.
 END METHOD MANAGE-STOCK.
 END OBJECT.
 END CLASS DBACCESS-CLASS.
```

Explanation of diagram

- [1] Defines method data.
- [2] Defines the interface of the inventory management method.
- [3] Defines the procedure of inventory management method.

The inventory management method takes out the inventory quantity of a product number from the STOCK table (inventory table), computes it for stocking and shipping, and stores it again in the table.

Program example of calling the method of the database access class

```
*>    :
 CONFIGURATION SECTION.
  REPOSITORY.
     CLASS DBACCESS-CLASS.
DATA DIVISION.
 WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  SQLSTATE  PIC   X(5).
    EXEC SQL END   DECLARE SECTION END-EXEC.
 01  OBJECT-DATA.
   02  DBACCESS-OBJECT  OBJECT REFERENCE DBACCESS-CLASS.
 01  PRODUCT-NUMBER         PIC S9(4) COMP-5.
 01  INPUT-OUTPUT-ID        PIC S9(4) COMP-5.
 01  INPUT-OUTPUT-QUANTITY  PIC S9(9) COMP-5.
 01  END-REQUEST            PIC X(1).
 PROCEDURE DIVISION.
    EXEC SQL CONNECT TO DEFAULT END-EXEC.
    INVOKE  DBACCESS-CLASS "NEW" RETURNING DBACCESS-OBJECT.              *>   [1]
    PERFORM TEST AFTER UNTIL END-REQUEST = "Y"                           *>-+
      DISPLAY "ENTER A PRODUCT NUMBER, INPUT-OUTPUT ID (1 OR 2), "       *> |
      DISPLAY "AND INPUT-OUTPUT QUANTITY."                               *> |
      ACCEPT  PRODUCT-NUMBER                                             *> |
      ACCEPT  INPUT-OUTPUT-ID                                            *> | [2]
      ACCEPT  INPUT-OUTPUT-QUANTITY                                      *> |
      INVOKE  DBACCESS-OBJECT "MANAGE-STOCK"                             *> |
              USING PRODUCT-NUMBER INPUT-OUTPUT-ID INPUT-OUTPUT-QUANTITY *> |
      DISPLAY "DO YOU WANT TO END INVENTORY MANAGEMENT? (Y/N)"           *> |
      ACCEPT  END-REQUEST                                                *> |
    END-PERFORM.                                                         *>-+
    EXEC SQL COMMIT WORK END-EXEC.
    EXEC SQL DISCONNECT DEFAULT END-EXEC.
    STOP RUN.
```

Explanation of diagram

- [1] Creates an object instance in the NEW method.
- [2] Requests the input of a product number, input-output ID, and input-output quantity, and calls the inventory management method. Repeats the processing until Y is input to end-request.

17.2.6.3 Using a Connection for Each Object Instance

This section explains how to use a connection for each object instance, thereby making it possible to mange transactions for each object instance in a distributed object environment.

Creating a Program Using the Connection for Each Object Instance

An example class definition using the connection for each object instance is shown below.

```
CLASS-ID. DBACCESS-CLASS INHERITS FJBASE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
 REPOSITORY.
    CLASS FJBASE.
 OBJECT.
 DATA DIVISION.
 PROCEDURE DIVISION.
 METHOD-ID.  MANAGE-STOCK.
 DATA DIVISION.                                        *> [1]
 WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 SQLSTATE            PIC X(5).
 01 INVENTORY-QUANTITY  PIC S9(9) COMP-5.
 01 PRODUCT-NUMBER      PIC S9(4) COMP-5.
 01 INPUT-OUTPUT-ID     PIC S9(4) COMP-5.
 01 INPUT-OUTPUT-QUANTITY PIC S9(9) COMP-5.
    EXEC SQL END   DECLARE SECTION END-EXEC.
 01  END-REQUEST        PIC X(1).
 01  REFLECTION-REQUEST PIC X(1).
 PROCEDURE   DIVISION.                                  *> [2]
     EXEC SQL CONNECT TO DEFAULT END-EXEC.
     PERFORM TEST AFTER UNTIL END-REQUEST = "Y"
       DISPLAY "ENTER A PRODUCT NUMBER, INPUT-OUTPUT ID (1 OR 2), "
       DISPLAY "AND INPUT-OUTPUT QUANTITY."
       ACCEPT  PRODUCT-NUMBER
       ACCEPT  INPUT-OUTPUT-ID
       ACCEPT  INPUT-OUTPUT-QUANTITY
       IF INPUT-OUTPUT-ID = 1 THEN
         EXEC SQL
           UPDATE STOCK SET QOH = QOH + :INPUT-OUTPUT-QUANTITY
             WHERE GNO = :PRODUCT-NUMBER
         END-EXEC
       ELSE
         EXEC SQL
           UPDATE STOCK SET QOH = QOH - :INVENTORY-QUANTITY
             WHERE GNO = :PRODUCT-NUMBER
         END-EXEC
       END-IF
       EXEC SQL
         SELECT QOH INTO :INVENTORY-QUANTITY FROM STOCK
           WHERE GNO = :PRODUCT-NUMBER
       END-EXEC
       DISPLAY "CURRENT INVENTORY QUANTITY" INVENTORY-QUANTITY
       DISPLAY "DO YOU WANT TO REFLECT THE RESULT OF CHANGE? (Y/N)"
       ACCEPT REFLECTION-REQUEST
       IF REFLECTION-REQUEST = "Y" THEN
         EXEC SQL COMMIT WORK END-EXEC
       ELSE
         EXEC SQL ROLLBACK WORK END-EXEC
       END-IF
       DISPLAY "DO YOU WANT TO END INVENTORY MANAGEMENT? (Y/N)"
       ACCEPT END-REQUEST
     END-PERFORM.
     EXEC SQL DISCONNECT DEFAULT END-EXEC.
     EXIT METHOD.
 END METHOD MANAGE-STOCK.
 END OBJECT.
 END CLASS DBACCESS-CLASS.
```

Explanation of diagram

- [1] Defines method data.
- [2] Defines procedure of the inventory management method. This method performs consecutive database access from connection setup to data manipulation and connection release. The data manipulation requests the input of a product number, input-output ID, and input-output quantity, and recomputes an inventory quantity for stocking and shipping. It then displays a post-recomputation inventory quantity and requests the user to specify whether the result of the change is to be posted. The data manipulation is repeated until Y is input to end-request.

The following shows a program example calling the database access class method (inventory management method).

```
*>    :
 CONFIGURATION SECTION.
 REPOSITORY.
     CLASS DBACCESS-CLASS.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  OBJECT-COUNT   PIC  S9(1).
 01  OBJECTS.
   02  DBACCESS-OBJECT OCCURS 3 OBJECT REFERENCE DBACCESS-CLASS.
 PROCEDURE DIVISION.
     PERFORM WITH TEST AFTER VARYING OBJECT-COUNT FROM 1 BY 1 *>-+
          UNTIL OBJECT-COUNT = 3                              *> |
       INVOKE  DBACCESS-CLASS "NEW"                           *> | [1]
                  RETURNING DBACCESS-OBJECT(OBJECT-COUNT)     *> |
       INVOKE  DBACCESS-OBJECT(OBJECT-COUNT) "MANAGE-STOCK"   *> |
     END-PERFORM.                                             *>-+
     STOP RUN.
```

Explanation of diagram

- [1] Creates three object instances, using the NEW method, each of which invokes the data management method.

  Each of these inventory management methods is executed for a request from one or more clients.

Executing the Program Using a Connection for Each Object Instance

When using a connection for each object instance, specify the object instance in the range in which the connection becomes valid in the ODBC information file, and execute. For details, see "17.2.8.1.2 Creating an ODBC Information File".

Create the ODBC information file using the ODBC information setup tool. For details, see "17.2.8.2 Using the ODBC Information Setup Tool".

17.2.7 Compiling/Linking the Program

You can compile a COBOL program that accesses the databases with an ODBC driver using embedded SQL, without specifying any particular COBOL compiler options.

Note

- To enable case-sensitivity for host variable names, specify the NOALPHAL compiler option before compiling the program. See "A. 2.1 ALPHAL(lowercase handling (in the program))".
- Embedded SQL keywords cannot be used as user-defined names. See "17.2.9 Embedded SQL Keyword List".
- When compiling a multithread program, specify compile option THREAD (MULTI).

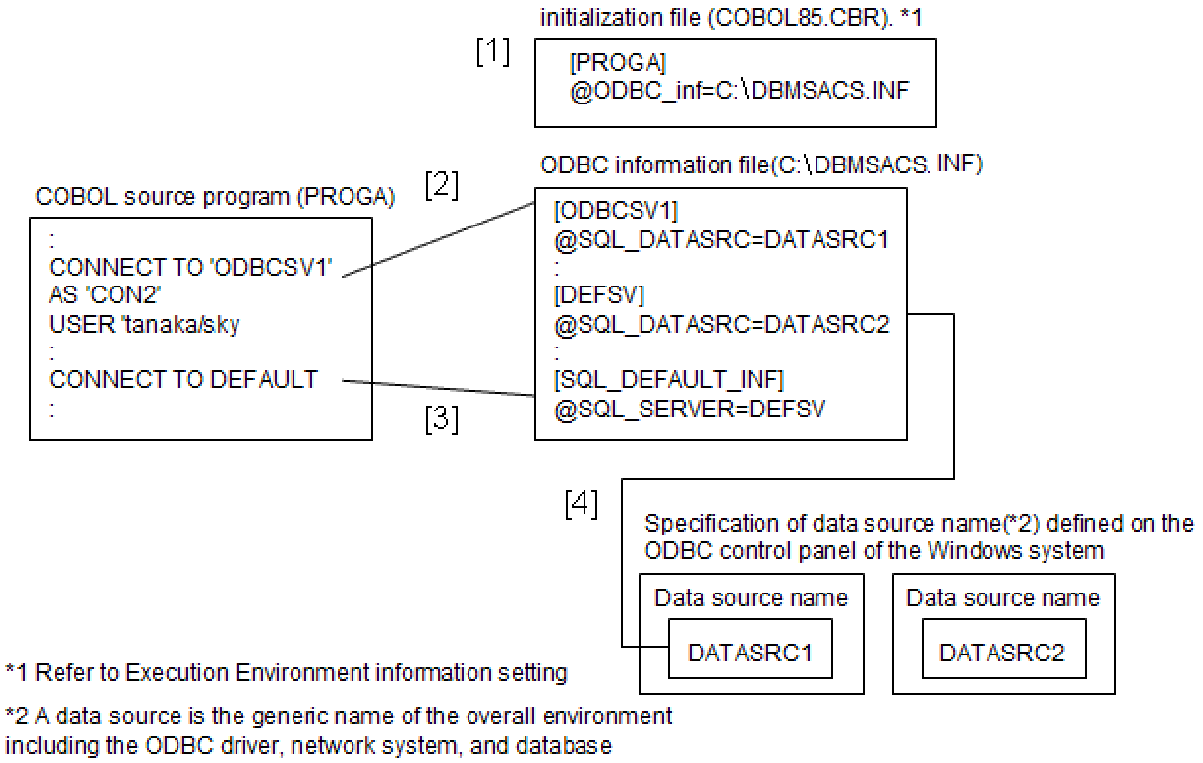
17.2.8 Executing the Program

This section explains how to construct the program execution environment, and how to use the ODBC Information Setup Tool.

17.2.8.1 Constructing the Program Execution Environment

A runtime initialization file (COBOL85.CBR), in which runtime environment information is specified, and the ODBC information file are required for program execution. Specify information in the files so that the files are related as follows: "Figure 17.3 Program execution environment"shows how the setting information in each file is associated.

Figure 17.3 Program execution environment



initialization file (COBOL85.CBR). *1
[1]
[PROGA]
@ODBC_inf=C:\DBMSACS.INF
ODBC information file(C:\DBMSACS. INF)
COBOL source program (PROGA)
[2]
:
CONNECT TO 'ODBCSV1'
AS 'CON2'
USER 'tanaka/sky
:
CONNECT TO DEFAULT
:
[ODBCSV1]
@SQL_DATASRC=DATASRC1
:
[DEFSV]
@SQL_DATASRC=DATASRC2
:
[SQL_DEFAULT_INF]
@SQL_SERVER=DEFSV
[3]
[4]
Specification of data source name(*2) defined on the ODBC control panel of the Windows system
Data source name
DATASRC1
Data source name
DATASRC2
*1 Refer to Execution Environment information setting
*2 A data source is the generic name of the overall environment including the ODBC driver, network system, and database

Description of Figure

- [1] Specify the ODBC information file name (C:\DBMSACS.INF).
- [2] If a server name is specified in the CONNECT statement, specify the server name in the ODBC information file.
- [3] If DEFAULT is specified in the CONNECT statement, specify a fixed character string indicating the definition of default connection information in the ODBC information file.
- [4] To define the data source name of each server specified in the ODBC information file, specify the data source name defined in the Windows system ODBC data source administrator.

17.2.8.1.1 Setting Runtime Environment Information

To select the ODBC environment as client-server linkage software, specify the information shown below in the Environment Variable Information. See "5.3.2 How to Set Runtime Environment Information" for more information.

@ODBC_Inf (specification of the ODBC information file name)

```
@ODBC_Inf=C:\DBMSACS.INF
```

Specify the file name that the COBOL runtime system will refer when using ODBC. See "17.2.8.1.2 Creating an ODBC Information File" for more information.

17.2.8.1.2 Creating an ODBC Information File

An ODBC information file mainly contains information for connecting a client and server. Use the CONNECT statement to specify the connection.

Use the ODBC Information Setup Tool to create an ODBC information file. See "17.2.8.2 Using the ODBC Information Setup Tool" for more information.

The contents of an ODBC information file is classified into server information, default connection information and connection validation scope.

Note

There are potential security problems if you set a password in the ODBC information file.

To avoid these problems, specify that a password must be entered to execute an application, rather then setting the password in the ODBC information file.

Defining Server Information

"Table 17.3 How to define server information" defines server information.

Table 17.3 How to define server information

Information Name	Definition	Remarks
[Server Name]	[Server Name]	[Connect Sentence server name or default connection information]
@SQL_DATASRC	Data source name	Specify the data source name defined (added) on the Windows system ODBC control panel.
@SQL_DATASRC_KIND	Type of Data Source - MACHINE_DS - FILE_DS	Type of Data source is specified. Default is MACHINE_DS. The Machine Data source is used if "MACHINE_DS" is specified. The machine Data source means both user Data source and System Data source. The File Data source is used if "FILE_DS" is specified. (*1)
@SQL_USERID	User ID	Specify the User ID to operate the data source.
@SQL_PASSWORD	Password	Specify the password to operate the data source. Use the ODBC Information Setup Tool to encrypt the password.
@SQL_ACCESS_MODE	Access mode - READ_ONLY - READ_WRITE	Specify an access mode for the data source. The default is READ_ONLY. To enable the read only mode, specify READ_WRITE. To enable the read-write mode, specify READ_WRITE. Operation in the specified mode can vary READ_WRITE based on the capabilities of the ODBC driver.
@SQL_COMMIT_MODE	Commit mode - MANUAL - AUTO	Specify a commit mode for the data source. The default is MANUAL. If MANUAL is specified, SQL operation is determined by specifying the COMMIT or ROLLBACK statement in the COBOL source program. If AUTO is specified, operation is determined each time an SQL statement is executed, regardless of the specification in the COBOL source program. If AUTO is specified, SQL statement processing is reflected into the database at execution of the SQL statement. As a

Information Name	Definition	Remarks
		result, the database cannot be restored to the original status with a ROLLBACK statement. Manual should be specified to avoid this problem.
@SQL_QUERY_TIMEOUT	Timeout(seconds)	Specify the number of seconds to wait for a SQL statement to execute before returning to the application. The timeout range must be between 0 and 4294967285 (seconds). The default is 0 meaning that there is not timeout limit. Operation of the timeout can vary depending on the ODBC driver. If errors occur for timeout the specification should be removed.
@SQL_CONCURRENCY	Cursor concurrency mode - READ_ONLY - LOCK - ROWVER - VALUES	Specify the cursor concurrency. Concurrency is the ability of more than one user to use the same data at the same time. The default is READ_ONLY when @SQL_CONCURRENCY is not specified.(*3)
@SQL_CURSOR_TYPE	Kind of cursor - FORWARD_ONLY - STATIC - KEYSET_DRIVEN - DYNAMIC	Specify the kind of cursor. The default is FORWARD_ONLY when @SQL_CURSOR_TYPE is not specified.(*2) The kind of the cursor greatly influences the performance. The performance improves for the cursor of the read-only if FORWARD_ONLY is specified. If the FORWARD_ONLY is specified, the FETCH PRIOR, FETCH FIRST, and FETCH LAST statement cannot be executed.
@SQL_ODBC_CURSORS	ODBC cursor library - USE_DRIVER - USE_ODBC	Specifies the availability of ODBC cursor library. The ODBC cursor library can alternate cursor processing executed by the ordinary data source. This library can update and delete the positioning data even if they are not supported by the data source. The default is USE_DRIVER. If the USE_DRIVER character string is specified, the ODBC cursor library is not used. If the USE_ODBC is specified, the ODBC cursor library is used. (*4)

- *1: User ID and Password described on the Server Information are valid if "FILE_DS" is specified as type of Data source, and if User ID and Pass word are specified on both the Server Information and the File Data source.
- *2: For details on the operation for each specified value, see "Table 17.4 Operation for the cursor concurrency mode of @SQL_CONCURRENCY ". The operation for each specified value, however, may depend on the ODBC driver. Before using the ODBC driver, see "17.2.13.3 Notes Specific to Each ODBC Driver".
- *3: For details on the operation for each specified value, see "Table 17.5 Operation for the cursor concurrency mode of @SQL_CURSOR_TYPE ". The operation for each specified value, however, may depend on the ODBC driver. Before using the ODBC driver, see "17.2.13.3 Notes Specific to Each ODBC Driver".
- *4: The following rules apply when the ODBC cursor library is used.
    - Always select one or more columns having unique values in the cursor declaration when using the UPDATE statement (positioning) or DELETE statement (positioning).
      The ODBC cursor library simulates the UPDATE statement (positioning) and DELETE statement (positioning) as the respective

UPDATE statement (searching) and DELETE statement (searching) and executes them. If a column having a unique value is not selected, the processing result may affect on multiple lines.

- The execution performance may drop when compared with the operation that does not use the ODBC cursor library.
- The VALUES option must be specified for concurrent cursor operations (@SQL_CONCURRENCY) and the STATIC option must be specified for the kind of cursor (@SQL_CURSOR_TYPE).

Table 17.4 Operation for the cursor concurrency mode of @SQL_CONCURRENCY

Mode	Operation		
READ_ONLY	FETCH statement is allowed. No position UPDATE and DELETE statements are allowed. (*2)		
LOCK	FETCH, position UPDATE and position DELETE statements are allowed. (*1)	NO CONCURRENCY	When processing FETCH, position UPDATE and position DELETE statements, the table is locked and operations from sever or other clients wait until operation is complete.
ROWVER		CONCURRENCY	FETCH statement using same data is allowed. When processing position UPDATE or DELETE statements, and a row is changed, the transaction containing the update or delete operation fails. If the row has not changed, the table is locked until the operation is complete. To determine if a row has changed, the rows and versions should be compared.
VALUE			Operation is the same as ROWVER. To determine if a row has changed, the data values should be compared.

- *1: A lock level for cursor depends on the data source.
- *2: Position UPDATE and DELETE statement is executable in some data sources.

Table 17.5 Operation for the cursor concurrency mode of @SQL_CURSOR_TYPE

Mode	Operation
FORWARD_ONLY	The cursor is opened with the cursor only for the forward direction.
STATIC	The cursor is opened with a static cursor.
KEYSET_DRIVEN	The cursor is opened with the key set driven cursor.
DYNAMIC	The cursor is opened with a dynamic cursor.

Note

The kind of the cursor depends on the data source (product related to the ODBC driver, the data base, and the data base). Please confirm whether there is a corresponding cursor type in the data base used. The kind of the cursor influences the value of @SQL_CONCURRENCY. Please refer to the relating driver.

Defining Default Connection Information

If DEFAULT is specified in the CONNECT statement, connection is established.

"Table 17.6 How to define default connection information" defines default connection information.

Table 17.6 How to define default connection information

Information Name	Definition	Remarks
[SQL_DEFAULT_INF]	Fixed character string	Specify the fixed character string (section name) indicating the start of definition of default connection information.

Information Name	Definition	Remarks
@SQL_SERVER	Server name	Specify the name of a server where default connection is to be established. This server name is used for retrieving the definition information of each server and for establishing connection for the data source for each server. The definition information must be specified.
@SQL_USERID	User ID	Specify the user ID for operating the data source of the default connection server.
@SQL_PASSWORD	Password	Specify the password for operating the data source of the default connection server. Use the ODBC Information Setup Tool to encrypt the password.

Defining Connection Validation Scope

For information about connection validation scope, see "Table 17.7 How to define connection validation scope"

Table 17.7 How to define connection validation scope

Information Name	Contents defined	Remarks
[CONNECTION_SCOPE]	Fixed character string	Specify a fixed character string (section name) that indicates the beginning of the definition of the connection validation scope.
@SQL_CONNECTION SCOPE	Connection validation Scope. - PROCESS - THREAD - OBJECT_INSTANCE	Specify connection validation scope. The default value is PROCESS. The connection validation scope is that in which a set connection can be used. (*1)

*1 : For information on the operation of each specified value, see "Table 17.8 Operation for values specified for the connection validation scope".

For more details, see "19.6.2 Using the Remote Database Access (ODBC)" and "17.2.6.3 Using a Connection for Each Object Instance"

Table 17.8 Operation for values specified for the connection validation scope

Value specified	Operation
PROCESS	A set connection can be used in the execution environment (process). Specify this value when executing single-thread programs. Notes on executing multithread programs - When two or more multithread programs share a connection, a transaction processed by one of those programs affects the data manipulated by the programs sharing the connection. - When two or more multithread programs use two or more connections, the embedded SQL statement executed first in each multithread program must be the CONNECT or SET CONNECTION statement. If the SQL statement is neither the CONNECT nor SET CONNECTION statement, it is uncertain which connection the SQL statement manipulates because each multithread program uses the current connection of that program. The above requirement does not apply when two or more multithread programs share one connection.

	- When a multithread program using cursors is executed in two or more threads, each thread has a cursor, and so cursors cannot be shared among threads.
THREAD	A set connection can be used in a run unit (thread). This value is usually specified when converting a single-thread program to a multithread program. When two or more connections are set up in a run unit, the connection specified in the last executed CONNECT or SET CONNECTION statement is used as the current connection of the run unit. (*1)
OBJECT-INSTANCE	A set connection can be used in an object instance. This value is usually specified when converting a single-thread program using an object-oriented function to a multithread program. When two or more connections are set up in an object instance, the connection specified in the last executed CONNECT or SET CONNECTION statement is assumed to be the current connection of the object instance. (*2)

- *1 : The embedded SQL statement described in a class definition (object-oriented programming function) is invalid.
- *2 : The embedded SQL statement described in a program definition is invalid.

17.2.8.2 Using the ODBC Information Setup Tool

The ODBC information setup tool allows you to set information in the ODBC information file in order to access remote databases via ODBC.

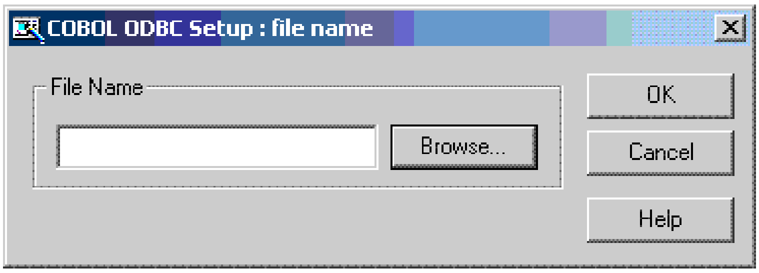
The following functions are provided in ODBC information setup tool.

- Select an ODBC information file
- Set server information
- Set default connection information
- Set connection validation scope.

To use the ODBC information setup tool, follow the steps below (Refer to the online help for additional details).

1. Start the ODBC information setup tool.

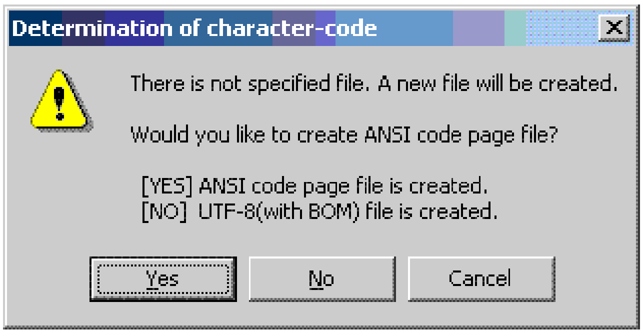
   Run the SQLODBCS.EXE to start the ODBC information setup tool.



2. Select an ODBC information file.

   Specify an ODBC information file to set up connection information.

The file can be specified in ANSI code page (SHIFT-JIS) and UTF-8 (with BOM). When you create a new file, select a character code using the following dialog. When "Yes" is selected, an ANSI code page file is created. When "No" is selected, a UTF-8 (with BOM) file is created.



Determination of character-code

There is not specified file. A new file will be created.

Would you like to create ANSI code page file?

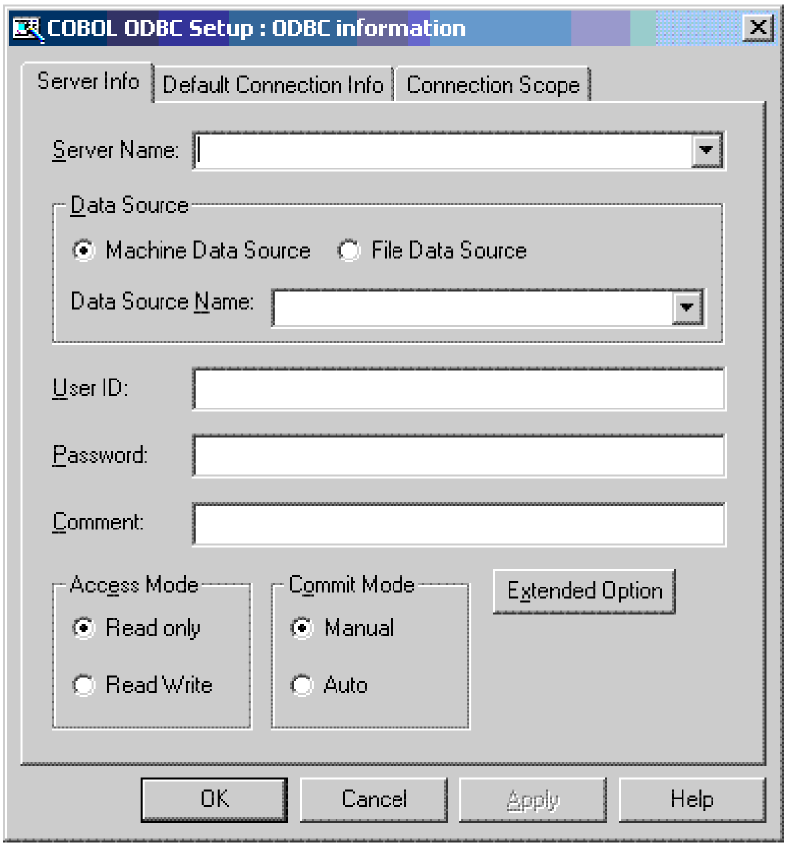
[YES] ANSI code page file is created.
[NO] UTF-8(with BOM) file is created.

When an existing file is opened, the code of the file (ANSI code page or UTF-8 (with BOM)), is automatically determined.

When a UTF-8 (with BOM) file is opened, "UTF-8" is displayed in the title of the ODBC information setup tool.

3. Setup the server information.

   Use the CONNECT statement or select default connection information to set the server name.



COBOL ODBC Setup : ODBC information

Server Name:

Data Source

● Machine Data Source ○ File Data Source

Data Source Name:

User ID:

Password:

Comment:

Access Mode

● Read only

○ Read Write

Commit Mode

● Manual

○ Auto

Information on each server is set by specifying the "Server name".

Each new set of "Server name" information is added to the ODBC information file. In the server information tag, each "Server name" section sets the following information.

Server information of ODBC information setting utility		Information name
Data source	Machine data source/file data source option button	@SQL_DATASRC_KIND
	Data source name	@SQL_DATASRC

Server information of ODBC information setting utility	Information name
User ID	@SQL_USERID
Password	@SQL_PASSWORD
Access mode	@SQL_ACCESS_MODE
Commit mode	@SQL_COMMIT_MODE

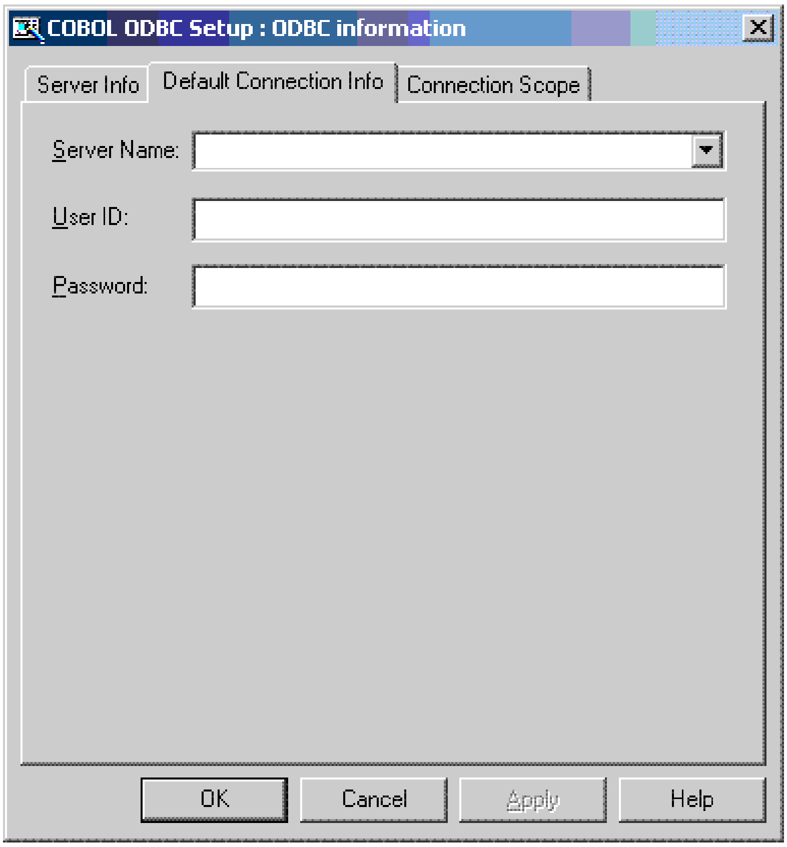
When the "Expand option" button is clicked, the expand option dialog is displayed. In the expand option tag, each "Server name" section sets the following information.

Expand option dialog of ODBC information setting utility	Information name
Cursor simultaneous execution	@SQL_CONCURRENCY
Kind of cursor	@SQL_CURSOR_TYPE
ODBC cursor library	@SQL_ODBC_CURSORS
Query timeout time	@SQL_QUERY_TIMEOUT

For details, refer to "17.2.8.1.2 Creating an ODBC Information File".

4. Setup the default connection information.

   Use the CONNECT statement with the DEFAULT specification to set default connection information (server name, user ID, and password).



In the default connection information tag, "SQL DEFAULT INFO" section sets the following information.

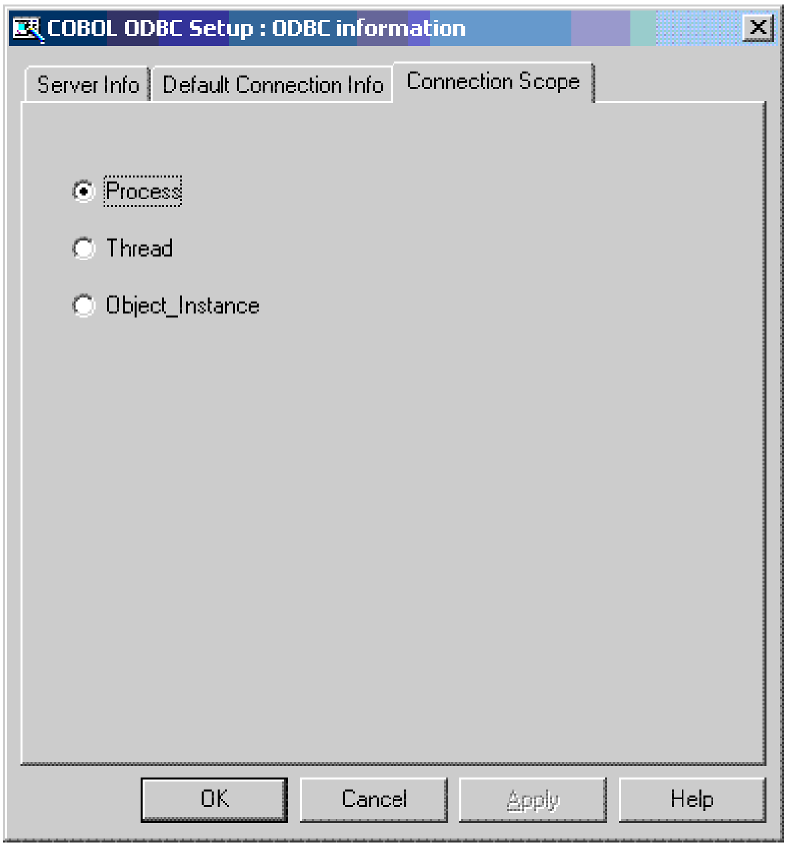
Default connection information of ODBC information setting utility	Information name
Server name	@SQL_SERVER

Default connection information of ODBC information setting utility	Information name
User ID	@SQL_USERID
Password	@SQL_PASSWORD

For details, refer to "Defining Default Connection Information" in "17.2.8.1.2 Creating an ODBC Information File".

5. Setup the connection validation scope.

   Specify the scope in which the connection becomes valid.



● Process
○ Thread
○ Object_Instance

In the connection useful range tag, the "CONNECTION SCOPE" section sets the following information.

Connection useful range of ODBC information setting utility	Information name
Option button of process/thread/object instance	@SQL_CONNECTION_SCOPE

For details, refer to "Defining Connection Validation Scope" in "17.2.8.1.2 Creating an ODBC Information File".

17.2.8.3 Maximum Length of Information Specified in the ODBC Information File

"Table 17.9 ODBC information file specifications" shows the maximum length of information specified in an ODBC information file.

Table 17.9 ODBC information file specifications

Information Type	Maximum Length	Remarks
User ID	32 bytes	The maximum length depends on the specification of the data source used for establishing the connection. Refer to the ODBC manual for more information about the ODBC driver environment.
Password	32 bytes	
Server name	32 bytes	-
Data source name	32 bytes	-

Note

Use the ODBC Information Setup Tool to set up a password. The password must be encrypted. Do not edit the password with an editor.

17.2.8.4 Preparing Linkage Software and the Hardware Environment

Prepare the linkage software and hardware environment for a COBOL application program to access a server database through ODBC using the following procedures:

Setting Up the ODBC Environment

- Install ODBC in the Windows system. If the ODBC environment is installed in the Windows system for the first time, use the ODBC setup supported with the ODBC driver.
- ODBC is added to the Windows control panel. Install the ODBC driver, then create a data source. Start the ODBC data source administrator (usually in a Windows control panel group), then set it up. The ODBC driver is installed and the data source is defined.

  Note

  When a COBOL program using the remote database access (ODBC) function is invoked from a service, the data source must be defined as a system data source. The system data source is defined for the computer, not for the user. All users, including service tasks, can recognize this data source. For details on the system data source, refer to the online Help information on the ODBC data source administrator. For more details, see "18.1.3 Programs Running under a Service".

- Prepare the ODBC driver environment. For information on the ODBC driver environment, refer to the manual and help information of the ODBC driver. The help information can be referenced by starting the ODBC data source administrator.

Confirming a Connection with the Data Source

A database usually supports programs for a client to operate a server database. Before executing a COBOL application program using ODBC, use the programs to check whether the client and server are connected normally.

Execute the COBOL application program after completing the setup of the ODBC control panel, with the runtime environment information that is placed in initialization file (COBOL85.CBR), and the ODBC information file.

17.2.9 Embedded SQL Keyword List

This section gives a keyword list for embedded SQL.

[A]

ABSOLUTE

ADA

ADD

ALL

ALLOCATE

ALTER

AND

ANY

ARE

AS

ASC

ASSERTION

AT

AUTHORIZATION

AVG

[B]

BEGIN

BETWEEN

BIND

BIT

BIT_LENGTH

BY

[C]

CASCADE

CASCADED

CASE

CAST

CATALOG

CHAR

CHAR_LENGTH

CHARACTER

CHARACTER_LENGTH

CHARACTER_SET_CATALOG

CHARACTER_SET_NAME

CHARACTER_SET_SCHEMA

CHECK

CLOSE

COALESCE

COBOL

COLLATE

COLLATION

COLLATION_CATALOG

COLLATION_NAME

COLLATION_SCHEMA

COLUMN

COMMIT

CONNECT

CONNECTION

CONSTRAINT

CONSTRAINTS

CONTINUE

CONVERT

CORRESPONDING COUNT

CREATE

CURRENT

CURRENT_DATE

CURRENT_TIME

CURRENT_TIMESTAMP_ CURSOR

[D]

DATA

DATE

DATETIME_INTERVAL_CODE

DATETIME_INTERVAL_PRECISION

DAY

DEALLOCATE

DEC

DECIMAL

DECLARE

DEFAULT

DEFERRABLE

DEFERRED

DELETE

DESC

DESCRIBE

DESCRIPTOR

DIAGNOSTICS

DICTIONARY

DISCONNECT

DISPLACEMENT

DISTINCT

DOMAIN

DOUBLE

DROP

[E]

ELSE

END

END-EXEC

ESCAPE

EXCEPT

EXCEPTION

EXEC

EXECUTE

EXISTS

EXTERNAL

EXTRACT

[F]

FALSE

FETCH

FIRST

FLOAT

FOR

FOREIGN

FORTRAN

FOUND

FROM

FULL

[G]

GET

GLOBAL

GO

GOTO

GRANT

GROUP

[H]

HAVING

HOUR

[I]

IDENTITY

IGNORE

IMMEDIATE

IN

INCLUDE

INDEX

INDICATOR

INITIALLY

INNER

INPUT

INSENSITIVE

INSERT

INTEGER

INTERSECT

INTERVAL

INTO

IS

ISOLATION

[J]

JOIN

[K]

KEY

[L]

LANGUAGE

LAST

LEFT

LENGTH

LEVEL

LIKE

LIST

LOCAL

LOWER

[M]

MATCH

MAX

MIN

MINUTE

MODULE

MONTH

MUMPS

[N]

NAME

NAMES

NATIONAL

NCHAR

NEXT

NONE

NOT

NULL

NULLABLE

NULLIF

NUMERIC

[O]

OCTET_LENGTH

OF

OFF

ON

ONLY

OPEN

OPTION

OR

ORDER

OUTER

OUTPUT

OVERLAPS

[P]

PARTIAL

PASCAL

PLI

POSITION

PRECISION

PREPARE

PRESERVE

PREVIOUS

PRIMARY

PRIOR

PRIVILEGES

PROCEDURE

PUBLIC

[R]

RELATIVE

RESTRICT

REVOKE

RIGHT

ROLLBACK

ROWS

[S]

SCALE

SCHEMA

SCROLL

SECOND

SECTION

SELECT

SEQUENCE

SET

SIZE

SMALLINT

SOME

SQL

SQLCA

SQLCODE

SQLERARY

SQLERRD

SQLERROR

SQLSTATE

SQLWARNING

START

SUBSTRING

SUM

SYSTEM

[T]

TABLE

TEMPORARY

THEN

TIME

TIMESTAMP

TIMEZONE_HOUR

TIMEZONE_MINUTE

TO

TRANSACTION

TRANSLATE

TRANSLATION

TRUE

TYPE

[U]

UNION

UNIQUE

UNKNOWN

UPDATE

UPPER

USAGE

USER

USING

[V]

VALUE

VALUES

VARCHAR

VARIABLES

VARYING

VIEW

[W]

WHEN

WHENEVER

WHERE

WITH

WORK

[Y]

YEAR

17.2.10 Available Host Variable in Embedded SQL Statements

"Table 17.10 Available host variable in embedded SQL statements" shows the available host variable in embedded SQL statements.

Table 17.10 Available host variable in embedded SQL statements

	Manipulate a single column at a time		Manipulate multiple columns at a time	
Embedded SQL statement	Host variable with a single column specified	Host variable with multiple column specified	Host variable with multiple rows specified	Host variable with a table specified
Data Manipulation without Using Cursor				
SELECT statement	Yes	Yes	Yes	Yes
DELETE statement (searched)	Yes	No	Yes	No
INSERT statement	Yes	Yes	Yes	Yes
UPDATE statement	Yes	No	Yes	No
Data Manipulation without Using Cursor				
OPEN statement	Yes	No	No	No
CLOSE statement	No	No	No	No
FETCH statement	Yes	Yes	Yes	Yes
DELETE statement (positioned)	No	No	No	No
UPDATE statement (positioned)	No	No	No	No
Dynamic SQL				
PREPARE statement	Yes	No	No	No
EXECUTE statement	Yes	Yes	Yes	Yes
EXECUTE IMMEDIATE statement	Yes	No	No	No
Dynamic SELECT statement	No	No	No	No
Dynamic declare cursor	No	No	No	No
Dynamic OPEN statement	Yes	No	No	No

	Manipulate a single column at a time		Manipulate multiple columns at a time	
Embedded SQL statement	Host variable with a single column specified	Host variable with multiple column specified	Host variable with multiple rows specified	Host variable with a table specified
Dynamic CLOSE statement	No	No	No	No
Dynamic FETCH statement	Yes	Yes	Yes	Yes

Yes : Can be specified

No : Cannot be specified

17.2.11 Correspondence Between ODBC-Handled Data and COBOL-Handled Data

COBOL handles ODBC data according to the corresponding definitions. To understand how an ODBC driver handles ODBC SQL data, refer to ODBC driver user's guide and online help.

Note

Data contents can only be assured when you use the data correspondence defined in the following tables.

Table 17.11 Arithmetic Data Correspondence

ODBC SQL Data Type		COBOL85 Description
Binary	SQL_SMALLINT (SMALLINT)	PIC S9(4) BINARY or PIC S9(4) COMP-5
	SQL_INTEGER (INTEGER)	PIC S9(9) BINARY or PIC S9(9) COMP-5
	SQL_BIGINT (BIGINT)(*1)	PIC S9(18) BINARY or PIC S9(18) COMP-5
Decimal	SQL_DECIMAL (DECIMAL)	PIC S9(p) PACKED-DECIMAL or PIC S9(p)V9(q) PACKED-DECIMAL 1 =< p =< 31 (*2), 1 =< q, p+q =< 31 (*2) The maximum precision is driver-specific.
	SQL_NUMERIC (NUMERIC)	{PIC S9(p) | PIC S9(p)V9(q)} [SIGN IS { LEADING SEPARATE CHARACTER | TRAILING }] 1 =< p =< 31(*2), 1 =< q, p+q =< 31(*2) The maximum precision is driver-specific.
Internal floating point	SQL_REAL (REAL)	COMP-1
	SQL_DOUBLE (FLOAT)	COMP-2

*1 : Compile in 31-digit extension mode to use this data type.

*2 : Up to 18 digits can be used when compiling in 18-digit compatible mode. Compile in 31-digit extension mode to use up to 31 digits.

If the RCS (SJIS) compile option is specified or no compile option is specified, the correspondence table for character data is as follows.

Table 17.12 Character Data Correspondence

ODBC SQL Data Type		COBOL85 Description
Fixed length	SQL_CHAR (CHAR)	PIC X(n) (*1) PIC N(n) (*1)

Variable	SQL_VARCHAR (VCHAR)	01 data-name-1.
		  49 data-name-2 PIC S9(m) BINARY. (*2)
		  49 data-name-3 PIC X(n). (*1)
		
		or
		
		01 data-name-1.
		  49 data-name-2 PIC S9(m) COMP-5. (*2)
		  49 data-name-3 PIC X(n). (*1)
		
		m = 4 or 9
		
		01 data-name-1. (*1)
		  49 data-name-2 PIC S9(m) BINARY. (*2)
		  49 data-name-3 PIC N(n). (*1)
		
		or
		
		01 data-name-1. (*1)
		  49 data-name-2 PIC S9(m) COMP-5. (*2)
		  49 data-name-3 PIC N(n). (*1)
		
		m = 4 or 9

- *1 : There are some restrictions on n, the number of characters, depending on the version of the ODBC driver manager supported by the database driver and the database driver specifications. For example, if ODBC2.0 is used, the following restrictions apply:

  ```
  X(n) 1=< n =< 254
  N(n) 1=< n =< 127
  ```

- *2 : The number of characters is specified for the length part of variable-length character data.

If the RCS (UTF16) compile option is specified, or no compile option is specified, the correspondence table for character data is as follows.

Table 17.13 Character Data Correspondence

ODBC SQL Data Type		COBOL85 Description
Fixed length	SQL_CHAR (CHAR)	PIC X(n) (*1)
	SQL_WCHAR(WCHAR)	PIC N(n) (*1) (*3)

Variable	SQL_VARCHAR (VCHAR)	01 data-name-1.
		  49 data-name-2 PIC S9(m) BINARY. (*2)
		  49 data-name-3 PIC X(n). (*1)
		
		or
		
		01 data-name-1.
		  49 data-name-2 PIC S9(m) COMP-5. (*2)
		  49 data-name-3 PIC X(n). (*1)
		
		m = 4 or 9

ODBC SQL Data Type		COBOL85 Description

	SQL_WVARCHAR (WVARCHAR)	`01  data-name-1.                      (*3)`
		`  49  data-name-2 PIC S9(m) BINARY.    (*2)`
		`  49 data-name-3 PIC N(n).            (*1)`
		
		`or`
		
		`01  data-name-1.                      (*3)`
		`  49  data-name-2 PIC S9(m) COMP-5.    (*2)`
		`  49  data-name-3 PIC N(n).           (*1)`
		
		`m = 4 or 9`

- *1 : There are some restrictions on n, the number of characters, depending on the version of the ODBC driver manager supported by the database driver and the database driver specifications. For example, if ODBC2.0 is used, the following restrictions apply:

```
X(n) 1=< n =< 254
N(n) 1=< n =< 127
```

- *2 : The number of characters is specified for the length part of variable-length character data.
- *3 : Depends on the database supporting Unicode and the products used.

Table 17.14 Date Correspondence

ODBC SQL data type		COBOL representation
Date data type	`SQL_DATE`	`PIC X(n)          (*1)`
	`SQL_TIMESTAMP`	

- *1 : There are some restrictions on n the number of characters depending on the version of the ODBC driver manager supported by the database driver and the database driver specifications.

17.2.12 SQLSTATE, SQLCODE, and SQLMSG

This section describes information that is shown in the notification area of the COBOL, ODBC driver manager, ODBC driver, or DBMS if an SQL statement is executed using ODBC.

The following table explains information posted as SQLSTATE, SQLCODE, and SQLMSG.

Table 17.15 SQLSTATE, SQLCODE, and SQLMSG information

Information	Value Posted	Explanation	Programmer Response
SQLSTATE	000000	Normal termination	-
	5 alphanumeric characters	Error or warning during execution of an SQL statement	CHECK SQLCODE and SQLMSG, and take action if error is detected. For SQL STATE details, see the ODBC environment manuals (covering ODBC drivers and DBMS).
SQLCODE	0	Normal termination	-
	Positive or negative integer other than 0	Error or warning during execution of an SQL statement	Check the cause of the error according to the ODBC environment (such as ODBC or DBMS) manual, and take corrective action on the error.
SQLMSG	Blank (no output)	Normal termination	-
	Message (character string)	The message explains an error or warning posted during execution of an SQL statement	Check the cause of the error according to the displayed message, and take corrective action on the error.

The following describes errors that might be detected by COBOL during execution of an embedded SQL statement. The information is posted as SQLSTATE, SQLCODE, and SQLMSG.

Table 17.16 SQLSTATE, SQLCODE, and SQLMSG error information

SQLSTATE	SQLCODE	SQLMSG	Programmer response
99999	-999999999	The connection has been established with the same connection name	A connection name must not be specified for more than one connection. Specify a unique name for each connection.
9999A	-999999990	The number of connections exceeds the maximum.	The number of connections exceeds the maximum specified in the COBOL system. Decrease the number of connections. The maximum number of connections can vary depending on the ODBC environment. Refer to the ODBC environment manual and take action.
9999B	-999999800	The specified connection does not exist.	SQL statements cannot be executed because the connection is not active. Check the SQL statement sequence in the program, and take corrective action on the error.
9999D	-999999200	The number of occurrences specified in the FOR clause is too large.	In the FOR clause, specify the number of occurrences which is equal to or less than the value set in the OCCURS clause.
9999E	-999999100	An incorrect value is specified in the FOR clause.	In the FOR clause, specify the number of occurrences which is equal to or greater than one (1).
999SA	-999999700	The cursor is not opened.	The cursor is not ready. Check the sequence of SQL statements using the cursor, and take corrective action for the error.
999SB	-999999600	The prepared statement is not prepared.	Check the sequence of dynamic SQL statements, and take corrective action for the error.
999SC	-999999500	The cursor has already been opened.	The cursor can be used. Examine the order of the SQL statement using the cursor and cope with it.
?????	-999999992	Invalid process occurred.	A system error occurred.

17.2.13 Notes on Using the ODBC Driver

This section explains usage notes for ODBC drivers. Pay special attention to this section, since it covers important usage information.

17.2.13.1 Notes on SQL Statement Syntax

DATA DIVISION

- The correspondence between ODBC data types and host variable data types is defined for COBOL. See "17.2.11 Correspondence Between ODBC-Handled Data and COBOL-Handled Data" for more information. Use a host variable corresponding to the data type that the ODBC driver prescribes. Data contents are assured only for the defined data correspondence.

- Some data types may not be handled by COBOL as host variables depending on the ODBC driver.

PROCEDURE DIVISION

- Use the COMMIT or ROLLBACK statements to terminate transactions and before using the DISCONNECT statement to terminate a connection. Otherwise, the operation process results may not be reflected in the database

- The SQL descriptor area cannot be used. Therefore, do not specify the following embedded SQL statements:

  - INTO and USING clauses in which a descriptor name is specified

    - SQL statements for the descriptor area: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR, and DESCRIBE statements

- You cannot specify the DEALLOCATE PREPARE statement.

- You cannot specify any SQL statement for data definition (DDL).

- When using a dynamic SQL statement such as the PREPARE, EXECUTE, or EXECUTE IMMEDIATE, you can only use NOT FOUND in the WHENEVER clause for following SQL statements:

    - SELECT statement (only with a PREPARE/EXECUTE statement)

    - UPDATE statement (search)

    - UPDATE statement (positioning)

    - DELETE statement (search)

    - DELETE statement (positioning)

    - INSERT statement

    - Dynamic UPDATE statement (search)

    - Dynamic UPDATE statement (search)

    - Dynamic DELETE statement (search)

    - Dynamic DELETE statement (positioning)

- The same statement identifier used in a dynamic cursor declaration cannot be specified in the EXECUTE statement. In the same way, the same statement identifier used in the EXECUTE statement cannot be specified to the dynamic cursor declaration.

- Details on the embedded SQL statement syntax comply with the database specification and the specification of the database related products.

17.2.13.2 Notes on Executing Embedded SQL Statements

- If you access to the database from NetCOBOL by using the ODBC environment, you need to build the ODBC driver for the database accessed, and the environment needed by the ODBC driver.

- Embedded SQL statements that can be used, and the methods of specifying them, vary between ODBC drivers.

  If embedded SQL statements are specified in the COBOL source program, refer to the product manuals as well as the COBOL specification for:

    - The ODBC driver

    - Software and hardware related to the ODBC driver

    - The database management system

- An optional value of the ODBC environment may be changed an embedded SQL statement, the ODBC information file, or the data source.

  In this case, informative messages are written out by each ODBC driver when the embedded SQL statement is executed. Because using the embedded exception declaration is regarded as an error, change the program if necessary.

- When an embedded SQL statement (other than the data manipulator statement) is executed dynamically, the following phenomena may occur depending on each ODBC driver:

  Do not dynamically execute embedded SQL statements except for data manipulation statements.

```
Phenomenon:
 SQLSTATE: 02000;
 SQLCODE: +100;
 SQLMSG: No data
```

- Use the COMMIT or ROLLBACK statements to terminate transactions and before using the DISCONNECT statement to terminate a connection, otherwise the results of processing may not be reflected in the database.

- If X'00' is stored as a value in character data, the storage and fetch results are not guaranteed.

- If you are using floating-point data, a conversion error may occur, because servers and clients may differ in their floating-point definitions. Therefore, you may not get expected results.

- Details on the embedded SQL statement syntax comply with the database specification, and the specification of the database-related products.

- You can use the UPDATE statement (positioning) and DELETE statement (positioning) with a dynamic SQL statement when OBJECT_INSTANCE or THREAD is specified for @SQL_CONNECTION_SCOPE (specification of the connection scope).

- It is decided whether position UPDATE or DELETE statement can be executed by combining @SQL_CONCURRENCY and @SQL_CURSOR_TYPE.

- If the cursor type is omitted or specified as FORWARD_ONLY in @SQL_CURSOR_TYPE, the following error occurs when FETCH PRIOR, FETCH FIRST, or FETCH LAST is used (error details will depend on the database). Specify a cursor type that corresponds to the database being used (ODBC driver, database, database-related product): specify STATIC, KEYSET_DRIVEN, or DYNAMIC in @SQL_CURSOR_TYPE.

  ```
  Phenomenon:
    SQLSTATE : S1106
    SQLCODE : +0
    SQLMSG : Fetch type has exceeded the range.
  ```

17.2.13.3 Notes Specific to Each ODBC Driver

This section provides the notes on specific ODBC drivers.

Note on use of Oracle ODBC driver

The following can not be called using CALL statements:

- Functions

- Packaged stored procedures

- Packaged functions

Notes on use of Microsoft SQL Native Client ODBC driver

- When the CONNECT statement is executed, the following event occurs with the following symptoms (information output) and the connection ends normally:

  ```
  SQLSTATE: 01000;
  SQLCODE:  +5701;
  SQLMSG: (SQL server) The database context is changed to a "database name"
  ```

  Specifying the WHENEVER statement before the CONNECT statement causes an error. Therefore, specify the WHENEVER statement after the CONNECT statement.

- The positional DELETE and positional UPDATE statements can be used only under specific conditions. Refer to "Specific Conditions" of "Notes Peculiar to Each ODBC Driver".
  Moreover, refer to the following for the note when you specify @SQL_CONCURRENCY. And refer to "Creating ODBC information file" and the manual of data source (ODBC driver, database and database-related products). Refer to "Creating ODBC information file".

- The positional DELETE and positional UPDATE statements can be used only when @SQL_CONCURRENCY is specified for server information in the ODBC information file. However, one or more unique indexes may have to exist in the table that becomes the object of a cursor definition. An example of creating a unique index is presented below:

  - Add a PRIMARY KEY restriction or a UNIQUE restriction to the arbitrary column when you create the table.

  - Create unique indexes to the arbitrary column of the existing table (for instance, CREATE UNIQUE INDEX...).

Note

The value in the arbitrary column must be unique. If there is a value that overlaps with another one, an error message is returned and the process fails.

- When a positional DELETE or UPDATE statement is executed, the following message is returned, and the statement fails (because @SQL_CURRENCY is specified without creating a unique index):

```
SQLSTATE: S1009;
SQLCODE: +00016929;
SQLMSG: [Microsoft][ODBC SQL Server Driver][SQL Server] Cursor is read only.
```

- The following causes an operational example when @SQL_CONCURRENCY is specified.

  Two clients, client A and client B in order, update the same data in the same column, same table and on the same server using a positional UPDATE statement.

  The lock, unlock and state of each client operates as follows (execution order is in parentheses):

```
Client A
 (1) DECLARE cursor-name CURSOR FOR ...
 (3) OPEN cursor-name
 (5) FETCH cursor-name INTO ...
 (7) UPDATE table-name .. WHERE CURRENT OF cursor-name
 (9) CLOSE cursor-name
 (11) COMMIT

Client B
 (2) DECLARE cursor-name CURSOR FOR ...
 (4) OPEN cursor-name
 (6) FETCH cursor-name INTO ...
 (8) UPDATE table-name .. WHERE CURRENT OF cursor-name
 (10) CLOSE cursor-name
 (12) COMMIT
```

  - Case 1:

    - @SQL_CONCURRENCY=LOCK is specified for client A and @SQL_CONCURRENCY=LOCK, ROWVER or VALUES is specified for client B

    - Result: When the (5) FETCH statement of client A is executed, the table specified for the (1) DECLARE statement is locked until the transaction of client A proceeds to completion with the (11) COMMIT statement. The (6) FETCH statement of client B waits for execution.

  - Case 2:

    - @SQL_CONCURRENCY=ROWVER or VALUES is specified for client A and @SQL_CONCURRENCY=LOCK is specified for client B

    - Result: When the (6) FETCH statement of client B is executed, the table specified for the (1) DECLARE statement is locked until the transaction of client B proceeds to completion with the (12) COMMIT statement. The (7) UPDATE statement of client A waits for execution.

  - Case 3:

    - @SQL_CONCURRENCY=ROWVER is specified for client A and @SQL_CONCURRENCY=ROWVER is specified for client B

    - Result: When the (7) UPDATE statement of client A is executed, the table specified for the (1) DECLARE statement is locked until the transaction of client A proceeds to completion with the (11) COMMIT statement. Also, the (8) UPDATE statement of client B error occurs. If the order of the (6) FETCH and (7) UPDATE statements are reversed, the (6) FETCH statement waits for execution.

- When the following options are specified with the cursor to which data is not updated, the improvement of the performance can be expected.

  - @SQL_CONCURRENCY = READ_ONLY

  - @SQL_CURSOR_TYPE = FORWARD_ONLY

- Connected error occurs when SQL statement is executed by one connection while using a predetermined result set of the cursor. This error can be evaded by changing the cursor type to the server cursor. The change to the server cursor is revocable by @SQL_CONCURRENCY and SQL_CURSOR_TYPE. Because it is READ_ONLY and @SQL_CURSOR_TYPE is FORWARD_ONLY, @SQL_CONCURRENCY is a predetermined result set in default in this value.

- When either another cursor is opened, or SQL statements having no relation with the open cursor are executed (and do not close the cursor), the following event occurs with the following symptoms:

  ```
  SQLSTATE: S1000;
  SQLCODE:  0
  SQLMSG: (SQL server) The connection is the busy state for the result of other commands.
  ```

  Close the open cursor, then execute the SQL statements or specify the @SQL_ODBC_MARS=ON. The value of @SQL_ODBC_MARS can be OFF (default) or ON. If specified OFF, MARS is not used; if specified ON, MARS is used. MARS can be used only with the SQL Native Client ODBC driver. For details on MARS, refer to Using Multiple Active Result Sets (MARS) in SQL Server 2005 Books Online.

  To execute these SQL statements, the data source environment and COBOL85 conditions should match.

  Refer to "Specific Conditions" of "Notes Peculiar to Each ODBC Driver"

- OPEN fails, and will generate the following events, when the cursor is defined and opened by the following cursor declaration:

  ```
  Cursor defined:
    DECLARE [cursor name] CURSOR FOR SELECT ... FOR UPDATE
  Event:
    SQLSTATE: 37000;
    SQLCODE:  +00001003;
    SQLMSG: (SQL Server) The FOR UPDATE phrase is permitted only with DECLARE CURSOR.
  ```

  The cursor operation cannot be executed under this condition.

  To execute the OPEN statement, the data source environment and NetCOBOL should have matching conditions.

  Refer to "Specific Conditions" of "Notes Peculiar to Each ODBC Driver"

Specific Conditions

This is the case where the value of @SQL_CONCURRENCY (concurrency of cursor) is LOCK, ROWVER or VALUES to be specified to the server information of an ODBC information file.

17.2.13.4 Quantitative Limits of Embedded SQL Statement at Execution

- An embedded SQL statement can be up to 16,384 bytes in length. The maximum length may be shorter depending on the ODBC driver environment.

  For example, the maximum length of an embedded SQL statement may be shorter than 4,096 bytes because of the maximum length of data transferred by the software product responsible for the network.

  If the ODBC driver processes data in an embedded SQL statement from COBOL, the embedded SQL statement may be longer than specified in the COBOL source program.

- The sum of input variable length and output variable length may depend on the ODBC driver environment.

- The maximum length of the data source message string that can be set for SQLMSG is 1024 bytes. Messages longer than 1024 bytes are truncated. The number of parameters in the stored procedure can be up to 256.

17.2.14 Deadlock Exits

The term deadlock refers to the state in which two or more programs that, on attempting to access the same database simultaneously, sit back in semi-permanent standby mode while waiting to be granted exclusive access to the database. When a deadlock occurs, the database notifies the programs.

If the program contains a deadlock exit procedure, a deadlock exit subroutine can be called whenever the program is deadlocked. The deadlock exit procedure defines the processing to be performed when a deadlock occurs.

Deadlock exit procedures are identified by USE FOR DEAD-LOCK statements. For details on the USE FOR DEAD-LOCK statement, see "USE FOR DEAD-LOCK Statement" of the "COBOL Language Reference".

17.2.14.1 Overview of Deadlock Exit Support

NetCOBOL provides you with the ability to return control from a deadlocked program to a program containing a deadlock exit procedure. You do this by invoking a NetCOBOL subroutine, COB_DEADLOCK_EXIT, from the deadlocked COBOL program.

The deadlock exit is registered in the NetCOBOL runtime system when the program containing the deadlock exit procedure is executed, and the registration is annulled when the EXIT PROGRAM statement of the program containing the deadlock exit is executed.

When the COB_DEADLOCK_EXIT subroutine is invoked, control is returned to the deadlock exit in the program in the call chain that is closest to the program that invoked the subroutine. At this time, any programs between the program that invoked the subroutine and the program containing the deadlock exit procedure to which control is returned are terminated as if the each executed an EXIT PROGRAM statement.

For details on invoking the COB_DEADLOCK_EXIT subroutine, see "G.1.5 Deadlock Exit Subroutine".

Note

- Any programs between the program that invoked the subroutine and the program containing the deadlock exit that are written in a different language, are not recovered. Also, when restarting processes with a deadlock exit, programs written in a different language must be re-entered. This means that programs written in a different language must be re-enterable and not require resource recovery.

- It is up to the COBOL code to determine whether or not a program is deadlocked.

- After its execution, a USE FOR DEAD-LOCK procedure must pass control to a procedure that is not a declaratives procedure in a GO TO statement. If the control is transferred to the end of the USE FOR DEAD-LOCK procedure, the program is abnormally terminated with error JMP0004I-U.

- Database transactions are cancelled when a deadlock is notified. Apart from the resources involved in transactions cancelled by the database, it is up to the COBOL code to determine which files and other resources are to be recovered. This requires careful architecture of your programs and file handling so that the appropriate USE FOR DEADLOCK procedures have access to files that may be affected by the recovery actions.

- With the deadlock exit you can resume execution in the code outside the USE FOR DEAD-LOCK DECLARATIVE procedure by using a GO TO statement. However, you need to be aware that the program state and environment remain the same as they were before the COB_DEADLOCK_EXIT subroutine was invoked. Thus if the program logic depends on any values or states, it is up to the deadlock exit procedure code to set those values and states appropriately for processing to resume.

- A deadlock exit subroutine should not be invoked in a USE procedure.

- If the compiler option LANGLVL(68/74) is specified, and there is a possibility that sections or paragraphs referenced in a PERFORM statement are executed by other methods (such as a GO TO statement), a deadlock exit subroutine should not be invoked in those sections or paragraphs.

Chapter 18 Server Type Applications

This chapter explains the background processing function.

18.1 Background Processing

18.1.1 How to suppress an I/O operation on the screen

Operation of the screen is not generally necessary in background processing. This section explains how to suppress the screen handling function.

The screen is displayed when user executes a function that uses the screen in the source program, and when the COBOL runtime system notifies you of the application status (independent of the user function).

Examples follow:

- Screen I/O functions users can use in the source program:
  - Screen function
  - ACCEPT/DISPLAY function
- The functions by which the COBOL runtime uses the screen as user interface
  - Execution time messages
  - NetCOBOL icons
  - A confirmation message displayed when the console screen is closed

If your application does not perform I/O with the screen, do not specify any screen I/O functions in the source program. However, the ACCEPT/DISPLAY functions can perform I/O with a file instead of using screen I/O. See "10.1 ACCEPT/DISPLAY Function"for details.

This section explains how to suppress the function by which the COBOL runtime uses the screen as a user interface.

- To change the output target of a message of execution time, the message can be output to a file by specifying the file name in an environment variable. Also, the message can be output to the event log.
  - How to output the message to a file by specifying the file name in an environment variable

    Specify the following execution environment:

    ```
    @MessOutFile = file name
    ```

    For details, see "5.4.1.56 @MessOutFile(Set a Message Output File)".

  - How to output the message to the event log:

    Specify the following execution environment:

    ```
    @CBR_MESSAGE = EVENTLOG.
    ```

    For details, see "5.4.1.30 @CBR_MESSAGE(Specify the runtime message output destination)".

  When both of the above specifications are executed, the priority is as shown below.

  ```
  (High)  @CBR_MESSAGE > @MessOutFile  (Low)
  ```

  To inhibit all messages of execution time except U level message, and to output the U level message, specify the following environment variable information:

  ```
  @NoMessage = YES
  ```

  For details, see "5.4.1.58 @NoMessage(Set to Suppress Runtime Messages)".

- To suppress displaying NetCOBOL icons, specify the following environment variable:

  ```
  @ShowIcon=NO
  ```

  See "5.4.1.64 @ShowIcon(Specifying the suppression of COBOL icon display)" for details.

- To suppress a confirmation message displayed when the console screen is closed, specify the following environment variable:

  ```
  @WinCloseMsg=OFF
  ```

  See "5.4.1.65 @WinCloseMsg(Set a Display Message When the Window Close)" for details.

  If log information is written to the console screen by a DISPLAY statement, a message to notify you that COBOL has closed the console screen is displayed when the application is closed. If you want to suppress this message, specify this runtime environment variable.

18.1.2 How to Use the Command Prompt Window

You can perform input of an ACCEPT statement, output of a DISPLAY statement, and display of a runtime message with the command prompt window.

When you activate a COBOL application from the command prompt window (or from a batch file), the command prompt window and the window generated by the batch file are used as a console window.

When you activate a COBOL application from an icon, the command prompt window is generated at the same time and it is used as a console window.

Specify the compile option MAIN(MAIN) if the main program is written in COBOL. Specify the execution environment variable @CBR_CONSOLE=SYSTEM if the main program is written in another language.

Specifying of the environment variable information @CBR_CONSOLE=SYSTEM may not work correctly when a main program is written in COBOL and the compile option MAIN(WINMAIN) is specified, or if a main program is written in another language and the WinMain function is used. You should use @CBR_CONSOLE=COBOL (the default value) in this case. The following is the behavior when the environment variable information @CBR_CONSOLE=SYSTEM is specified.

- A new command prompt window is generated.

  A window generated by the command prompt window, or a batch file you activated, cannot be used as a console screen. A new command prompt window is generated when the console function is used, and is used as a console screen.

- You may get I/O failures. Avoid the failure by using START command. If input/output to the command prompt window fails, you may not be able to confirm the event of an error because writing a runtime message to the command prompt window also fails. You should write the run time message to a file.

18.1.3 Programs Running under a Service

Application environment under a service

When a COBOL program is invoked from a service, it may fail to obtain printer or network drive information needed to run normally.

The possible cause of this is that the COBOL program invoked from the service cannot reference information related to each user in the system when attempting to obtain such information as printer information from the system.

Normally, a COBOL program invoked from a service runs with the system account. In this case, user-specific information (such as registry HKEY_CURRENT_USER) is not available to the system service. Therefore, the COBOL program cannot obtain user-specific information such as the default printer. User-specific information is set up in the system when the user logs onto the system.

Some services permit the user to define a COBOL program so that it can run with a specific user account. The user can log on to the system in advance with the user account with which the COBOL program runs. The COBOL program can then reference user setup information. A service may also be created so a COBOL program can reference user-specific information as system information even when running with the system account. However, as explained earlier, COBOL programs invoked from a service normally cannot reference user-specific information.

Notes on programming

When creating a COBOL program invoked from a service, note the following:

Environment variables

Environment variable information set in the user environment variables cannot be referenced. Set environment variable information required for the COBOL program in the system environment variables.

Printer information

When a COBOL program using a print file or presentation file with or without the FORMAT clause (destination PRT) is executed under a service, specify a printer name explicitly for the file output destination. Printing may fail with a file allocation error if the default printer, a local printer port name (LPTn:), or a communication port name (COMn:) is specified for the output destination.

See

Examples of file allocation, enabling printing under a service

- For a print file without the FORMAT clause
    - Specify the ASSIGN clause.

      SELECT print-file ASSIGN TO S-PRTF.
    - Specify environment variable information.

      PRTF = PRTNAME: FUJITSU VSP4620A
- For a print file with the FORMAT clause
    - Specify a printer information file.

      PRTDRV FUJITSU VSP4620A

Network drive information

Network drive information cannot be referenced. To access a file in a network connection environment, use the UNC specification.

Example

file-ID = \\FILESERVER\FILE\WORK.DAT

ODBC information

User data sources cannot be referenced. Define the data sources as system data sources. For details, see "17.2.8.4 Preparing Linkage Software and the Hardware Environment".

Screen input/output function

Most services do not support input/output through the screen. If a COBOL program invoked from a service not supporting input-output through the screen attempts to input-output through the screen, an invisible screen enters an input wait state, thereby appearing as though the system failed to respond.

To avoid this problem, suppress the COBOL input/output function using the screen. For more information, see "18.1.1 How to suppress an I/O operation on the screen".

Program debugging

To debug a COBOL program invoked from a service, use attached debugging function of NetCOBOL Studio. For more information, refer to "NetCOBOL Studio User's Guide ".

Service

A service for calling a COBOL program must be created for an application running under the service to reference user-specific information as system information.

18.2 Event Log

The event log is a function supported by Windows.
Applications record error information in the event log. This function enables the system administrator to centrally manage application conditions (event monitoring) and thereby enhance troubleshooting efficiency.
This section explains the function for outputting runtime messages or user-defined messages to the event log. For event monitoring, the user can use Event Viewer. For details, refer to the product manual or online Help file.

18.2.1 Function for Outputting Runtime Messages to the Event Log

Runtime messages output by the COBOL runtime system can be output to the event log. Information output to each item of the event log is as follows.

- The message number is output to the Source.
- "NetCOBOL x64" is output to the Event ID.
- The severity code is output to the Type.
- The message is output the Description.

In terms of type, however, level classifications are different for the system than for COBOL runtime messages and so are related as shown below.

Table 18.1 Correspondence between runtime message severity code and event log type

Severity code	Event log type
I (INFORMATION)	WARNING
W (WARNING)	
E (ERROR)	
U (UNRECOVERABLE)	ERROR

Environment variable @CBR_MESSAGE must be specified to use the function to output runtime messages to the event log. For details, see "5.4.1.30 @CBR_MESSAGE(Specify the runtime message output destination)".

See "Runtime Messages" in NetCOBOL Messages for information on the runtime message severity codes.

Note

The user can output a runtime message to another computer (Windows) in the network provided that NetCOBOL is installed in the output destination computer.

18.2.2 Function that Outputs User-defined Information to the Event Log

The user can call the event log output subroutine (COB_REPORT_EVENT) to output information, such as character strings defined by the user in COBOL source programs, to the event log. The items and information the user can specify for the subroutine for output are summarized in the table below. See "G.1.2 Subroutine for Outputting an Event Log" for the interface of the event log output subroutine.

Table 18.2 Event log items that can be specified

Item	Description
Source	Use this item with the event ID to identify an event. The name of an application is normally used for the source. The COBOL interface permits an arbitrary character string in up to 256 bytes to be specified. However, the maximum length of character strings for output is dependent on the quantitative limit of the system. The default is "COBOL Application x64".
Event ID	Use this item with the source to identify an event. A value from 0 to 999 can be specified.
Type	Specify Information, Warning, or Error for the type of event.

Item	Description
Description	Provide an explanation for an event using up to 1,024 bytes. If this item is omitted, no information is output to the Description.
Data	Specify the address and length of the data field to be output if required. The maximum length of the data field that can be output is dependent on the quantitative limit of the system. If this item is omitted, no information is output to the Data.

Note

- User-defined information can be output to another computer (Windows(x64)) in the network. The output destination computer is the same as that to which runtime messages are output. NetCOBOL or NetCOBOL runtime system must be installed in the output destination computer in the same manner as runtime messages. See "18.2.1 Function for Outputting Runtime Messages to the Event Log" for details.

- To output an arbitrary character string other than the default to the Source, set information in the registry of the output destination computer. To set or delete registry information for this function, use the "Registration tool for event logging subroutine". A user having registry key access authority (value referencing/setting or subkey creation/deletion) such as that in the Administrators group should set or delete registry information.

Chapter 19 Multithread Programs

This chapter describes multithread programs.

Creating multithread programs enables COBOL programs to be executed in multithread environments.

19.1 Overview

If a COBOL application is used as a server application, for example by using distributed objects, the executable module may be started for many clients, thereby causing the server load to be greatly increased. Applying multithreading can significantly reduce the resources used, resulting in improved execution performance.

19.1.1 Features

Use of existing COBOL resources in multithread environments

Existing COBOL resources can be used in multithread environments simply by recompiling the programs.

Data and file sharing among threads

It is possible to create programs that make the best use of a multithread feature, sharing resources such as data and files among threads. To create such programs, programmers generally have to program complicated programs so that plural threads will not access a resource concurrently (thread synchronization control). However, in COBOL, the COBOL runtime system performs complicated thread synchronization control automatically, and a subroutine that controls thread synchronization is provided. Thus, programs can be created easily.

Debug supported for multithread programs

Multithread programs can be debugged by using the following functions provided by NetCOBOL.

- The TRACE, CHECK, and COUNT functions
- The debugging function of NetCOBOL Studio

A subroutine used for obtaining process IDs and thread IDs is provided to support debugging of multithread programs.

Note

COBOL provides no function for directly activating threads.

19.1.2 Function Scope

All COBOL functions can be used in multithread environments. However, when you use a function that uses a related product, you should verify that the product is supported.

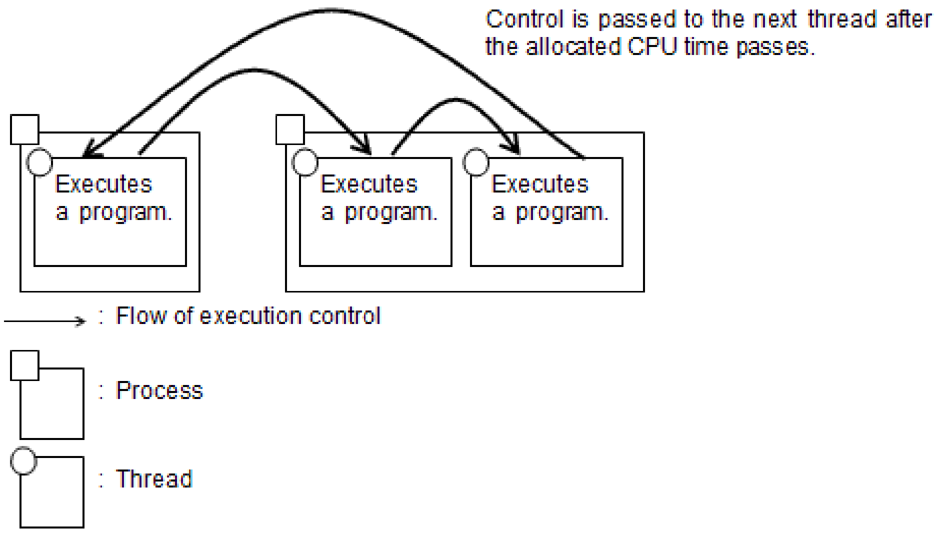
Refer to the software description for the related product.

19.2 Multithread Advantages

19.2.1 What is a Thread?

A thread is a minimum run unit for which execution scheduling is controlled by the operating system. The operating system schedules a thread to be executed, controls its execution time. After the allocated CPU time has passed, the thread is interrupted and CPU time is allocated to the next thread.

A thread is contained in a process. A thread executes a program. The process is composed of resources such as program codes, data, opened files in memory, and dynamically allocated memory. At least one thread is contained in a process.

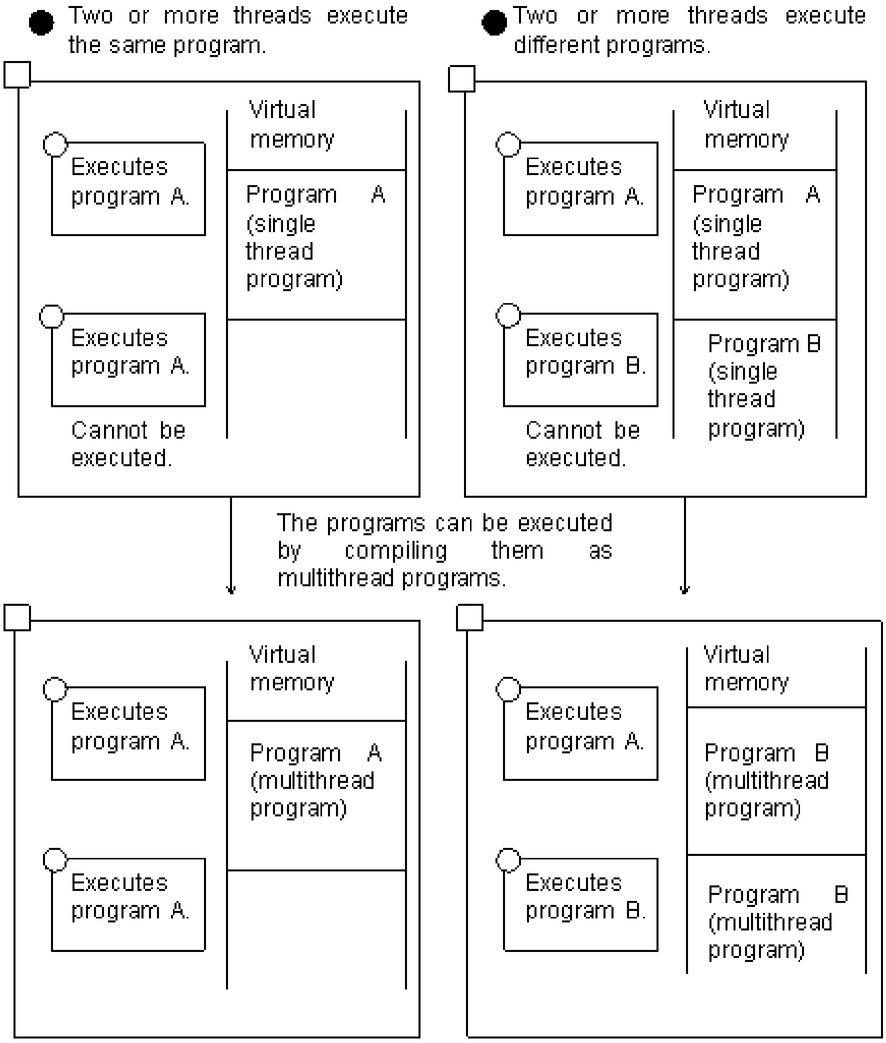

Control is passed to the next thread after the allocated CPU time passes.
Executes a program.
Executes a program.
Executes a program.
: Flow of execution control
: Process
: Thread

The above symbols are used in the following description.

19.2.2 What is a Multithread Program?

For single-thread programs, only one thread can execute COBOL programs in a process. Two or more threads of a multithread program may not execute the same single-thread program or different single-thread programs in a single-thread compilation unit. (see the figure below). For multithread programs, two or more threads (multithreads) in a process can execute COBOL programs.

For COBOL, a program compiled with compiler option THREAD (SINGLE) becomes a single-thread program and a program compiled with compiler option THREAD (MULTI) becomes a multithread program.


● Two or more threads execute the same program.
● Two or more threads execute different programs.
Executes program A.
Executes program A.
Cannot be executed.
Virtual memory
Program A (single thread program)
Executes program A.
Executes program B.
Cannot be executed.
Virtual memory
Program A (single thread program)
Program B (single thread program)
The programs can be executed by compiling them as multithread programs.
Executes program A.
Executes program A.
Virtual memory
Program A (multithread program)
Executes program A.
Executes program B.
Virtual memory
Program B (multithread program)
Program B (multithread program)

19.2.3 Multithread Efficiencies

The efficiencies given below can be implemented by compiling server applications, which use server-activated functions such as distributed objects, as multithread programs. Thus, the applications can be used as high-performance server applications.

High-speed startup

To execute a single-thread program, a process needs to be activated. To activate the process, process space needs to be allocated and an executable file and DLL, if necessary, need to be read from a disk and loaded in the space. Such processing is not required, after the first invocation, if only threads are activated, thereby reducing the activation time.

Memory saving and simple sharing of resources

Because all threads in a process share process space, memory can be saved and resources can be shared among the threads.

19.3 Basic Operation of Multithread Programs

19.3.1 Execution Environment and Run Unit

As described in "9.1.2 COBOL Inter-Language Environment", COBOL supports execution environments and run units. A runtime environment is supported for each process, and a run unit is supported for each thread.

Runtime environment

The runtime environment of a COBOL multithread program is set up when it is called by a process for the first time, and is closed when the process terminates or JMPCINT4 is called. Similarly to a single-thread program, information necessary for executing the

COBOL program, such as an initialization file for execution, is loaded when the runtime environment is set up. When it is closed, resources controlled in the process are released. The resources include factory objects, object instances, system console windows, files used by the ACCEPT/DISPLAY statement, external data/file shared by threads (compiled with specification of compiler option SHREXT, which is a multithread-specific function) and so on.

Run unit

The timing of starting and terminating a run unit of a multithread program is the same as that of a single-thread program. Because two or more threads execute a COBOL program, however, two or more run units exist in one process. When a run unit terminates, resources controlled in the thread are released. The resources include data declared in program definitions (excluding external data/file shared by threads), COBOL console windows, screen windows, and so on.

Note

Be sure to call JMPCINT3 when calling JMPCINT2. Otherwise, the COBOL run unit will not terminate and the resources used by the thread will not be released, thereby causing a memory leak.

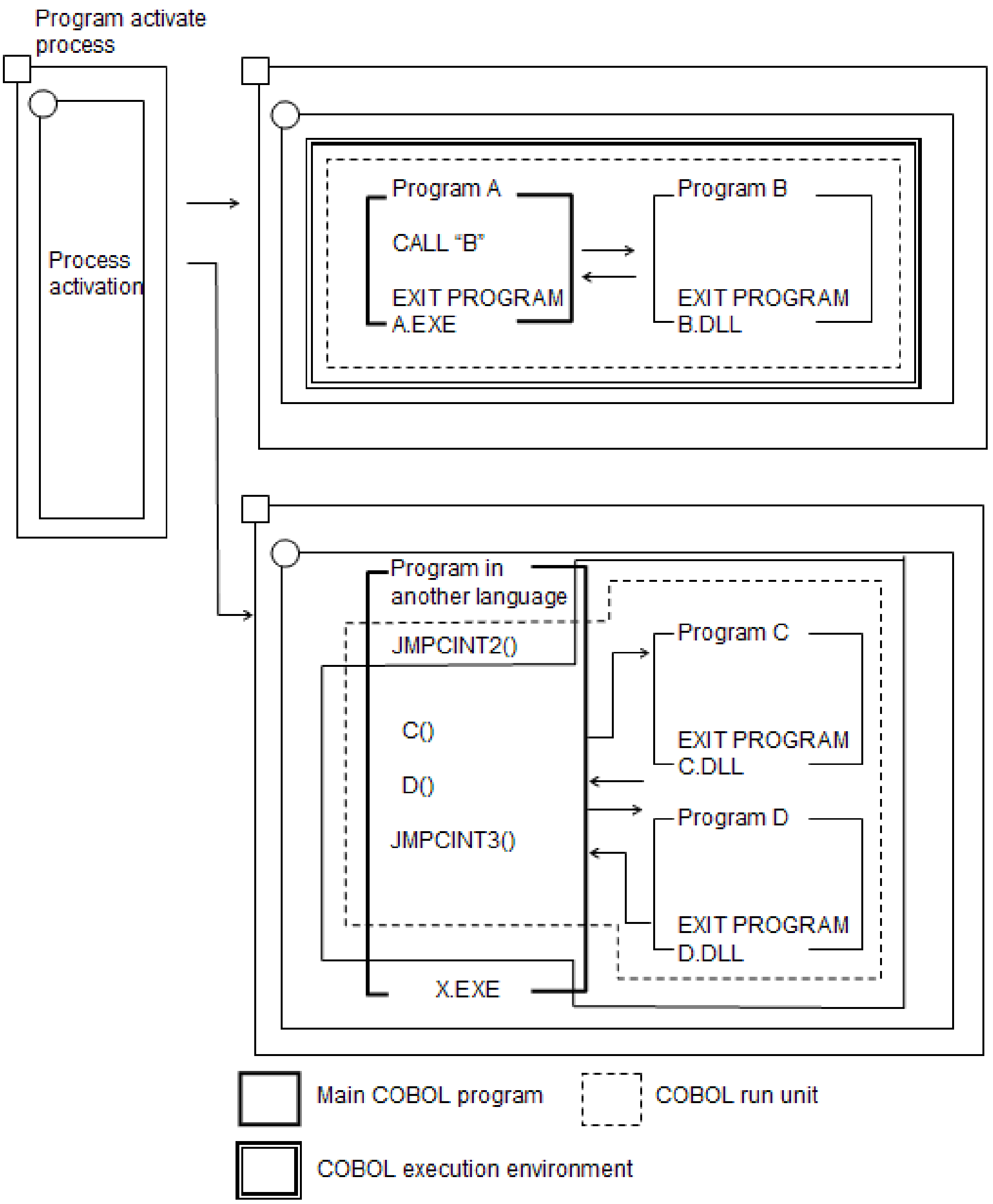
JMPCINT4

This subroutine is provided to close an runtime environment before process termination. The runtime environment can be closed by calling this subroutine from a program in another language. Call this subroutine after all run units in a process terminate. Note that COBOL programs terminate abnormally if this subroutine is called while the programs are being executed. This is because the runtime environment is closed. Refer to "G.2 Subroutines Used to Link to Another Language" for the calling format of JMPCINT4.

The figures given below show the relationships between the runtime environment and run unit of single-thread programs and multithread programs. For a single-thread program, a process is activated and the thread in the process executes the program. For a multithread program, another thread in the process is activated and the thread executes the program.

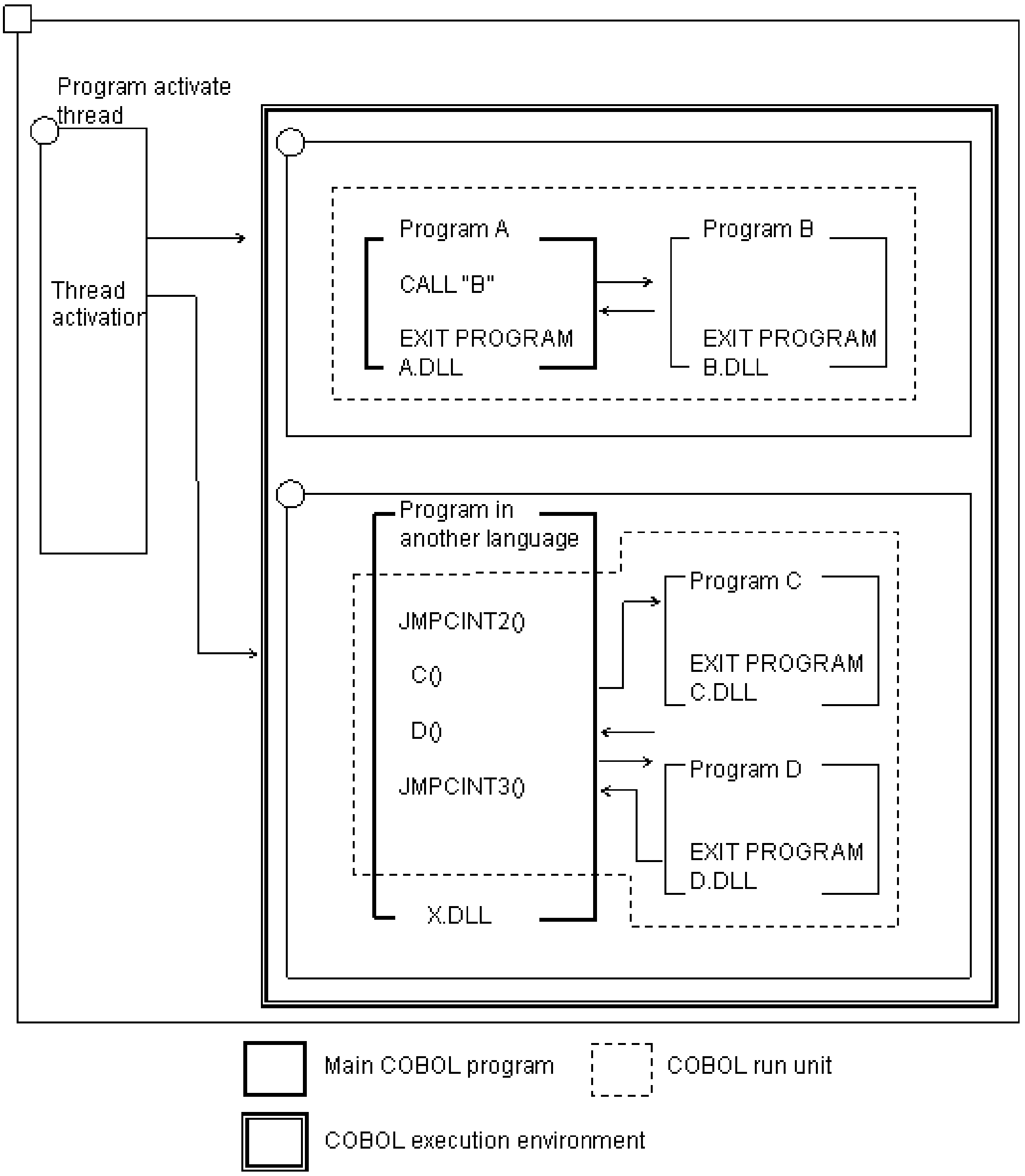
Single-thread program

For a single-thread program, only one run unit exists in one process because only one thread can execute COBOL programs. The runtime environment is closed when the run unit is terminated.



Multithread program

For a multithread program, two or more run units can exist in one process because two or more threads can execute COBOL programs concurrently in the process. The runtime environment is closed when all threads cease to exist and the process terminates.



19.3.2 Data Treatment of Multithread Programs

This section describes how multithread programs control data areas.

Multithread programs treat data acquired and managed by a process (runtime environment), a thread (run unit), and a calling unit (from calling to return).

Data acquired and managed by process

- External data and files shared among threads (*1)
- Factory object
- Object instance

*1 : Data or file for which the EXTERNAL clause is specified in COBOL source programs compiled with specification of compiler option SHREXT in addition to compiler option THREAD (MULTI)

Data acquired and managed by thread

- Data declared in program definitions (*2)

*2 : External data and files shared among threads are excluded.

Data acquired and managed by calling unit

- Data declared in method definitions

The following subsection describes these data items.

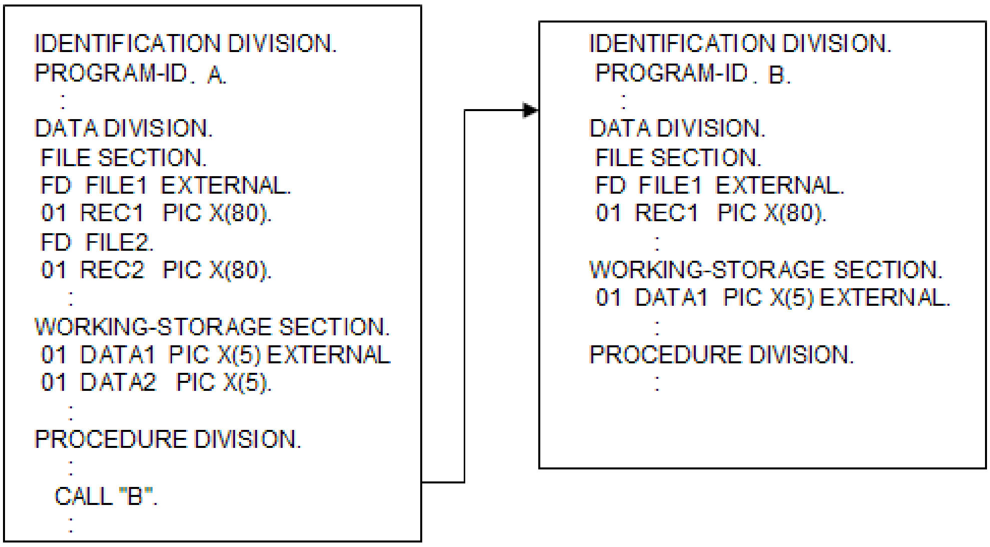
19.3.2.1 Data Declared in Program Definitions

Data declared in program definitions is allocated for each thread. This area is allocated when a run unit starts, and is closed when the run unit terminates. Files and cursors not closed by the thread are forcibly closed at termination of the run unit.

Note

The cursors are closed only if the remote database access function is used. If the precompiler is used, the run unit terminates with the cursors kept opened. Therefore, be sure to close them before the run unit terminates.

Data and files declared in program definitions



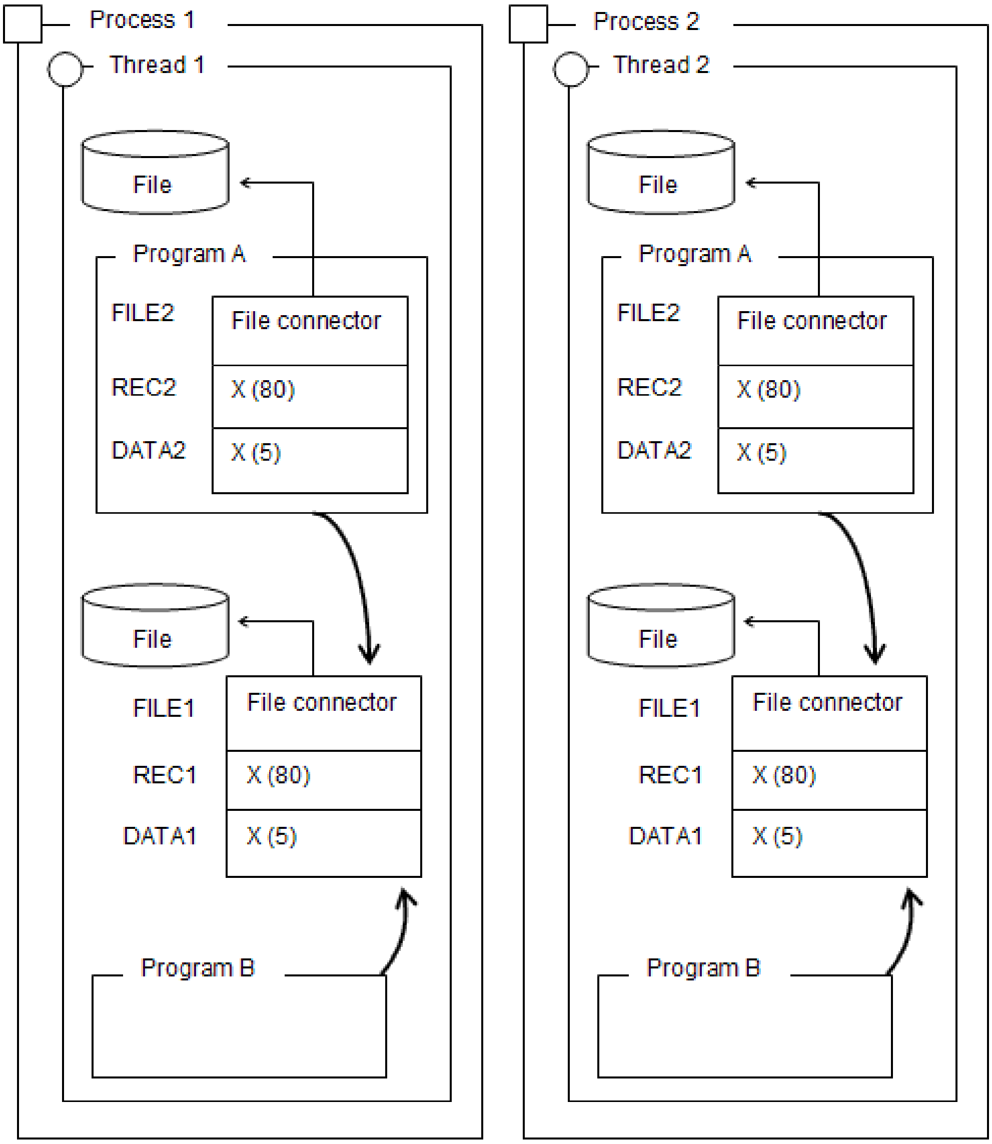
```
IDENTIFICATION DIVISION.
PROGRAM-ID. A.
    :
DATA DIVISION.
 FILE SECTION.
 FD  FILE1  EXTERNAL.
 01  REC1   PIC X(80).
 FD  FILE2.
 01  REC2   PIC X(80).
     :
WORKING-STORAGE SECTION.
 01  DATA1  PIC X(5) EXTERNAL
 01  DATA2   PIC X(5).
     :
PROCEDURE DIVISION.
     :
   CALL "B".
     :
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. B.
    :
DATA DIVISION.
 FILE SECTION.
 FD  FILE1  EXTERNAL.
 01  REC1   PIC X(80).
         :
WORKING-STORAGE SECTION.
 01  DATA1   PIC X(5) EXTERNAL.
         :
PROCEDURE DIVISION.
         :
```

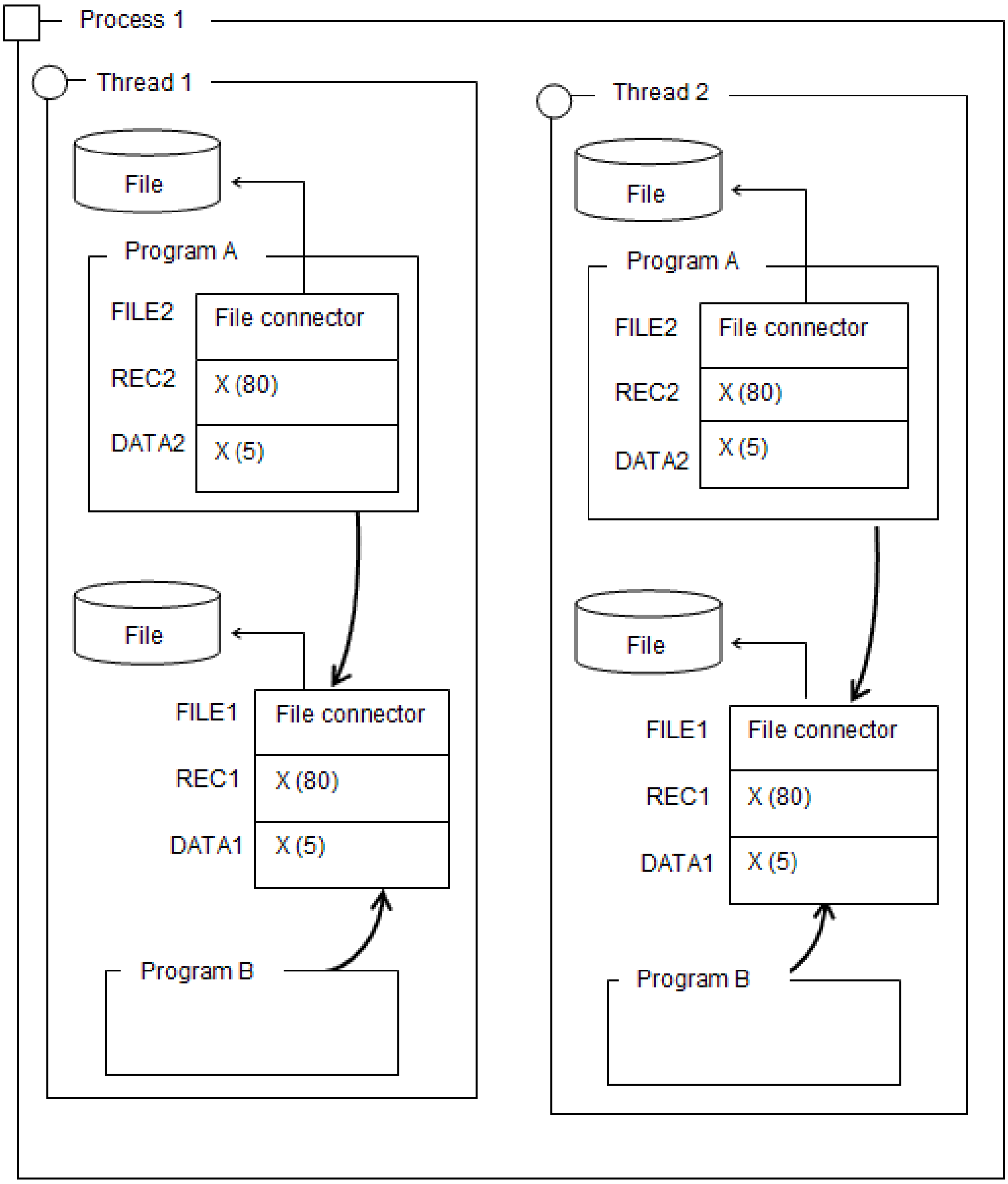
The figure given below shows activation of two copies of the above program. If the program is activated as a single-thread program, two processes are activated. If the program is activated as a multithread program, two threads are activated.

As shown in the figure, for the single-thread program, data is allocated for each process, and for the multithread program, the data is allocated for each thread.

Single-thread program



Multithread program



19.3.2.2 Factory Object and Object Instance

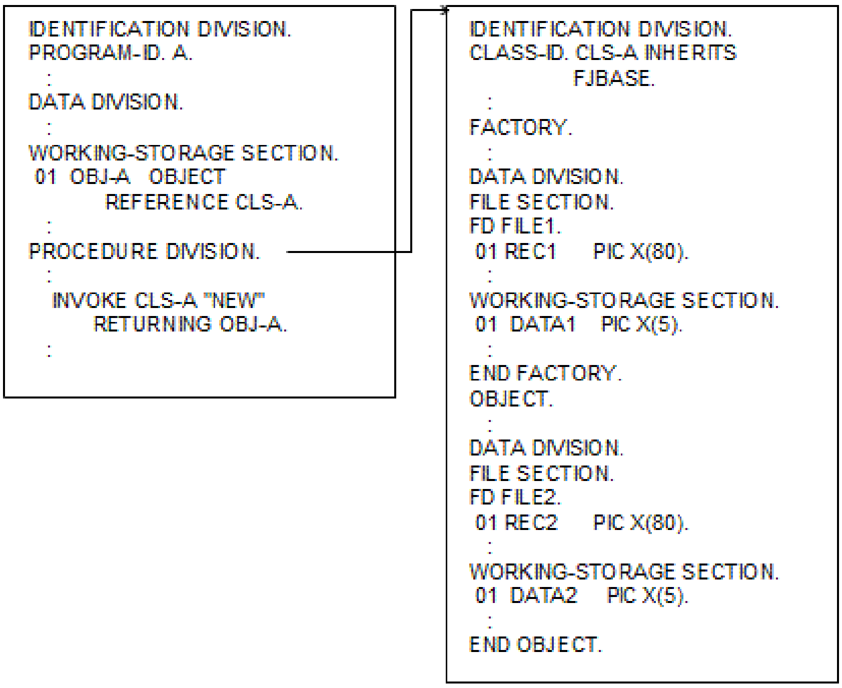
Factory objects and object instances are controlled by processes.

Only one factory object of each class exists in one process; therefore, the object is shared by threads in a multithread program. For more information, refer to "19.4.2.2 Factory Object".

Object instances can be shared by threads through factory objects. For more information, refer to "19.4.2.3 Object Instance".

Factory objects and object instances not released are released at termination of an runtime environment.

Data and files declared in factory definitions and object definitions



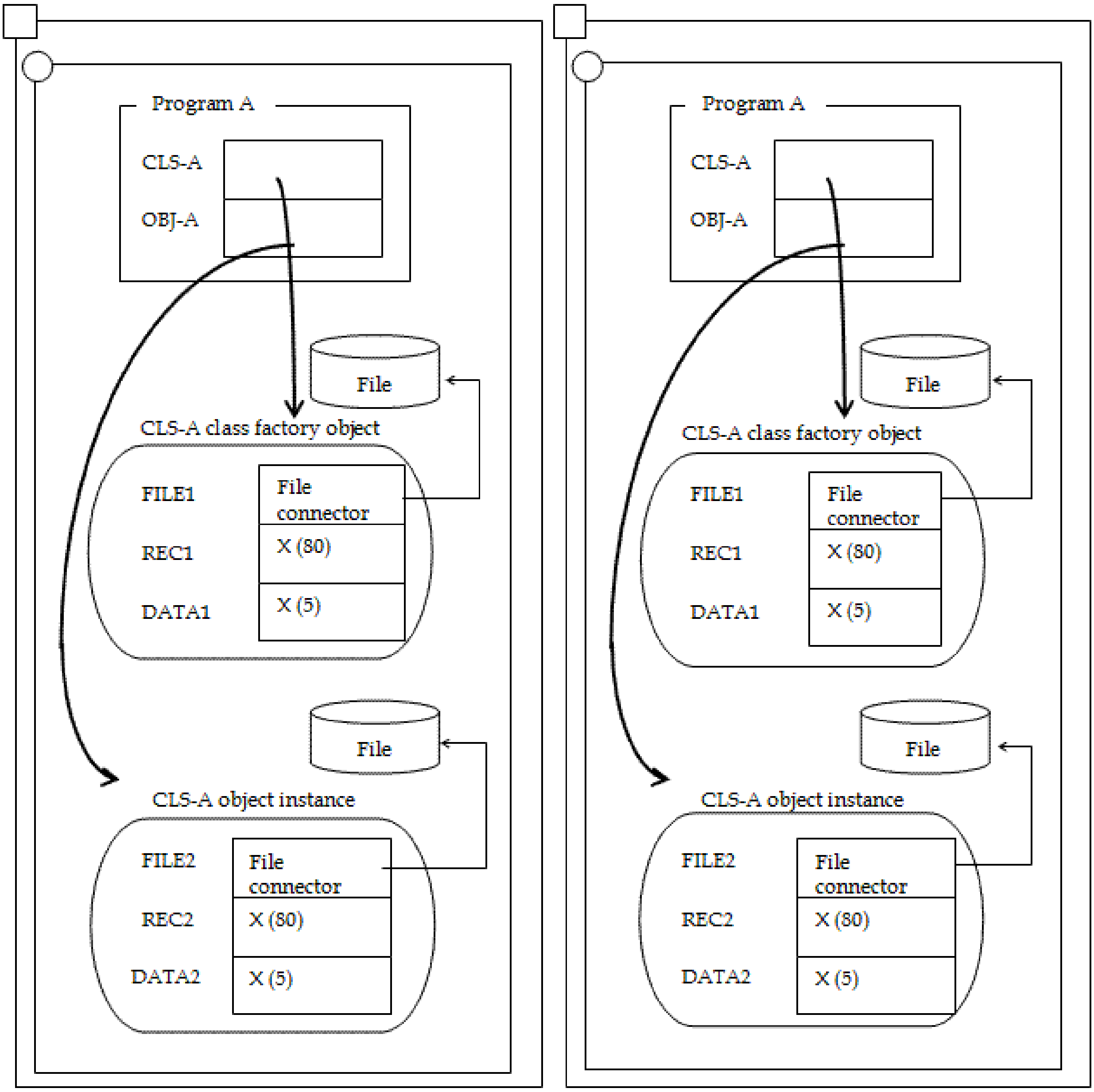
```
IDENTIFICATION DIVISION.
PROGRAM-ID. A.
  :
DATA DIVISION.
  :
WORKING-STORAGE SECTION.
 01  OBJ-A   OBJECT
           REFERENCE CLS-A.
  :
PROCEDURE DIVISION.
  :
   INVOKE CLS-A "NEW"
        RETURNING OBJ-A.
  :
```

```
IDENTIFICATION DIVISION.
CLASS-ID. CLS-A INHERITS
              FJBASE.
  :
FACTORY.
  :
DATA DIVISION.
FILE SECTION.
FD FILE1.
 01 REC1      PIC X(80).
  :
WORKING-STORAGE SECTION.
 01  DATA1    PIC X(5).
  :
END FACTORY.
OBJECT.
  :
DATA DIVISION.
FILE SECTION.
FD FILE2.
 01 REC2      PIC X(80).
  :
WORKING-STORAGE SECTION.
 01  DATA2    PIC X(5).
  :
END OBJECT.
```

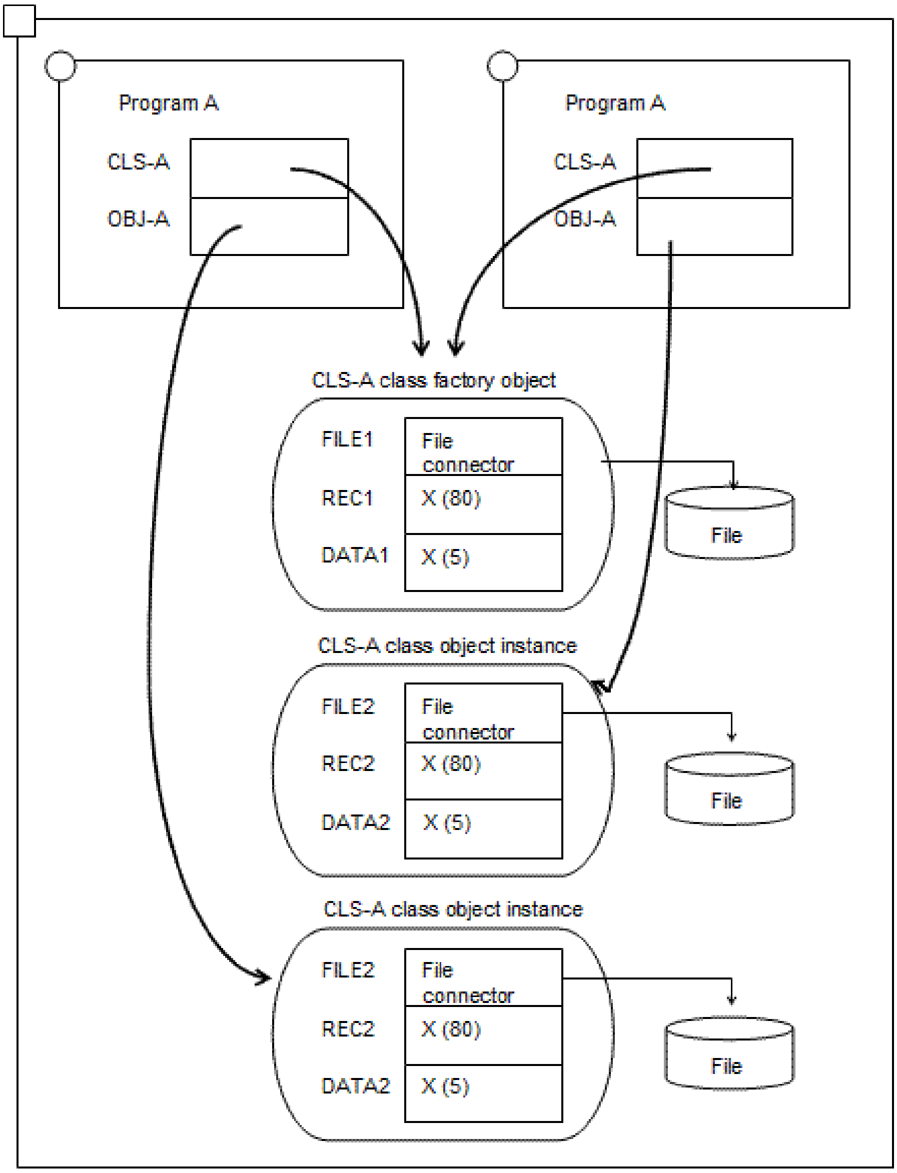
In the figure given below, two of the above programs are activated. If the program is activated as a single-thread program, two processes are activated. If the program is activated as a multithread program, two threads are activated.

As shown in the figure, the process controls the objects in both the single-thread program and the multithread program. Therefore, in the multithread program, the factory object is always shared by threads.

Single-thread program



Multithread program



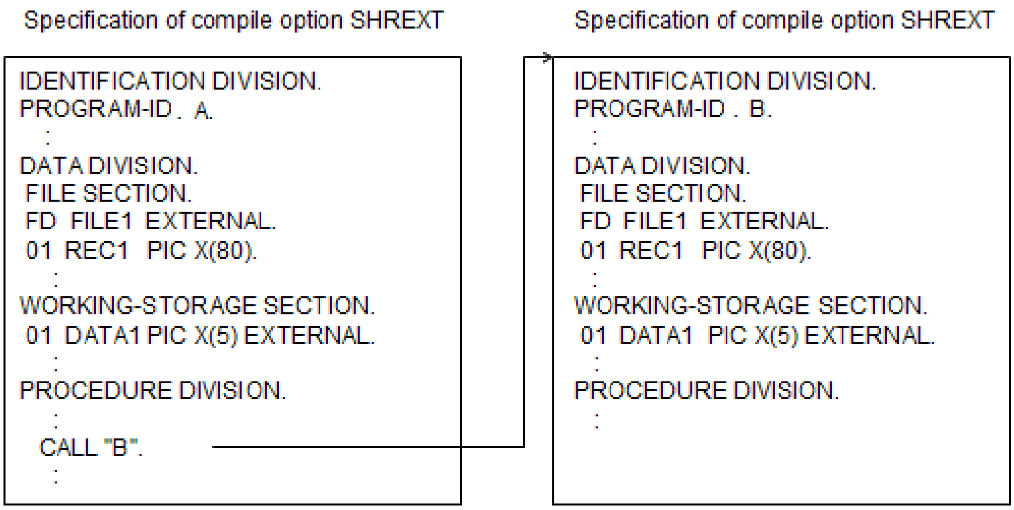
19.3.2.3 Data Declared in Method Definitions

Data declared in method definitions is allocated by using the same method as that for a single-thread program. That is, the data is allocated when a method is called and released when control is returned to the caller of the method. Files not closed by the method are forcibly closed when control is returned.

19.3.2.4 External Data and File Shared among Threads

A data area can be shared among threads by compiling a program with specification of an EXTERNAL clause in a data description entry or a file description entry and the compiler option SHREXT in addition to the compiler option THREAD (MULTI).

The figure given below shows data area allocation when data and files are shared by threads. In this figure, multithread programs are executed by two threads.

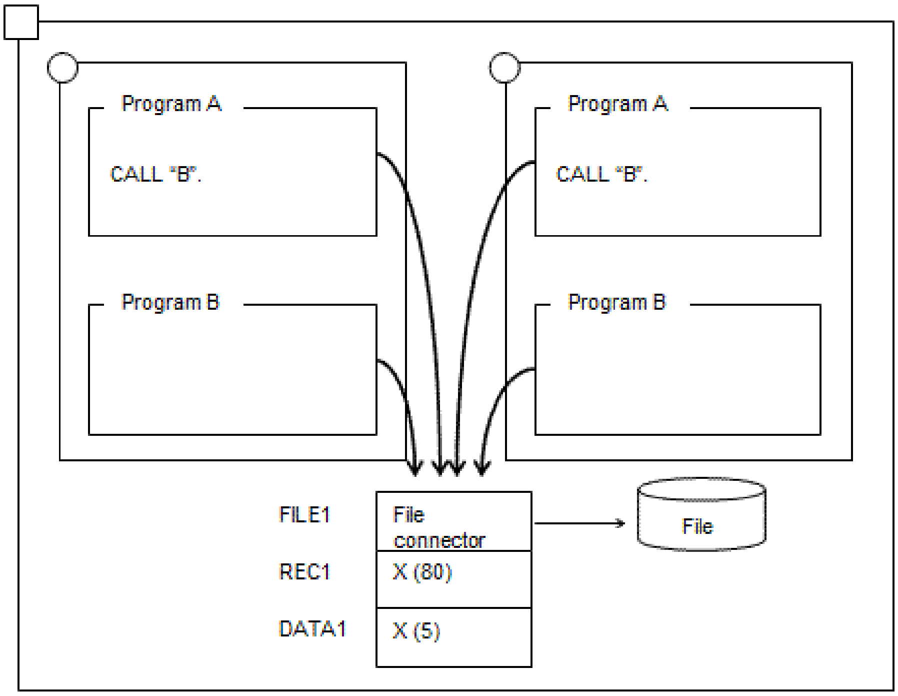


Specification of compile option SHREXT

```
IDENTIFICATION DIVISION.
PROGRAM-ID . A.
  :
DATA DIVISION.
 FILE SECTION.
 FD  FILE1  EXTERNAL.
 01  REC1   PIC X(80).
  :
WORKING-STORAGE SECTION.
 01  DATA1 PIC X(5) EXTERNAL.
  :
PROCEDURE DIVISION.
  :
   CALL "B".
  :
```

Specification of compile option SHREXT

```
IDENTIFICATION DIVISION.
PROGRAM-ID . B.
  :
DATA DIVISION.
 FILE SECTION.
 FD  FILE1  EXTERNAL.
 01  REC1   PIC X(80).
  :
WORKING-STORAGE SECTION.
 01  DATA1  PIC X(5) EXTERNAL.
  :
PROCEDURE DIVISION.
  :
```



19.3.3 Program Execution and Thread Mode

A process in which COBOL programs operate supports thread mode. Thread mode is of two types: Single-thread mode and multithread mode. In single-thread mode, only one thread in the process can execute COBOL programs. In multithread mode, two or more threads in the process can execute COBOL programs. Therefore, in multithread mode, programs compiled with specification of the compiler option THREAD (SINGLE) cannot be executed.

Thread mode in a process is determined by the first COBOL program executed . If the program was compiled with compiler option THREAD (SINGLE), the process supports single-thread mode. If the program was compiled with compiler option THREAD (MULTI), the process supports multithread mode.

Program execution in multithread mode

To execute programs in multithread mode, compile all of the programs in the execution environment with compiler option THREAD (MULTI).

Creation of components common to single-thread and multithread mode

Programs compiled with compiler option THREAD (MULTI) can operate also in single-thread mode. Compiling with compiler option THREAD (MULTI) allows common components that can operate in both single-thread and multithread mode to be created. However, if a program compiled with compiler option THREAD (MULTI) is executed in single-thread mode, the execution performance is degraded even in comparison to execution of the program compiled with compiler option THREAD (SINGLE). Be sure to read the following note.

Forcible execution in single-thread mode

Even if a COBOL program compiled with compiler option THREAD (MULTI) was executed first in a process, the program can be executed forcibly in single-thread mode by specifying environment variable @CBR_THREAD_MODE=SINGLE. Refer to "19.7.2.1.1 Format for Specifying Runtime Environment Information".

Note

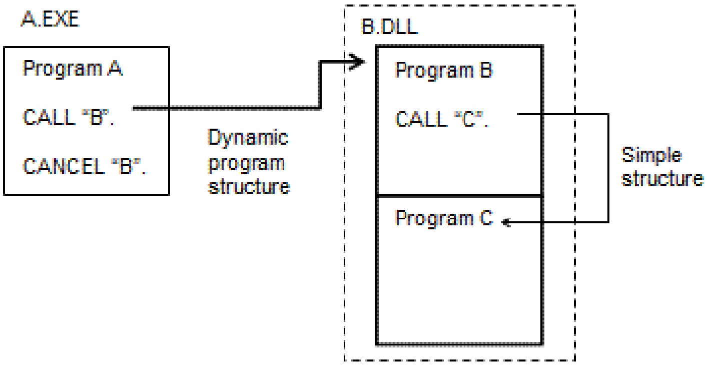
When executing a multithread program in single-thread mode, caution needs to be exercised in deleting the program by using a CANCEL statement. An example is shown below.

Example:

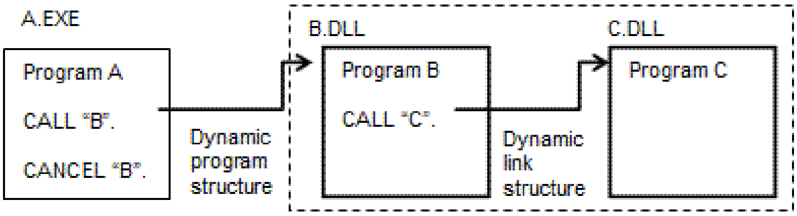
In the figure given below, programs B and C compiled with compiler option THREAD (MULTI) are in the simple structure ((a) in the figure) or dynamic link structure ((b) in the figure). If CANCEL "B" is executed, program B is deleted from virtual memory, thereby causing program C to be deleted. Therefore, the CANCEL statement cannot be used.

The CANCEL statement can be used by changing the linkage between programs B and C to the dynamic program structure ((c) in the figure).

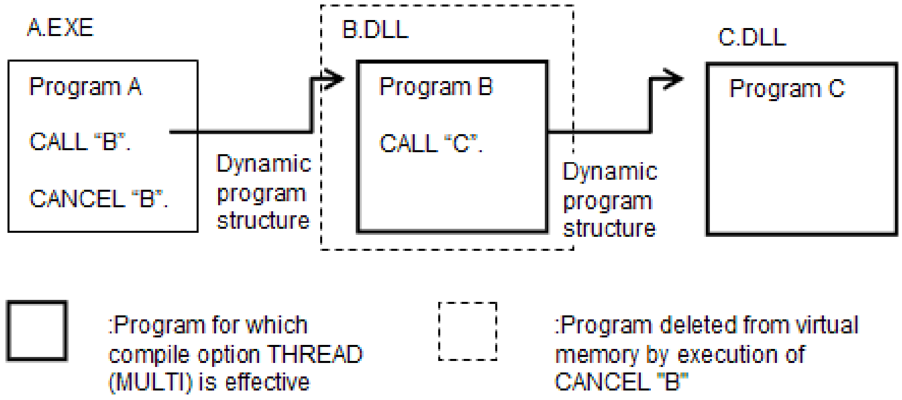
(a) Programs B and C are in the simple structure.



(b) Programs B and C are in the dynamic link structure.



(c) Programs B and C are in the dynamic program structure.


A.EXE
Program A
CALL "B".
CANCEL "B".
Dynamic program structure
B.DLL
Program B
CALL "C".
Dynamic program structure
C.DLL
Program C
:Program for which compile option THREAD (MULTI) is effective
:Program deleted from virtual memory by execution of CANCEL "B"

19.4 Resource Sharing among Threads

COBOL multithread programs enable easy creation of programs that make the best use of multithread features sharing resources such as data and files among threads.

19.4.1 Contention Status

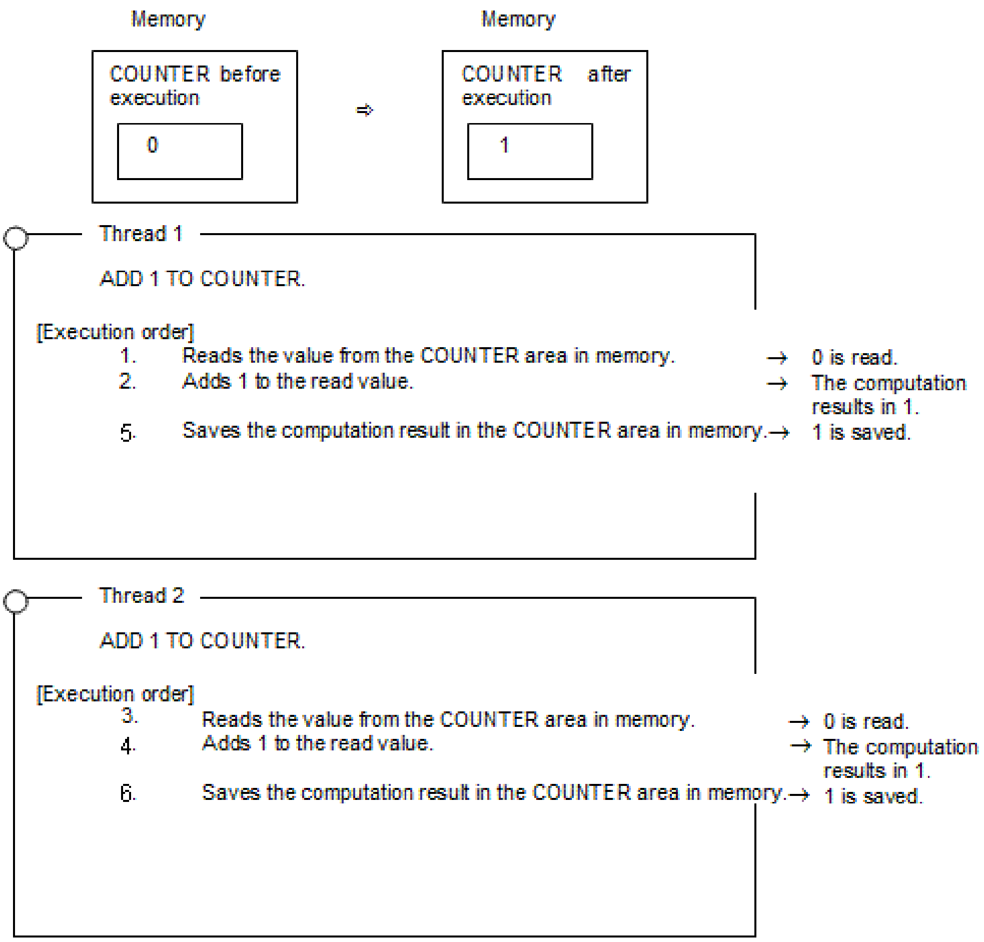
This section describes contention that may occur when resources are shared among threads.

Because the system forcibly switches threads, the execution order of the threads cannot be predicted. Therefore, when resources are shared among threads, the execution order of the threads may affect program results. This is called the contention status. It is explained by using an example.

Suppose that two threads execute statement ADD 1 TO COUNTER for data COUNTER shared among threads. ADD 1 TO COUNTER is one statement, which is to be expanded to some machine language statements, and the system executes them in the following order:

1. Reads out the value in the COUNTER area in memory.
2. Adds 1 to the read value.
3. Saves the computation result in the COUNTER area in memory.

If the execution order of threads 1 and 2 is changed as shown below, the COUNTER value does not result in 2 but 1.



Thread 1

ADD 1 TO COUNTER.

[Execution order]

1. Reads the value from the COUNTER area in memory. → 0 is read.
2. Adds 1 to the read value. → The computation results in 1.
5. Saves the computation result in the COUNTER area in memory. → 1 is saved.

Thread 2

ADD 1 TO COUNTER.

[Execution order]

3. Reads the value from the COUNTER area in memory. → 0 is read.
4. Adds 1 to the read value. → The computation results in 1.
6. Saves the computation result in the COUNTER area in memory. → 1 is saved.

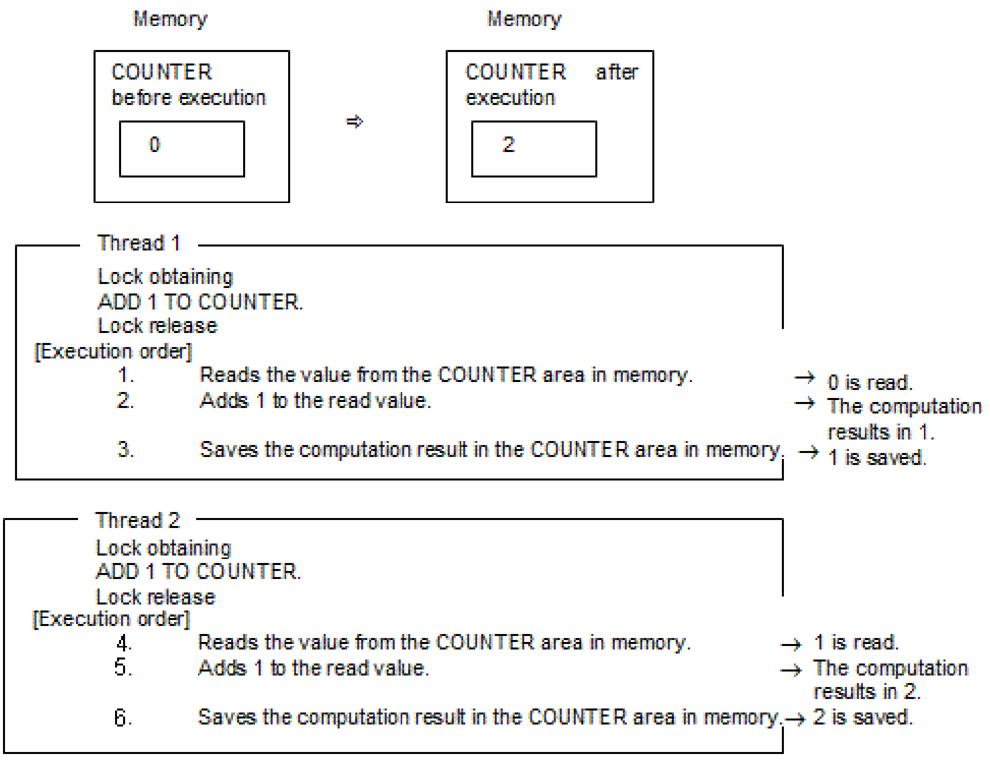
This contention status occurs under the conditions given below. If all threads only reference the same data, it can be accessed concurrently.

- A thread that updates the same data and another thread that references the same data access the data concurrently.
- Two threads that update the same data access it concurrently.

To prevent concurrent access to the same data from two or more threads, synchronization control among threads is required.

To implement this control, a lock mechanism is provided. The right to execute with exclusive access to data areas is called the lock. Only one thread can obtain the lock and only the thread obtaining the lock can be executed. Other threads that attempt to obtain the lock wait until the thread releases the lock.

By using the lock feature, thread 2 is executed after execution of thread 1 in the above example. The COUNTER value becomes 2. (In this example, thread 1 is assumed to obtain the lock first.)



Thread 1
Lock obtaining
ADD 1 TO COUNTER.
Lock release
[Execution order]
1. Reads the value from the COUNTER area in memory. → 0 is read.
2. Adds 1 to the read value. → The computation results in 1.
3. Saves the computation result in the COUNTER area in memory. → 1 is saved.

Thread 2
Lock obtaining
ADD 1 TO COUNTER.
Lock release
[Execution order]
4. Reads the value from the COUNTER area in memory. → 1 is read.
5. Adds 1 to the read value. → The computation results in 2.
6. Saves the computation result in the COUNTER area in memory. → 2 is saved.

19.4.2 Resource Sharing

In COBOL, the following resources can be shared among threads:

- External data and external file shared among threads

- Factory object

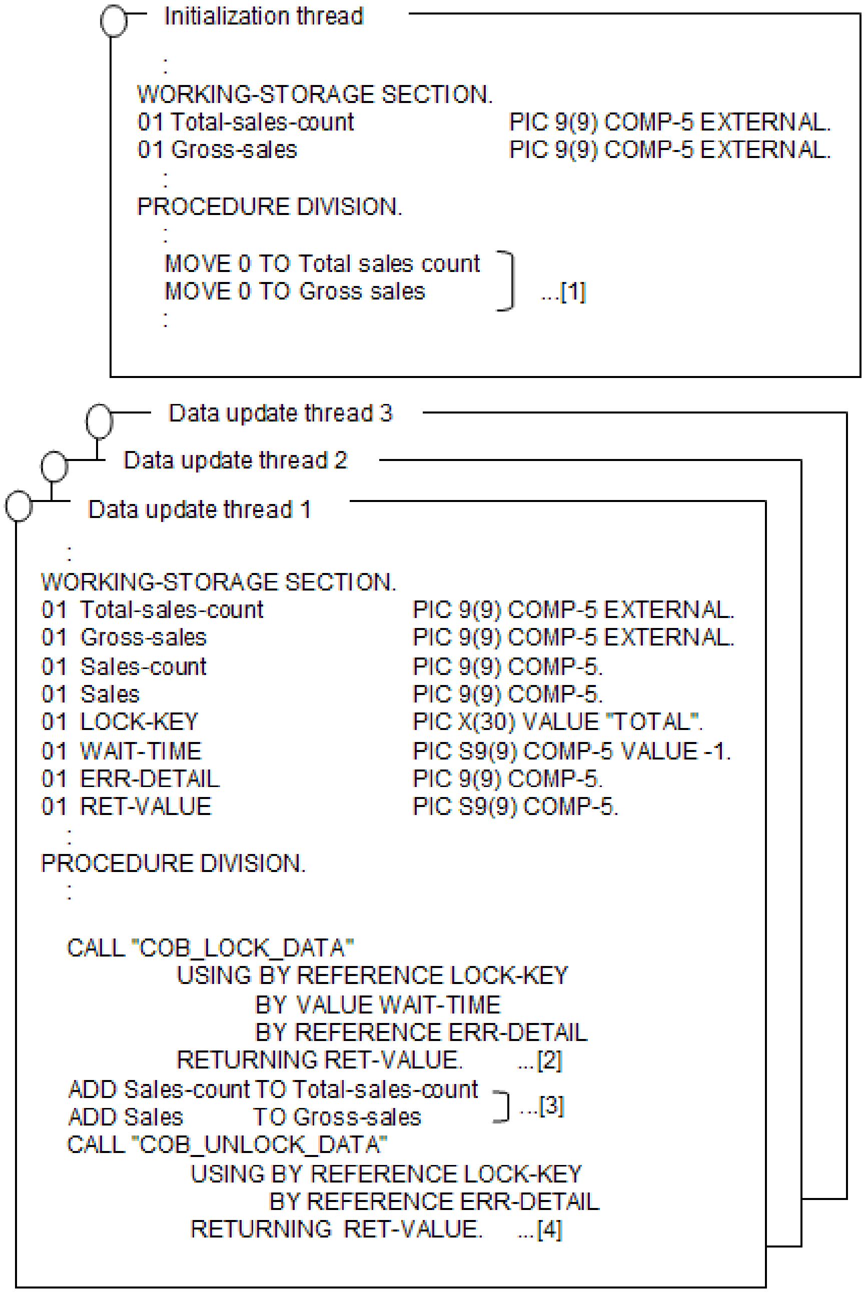
- Object instance

The COBOL runtime system performs synchronization control automatically for external files shared among threads for each I/O statement and for factory objects for each factory object. A subroutine for controlling synchronization directly from a program is provided. For information on this subroutine, refer to "19.9 Thread Synchronization Control Subroutine".

19.4.2.1 External Data and External File Shared among Threads

Common data areas can be used by threads by compiling with specification of an EXTERNAL clause in a data description entry or a description file entry and compiler option SHREXT, in addition to compiler option THREAD (MULTI).

In the example given below, external data is shared among threads. Refer to "19.6.1.1 External File Shared between Threads" for information on sharing external files among threads.



```
  :
WORKING-STORAGE SECTION.
01 Total-sales-count            PIC 9(9) COMP-5 EXTERNAL.
01 Gross-sales                  PIC 9(9) COMP-5 EXTERNAL.
  :
PROCEDURE DIVISION.
  :
  MOVE 0 TO Total sales count  ]
  MOVE 0 TO Gross sales        ]  ...[1]
  :
```

```
  :
WORKING-STORAGE SECTION.
01  Total-sales-count           PIC 9(9) COMP-5 EXTERNAL.
01  Gross-sales                 PIC 9(9) COMP-5 EXTERNAL.
01  Sales-count                 PIC 9(9) COMP-5.
01  Sales                       PIC 9(9) COMP-5.
01  LOCK-KEY                    PIC X(30) VALUE "TOTAL".
01  WAIT-TIME                   PIC S9(9) COMP-5 VALUE -1.
01  ERR-DETAIL                  PIC 9(9) COMP-5.
01  RET-VALUE                   PIC S9(9) COMP-5.
  :
PROCEDURE DIVISION.
  :

  CALL "COB_LOCK_DATA"
          USING BY REFERENCE LOCK-KEY
                BY VALUE WAIT-TIME
                BY REFERENCE ERR-DETAIL
          RETURNING RET-VALUE.      ...[2]
  ADD Sales-count TO Total-sales-count ]
  ADD Sales       TO Gross-sales       ] ...[3]
  CALL "COB_UNLOCK_DATA"
           USING BY REFERENCE LOCK-KEY
                 BY REFERENCE ERR-DETAIL
           RETURNING  RET-VALUE.    ...[4]
```

Explanation of diagram:

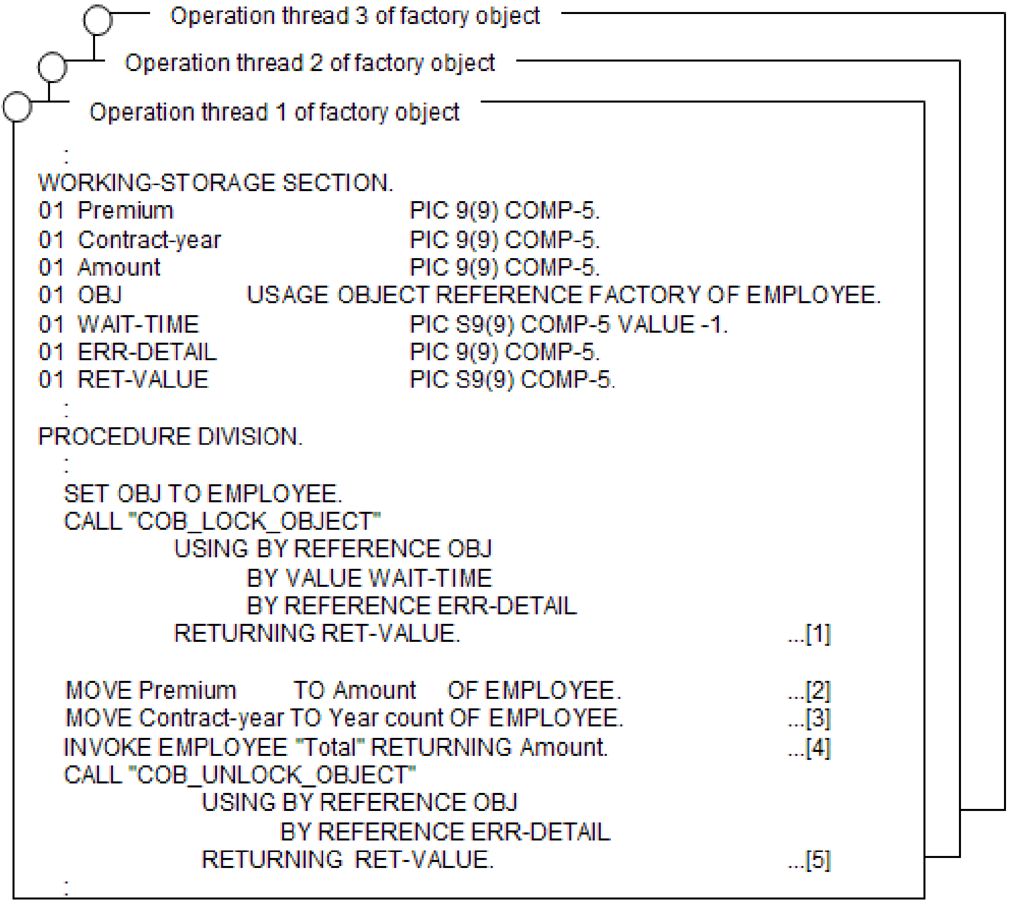
- [1] The initialization thread is activated only once in an execution environment. Shared data is initialized by the initialization thread.
- [2] The lock for data name TOTAL is obtained by using the data lock subroutine to prevent the shared data from being accessed by two or more threads concurrently. Refer to "19.9.1 Data Lock Subroutines" for more information on the data lock subroutine.
- [3] The value is added to the shared data. This processing is performed by the thread that obtained the lock in step [2].
- [4] The lock obtained for data name TOTAL is released. Another thread can obtain the lock by using the same processing as [2].

19.4.2.2 Factory Object

Factory objects are shared among threads, so data and files can be shared among threads by using factory data.

The COBOL runtime system controls thread synchronization automatically, so that two or more threads will not access factory data concurrently. (For more information, refer to the "Synchronization control by the COBOL runtime system" given below.) If processing is completed by calling a factory method only once, the program need not control thread synchronization.

However, if as shown in the example given below, processing is completed by calling methods two or more times, the object lock subroutine is required to control thread synchronization. This subroutine controls synchronization for each object. A thread obtaining the lock of an object can own the object until the lock is released.


Operation thread 3 of factory object
Operation thread 2 of factory object
Operation thread 1 of factory object

```
  :
WORKING-STORAGE SECTION.
01  Premium                       PIC 9(9) COMP-5.
01  Contract-year                 PIC 9(9) COMP-5.
01  Amount                        PIC 9(9) COMP-5.
01  OBJ            USAGE OBJECT REFERENCE FACTORY OF EMPLOYEE.
01  WAIT-TIME                     PIC S9(9) COMP-5 VALUE -1.
01  ERR-DETAIL                    PIC 9(9) COMP-5.
01  RET-VALUE                     PIC S9(9) COMP-5.
  :
PROCEDURE DIVISION.
  :
  SET OBJ TO EMPLOYEE.
  CALL "COB_LOCK_OBJECT"
           USING BY REFERENCE OBJ
                 BY VALUE WAIT-TIME
                 BY REFERENCE ERR-DETAIL
           RETURNING RET-VALUE.                      ...[1]

  MOVE Premium       TO Amount    OF EMPLOYEE.       ...[2]
  MOVE Contract-year TO Year count OF EMPLOYEE.      ...[3]
  INVOKE EMPLOYEE "Total" RETURNING Amount.          ...[4]
  CALL "COB_UNLOCK_OBJECT"
             USING BY REFERENCE OBJ
                   BY REFERENCE ERR-DETAIL
             RETURNING  RET-VALUE.                   ...[5]
  :
```

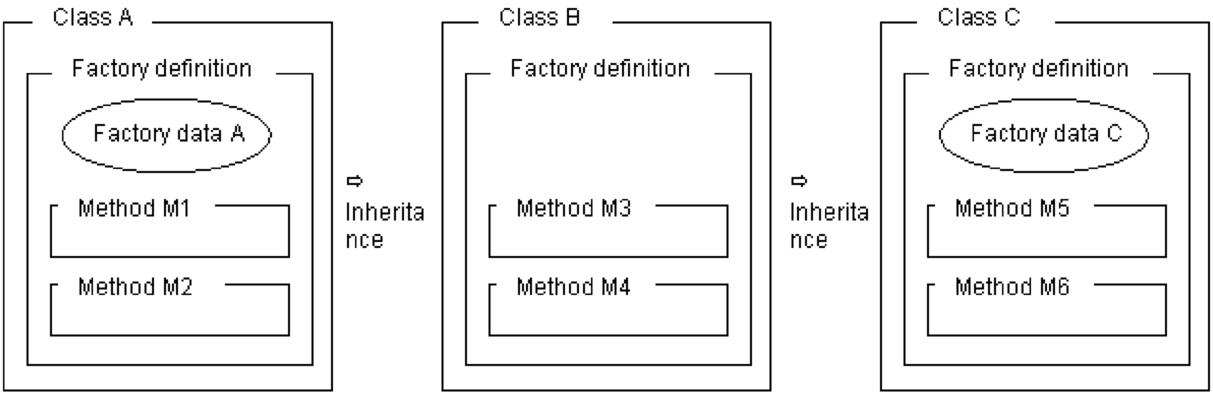
Explanation of diagram:

- [1] The object lock subroutine is used for obtaining the lock of the factory object to prevent factory data of the EMPLOYEE class from being accessed by two or more threads concurrently. Refer to "19.9.2 Object Lock Subroutines" for the object lock subroutine.
- [2] The property method of the EMPLOYEE-class factory object is called to set the premium in factory data.
- [3] The property method of the EMPLOYEE-class factory object is called to set the contract year in factory data.
- [4] The total method of the EMPLOYEE-class factory object is called to obtain the amount. Processing steps [2] to [4] are executed by the thread that obtained the lock in step [1].
- [5] The lock of the factory object is released. Releasing the lock enables another thread to obtain the lock by using the same method as that in step [1].

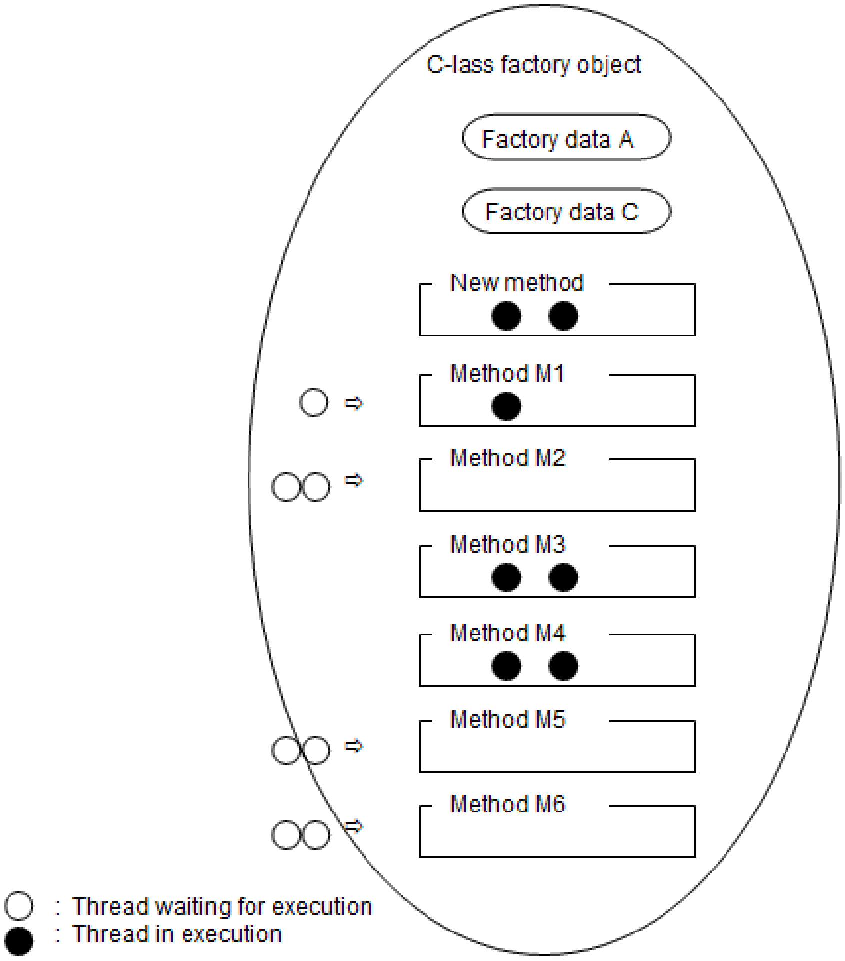
Synchronization control by the COBOL runtime system

The thread synchronization control that the COBOL runtime system performs automatically for factory data is described below.

The COBOL runtime system controls threads so that only one thread at a time executes the factory method of a class, for which factory data is defined explicitly, in a factory object. The operation is explained, using the example given below in which class C is in an inheritance relationship.



The C-class factory object has factory data defined explicitly in Classes A and C. Therefore, methods defined explicitly in class A (methods M1 and M2) and methods defined explicitly in class C (methods M5 and M6) are not executed by two or more threads concurrently. The other methods may executed by two or more threads concurrently.



In the above example, one thread executes method M1. Therefore, another thread which is to execute method M1 and threads which are to execute methods M2, M5, and M6 are placed in the execution wait state. Each of the other methods is concurrently executed by two or more threads. Thus, synchronization of access to factory data is automatically controlled by the COBOL runtime system. Thread synchronization need not be controlled for processing that is completed by calling a method only once.

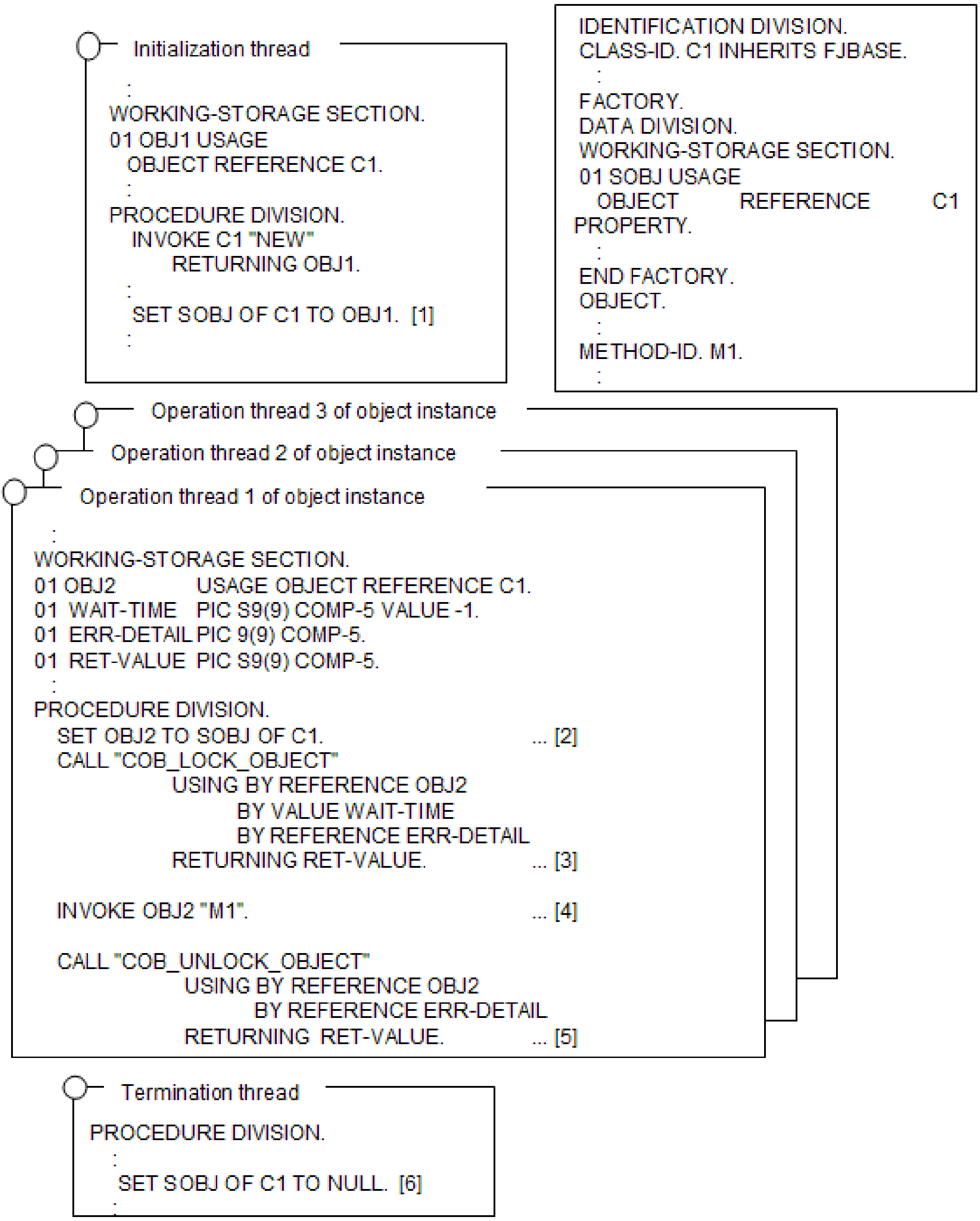
Information

The above synchronization control is performed for each factory object. Therefore, synchronization control of factory objects of the local class is not affected by any method executed in a factory object of an inherited class.

19.4.2.3 Object Instance

Object data can be shared among threads by passing an object reference of an object instance between threads through factory data. The COBOL runtime system does not control synchronization of object instances. So, to support object data, synchronization for each object needs be controlled by the object lock subroutine.

In the example given below, an object instance is shared among threads through factory data.



Initialization thread

```
  :
WORKING-STORAGE SECTION.
01 OBJ1 USAGE
   OBJECT REFERENCE C1.
  :
PROCEDURE DIVISION.
    INVOKE C1 "NEW"
           RETURNING OBJ1.
  :
    SET SOBJ OF C1 TO OBJ1.  [1]
  :
```

```
IDENTIFICATION DIVISION.
CLASS-ID. C1 INHERITS FJBASE.
   :
FACTORY.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 SOBJ USAGE
   OBJECT        REFERENCE        C1
PROPERTY.
   :
END FACTORY.
OBJECT.
   :
METHOD-ID. M1.
   :
```

Operation thread 3 of object instance

Operation thread 2 of object instance

Operation thread 1 of object instance

```
  :
WORKING-STORAGE SECTION.
01 OBJ2           USAGE OBJECT REFERENCE C1.
01  WAIT-TIME     PIC S9(9) COMP-5 VALUE -1.
01  ERR-DETAIL    PIC 9(9) COMP-5.
01  RET-VALUE     PIC S9(9) COMP-5.
  :
PROCEDURE DIVISION.
    SET OBJ2 TO SOBJ OF C1.                      ... [2]
    CALL "COB_LOCK_OBJECT"
              USING BY REFERENCE OBJ2
                    BY VALUE WAIT-TIME
                    BY REFERENCE ERR-DETAIL
              RETURNING RET-VALUE.               ... [3]

    INVOKE OBJ2 "M1".                            ... [4]

    CALL "COB_UNLOCK_OBJECT"
               USING BY REFERENCE OBJ2
                     BY REFERENCE ERR-DETAIL
               RETURNING  RET-VALUE.             ... [5]
```

Termination thread

```
PROCEDURE DIVISION.
  :
  SET SOBJ OF C1 TO NULL.  [6]
  :
```

Explanation of diagram:

- [1] The initialization thread is activated in the execution environment only once. In the initialization thread, the property method of the C1-class factory object is called, and the C1-class object instance is set in factory data.
- [2] The property method of the C1-class factory object is called, and the C1-class object instance set in step [1] is obtained from the factory data.
- [3] The lock for the C1-class object instance is obtained by the object lock subroutine, so that the object instance will not be used by two or more threads concurrently. The lock is not required if no object data is supported, or the object data is referenced only. Refer to "19.9.2 Object Lock Subroutines" for the object lock subroutine.
- [4] The M1 method of the C1-class object instance is called to perform processing. The processing is executed by the thread that obtained the lock in step [3].
- [5] The object instance lock is released. Releasing the lock enables another thread to obtain the lock by using the method described in step [3].
- [6] The termination thread is called last in the execution environment. In this thread, the property method of the C1-class factory object is called to delete the object instance set in the factory data.

19.5 Basic Use

As described in "19.3.2 Data Treatment of Multithread Programs", a single-thread program can be compiled to a program that can be executed in multithread mode only by specifying compiler option THREAD (MULTI). This is true only if the single-thread program has neither data nor files in an OO factory object. Therefore, existing programs can be transferred to multithread environments easily only by recompiling them. (Refer to "19.7.1 Compilation and Link" for compile and link.)

Be sure to read "19.4 Resource Sharing among Threads" if the factory object contains factory data and files.

The subsection below explains the function that should be noted when a single-thread program is compiled to a multithread program.

19.5.1 Use of the Input-Output Module

This section describes how to develop a multithread program that accesses files by use of the input-output module.

19.5.1.1 Transformation to a Multithread Program

To transform a single-thread program to a multithread program, the program need not be modified. The program can be used as a multithread program simply by recompiling the program with specification of compiler option THREAD (MULTI) and relinking the program.The subsections given below describe programs sharing the same file. These subsections also describe a method for executing the same program and operating separate files together with threads.

19.5.1.2 Sharing the Same File

A file in an external medium is related to a program through a file connector. The file connector has an internal attribute or an external attribute. A file can be shared among threads by using separate file connectors, regardless of the internal and external attributes.

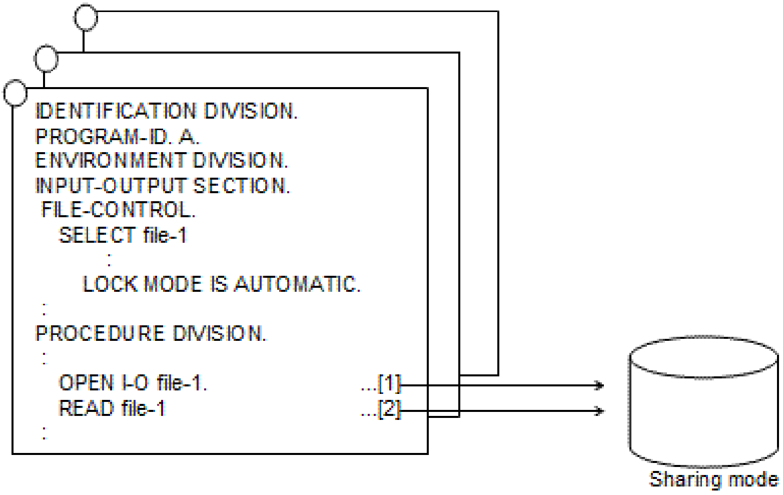
How the same file is shared by operating internal attribute file connectors is described below.

Files defined in program/method/object

A file can be shared among threads by allocating it to files defined in a file description entry in a program, a method, or an object if the files have internal-attribute file connectors.

For a file defined in a file description in an object, similarly to a file defined in a program, the file can be shared by using the object reference identifier of each object instance.

Below is shown the sharing of a file defined in a program.



```
IDENTIFICATION DIVISION.
PROGRAM-ID. A.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT file-1
          :
      LOCK MODE IS AUTOMATIC.
 :
PROCEDURE DIVISION.
 :
   OPEN I-O file-1.                   ...[1]
   READ file-1                        ...[2]
 :
```

Explanation of diagram:

- [1] The file is opened in sharing mode.
- [2] A record is read.

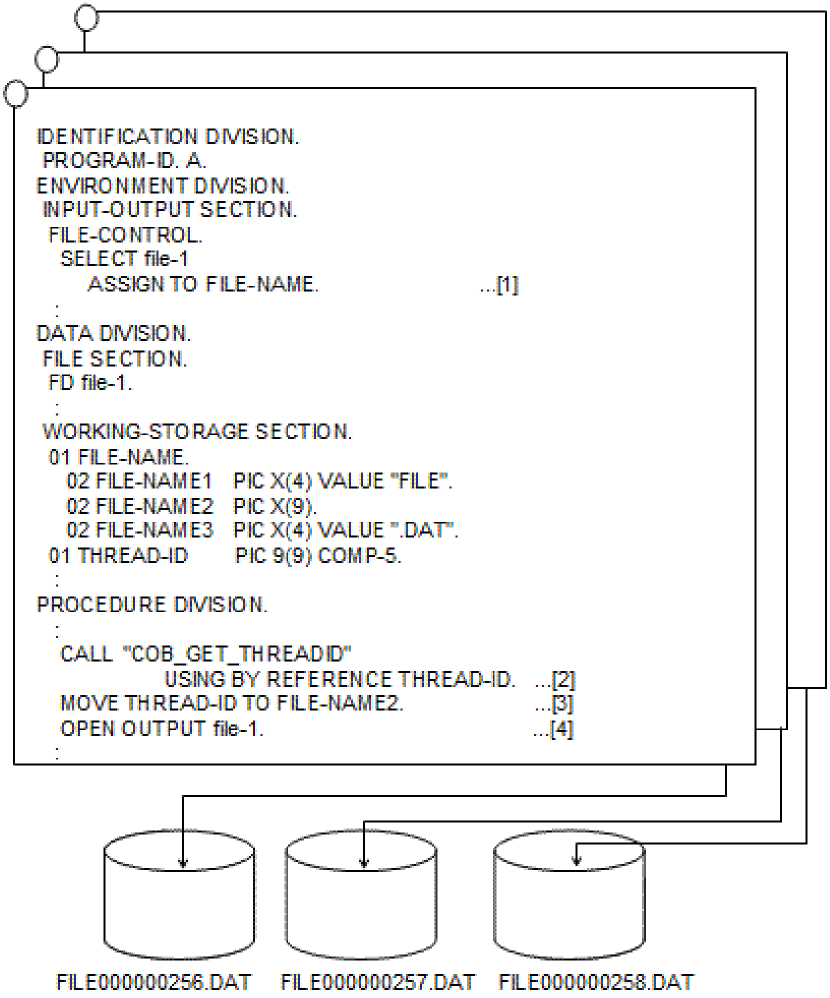
Similarly to a single-thread program, management of shared files is performed by processing in accordance with exclusive control of files. For information on the exclusive control of files, see "6.7.2 Exclusive Control of Files".

Note

A file defined in a file description entry in a factory object is shared among threads. For information on this type of file, see "19.6 Advanced Use".

19.5.1.3 Access of a Separate File for Each Thread by Executing the Same Program

Below is described how a separate file is managed in each thread by executing the same program with different file connectors.



```
IDENTIFICATION DIVISION.
 PROGRAM-ID. A.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
        ASSIGN TO FILE-NAME.                    ...[1]
    :
DATA DIVISION.
 FILE SECTION.
  FD file-1.
    :
 WORKING-STORAGE SECTION.
  01 FILE-NAME.
     02 FILE-NAME1   PIC X(4) VALUE "FILE".
     02 FILE-NAME2   PIC X(9).
     02 FILE-NAME3   PIC X(4) VALUE ".DAT".
  01 THREAD-ID       PIC 9(9) COMP-5.
    :
PROCEDURE DIVISION.
    :
    CALL  "COB_GET_THREADID"
                 USING BY REFERENCE THREAD-ID.   ...[2]
    MOVE THREAD-ID TO FILE-NAME2.                ...[3]
    OPEN OUTPUT file-1.                          ...[4]
    :
```

Explanation of diagram:

- [1] The data name is described in the ASSIGN clause.
- [2] The "thread obtain" subroutine is called to obtain the thread ID.
- [3] The thread ID obtained in step [2] is set in the data name.
- [4] The OPEN statement (OUTPUT mode) is executed to create the file.

As shown above, the thread ID is obtained by the subroutine and inserted as part of the file-name, thus enabling a separate file to be accessed in each thread by executing the same program. For information on the thread obtain subroutine, refer to "G.1.4 Subroutine for Obtaining a Thread ID".

Note

The thread ID changes each time the program is executed. Therefore, if the thread ID is used as the file name, use it as a temporary work file.

To delete files that are no longer required at termination of a thread from a program, use the delete function of the COBOL File Utility Function. For information on this function, refer to "6.8.4 COBOL File Utility Functions".

19.5.1.4 Note

There is a possibility of becoming dead-locked status in the order of the file designation when the multi-thread program that specifies two or more files by one OPEN statement is multiple operated. Please describe the OPEN statement at each file when you multiple operate the multi-thread program or make the order of specifying the file name described in one OPEN statement the same.

19.5.2 base Access (ODBC)

This section describes how to transfer an existing program that performs remote database access (ODBC) to a multithread program.

19.5.2.1 Transformation to a Multithread Program

To transform a single-thread program to a multithread program, the program does not need to be modified. Recompile the program with specification of the compiler option THREAD (MULTI) and relink the program.

Specify THREAD for the connection validation scope of the ODBC information file to be used at execution of the program. Refer to "17.2.8.1.2 Creating an ODBC Information File".

Modify the ODBC information file by using the ODBC information setup tool. Refer to "17.2.8.2 Using the ODBC Information Setup Tool".

Specifying the thread for the connection validation scope assures connection for each thread. An application in which an embedded SQL statement is described in a class definition (when an OO function is used), however, will not operate in this environment. Refer to "19.5.2.2 Transformation to a Multithread Program (When an OO Function Is Used)".

19.5.2.2 Transformation to a Multithread Program (When an OO Function Is Used)

To transform a single-thread program that uses an OO function to a multithread program, the program does not need to be changed. Recompile the program with specification of the compiler option THREAD (MULTI) and relink the program.

Specify the OBJECT_INSTANCE for the connection validation scope in the ODBC information file to be used at execution of the program. Refer to "17.2.8.1.2 Creating an ODBC Information File".

Modify the ODBC information file by using the ODBC information setup tool. Refer to "17.2.8.2 Using the ODBC Information Setup Tool".

Specifying object instance for the connection validation range assures connection for each object instance. An application in which an embedded SQL statement is described in a program definition, however, does not operate in this environment. Processing from database connection to disconnection should be completed in each object instance.

19.5.2.3 Notes

- If two or more threads access the same database table, threads may be kept waiting, or an error may be returned so that the database assures transaction consistencies.
  Refer to a database manual for transaction management. Transaction management may differ depending on the value specified for concurrent execution of the ODBC information file cursor. Refer to "17.2.8.1.2 Creating an ODBC Information File".

- Some ODBC drivers allow multithread (thread safety) operation to be specified during data source setting. When specifying this operation, be sure to enable multithread (thread safety).

- Some ODBC drivers do not support multithread mode. For these drivers, multithread programs do not operate normally.

19.5.3 SymfoWARE Linkage with Use of Precompiler

A multithread program that accesses a SymfoWARE RDB can also be created by using a precompiler.

19.5.3.1 Program Description

Create a COBOL program that accesses a database by describing an embedded SQL statement. No multithread-specific description is required.

19.5.3.2 Program Compile and Link

To execute compiling and linkage using the sqlcobol command, specify the -T option.

Note

To execute pre-compiling, compiling, and linkage separately, follow the procedure below.

- Specify the -T pre-compile option.
- For details about compiling and linkage, refer to "19.7 Procedure: From Compilation to Execution". Also link F3CWDRVM.LIB that is offered by SymfoWARE at the time of linkage.

19.5.3.3 Program Execution

To make it possible for a SymfoWARE multithread program that uses a pre-compiler to run, you must set the following environment variable information. However, if you use an SQL expansion interface to run a multithread program that is conscious of the session, there is no need to make settings.

```
@CBR_SYMFOWARE_THREAD=MULTI
```

Refer to @CBR_SYMFOWARE_THREAD (specification to enable operation of a multithread program linked to SymfoWARE using a precompiler) in "19.7.2.1.1 Format for Specifying Runtime Environment Information".

Note

Notes on development of multithread programs linking to SymfoWARE using a precompiler are described below.

- Neither connection nor cursor can be shared among threads. For example, a thread cannot access data by using a connection set by another thread and a thread cannot operate a cursor opened by another thread.
- An application created with specification of a multithread option during precompile cannot be executed in single-thread mode.

19.5.4 Using the ACCEPT and DISPLAY Statements

This section describes how to use the ACCEPT and DISPLAY statements in multithread environments.

19.5.4.1 ACCEPT/DISPLAY Statements

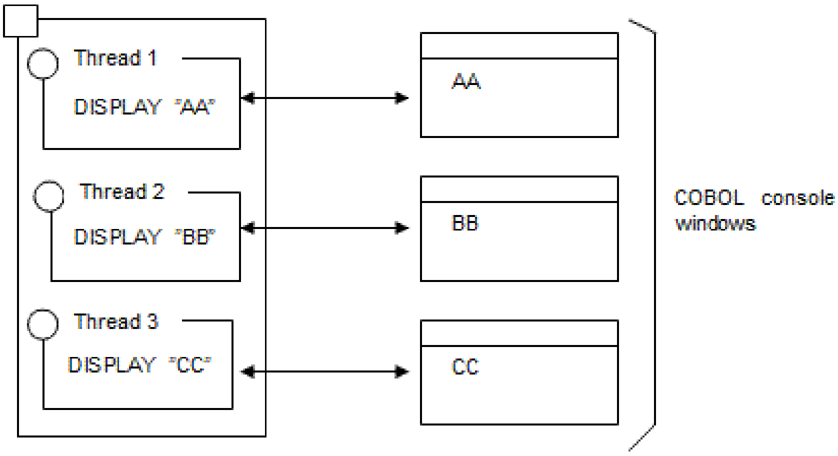
In multithread mode, a COBOL console window is displayed separately for each thread. For ACCEPT/DISPLAY function using system console windows (including the command prompt window) and files, one window and one file are shared among processes.

Input-output destination	DISPLAY		ACCEPT	
	Single-thread mode	Multithread mode	Single-thread mode	Multithread mode
COBOL console	Process	Each thread	Process	Each thread
System console	Process	Process	Process	Process
File	Process	Process	Process	Process

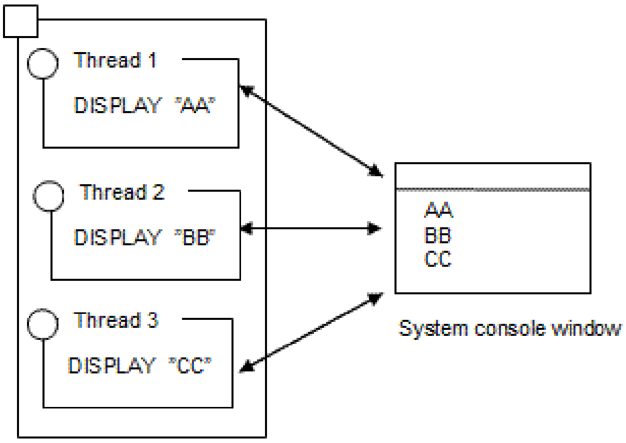
Example

Input-output example

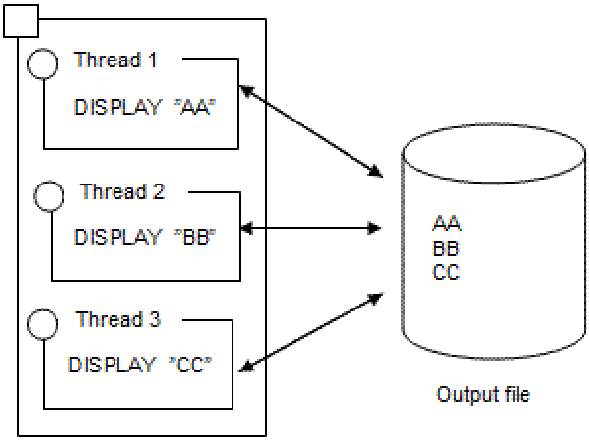
- When COBOL console windows are used



- When a system console window is used



- When a file is used



Synchronization of data input-output data is controlled for each ACCEPT or DISPLAY statement if the ACCEPT/DISPLAY function is performed using the system console window and file.

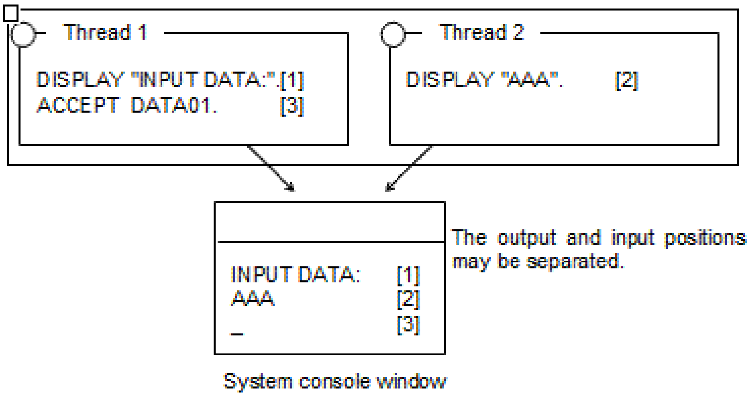
Synchronization is controlled for each statement. However, the execution order of each statement depends on the order of thread control of the system. Therefore, the result may vary with each execution.

To control synchronization of the execution order, for example, to execute an ACCEPT statement immediately after a DISPLAY statement during use of a system console window, use of the thread synchronization control subroutine enables the ACCEPT statement to be executed immediately after output by the DISPLAY statement. Refer to "19.9 Thread Synchronization Control Subroutine".

Example

Synchronization control of two or more ACCEPT and DISPLAY statements ([1] to [3]: Execution order)

- When the thread synchronization control subroutine is not used



```
DISPLAY "INPUT DATA:".[1]
ACCEPT  DATA01.       [3]
```

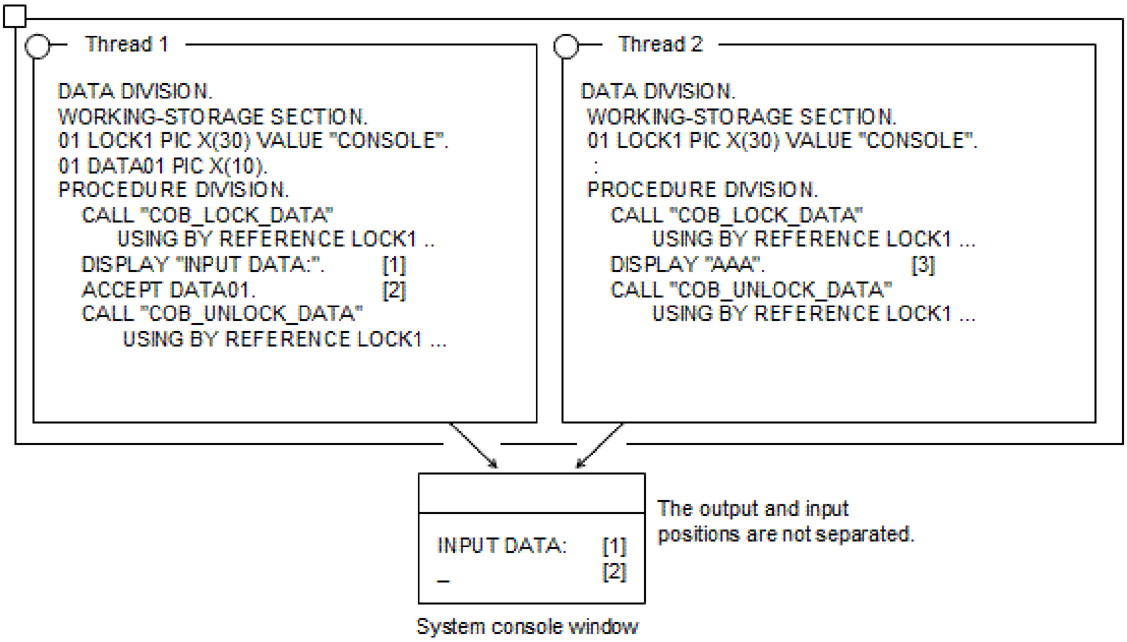
```
DISPLAY "AAA".        [2]
```

```
INPUT DATA:    [1]
AAA            [2]
_              [3]
```

The output and input positions may be separated.

System console window

- When the thread synchronization control subroutine is used



```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 LOCK1 PIC X(30) VALUE "CONSOLE".
01 DATA01 PIC X(10).
PROCEDURE DIVISION.
   CALL "COB_LOCK_DATA"
        USING BY REFERENCE LOCK1 ..
   DISPLAY "INPUT DATA:".        [1]
   ACCEPT DATA01.                [2]
   CALL "COB_UNLOCK_DATA"
        USING BY REFERENCE LOCK1 ...
```

```
DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 LOCK1 PIC X(30) VALUE "CONSOLE".
  :
 PROCEDURE DIVISION.
    CALL "COB_LOCK_DATA"
         USING BY REFERENCE LOCK1 ...
    DISPLAY "AAA".               [3]
    CALL "COB_UNLOCK_DATA"
         USING BY REFERENCE LOCK1 ...
```

```
INPUT DATA:    [1]
_              [2]
```

The output and input positions are not separated.

System console window

Note

- Which is to be used, a console screen or a file, by the DISPLAY and ACCEPT statements, and which file ID name is validated when the file is used, depend on the compiler option of the object that contains the DISPLAY and ACCEPT statements and that operates first. Therefore, specify identical compiler options for compilation as much as possible.

- In multithread mode, click the Close button on the COBOL console screen to terminate a process.

- Under the multithreaded environment, when the standard output and the standard error output of the system are output to the same file by specifying the redirection, the content of the output file is not guaranteed.

  Please specify the file to environment variable information @MessOutFile when you output the result of the DISPLAY statement associated with execution message and function-name SYSERR that runtime system outputs to the arbitrary file.

19.5.4.2 Command Line Argument and Environment Variable Operation Functions

19.5.4.2.1 Command line argument operation function

The position of an argument to be used by the command line argument operation function is specified for each thread. Therefore, even if two or more threads use command line arguments, thread operation is kept unaffected.

```
ENVIRONMENT     DIVISION.
CONFIGURATION   SECTION.
SPECIAL-NAMES.
    ARGUMENT-NUMBER IS ARGNUM       *>[1]
    ARGUMENT-VALUE  IS ARGVAL.      *>[2]
DATA            DIVISION.
WORKING-STORAGE SECTION.
01 DATA01 PIC X(10).
PROCEDURE     DIVISION.
    DISPLAY 3 UPON ARGNUM.          *>[1]
    ACCEPT  DATA01 FROM ARGVAL.     *>[2]
```

Explanation of diagram:

- [1] Specify an argument position for each thread.
- [2] The value of an argument is common in the process.

19.5.4.2.2 Environment variable operation function

Environment variable names to be used by the environment variable functions are specified for each thread. Therefore, an environment variable name allocated to ENVIRONMENT-NAME in a SPECIAL-NAMES paragraph does not need to be identical in two or more threads. However, environment variable values are common in a process. So if an environment variable is operated in multithread mode, each threads is affected.

```
ENVIRONMENT     DIVISION.
CONFIGURATION   SECTION.
SPECIAL-NAMES.
    ENVIRONMENT-NAME  IS ENVNAME    *>[1]
    ENVIRONMENT-VALUE IS ENVVAL.    *>[2]
DATA            DIVISION.
WORKING-STORAGE SECTION.
01 DATA01 PIC X(10).
PROCEDURE     DIVISION.
    DISPLAY "ABC" UPON ENVNAME.     *>[1]
    ACCEPT  DATA01 FROM ENVVAL.     *>[2]
```

Explanation of diagram:

- [1] The environment name is specific to the thread.
- [2] The environment value is common in the process.

19.5.5 Using the Screen Module

The screen module in the multithread mode has a window for each thread.

The process ends when the x (close) button on the screen window is clicked.

19.6 Advanced Use

19.6.1 Using the Input-Output Module

The basic use method explained the sharing of the file that operated a different file connector between threads.

This subsection explains how to share files by operating the same file connector between threads.

To operate the same file connector between threads, use the following methods:

- External file shared between threads
- File defined in the factory object
- File defined within the object

The following explains how to use each file:

If a file is shared by operating the same file connector, thread contention occurs. To prevent thread contention, thread synchronization control is required. This control is explained in the method of using each file.

Note

When a single file is accessed by multiple threads using the same file connector, the status of the file position indicator is changed by executing any input-output statement, no matter which thread executes it.

Therefore, to perform operation in multithread mode, a program must be designed considering the status of the file position indicator.

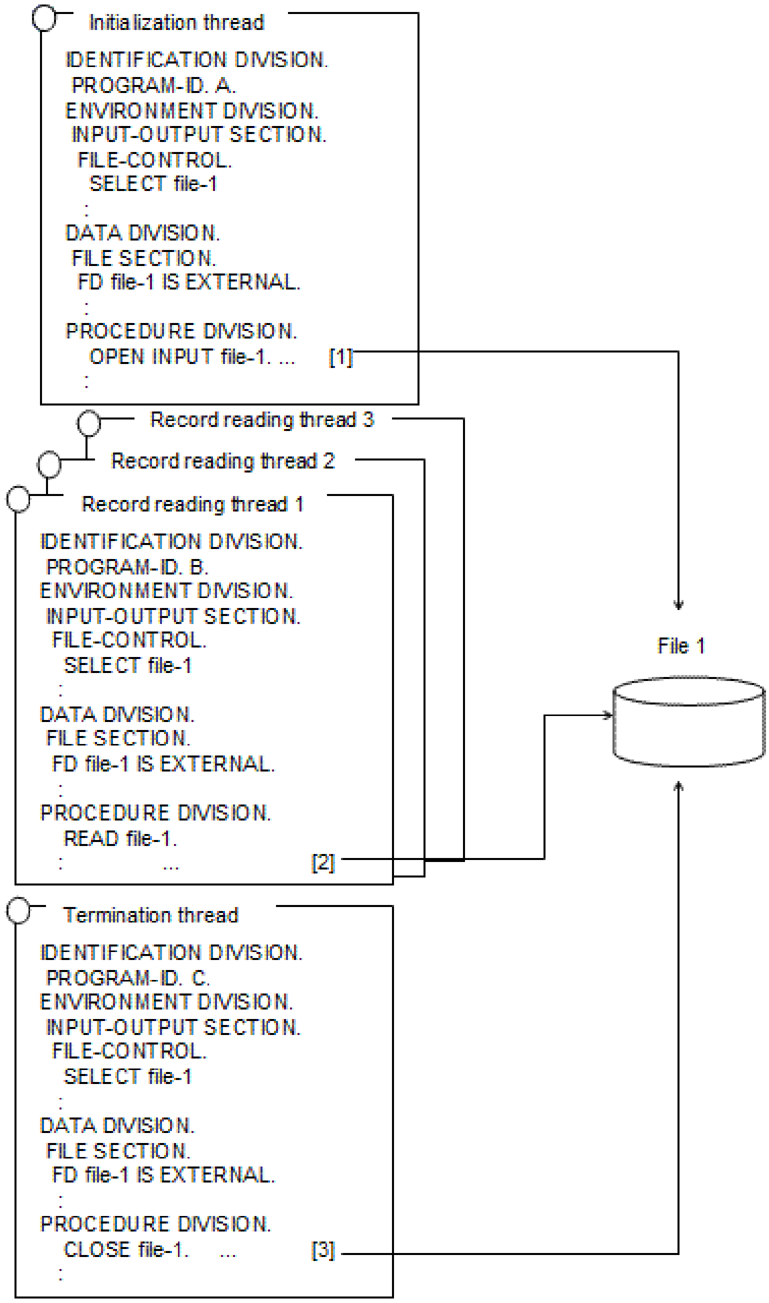
19.6.1.1 External File Shared between Threads

If the EXTERNAL clause is specified in the file description entry, an external attribute is assigned to the file connector. For the file connector having the external attribute, the same file connector can be shared between programs.

To share the same file connector among two or more threads, specify the compiler option SHREXT in addition to the compiler option THREAD (MULTI).

The EXTERNAL clause can also be specified in the file description entry in the method.

The following shows an example of the program using the external file shared between threads:



```
IDENTIFICATION DIVISION.
 PROGRAM-ID. A.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
   :
DATA DIVISION.
 FILE SECTION.
  FD file-1 IS EXTERNAL.
   :
PROCEDURE DIVISION.
    OPEN INPUT file-1. ...      [1]
   :
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. B.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
   :
DATA DIVISION.
 FILE SECTION.
  FD file-1 IS EXTERNAL.
   :
PROCEDURE DIVISION.
    READ file-1.
   :             ...            [2]
```

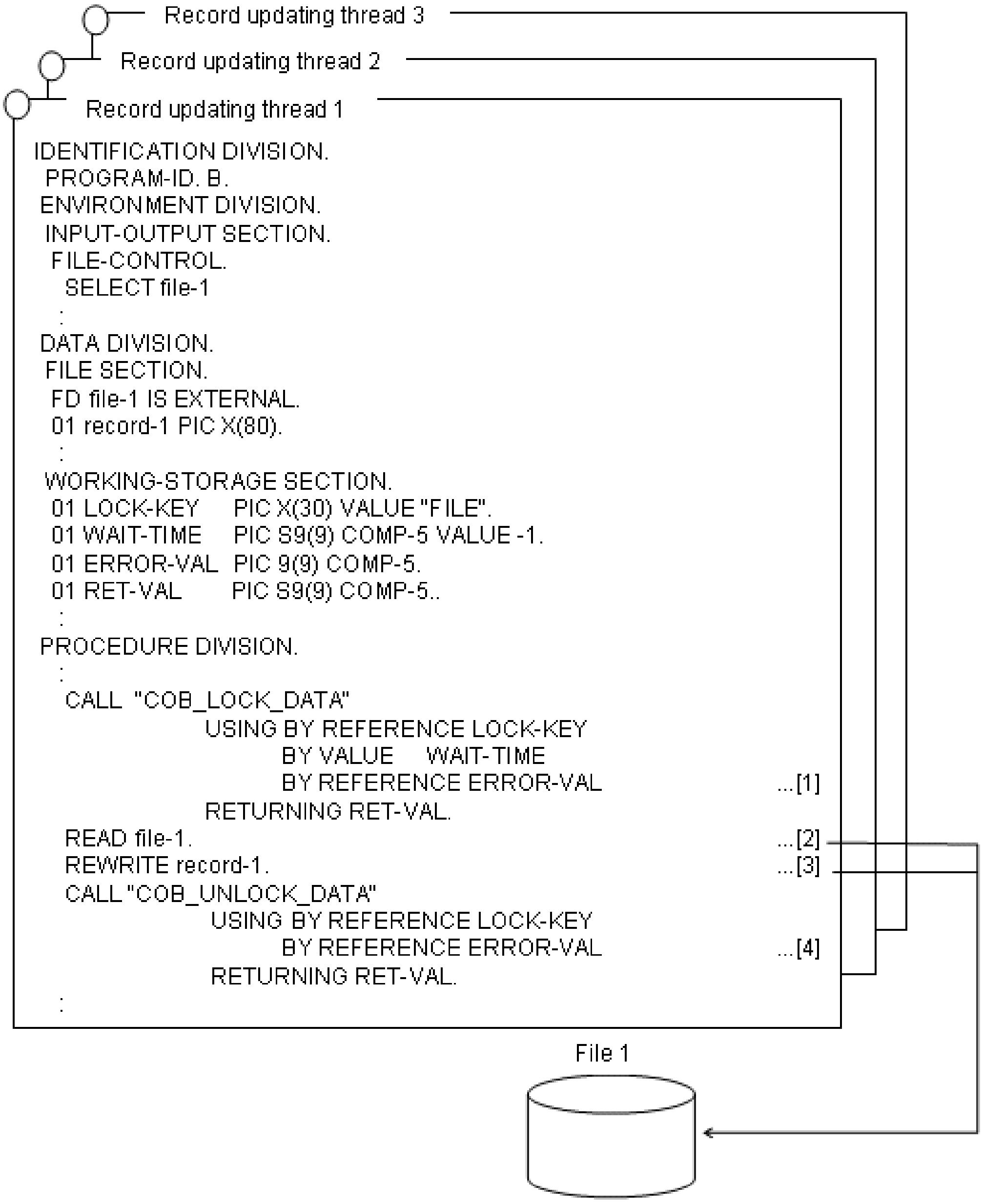
```
IDENTIFICATION DIVISION.
 PROGRAM-ID. C.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
   :
DATA DIVISION.
 FILE SECTION.
  FD file-1 IS EXTERNAL.
   :
PROCEDURE DIVISION.
    CLOSE file-1.    ...        [3]
   :
```

Explanation of diagram:

- [1] The initialization thread is activated only once in the execution environment. The initialization thread opens a file having the file connector assigned the external attribute.
- [2] Two or more record reading threads (three in this program example) are simultaneously activated. The record reading thread reads a record for the file opened by the initial thread.
- [3] The termination thread is activated last, and only once in the runtime environment. The last thread closes the file opened by the initial thread.

For an external file, the COBOL runtime system automatically performs thread synchronization control for a single input-output statement. However, the desired processing to be performed in executing two or more input-output statements requires thread synchronization control. To perform synchronization control for the external file, the data lock subroutine and data unlock subroutine are used. Using these subroutines prevents another READ statement from being executed by another thread between the READ statement in [2] and the REWRITE statement in [3].

The following shows an example of the thread synchronization control program for two or more input-output statements:



```
IDENTIFICATION DIVISION.
 PROGRAM-ID. B.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
    :
DATA DIVISION.
 FILE SECTION.
  FD file-1 IS EXTERNAL.
  01 record-1 PIC X(80).
    :
 WORKING-STORAGE SECTION.
  01 LOCK-KEY     PIC X(30) VALUE "FILE".
  01 WAIT-TIME    PIC S9(9) COMP-5 VALUE -1.
  01 ERROR-VAL    PIC 9(9) COMP-5.
  01 RET-VAL      PIC S9(9) COMP-5..
    :
PROCEDURE DIVISION.
    :
    CALL  "COB_LOCK_DATA"
                    USING BY REFERENCE LOCK-KEY
                             BY VALUE     WAIT-TIME
                             BY REFERENCE ERROR-VAL                 ...[1]
                    RETURNING RET-VAL.
    READ file-1.                                                    ...[2]
    REWRITE record-1.                                               ...[3]
    CALL "COB_UNLOCK_DATA"
                     USING BY REFERENCE LOCK-KEY
                             BY REFERENCE ERROR-VAL                 ...[4]
                     RETURNING RET-VAL.
    :
```

Explanation of diagram:

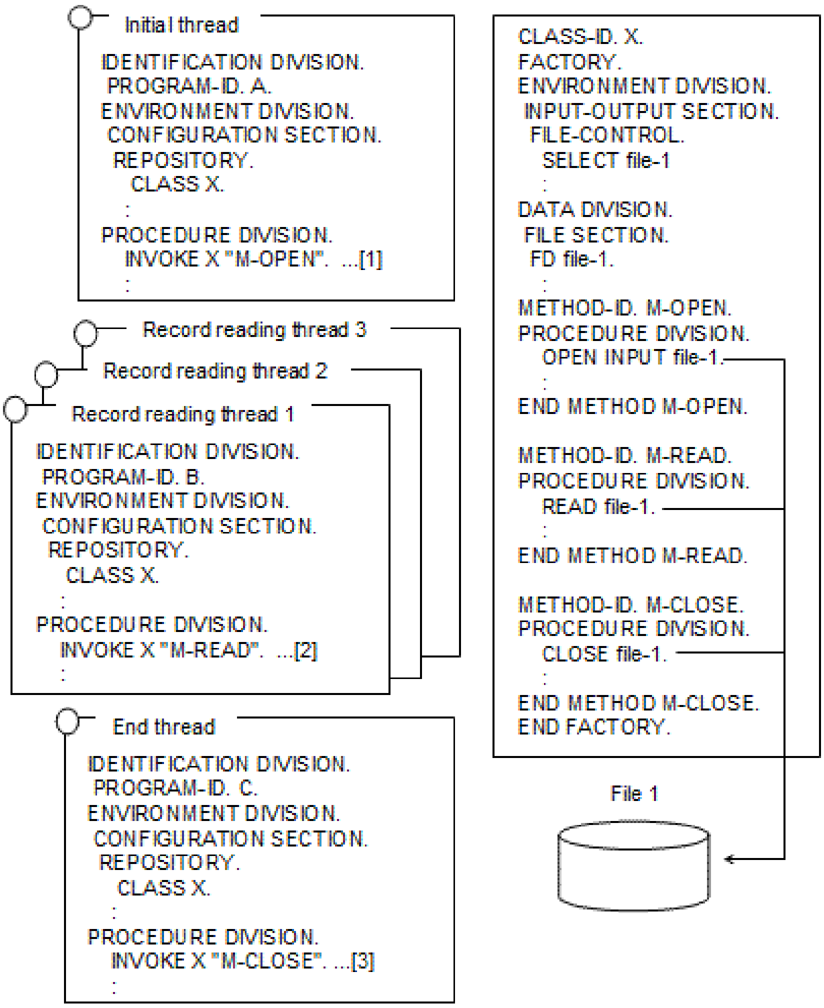
- [1] The data lock subroutine acquires the lock for the lock key associated with the data name FILE.
- [2] The file-1 record is read.
- [3] The record read in [2] is updated.
- [4] The data lock subroutine releases the lock acquired for the lock key associated with data name FILE.

For information on the data lock subroutine, refer to "19.9.1 Data Lock Subroutines".

19.6.1.2 File Defined in the Factory Object

As is the case with an external file, a file defined in the file description entry within the factory object can share the same file connector.

The following shows an example of the program that uses a file in the factory object:



```
IDENTIFICATION DIVISION.
 PROGRAM-ID. A.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
  REPOSITORY.
     CLASS X.
    :
PROCEDURE DIVISION.
    INVOKE X "M-OPEN".  ...[1]
    :
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. B.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
  REPOSITORY.
     CLASS X.
    :
PROCEDURE DIVISION.
    INVOKE X "M-READ".  ...[2]
    :
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. C.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
  REPOSITORY.
     CLASS X.
    :
PROCEDURE DIVISION.
    INVOKE X "M-CLOSE". ...[3]
    :
```

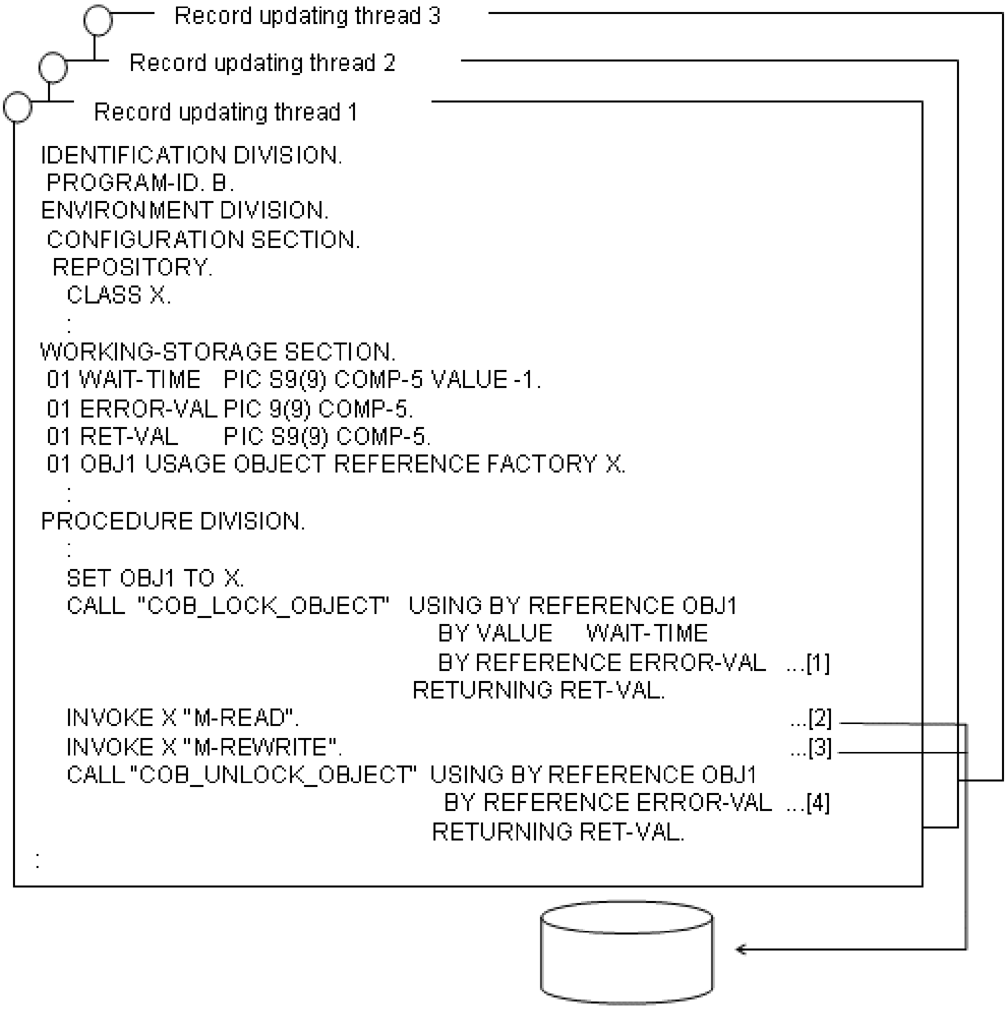
```
CLASS-ID. X.
FACTORY.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
    :
DATA DIVISION.
 FILE SECTION.
  FD file-1.
    :
METHOD-ID. M-OPEN.
PROCEDURE DIVISION.
    OPEN INPUT file-1.
    :
END METHOD M-OPEN.

METHOD-ID. M-READ.
PROCEDURE DIVISION.
    READ file-1.
    :
END METHOD M-READ.

METHOD-ID. M-CLOSE.
PROCEDURE DIVISION.
    CLOSE file-1.
    :
END METHOD M-CLOSE.
END FACTORY.
```

Explanation of diagram:

- [1] The initialization thread is activated only once in the runtime environment. It calls the factory method M-OPEN to open the file.
- [2] Two or more record reading threads (three in this program example) are activated simultaneously. The record reading thread calls the factory method M-READ and reads a record for the file opened by the initial thread.
- [3] The M-CLOSE thread is activated last and only once in the execution environment. It calls the factory method M-CLOSE and closes the file opened by the initial thread.

When the processing is completed with one call of the factory method, the COBOL runtime system automatically performs thread synchronization control. However, to perform the desired processing by calling the factory method two or more times, thread synchronization control must be executed.

The following shows an example of the thread synchronization control program for several calls of the factory method:



```
IDENTIFICATION DIVISION.
 PROGRAM-ID. B.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
  REPOSITORY.
    CLASS X.
    :
WORKING-STORAGE SECTION.
 01 WAIT-TIME   PIC S9(9) COMP-5 VALUE -1.
 01 ERROR-VAL PIC 9(9) COMP-5.
 01 RET-VAL      PIC S9(9) COMP-5.
 01 OBJ1 USAGE OBJECT REFERENCE FACTORY X.
    :
PROCEDURE DIVISION.
    :
    SET OBJ1 TO X.
    CALL  "COB_LOCK_OBJECT"   USING BY REFERENCE OBJ1
                                    BY VALUE     WAIT-TIME
                                    BY REFERENCE ERROR-VAL   ...[1]
                                  RETURNING RET-VAL.
    INVOKE X "M-READ".                                       ...[2]
    INVOKE X "M-REWRITE".                                    ...[3]
    CALL "COB_UNLOCK_OBJECT"  USING BY REFERENCE OBJ1
                                    BY REFERENCE ERROR-VAL   ...[4]
                                  RETURNING RET-VAL.
:
```

Explanation of diagram:

- [1] The object lock subroutine acquires the lock of the factory object.
- [2] The factory method M-READ is called to read the file 1 record.
- [3] The factory method M-REWRITE is called to update the record that was read in [2].
- [4] The object lock subroutine unlocks the factory object.
- For information on the object lock subroutine, refer to "19.9.2 Object Lock Subroutines".

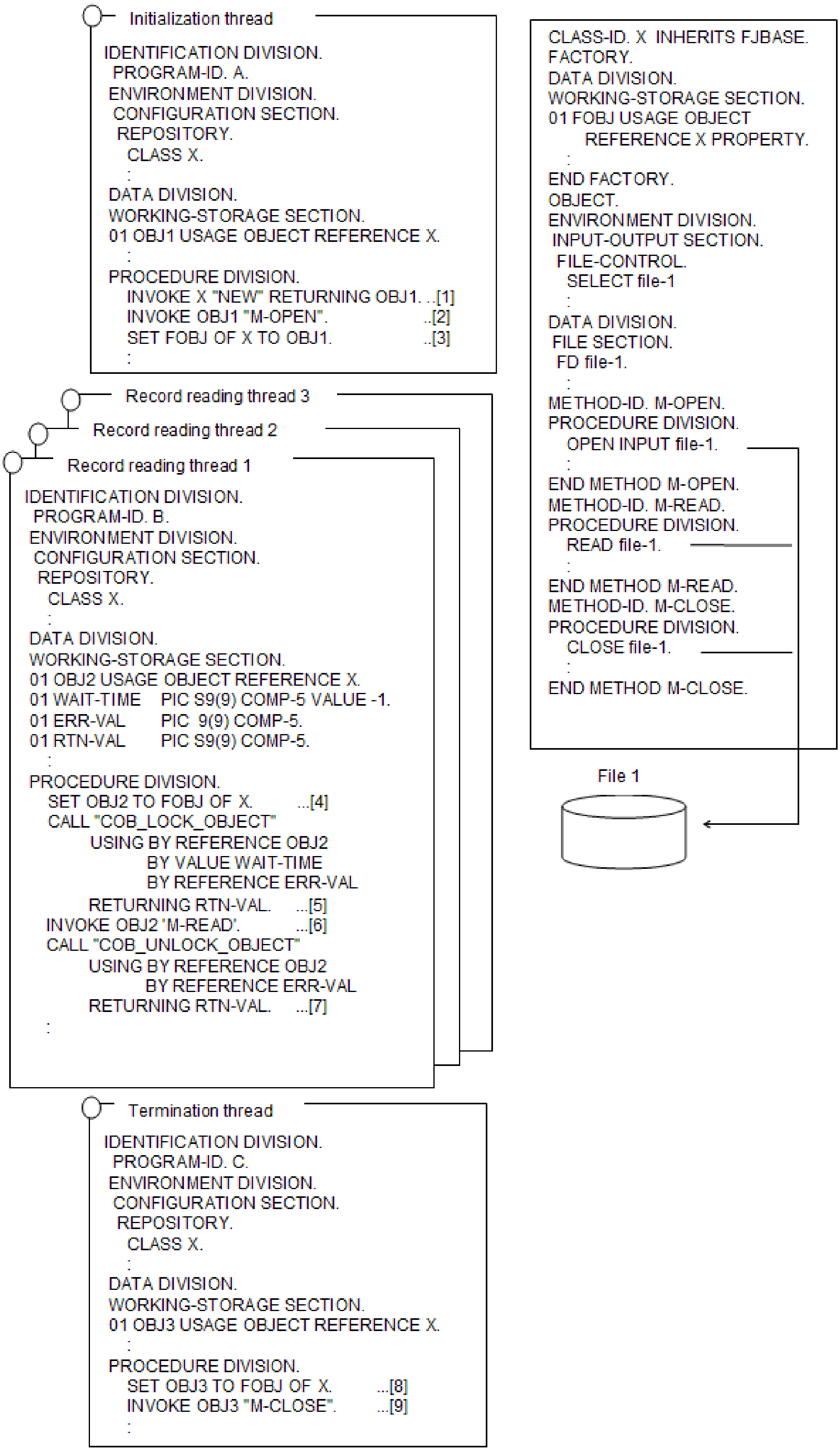
19.6.1.3 File Defined in the Object

As is the case with an external file, a file defined in the file description entry in the object can share the same file connector by sharing a single object instance.

The COBOL runtime system does not perform thread synchronization control for the object instance. Therefore, to operate a file in the object by sharing a single object instance, thread synchronization control must be executed.

Synchronization control between threads for the file in the object is executed by using the object lock subroutine.

The following shows an example of the program that uses the file in the object:



```
IDENTIFICATION DIVISION.
 PROGRAM-ID. A.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
  REPOSITORY.
    CLASS X.
    :
DATA DIVISION.
WORKING-STORAGE SECTION.
01 OBJ1 USAGE OBJECT REFERENCE X.
    :
PROCEDURE DIVISION.
    INVOKE X "NEW" RETURNING OBJ1. ..[1]
    INVOKE OBJ1 "M-OPEN".          ..[2]
    SET FOBJ OF X TO OBJ1.         ..[3]
    :
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. B.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
  REPOSITORY.
    CLASS X.
    :
DATA DIVISION.
WORKING-STORAGE SECTION.
01 OBJ2 USAGE OBJECT REFERENCE X.
01 WAIT-TIME    PIC S9(9) COMP-5 VALUE -1.
01 ERR-VAL      PIC  9(9) COMP-5.
01 RTN-VAL      PIC S9(9) COMP-5.
    :
PROCEDURE DIVISION.
    SET OBJ2 TO FOBJ OF X.         ...[4]
    CALL "COB_LOCK_OBJECT"
         USING BY REFERENCE OBJ2
               BY VALUE WAIT-TIME
               BY REFERENCE ERR-VAL
         RETURNING RTN-VAL.    ...[5]
    INVOKE OBJ2 'M-READ'.      ...[6]
    CALL "COB_UNLOCK_OBJECT"
         USING BY REFERENCE OBJ2
               BY REFERENCE ERR-VAL
         RETURNING RTN-VAL.    ...[7]
    :
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. C.
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
  REPOSITORY.
    CLASS X.
    :
DATA DIVISION.
WORKING-STORAGE SECTION.
01 OBJ3 USAGE OBJECT REFERENCE X.
    :
PROCEDURE DIVISION.
    SET OBJ3 TO FOBJ OF X.       ...[8]
    INVOKE OBJ3 "M-CLOSE".       ...[9]
    :
```

```
CLASS-ID. X  INHERITS FJBASE.
FACTORY.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 FOBJ USAGE OBJECT
       REFERENCE X PROPERTY.
    :
END FACTORY.
OBJECT.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-1
    :
DATA DIVISION.
 FILE SECTION.
  FD file-1.
    :
METHOD-ID. M-OPEN.
PROCEDURE DIVISION.
    OPEN INPUT file-1.
    :
END METHOD M-OPEN.
METHOD-ID. M-READ.
PROCEDURE DIVISION.
    READ file-1.
    :
END METHOD M-READ.
METHOD-ID. M-CLOSE.
PROCEDURE DIVISION.
    CLOSE file-1.
    :
END METHOD M-CLOSE.
```

Explanation of diagram:

The initialization threads ([1] to [3]) are activated only once at the beginning in the runtime environment.

- [1] The class 'X' object instance is acquired.
- [2] The method M-OPEN is called to open the file.
- [3] The object instance is inserted into factory data FOBJ that includes the PROPERTY clause.
- Two or more record reading threads ([4] to [7]) (three in this program example) are simultaneously activated.
- [4] The factory data FOBJ is inserted into the object instance OBJ2. This enables the sharing of the object instance used in program A.
- [5] The object lock subroutine acquires the lock of the object instance.
- [6] The method M-READ is called to read the record of the file opened by program A.
- [7] The object lock subroutine unlocks the object instance.
- The end threads ([8] to [9]) are lastly activated only once in the runtime environment.
- [8] The factory data FOBJ is inserted into the object instance OBJ3. This enables the sharing of the object instance used by program A.
- [9] The method M-CLOSE is called to close the file opened by the initialization thread.

For information on the object lock subroutine, refer to "19.9.2 Object Lock Subroutines".

19.6.2 Using the Remote Database Access (ODBC)

In the basic use methods above, a single-thread program is transformed, intact, to a multithread program.

This subsection explains a slightly more advanced use, constructing a new multithread program that makes the use of multithread advantages.

19.6.2.1 Sharing the Connection between Threads

Sharing the connection between threads enables connection and disconnection processing having a large overhead to be separated from processing into initialization and termination threads, respectively. This function also enables data to be manipulated with two or more threads. It increases application performance. However, to share the connection between threads, the following points must be noted.

When two or more threads use a single connection, they also share the transaction. This means that transaction processing performed by a thread affects the data manipulation of all threads that share the connection. There is no problem with this effect when a multithread program that does not change the database table is run. However, when a multithread program that changes the database table is run, it must be programmed so that each of the threads to be simultaneously executed performs transaction processing by using a different connection.

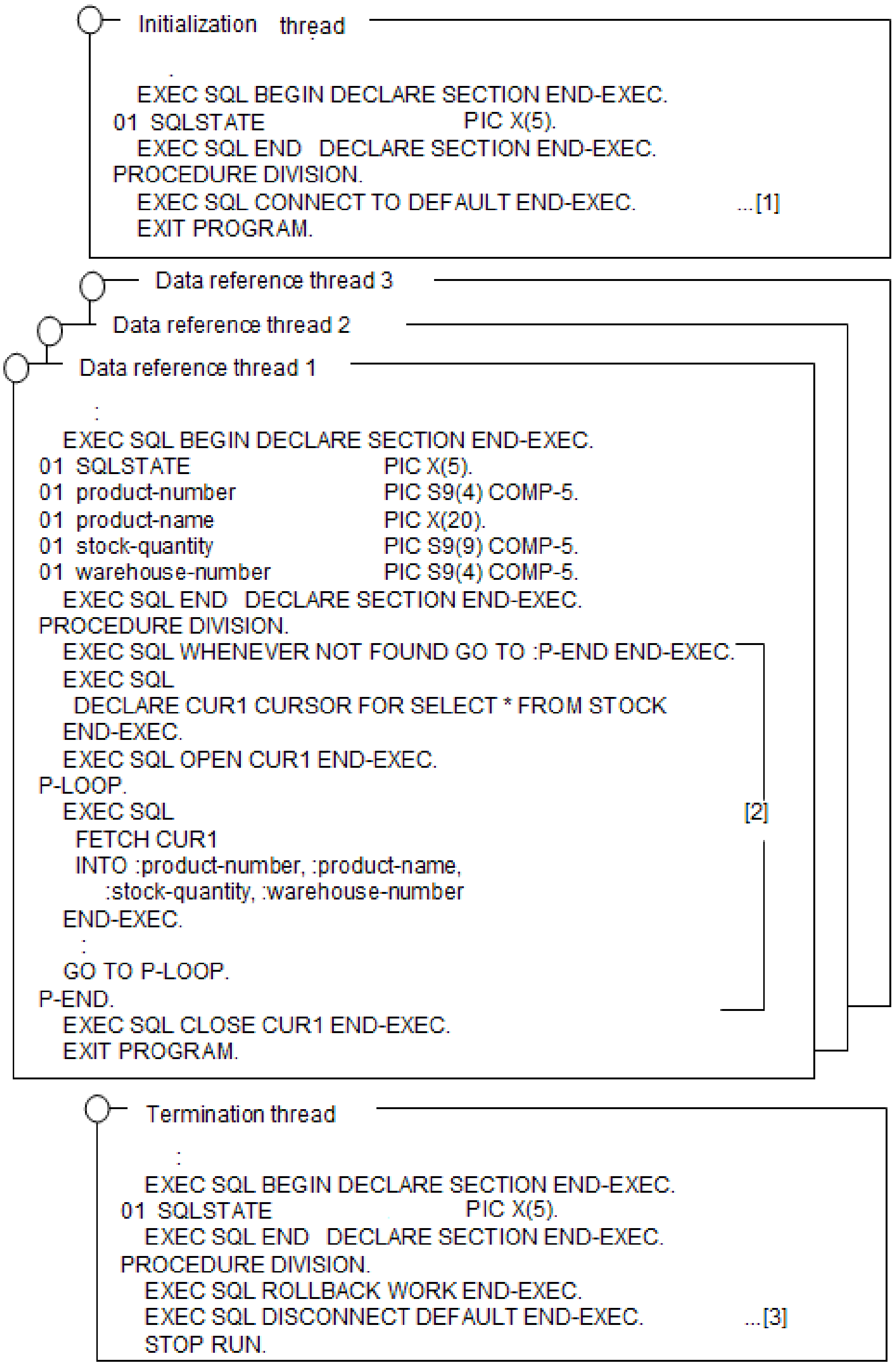
The following explains an example in which the database table is not changed (merely referenced) and an example of updating the database table:

These examples use the STOCK table of the sample database. Refer to "17.2.3.1 Sample Database".

Referencing data at connection sharing

The following shows an example in which the connection is shared between threads and the database table is not changed (merely referenced):

Figure 19.1 Example of referencing data at connection sharing



```
        :
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  SQLSTATE                       PIC X(5).
   EXEC SQL END   DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
   EXEC SQL CONNECT TO DEFAULT END-EXEC.              ...[1]
   EXIT PROGRAM.
```

```
        :
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  SQLSTATE                       PIC X(5).
01  product-number                 PIC S9(4) COMP-5.
01  product-name                   PIC X(20).
01  stock-quantity                 PIC S9(9) COMP-5.
01  warehouse-number               PIC S9(4) COMP-5.
   EXEC SQL END   DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
   EXEC SQL WHENEVER NOT FOUND GO TO :P-END END-EXEC.
   EXEC SQL
    DECLARE CUR1 CURSOR FOR SELECT * FROM STOCK
   END-EXEC.
   EXEC SQL OPEN CUR1 END-EXEC.
P-LOOP.
   EXEC SQL                                                [2]
    FETCH CUR1
    INTO :product-number, :product-name,
         :stock-quantity, :warehouse-number
   END-EXEC.
      :
   GO TO P-LOOP.
P-END.
   EXEC SQL CLOSE CUR1 END-EXEC.
   EXIT PROGRAM.
```

```
        :
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  SQLSTATE                       PIC X(5).
   EXEC SQL END   DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
   EXEC SQL ROLLBACK WORK END-EXEC.
   EXEC SQL DISCONNECT DEFAULT END-EXEC.              ...[3]
   STOP RUN.
```

Explanation of diagram:

- [1] The initialization thread is activated only once in the runtime environment.

  The initialization thread executes the CONNECT statement to establish the connection to the server.

- [2] The data reference thread is activated for each request made from a client.

  In the data reference thread, data is fetched, line by line, from the STOCK table by using the cursor, and a value for each column is set in the area of the associated host variable. If there is no data to be fetched, control branches to the position of the procedure name P-END specified by the WHENEVER statement to end the data reference thread.

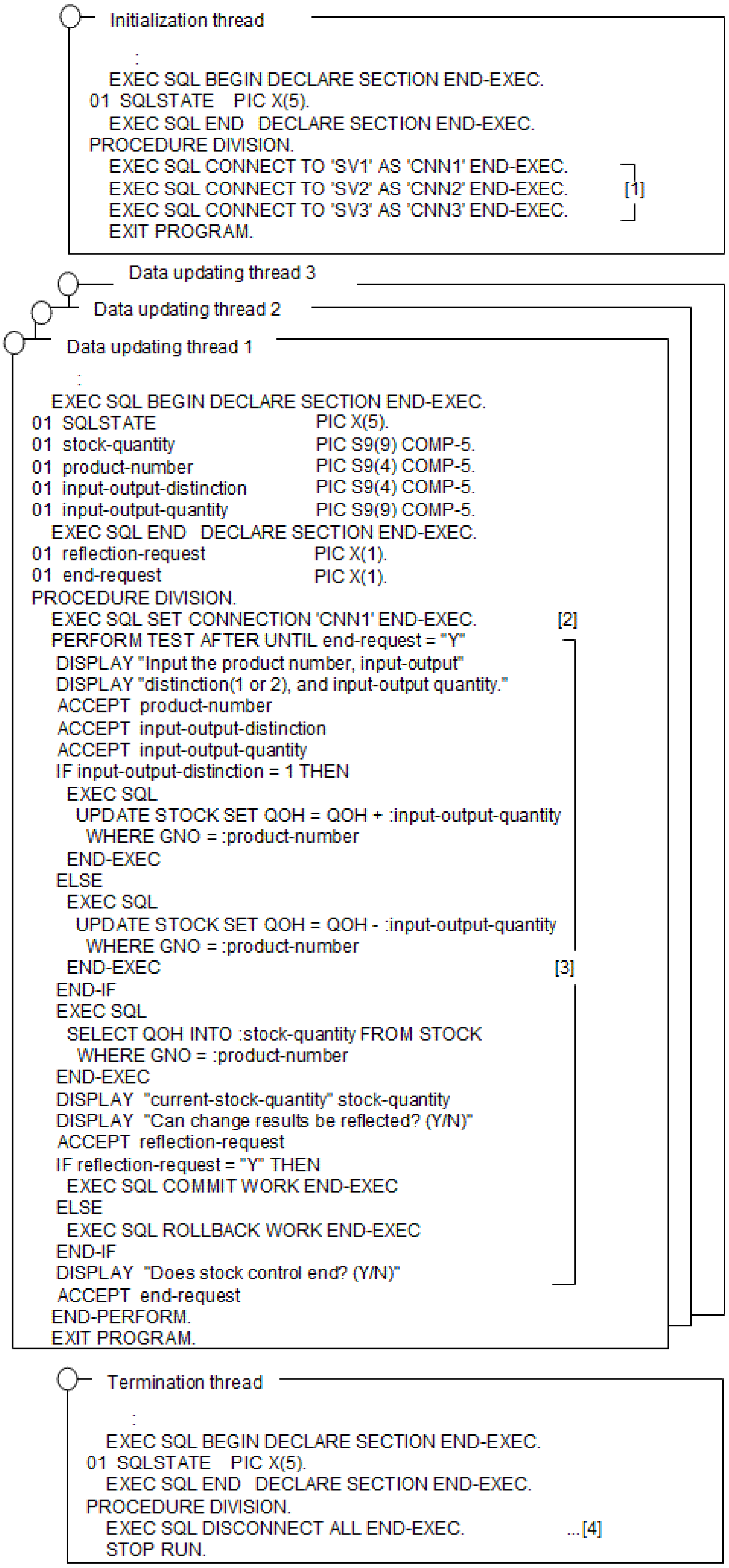
- [3] The termination thread is activated only once in the execution environment.

  The termination thread executes the ROLLBACK statement to end the transaction, and executes the DISCONNECT statement to disconnect the connection to the server.

Updating the data at connection sharing

Figure "Example of updating data at connection sharing" shows an example of the COBOL program that shares the connection between threads and updates the database table.

Figure 19.2 Example of updating data at connection sharing



Initialization thread

```
      :
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  SQLSTATE    PIC X(5).
    EXEC SQL END   DECLARE SECTION END-EXEC.
 PROCEDURE DIVISION.
    EXEC SQL CONNECT TO 'SV1' AS 'CNN1' END-EXEC.      ┐
    EXEC SQL CONNECT TO 'SV2' AS 'CNN2' END-EXEC.      [1]
    EXEC SQL CONNECT TO 'SV3' AS 'CNN3' END-EXEC.      ┘
    EXIT PROGRAM.
```

Data updating thread 3

Data updating thread 2

Data updating thread 1

```
      :
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  SQLSTATE                      PIC X(5).
 01  stock-quantity                PIC S9(9) COMP-5.
 01  product-number                PIC S9(4) COMP-5.
 01  input-output-distinction      PIC S9(4) COMP-5.
 01  input-output-quantity         PIC S9(9) COMP-5.
    EXEC SQL END   DECLARE SECTION END-EXEC.
 01  reflection-request            PIC X(1).
 01  end-request                   PIC X(1).
 PROCEDURE DIVISION.
    EXEC SQL SET CONNECTION 'CNN1' END-EXEC.              [2]
    PERFORM TEST AFTER UNTIL end-request = "Y"            ┐
     DISPLAY "Input the product number, input-output"
     DISPLAY "distinction(1 or 2), and input-output quantity."
     ACCEPT  product-number
     ACCEPT  input-output-distinction
     ACCEPT  input-output-quantity
     IF input-output-distinction = 1 THEN
       EXEC SQL
        UPDATE STOCK SET QOH = QOH + :input-output-quantity
          WHERE GNO = :product-number
       END-EXEC
     ELSE
       EXEC SQL
        UPDATE STOCK SET QOH = QOH - :input-output-quantity
          WHERE GNO = :product-number
       END-EXEC                                           [3]
     END-IF
     EXEC SQL
       SELECT QOH INTO :stock-quantity FROM STOCK
         WHERE GNO = :product-number
     END-EXEC
     DISPLAY  "current-stock-quantity" stock-quantity
     DISPLAY  "Can change results be reflected? (Y/N)"
     ACCEPT  reflection-request
     IF reflection-request = "Y" THEN
       EXEC SQL COMMIT WORK END-EXEC
     ELSE
       EXEC SQL ROLLBACK WORK END-EXEC
     END-IF
     DISPLAY  "Does stock control end? (Y/N)"             ┘
     ACCEPT  end-request
    END-PERFORM.
    EXIT PROGRAM.
```

Termination thread

```
      :
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  SQLSTATE    PIC X(5).
    EXEC SQL END   DECLARE SECTION END-EXEC.
 PROCEDURE DIVISION.
    EXEC SQL DISCONNECT ALL END-EXEC.                ...[4]
    STOP RUN.
```

Explanation of diagram:

- [1] The initialization thread is activated only once in the runtime environment.

  The initialization thread executes CONNECT statements only by the number of data change threads to be simultaneously activated (three in this program example), to establish the connection to the server.

- [2] Two or more data updating threads (three in this program example) are activated at the same time.

  The data updating thread executes the SET CONNECTION statement to determine the current connection to be used. Specify the connection name (CNN1 to CNN3) that is different for each data change thread.

- [3] A request is made to input the product number, input-output distinction, and input-output quantity, and the stock quantity is calculated by isolating the product stocking or taking. Next, the stock quantity after recalculation is displayed, and a request is made to input information on whether to post the change results to the table.

  A processing series is repeated until Y is entered for the end request.

- [4] The termination thread is activated only once in the execution environment.

  The end thread executes the DISCONNECT statement to perform disconnection from all servers.

Setting an environment at execution

To share the connection between threads, specify a process in the connection scope of the ODBC information file. Refer to "17.2.8.1.2 Creating an ODBC Information File".

Use the ODBC information setup tool to create the ODBC information file. Refer to "17.2.8.2 Using the ODBC Information Setup Tool".

The connection can be shared among threads by specifying the process in the connection scope.

19.6.2.2 Notes

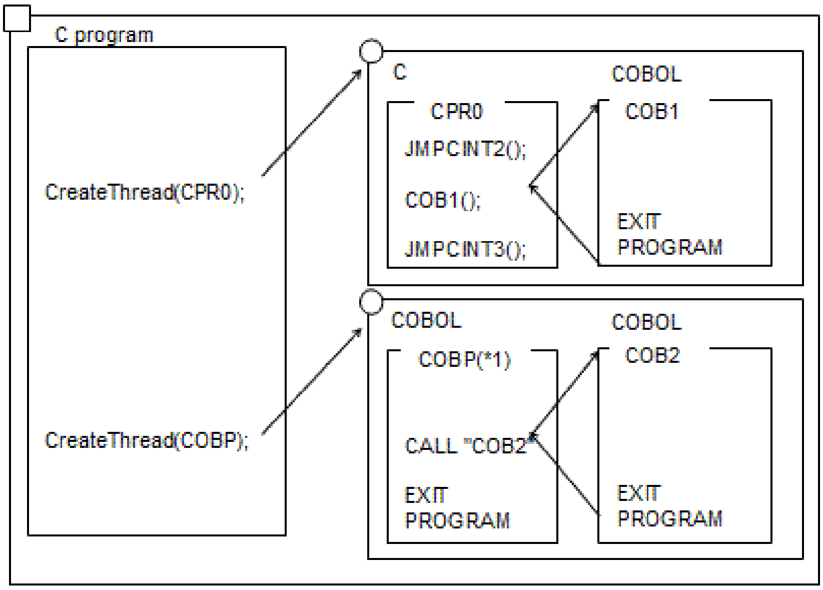
For details, refer to "19.5.2.3 Notes".

19.6.3 Activating the COBOL Program from the C Program as a Thread

This subsection explains how to activate the COBOL program from the C program as a thread.

19.6.3.1 Overview

Unlike cases where COBOL is called from the C program, use the Windows application programming interface(API) function to activate the COBOL program as a thread. When the EXIT PROGRAM statement is executed with the activated COBOL program thread, the thread merely ends and control does not return to the calling source. When a COBOL program is called from the activated COBOL program thread and the EXIT PROGRAM statement is executed by the called COBOL program, control returns immediately after the call. This rule applies to cases where the C program is activated as a thread.



19.6.3.2 Activation

To activate the COBOL program from the C program as a thread, use the Windows API CreateThread. The C program that activated the thread continues processing without waiting for the end of the thread. Accordingly, the C program may terminate before the COBOL program ends.

To wait for the end of the thread activated by the C program, use the Windows API WaitForSingleObject. To wait for the end of two or more threads, use the Windows API WaitForMultipleObjects.

19.6.3.3 Passing Parameters

To pass an argument to a COBOL program activated as a thread from a C program, specify an actual argument in the fourth argument of Windows API CreateThread. Only one actual argument can be specified. The actual argument value that can be passed to the COBOL program must be a storage area address. In the COBOL program, the content of the area existing in the address specified in the actual argument is received by specifying the data name in the procedure division header (PROCEDURE DIVISION) or the USING phase of the ENTRY statement. If CONST is specified in the definition of the argument, the content of the argument must remain unchanged.

19.6.3.4 Return Code (Function Value)

Use the Windows API GetExitCodeThread to fetch a value specified in the item of the RETURNING phase of the procedure division header (PROCEDURE DIVISION) or in the special register PROGRAM-STATUS. The item to be specified in the RETURNING phase of the procedure division header must correspond with the data type ([1], [2], and [3] in the figure below) of the C program. For information on the correspondence between a data type and item, refer to "9.3.3 Correspondence of COBOL and C Data Types".

Function C

```
extern int                //[1]
  COB(int *);


                :
int WINAPI WinMain (  ~  )
 {
/* Thread handle                                    */
  HANDLE cobhnd;
/* Thread ID                                        */
  int    cobthid;
/* Parameter to be passed to the COBOL program      */
  int    cobprm;
/* Return code                                      */
  int    cobrcd;        ...[2]
```

```
/* The COBOL program (COB) is activated as a thread.*/
  cobhnd = CreateThread(NULL,
                     0,
                     (LPTHREAD_START_ROUTINE)COB,
                     &cobprm,
                     0,
                     &cobthid);
                 :
  GetExitCodeThread(cobhnd,&cobrcd);
```

COBOL Program

```
 IDENTIFICATION DIVISION.
  PROGRAM-ID. COB.
 DATA DIVISION.
  LINKAGE SECTION.
  01 PRM     PIC S9(9) COMP-5.
  01 RTN-ITM PIC S9(9) COMP-5. *>[3]

 PROCEDURE DIVISION
                    USING PRM
                    RETURNING RTN-ITM.
*>          :
     MOVE 0 TO RTN-ITM.
     IF PRM < 0
      THEN MOVE 99 TO RTN-ITM
```

Note

- If the RETURNING phase is specified in the procedure division header, no value specified in the special register PROGRAM-STATUS is passed to the C program that activated the thread.
  To pass the function value with the special register PROGRAM-STATUS, the C program that activated the thread must receive the function value as the long long int type.

  Function C

```
extern long long int
 COB(int *);
//     :
int WINAPI WinMain (  ~  )
 {
  HANDLE cobhnd;
  int    cobthid;
  int    cobprm;
  long int cobrcd;
//     :
  cobhnd = CreateThread(NULL,
                     0,
                     (LPTHREAD_START_ROUTINE)COB,
                     &cobprm,
                     0,
                     &cobthid);
//     :
  GetExitCodeThread(cobhnd,&cobrcd);
```

  COBOL Program

```
 IDENTIFICATION DIVISION.
  PROGRAM-ID. COB.
 DATA DIVISION.
  LINKAGE SECTION.
  01 PRM PIC S9(9) COMP-5.
*>       :
 PROCEDURE DIVISION USING PRM.
*>        :
     IF PRM < 0
      THEN MOVE 99 TO PROGRAM-STATUS
```

- When the resetting value is taken out with GetExitCodeThread, the value that can be returned as a resetting value is a range of 32 bits.

19.6.3.5 Compilation and Link

This subsection explains the compilation and link, taking the program shown below as an example.

C program (CPROG.C)

This program activates COBOL programs COBTHD1, COBTHD2, and COBTHD3 as threads, and acquires the return code of each COBOL program.

```
#include "windows.h"
extern int COBTHD1(int *);
extern int COBTHD2(int *);
extern int COBTHD3(int *);

//      :

int WINAPI WinMain (  ~  )
 {
/* Data declaration  */
  HANDLE cobhnd1;
  HANDLE cobhnd23[2];
  DWORD  cobrcd1;
  DWORD  cobrcd23[2];
  int    cobtid1;
  int    cobtid23[2];
  int    cobprm1;
  int    cobprm23[2];
/* Activate COBTHD1 by specifying 1 in the parameter.  */
  cobprm1 = 1;
  cobhnd1 = CreateThread(NULL,
                         0,
                         (LPTHREAD_START_ROUTINE)COBTHD1,
                         &cobprm1,
                         0,
                         &cobtid1);
/* The program waits for the COBTHD1 to end.           */
 WaitForSingleObject(cobhnd1,INFINITE);
/* The program acquires the return code of COBTHD1.    */
 GetExitCodeThread(cobhnd1,&cobrcd1);
/* Specify parameters 2 and 3 to activate COBTHD2 and COBTHD3.  */
 cobprm23[0] = 2;
 cobprm23[1] = 3;
 cobhnd23[0] = CreateThread(NULL,
                         0,
                         (LPTHREAD_START_ROUTINE)COBTHD2,
                         &cobprm23[0],
                         0,
                         &cobtid23[0]);
 cobhnd23[1] = CreateThread(NULL,
                         0,
                         (LPTHREAD_START_ROUTINE)COBTHD3,
                         &cobprm23[1],
                         0,
                         &cobtid23[1]);
/* The program waits for COBTHD2 and COBTHD3 to end. */
 WaitForMultipleObjects(2,cobhnd23,TRUE,INFINITE);
/* The program acquires the return codes of COBTHD2 and COBTHD3.  */
 GetExitCodeThread(cobhnd23[0],&cobrcd23[0]);
 GetExitCodeThread(cobhnd23[1],&cobrcd23[1]);
/* The program closes the handle of the activated thread.  */
 CloseHandle(cobhnd1);
 CloseHandle(cobhnd23[0]);
 CloseHandle(cobhnd23[1]);
 return TRUE;
}
```

COBOL program (COBTHD1.COB)

This program is activated from the C program (CPROG.C) as a thread. If the parameter value is 1, the program returns return value 0. If it is a value other than 1, the program returns return value -1.

COBOL Program

```
 IDENTIFICATION DIVISION.
  PROGRAM-ID. COBTHD1.
 DATA DIVISION.
  LINKAGE SECTION.
   01 PRM1 PIC S9(9) COMP-5.
 PROCEDURE DIVISION USING PRM1.
     IF PRM1 = 1
      THEN MOVE  0 TO PROGRAM-STATUS
      ELSE MOVE -1 TO PROGRAM-STATUS
     END-IF.
     EXIT PROGRAM.
```

COBOL program (COBTHD2.COB)

This program is activated from the C program (CPROG.C) as a thread. If the parameter value is 2, the program returns return value 0. If it is a value other than 2, the program returns return value -1.

COBOL Program

```
 IDENTIFICATION DIVISION.
  PROGRAM-ID. COBTHD2.
 DATA DIVISION.
  LINKAGE SECTION.
   01 PRM2 PIC S9(9) COMP-5.
 PROCEDURE DIVISION USING PRM2.
     IF PRM2 = 2
      THEN MOVE  0 TO PROGRAM-STATUS
      ELSE MOVE -1 TO PROGRAM-STATUS
     END-IF.
     EXIT PROGRAM.
```

COBOL program (COBTHD3.COB)

This program is activated from the C program (CPROG.C) as a thread. If the parameter value is 3, the program returns return value 0. If it is a value other than 3, the program returns return value -1.

COBOL Program

```
 IDENTIFICATION DIVISION.
  PROGRAM-ID. COBTHD3.
 DATA DIVISION.
  LINKAGE SECTION.
   01 PRM3 PIC S9(9) COMP-5.
 PROCEDURE DIVISION USING PRM3.
     IF PRM3 = 3
```

```
      THEN MOVE  0 TO PROGRAM-STATUS
      ELSE MOVE -1 TO PROGRAM-STATUS
     END-IF.
     EXIT PROGRAM.
```

Compilation

- Compiling C program

```
CL -c -MD CPROG.C
```

- Compiling COBOL program COBTHD1

```
COBOL -WC,"THREAD(MULTI)" COBTHD1.COB
```

- Compiling COBOL program COBTHD2

```
COBOL -WC,"THREAD(MULTI)" COBTHD2.COB
```

- Compiling COBOL program COBTHD3

```
COBOL -WC,"THREAD(MULTI)" COBTHD3.COB
```

Note

To compile the COBOL program, be sure to specify the compiler option, THREAD (MULTI).

Link

For how to link objects, see "DLL Entry Object Linking", "5.3.2.3.4 Using the runtime initialization file under DLL".

- Linking the COBOL program COBTHD1

```
LINK COBTHD1.OBJ F4AGCBDM.OBJ F4AGCIMP.LIB KERNEL32.LIB /DLL /ENTRY:COBDMAIN /OUT:COBTHD1.DLL
```

- Linking COBOL program COBTHD2

```
LINK COBTHD2.OBJ F4AGCBDM.OBJ F4AGCIMP.LIB KERNEL32.LIB /DLL /ENTRY:COBDMAIN /OUT:COBTHD2.DLL
```

- Linking COBOL program COBTHD3

```
LINK COBTHD3.OBJ F4AGCBDM.OBJ F4AGCIMP.LIB KERNEL32.LIB /DLL /ENTRY:COBDMAIN /OUT:COBTHD3.DLL
```

See

COBTHD1.OBJ	COBOL program COBTHD1 object
COBTHD2.OBJ	COBOL program COBTHD2 object
COBTHD3.OBJ	COBOL program COBTHD3 object
F4AGCBDM.OBJ	DLL entry object
F4AGCIMP.LIB	COBOL runtime system import library
KERNEL32.LIB	Windows API import library

- Linking the C program

```
LINK CPROG.OBJ COBTHD1.LIB COBTHD2.LIB COBTHD3.LIB /OUT:CPROG.EXE
```

See

CPROG.OBJ	C program CPROG object
COBTHD1.LIB	COBOL program COBTHD1 import library
COBTHD2.LIB	COBOL program COBTHD2 import library
COBTHD3.LIB	COBOL program COBTHD3 import library

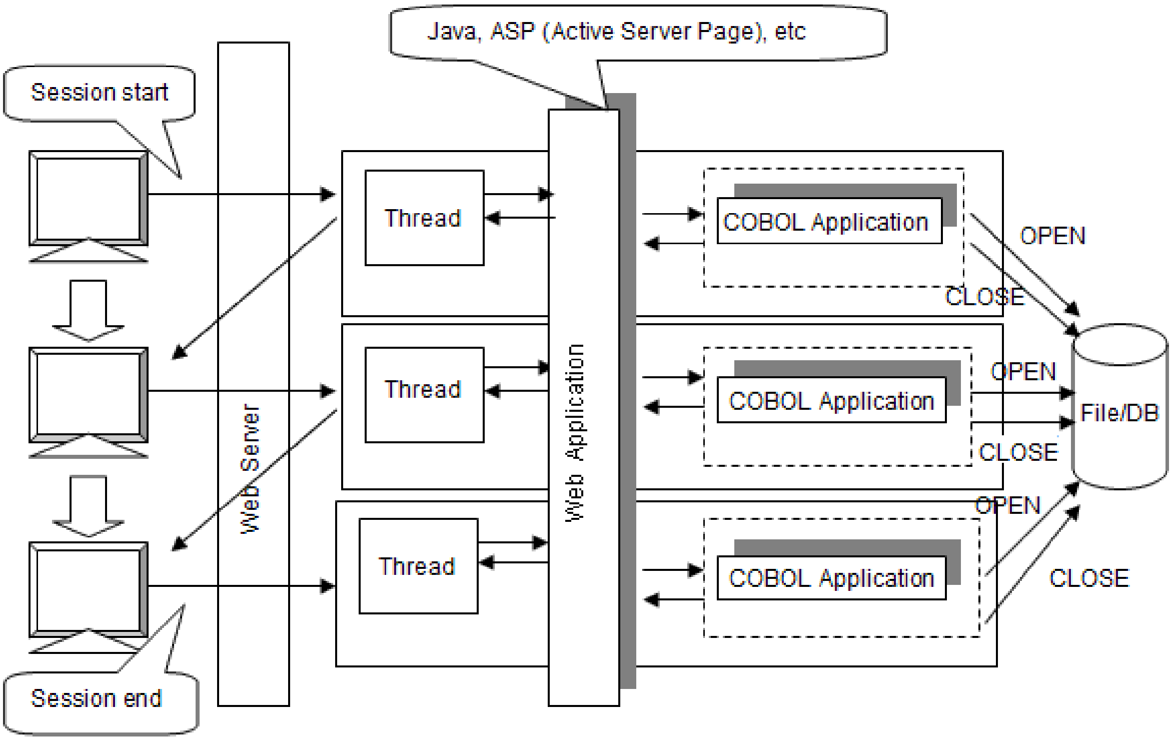
19.6.4 Method to relay run unit data between multiple threads

This section describes a method to relay run unit data between multiple threads.

19.6.4.1 Overview

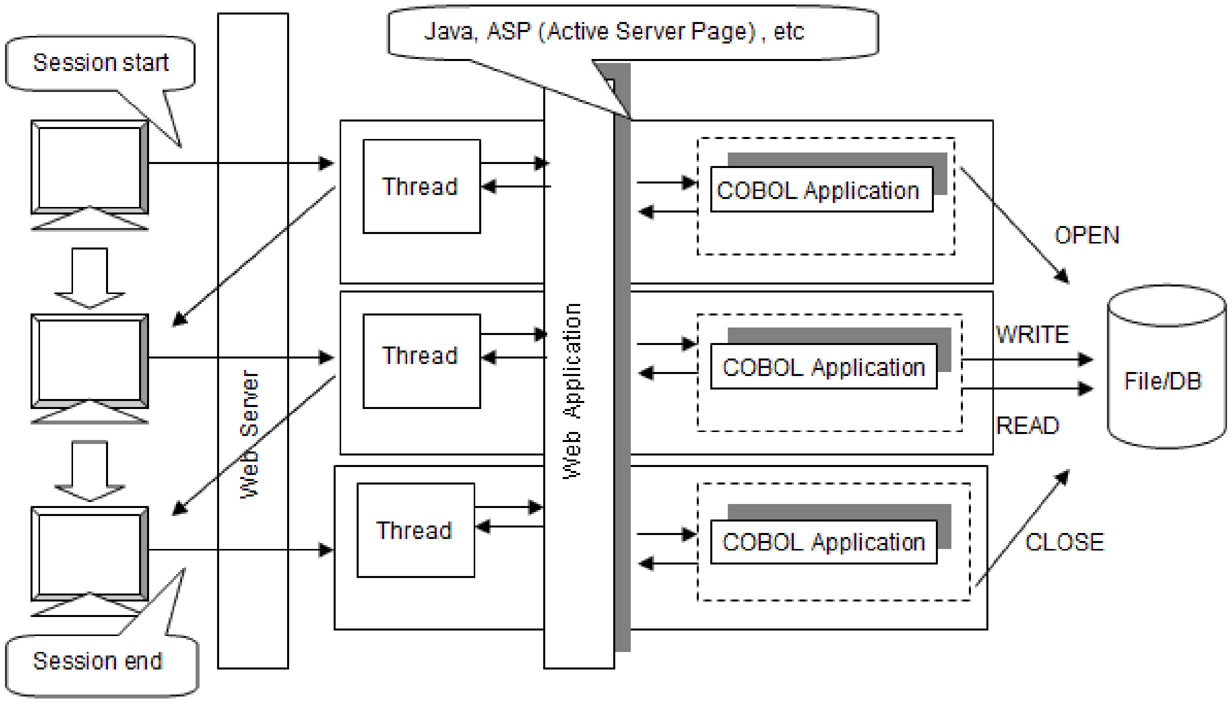
When a server application of COBOL runs over multiple threads, the COBOL run unit starts when the server application of COBOL is called by a client and it ends when control returns to the client. Consequently, resources which are valid in run units, such as a file connector, DB cursor, and data described in a WORKING-STORAGE section is released at the moment running ends. For example, COBOL run unit resources are created and released each time the client calls in a Web application. COBOL run unit resources cannot therefore be held in a session which runs over several threads.

Figure 19.3 When run unit data is not relayed



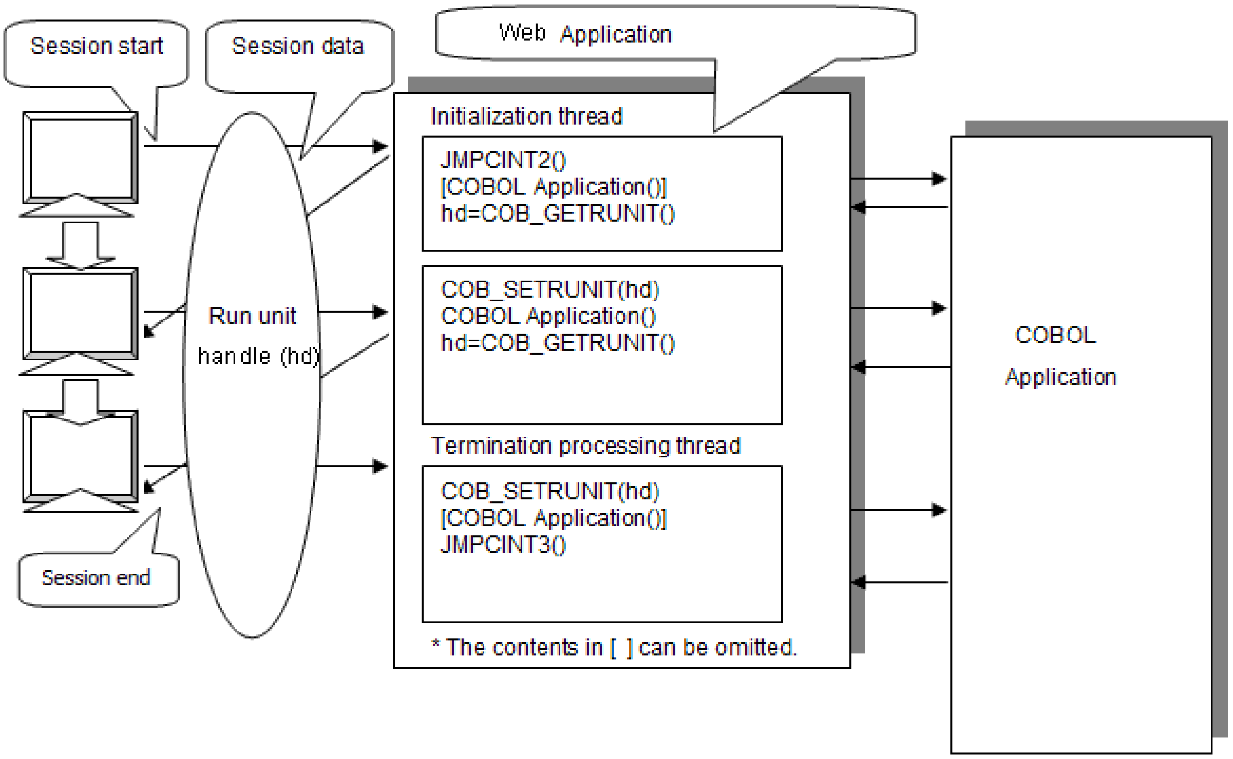
So that a COBOL application may relay run unit resources in a session which runs over several threads, it must run in one thread each time the client calls a COBOL application. However, the thread by which the server application executed is not fixed to the same thread. In this case, run unit resources can be relayed between the threads by using two types of subroutines: (1) To return a COBOL run unit handle; and (2) To notify the COBOL run time system of the run unit handle. The linkage between a run unit handle to be relayed and the session is made by the caller of a COBOL application, using a management technique such as a cookie.

Figure 19.4 When run unit data is relayed



19.6.4.2 Method of use

A COBOL execution-unit handle can be shared by threads by calling a subroutine offered by COBOL as shown below.



If run unit resources created in the other thread (such as file connector, DB cursor, and data described in the WORKING-STORAGE section) can be relayed, files do not need to be opened and closed each time the client calls. Thus, overhead can be reduced and operations can be made more efficient.

19.6.4.3 Use of subroutine

19.6.4.3.1 COBOL run unit handle acquiring subroutine

Use the COB_GETRUNIT subroutine when acquiring a handle that identifies a run unit of COBOL.

JMPCINT2 should be called in the start of the processing of the initialization thread and the run unit should be started.

Specification method

Description of call (C language calling)

```
/* Type declaration division */
  extern unsigned long long COB_GETRUNIT(void);
/* Data declaration division */
  unsigned long long hd;  /* COBOL run unit handle storage area */
/* Procedure division */
  hd = COB_GETRUNIT();
```

Interface

Parameter :

none

Return code :

Normal : COBOL run unit handle

Error : zero

19.6.4.3.2 COBOL run unit handle setting subroutine

Use the COB_SETRUNIT subroutine when setting up a COBOL run unit handle in the thread of a caller.

JMPCINT3 should be called in the end of the end thread and at the end the execution unit.

Specification method

Description of call (C language calling)

```
/* Type declaration division */
  extern int COB_SETRUNIT(unsigned long long hd);
/* Data declaration division */
  extern unsigned long long hd; /* COBOL run unit handle storing area */
                           /*       (acquired by COB_GETRUNIT) */
/* Procedure division */
  hd = COB_SETRUNIT(hd);
```

Interface

Parameter :

hd : COBOL execution-unit handle

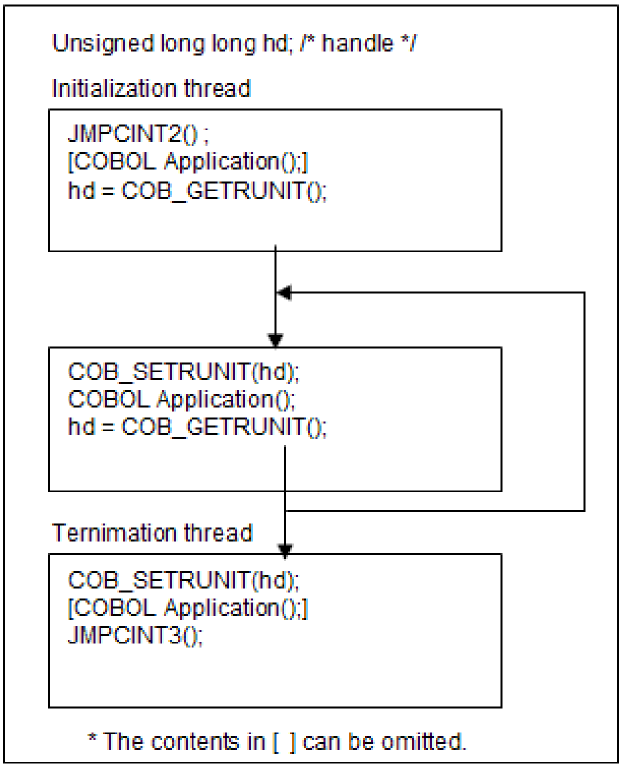
Return code :

Normal : zero

Error :

- -1 Input parameter is not correct.

- -2 A COBOL run unit data already exist.
- -99 Other error (contact a system engineer).



```
JMPCINT2() ;
[COBOL Application();]
hd = COB_GETRUNIT();
```

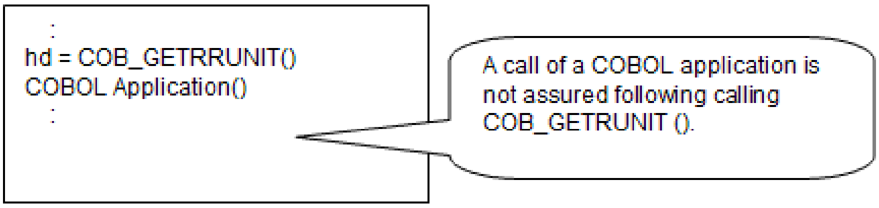
```
COB_SETRUNIT(hd);
COBOL Application();
hd = COB_GETRUNIT();
```

```
COB_SETRUNIT(hd);
[COBOL Application();]
JMPCINT3();
```

* The contents in [ ] can be omitted.

19.6.4.4 Note

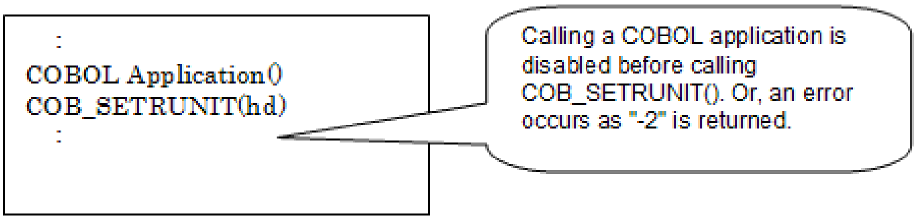
- Where an error has occurred such that a COBOL application running does not end (by time out), an execution-unit handle must be set up by the COB_SETRUNIT subroutine and then a JMPCINT3 function must be called to end the operation. Unless the JMPCINT3 function is called, execution-unit data remains unreleased. This will lead to a memory leak.
- The threads in one session must be controlled synchronously. Two or more threads cannot share a COBOL run unit at the same time. Each COBOL run unit must be used only by one thread.
- When the COB_GETRUNIT subroutine is called, the COBOL run unit of a called thread is cleared. Consequently, a COBOL application cannot be called from the same thread after the COB_GETRUNIT subroutine is called.



```
   :
hd = COB_GETRRUNIT()
COBOL Application()
   :
```

A call of a COBOL application is not assured following calling COB_GETRUNIT ().

- When a COB_SETRUNIT subroutine is called, the COBOL run unit of the called thread is overwritten. Consequently, a COBOL application cannot be called from the thread before calling COB_SETRUNIT subroutine.



```
   :
COBOL Application()
COB_SETRUNIT(hd)
   :
```

Calling a COBOL application is disabled before calling COB_SETRUNIT(). Or, an error occurs as "-2" is returned.

- Process identification (PID=) and thread identification (TID=) in COUNT information are process identification and thread identification when the COUNT function outputs information. Please refer to "COUNT Information" in "NetCOBOL Debugging Guide" on debugging function for details at the output time.
- The restrictions on the uses of these subroutines are as follows:
  - Screen handling function: The screen handling function cannot be used.
  - ACCEPT/DISPLAY function: The COBOL console window cannot be used in function-name CONSOLE, SYSIN, and SYSOUT. In this case, the system console window or file must be used as the input/output destination of data.
  - Presentation file function: The presentation file cannot be relayed between the threads. Operation must be done in the same thread from an OPEN statement to a CLOSE statement.
  - SymfoWARE linkage with use of pre-compiler: The connection and the cursor cannot be relayed between the threads. Operation must be done in the same thread from connecting to disconnecting and from an open of the cursor to a close of the cursor.

19.6.4.5 Compile and Link

Compile

Special specification is not necessary.

Link

When using the COB_GETRUNIT subroutine and COB_SETRUNIT subroutine, F4AGCIMP.LIB must be linked at the time of linking the module of the caller.

19.7 Procedure: From Compilation to Execution

19.7.1 Compilation and Link

This subsection explains how to compile and link the multithread program.

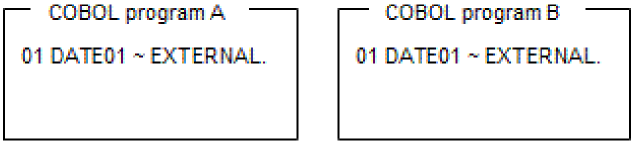
The multithread program differs from the single-thread program in that it requires the compiler option THREAD (MULTI). To use the external data shared between threads or external file shared between threads, the compiler option SHREXT is required in addition to THREAD (MULTI).

The link procedure for a multithread program is the same as that for a single-thread program.

For information on the compilation and link methods, refer to "Chapter 3 Compiling Programs" and "Chapter 4 Linking Programs".

19.7.1.1 Creating a DLL containing only COBOL Programs

COB.DLL is created by COBOL programs A and B.



```
01 DATE01 ~ EXTERNAL.
```

```
01 DATE01 ~ EXTERNAL.
```

19.7.1.1.1 Compilation

- Compiling COBOL program A

```
COBOL -WC,"THREAD(MULTI),SHREXT" A.COB
```

- Compiling COBOL program B

```
COBOL -WC,"THREAD(MULTI),SHREXT" B.COB
```

Note

To use the external data shared between threads or external file shared between threads, specify the compiler option SHREXT in addition to THREAD (MULTI).

19.7.1.1.2 Link

For how to link objects, see "DLL Entry Object Linking" in "5.3.2.3.4 Using the runtime initialization file under DLL".

- Creating COB.DLL by linking COBOL programs A and B

```
LINK A.OBJ B.OBJ F4AGCBDM.OBJ F4AGCIMP.LIB KERNEL32.LIB /DLL  /ENTRY:COBDMAIN /OUT:COB.DLL
```

See

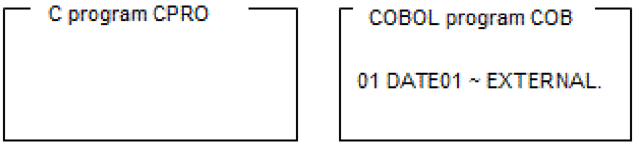
A.OBJ	COBOL program A object file
B.OBJ	COBOL program B object file
F4AGCBDM.OBJ	DLL entry object
F4AGCIMP.LIB	COBOL runtime system import library
KERNEL32.LIB	Windpws API import library

Note

Be sure to specify COBDMAIN in the LINK option/ENTRY.

19.7.1.2 Creating DLL with the COBOL Program and C Program

CCOB.DLL is created with C program CPRO and COBOL program COB.



19.7.1.2.1 Compilation

- Compiling C program CPRO

```
CL -c -MD CPRO.C
```

- Compiling COBOL program COB

```
COBOL -WC,"THREAD(MULTI),SHREXT" COB.COB
```

Note

- Compile the C program so that it runs under the multithread environment. (The compiler option /MD is required for Visual C++ (R)).

- To enable the COBOL program to use the external data shared between threads or the external file shared between threads, specify the compiler option SHREXT in addition to THREAD (MULTI).

19.7.1.2.2 Link

For how to link objects, see "DLL Entry Object Linking" in "5.3.2.3.4 Using the runtime initialization file under DLL".

- Creating CCOB.DLL by linking C program CPRO and COBOL program COB

```
LINK CPRO.OBJ COB.OBJ F4AGMLDM.OBJ F4AGCIMP.LIB KERNEL32.LIB MSVCRT.LIB /DLL /OUT:CCOB.DLL
```

See

CPRO.OBJ	Object file of C program CPRO
COB.COB	Object file of COBOL program COB
F4AGMLDM.OBJ	DLL entry object
F4AGCIMP.LIB	COBOL runtime system import library
KERNEL32.LIB	Windows API import library
MSVCRT.LIB	C runtime library. (This library is required when the program is compiled with the compiler option /MD)

19.7.2 Execution

This subsection explains the procedure for executing the multithread program.

19.7.2.1 Setting Runtime Environment Information

This subsection explains the procedure for setting execution environment information. For the explanations that overlap those for the single-thread mode, refer to "Chapter 5 Executing Programs".

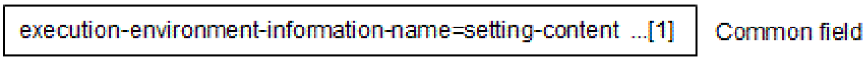
Content of the initialization file

The following explains the specified content of the initial file for execution that is used by a COBOL application that runs in the multithread mode.

In the multithread mode, the content of the initialization file consists only of the common field.

Information specified in the section is ignored.

The following shows the coding format of the content of the initial file for execution:



```
execution-environment-information-name=setting-content ...[1]
```

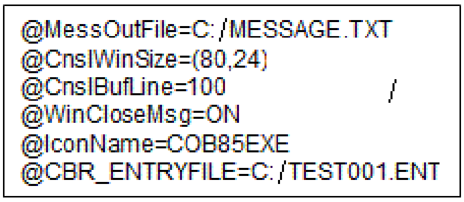
Explanation of diagram:

[1] Environment variable information (common field)

In this field, specify the environment variable information common to each program. For information on the specification format of each piece of environment variable information, refer to "5.4.1 Environment Variables". Two or more pieces of environment variable information cannot be specified in a single line. The runtime environment information specified here is reflected in the environment variable of the application. Accordingly, the information is valid until the application ends.

The content specified here is valid for any thread because it is shared within a process.

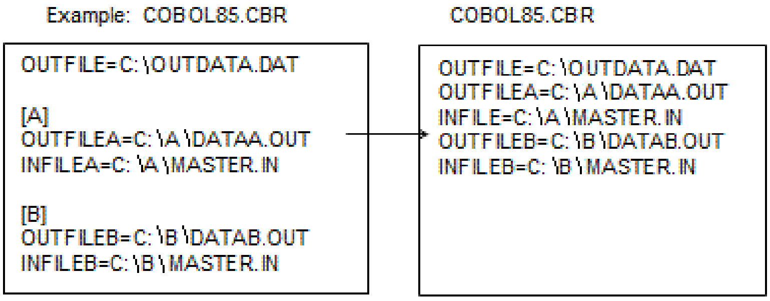
Example of coding the initialization file



```
@MessOutFile=C:/MESSAGE.TXT
@CnslWinSize=(80,24)
@CnslBufLine=100
@WinCloseMsg=ON
@IconName=COB85EXE
@CBR_ENTRYFILE=C:/TEST001.ENT
```

The following must be noted to transfer the single-thread application to the multithread environment:

- Specify the information coded in the section, in the common field.



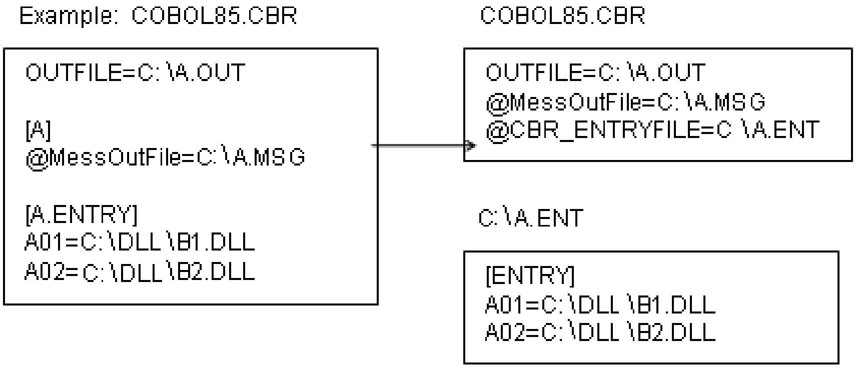
Example: COBOL85.CBR

```
OUTFILE=C:\OUTDATA.DAT

[A]
OUTFILEA=C:\A\DATAA.OUT
INFILEA=C:\A\MASTER.IN

[B]
OUTFILEB=C:\B\DATAB.OUT
INFILEB=C:\B\MASTER.IN
```

COBOL85.CBR

```
OUTFILE=C:\OUTDATA.DAT
OUTFILEA=C:\A\DATAA.OUT
INFILE=C:\A\MASTER.IN
OUTFILEB=C:\B\DATAB.OUT
INFILEB=C:\B\MASTER.IN
```

- When the entry data is coded, specify the entry data file, and then specify the @CBR_ENTRYFILE of the common field as the file name.



Example: COBOL85.CBR

```
OUTFILE=C:\A.OUT

[A]
@MessOutFile=C:\A.MSG

[A.ENTRY]
A01=C:\DLL\B1.DLL
A02=C:\DLL\B2.DLL
```

COBOL85.CBR

```
OUTFILE=C:\A.OUT
@MessOutFile=C:\A.MSG
@CBR_ENTRYFILE=C \A.ENT
```

C:\A.ENT

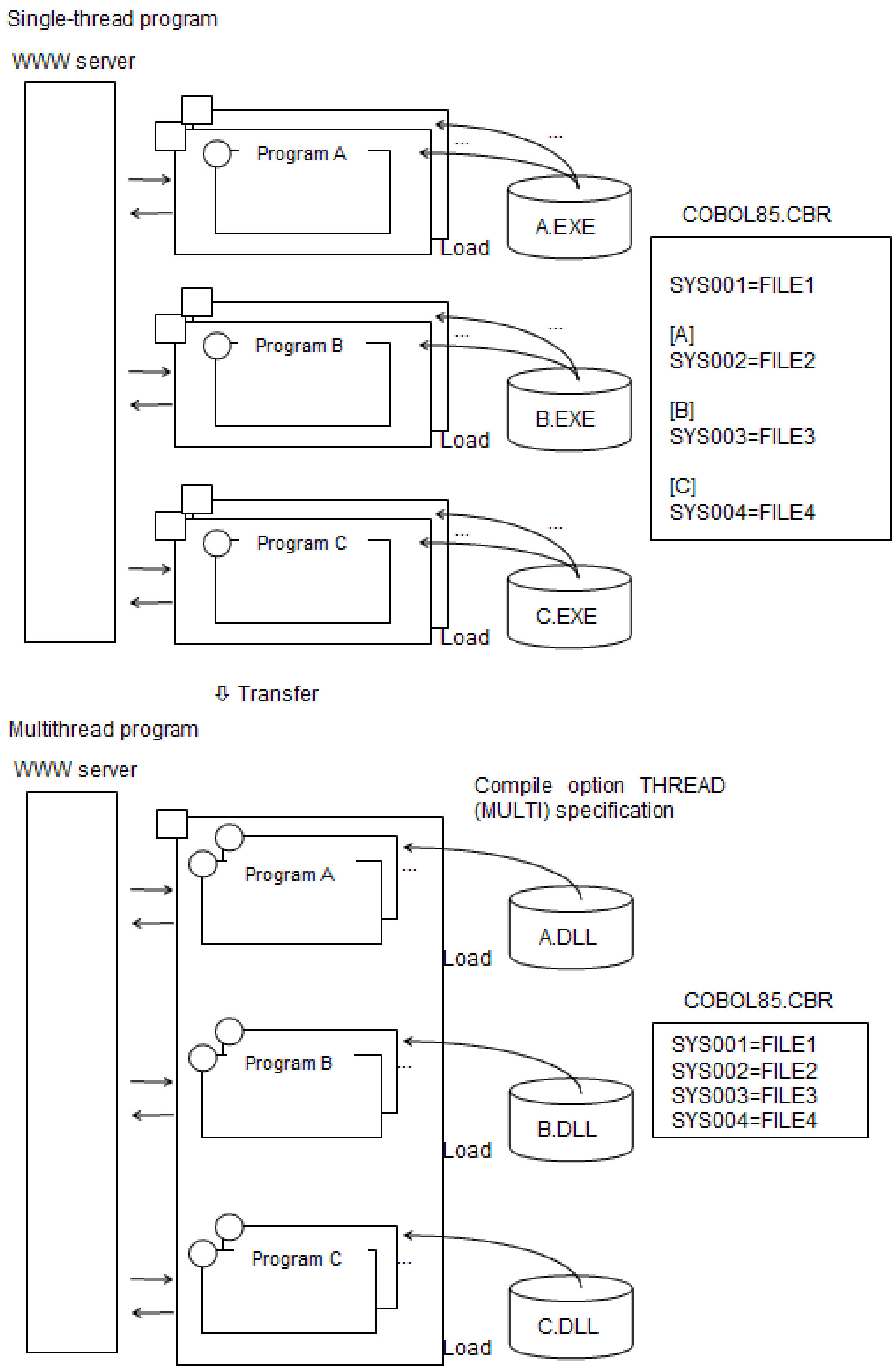
```
[ENTRY]
A01=C:\DLL\B1.DLL
A02=C:\DLL\B2.DLL
```

- Under the server environment, if a relative path is specified as the environment variable for the file name, the relative point from the origin may vary depending on the server. In such case, if it is possible to make specifications using an absolute path, specify the file name with an absolute path.

The following table summarizes the thread modes and the valid descriptions within the initialization file.

	Single-thread mode	Multithread mode
Section	Valid	Invalid
Common field	Valid	Valid

The following figure shows an example of transferring single-thread programs A, B, and C used by the WWW server to multithread programs.



Using COBOL85.CBR in the DLL storage position in the server environment

A single initialization file is used in a process.

Different pieces of execution environment information can be used for each process by executing two or more COBOL programs (DLL) in different processes from the server control program such as IIS.

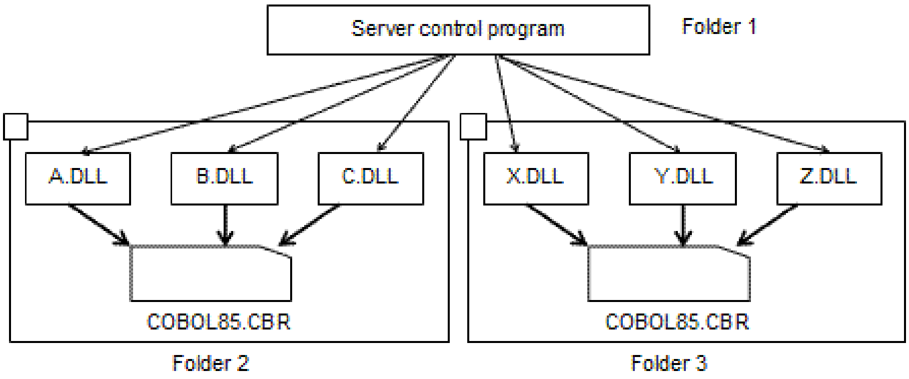
In this case, store the DLL of the COBOL program in a single folder for each application that runs in the same process, then specify the required information in the common field.

Note that, if all DLLs and COBOL programs that run as an application in a single execution environment are not in a single folder, the folder from which the COBOL85.CBR file is to be read may not be correctly identified.

To read the COBOL85.CBR file from the DLL storage position, the DLL entry object must be linked to DLL. Refer to "5.3.2.3.4 Using the runtime initialization file under DLL".

Example: Using the COBOL85.CBR file of the same folder for each process

In the following example, folders 1 to 3 are different from one another.



19.7.2.1.1 Format for Specifying Runtime Environment Information

This subsection explains the environment variable information that is valid only in the multithread mode.

@CBR_SYMFOWARE_THREAD(Specification enabling the multithread operation using linkage with SymfoWARE)

```
@CBR_SYMFOWARE_THREAD=MULTI
```

A multithread program linking with SymfoWARE using the precompiler is made operable. Refer to "19.5.3 SymfoWARE Linkage with Use of Precompiler".

@CBR_THREAD_MODE(Specifying the thread mode)

```
@CBR_THREAD_MODE=SINGLE
```

A program compiled by the multithread compiler option is run in single-thread mode, not in multithread mode. If this specification is omitted, the program runs in the mode specified by the compiler option.

This environment variable is invalidated even if it is specified in the initialization file. Specify it directly for the environment variable. Refer to "19.3.3 Program Execution and Thread Mode".

@CBR_THREAD_TIMEOUT(Specifying the wait time of the thread synchronization control subroutine)

```
@CBR_THREAD_TIMEOUT=wait-time (s)
```

If an infinite wait is specified for the thread synchronization control subroutine, specify this format to change the wait time. Specify the wait time as a number from 0 to up to 32 digits (s). If the wait time is omitted, an infinite wait is used. Refer to "19.9 Thread Synchronization Control Subroutine".

@CBR_SSIN_FILE(Specifying that opens input file of each thread)

```
@CBR_SSIN_FILE=THREAD
```

The input file of each thread is opened. Refer to "10.1.6.5 File Input Extension Function for the ACCEPT Statement".

19.8 Debugging the Multithread Program

The debug method is generally the same for the following functions provided by COBOL even if a multithread program is used:

- TRACE function
- CHECK function
- COUNT function
- Debugging function of NetCOBOL Studio

However, some notes must be observed for operations specific to a multithread program.

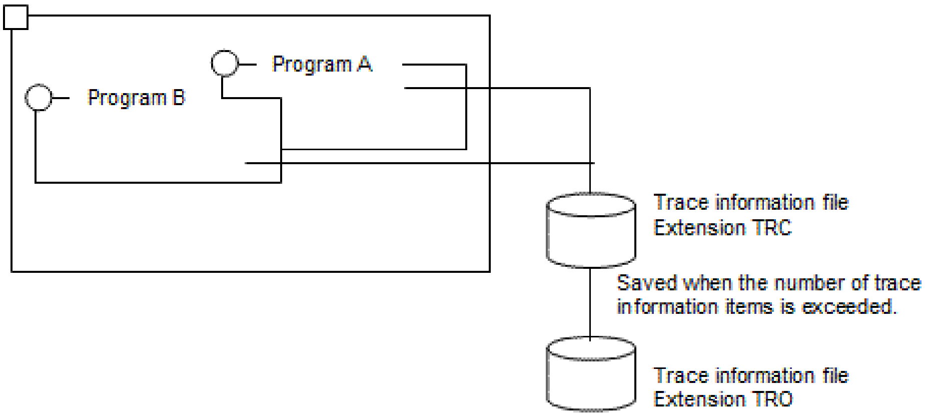
The notes to be observed are explained below for each function.

19.8.1 TRACE Function

The content of the trace information remains unchanged. Refer to "Using the TRACE Function" in the "NetCOBOL Debugging Guide".

However, trace information collected from each thread is stored in a single file (extension TRC).

The following figure shows the TRACE function:



The following figure explains how to read trace information:

```
NetCOBOL DEBUG INFORMATION   DATE 2007-07-05  TIME 20:29:52  PID=00000125

TRACE INFORMATION

 [1]     1  A    DATE 2007-07-05  TIME 20:27:50  TID=0000010C
         2              22.1 TID=0000010C
         3              23.1 TID=0000010C
 [1]     4  B    DATE 2007-07-05  TIME 20:27:50  TID=0000011E
         5              22.1 TID=0000011E
         6              23.1 TID=0000011E
         7              24.1 TID=0000011E
         8              25.1 TID=0000011E
 [2]'    9              26.1 TID=0000011E
        10              24.1 TID=0000010C
        11              25.1 TID=0000010C
        12              26.1 TID=0000010C
 [2]    13    THE INTERRUPTION WAS OCCURRED.PID=00000125,TID=0000011E
        14              27.1 TID=0000010C
        15              28.1 TID=0000010C
 [3]    16      EXIT-THREAD TID=0000010C
```

Explanation of diagram:

- [1] Thread ID assigned to the program

  The thread of program A is 10C.

  The thread of program B is 11E.

- [2] Exception notification message

  The exception occurred for program B whose thread ID is 11E.

- [3] Thread end notification message

  The thread ID ended for program A whose thread ID is 10C.

The following can be obtained from the above results:

- Program A whose thread ID is 10C run normally.
- Program B whose thread ID is 11E caused an exception in the execution statement in line 26.

Information

Using DISPLAY...UPON SYSERR enables any data to be output to trace information.

This function is useful in checking the transition of data used by the program.

To use DISPLAY...UPON SYSERR, specify YES in environment variable @CBR_SYSERR_EXTEND so that the thread ID is also output. Refer to "5.4.1.43 @CBR_SYSERR_EXTEND(Specify the SYSERR output information extension)".

Note

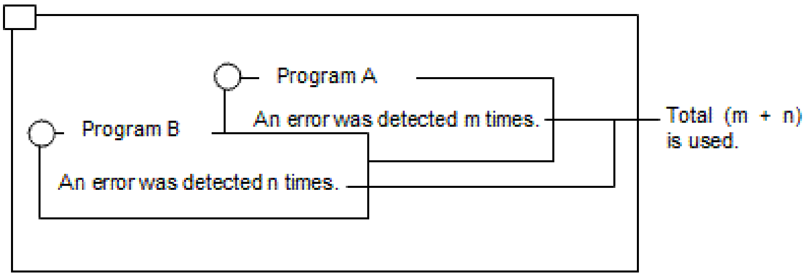
When many threads are executed at one time, a large amount of trace information is written in a single file. Adjust the number of trace information items (specification of "r" in environment variable @GOPT) so that much more trace information is output to a single file.

19.8.2 CHECK Function

When the CHECK function is valid, the content of the error message to be output and its detection method remain unchanged. Refer to "Using the CHECK Function" in the "NetCOBOL Debugging Guide".

However, the total number of message outputs detected within a process is used as the number of message outputs.

The following figure shows the CHECK function:


Program A
An error was detected m times.
Program B
An error was detected n times.
Total (m + n)
is used.

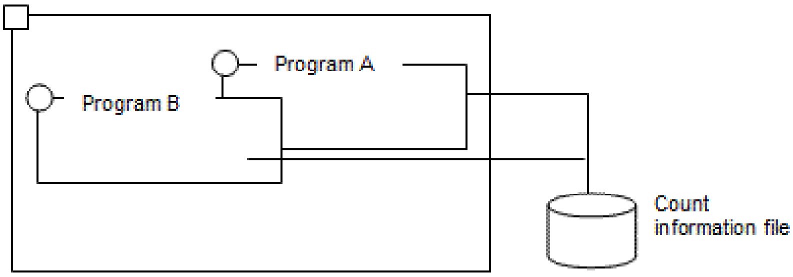
19.8.3 COUNT Function

The content of the count information remains unchanged. Refer to "Using the COUNT Function" in the "NetCOBOL Debugging Guide".

However, the following operations occur:

- Count information collected from each thread is stored in a single file.
- The total results in which count information is written are output for each thread. No total is made for the entire process.

The following figure shows the COUNT function.



The following figure shows how to read the count information to be output.

```
NetCOBOL COUNT INFORMATION(END OF RUN UNIT)  DATE 2007-07-05 TIME 21:38:59
                                          PID=000000B5  TID=000000D4
                                           [1]

  STATEMENT EXECUTION COUNT    PROGRAM-NAME : A

  :

COBOL COUNT INFORMATION(END OF RUN UNIT) DATE 2007-07-05 TIME 21:38:59
                                          PID=000000B5  TID=0000012A
                                           [1]

  STATEMENT EXECUTION COUNT    PROGRAM-NAME : B

  :
```

Explanation of diagram:

- [1] Thread ID assigned to the program

  The thread ID of program A is 0D4.

  The thread ID of program B is 12A.

19.8.4 Debugging function of NetCOBOL Studio

For details, refer to "NetCOBOL Studio User's Guide".

19.9 Thread Synchronization Control Subroutine

This section explains the subroutine for performing thread synchronization control. The thread synchronization control subroutine is divided into data lock subroutines and object lock subroutines.

19.9.1 Data Lock Subroutines

Subroutine name	Function
COB_LOCK_DATE	Acquires the lock for the lock key.
COB_UNLOCK_DATA	Releases the lock for the lock key.

When synchronization control is required between threads within the same process, a mutually exclusive lock can be provided by using the data lock subroutines. In this case, the data lock subroutines mutually acquire and release the lock for the lock key having the same data name. The data name to be specified here must be unique in the process.

When COB_LOCK_DATA is called, the lock key associated with the data name specified by the parameter is created to acquire the lock. If the lock key associated with the specified data name already exists, the lock is acquired for the lock key.

Only one thread can acquire the lock, and the thread that acquired the lock is executed. Other threads that attempted to acquire the lock for the same key must wait for the thread having the lock to release the lock.

The lock is released by calling COB_UNLOCK_DATA.

Calling by using the dynamic program structure

To call this subroutine with the dynamic program structure, the entry information shown below is required. For information on the method of specifying entry information, refer to "5.4.2 Entry Information for Subprograms".

```
[ENTRY]
COB_LOCK_DATA=F4AGEFNC.DLL
COB_UNLOCK_DATA=F4AGEFNC.DLL
```

19.9.1.1 COB_LOCK_DATA

Function

Acquires the lock for the lock key associated with the specified data name.

Format

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
  01  LOCK-KEY        PIC X(30).
  01  WAIT-TIME       PIC S9(9) COMP-5.
  01  ERR-DETAIL      PIC 9(9)  COMP-5.
  01  RET-VALUE       PIC S9(9) COMP-5.
*>    :
 PROCEDURE DIVISION.
*>    :
     CALL   "COB_LOCK_DATA"
                             USING   BY REFERENCE LOCK-KEY
                                     BY VALUE WAIT-TIME
                                     BY REFERENCE ERR-DETAIL
                             RETURNING  RET-VALUE.
```

Parameters

LOCK-KEY

Specifies the name of the lock key for which the lock is acquired, with up to 30 bytes. If the lock key name is less than 30 bytes, a space must be inserted at the end.

WAIT-TIME

Specifies the wait time (s) to be used until the lock is acquired. If -1 is specified, an infinite wait is set.

If the infinite wait is specified, a wait time can be changed by specifying the wait time (s) in environment variable @CBR_THREAD_TIMEOUT. This function can be used to specify a location where a deadlock occurred.

ERR-DETAIL

If a return value is -255, a Windows system error code is returned.

Return value

RET-VALUE

When the operation is successful, 0 is returned. When the thread mode is single thread, no lock is required. Accordingly, 1 is returned without acquiring the lock. There is no problem with the operation.

When the operation is unsuccessful, a negative value is returned. For details, refer to "19.9.3 Error Codes".

19.9.1.2 COB_UNLOCK_DATA

Function

Releases the lock for the lock key associated with the specified data name.

Format

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
  01  LOCK-KEY        PIC X(30).
  01  ERR-DETAIL      PIC 9(9)  COMP-5.
  01  RET-VALUE       PIC S9(9) COMP-5.
*>    :
 PROCEDURE DIVISION.
*>    :
     CALL   "COB_UNLOCK_DATA"
                             USING  BY REFERENCE LOCK-KEY
                                    BY REFERENCE ERR-DETAIL
                             RETURNING  RET-VALUE.
```

Parameters

LOCK-KEY

Specifies the name of the lock key for which the lock is acquired, up to 30 bytes. If the lock key name is less than 30 bytes, a space must be inserted at the end.

ERR-DETAIL

If a return value is -255, a Windows system error code is returned.

Return value

RET-VALUE

When the operation is successful, 0 is returned. When the thread mode is single thread, no lock is required. Accordingly, 1 is returned without releasing the lock. There is no problem with the operation.

When the operation is unsuccessful, a negative value is returned. For details, refer to "19.9.3 Error Codes".

19.9.2 Object Lock Subroutines

Subroutine name	Function
COB_LOCK_OBJECT	Acquires the lock for the object.
COB_UNLOCK_OBJECT	Releases the lock for the object.

When an object is shared among threads within the same process, a mutually exclusive lock can be provided by using these subroutines. In this case, these subroutines acquire and release the lock for the same object.

When COB_LOCK_OBJECT is called, the lock is acquired for an object by specifying this object.

Only one thread can acquire the lock, only the thread that acquired the lock can use the object. Other threads that attempt to acquire the lock for the same object have to wait for the thread having the lock to release the lock.

The lock is released by calling COB_UNLOCK_OBJECT.

Calling by using the dynamic program structure

To call this subroutine with the dynamic program structure, the entry information shown below is required. For information on the method of specifying entry information, refer to "5.4.2 Entry Information for Subprograms".

```
[ENTRY]
COB_LOCK_OBJECT=F4AGEFNC.DLL
COB_UNLOCK_OBJECT=F4AGEFNC.DLL
```

19.9.2.1 COB_LOCK_OBJECT

Function

Acquires the lock for the specified object.

Format

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
  01  OBJ              OBJECT REFERENCE class-name.
  01  WAIT-TIME        PIC S9(9) COMP-5.
  01  ERR-DETAIL       PIC 9(9)  COMP-5.
  01  RET-VALUE        PIC S9(9) COMP-5.
*>    :
 PROCEDURE DIVISION.
*>    :
     CALL   "COB_LOCK_OBJECT"
                               USING  BY REFERENCE OBJ
                                      BY VALUE WAIT-TIME
                                      BY REFERENCE ERR-DETAIL
                               RETURNING  RET-VALUE.
```

Parameters

OBJ

Specifies the object reference of the object for which the lock is acquired.

WAIT-TIME

Specifies the wait time (s) used until the lock is acquired. If -1 is specified, an infinite wait occurs.

If an infinite wait is specified, a wait time can be changed by specifying the wait time (s) in environment variable @CBR_THREAD_TIMEOUT. This function is used to specify the location where a deadlock occurred.

ERR-DETAIL

If a return value is -255, a Windows system error code is returned.

Return value

RET-VALUE

When the operation is successful, 0 is returned. When the thread mode is single thread, no lock is required. Accordingly, 1 is returned without acquiring the lock. There is no problem with the operation.

When the operation is unsuccessful, a negative value is returned. For details, refer to "19.9.3 Error Codes".

19.9.2.2 COB_UNLOCK_OBJECT

Function

Releases the lock for the specified object.

Format

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
  01  OBJ             OBJECT REFERENCE class-name.
  01  ERR-DETAIL      PIC 9(9)  COMP-5.
  01  RET-VALUE       PIC S9(9) COMP-5.
*>    :
 PROCEDURE DIVISION.
*>    :
     CALL   "COB_UNLOCK_OBJECT"
                                 USING  BY REFERENCE OBJ
                                        BY REFERENCE ERR-DETAIL
                                 RETURNING  RET-VALUE.
```

Parameters

OBJ

Specifies the object reference of the object for which the lock is released.

ERR-DETAIL

If a return value is -255, a Windows system error code is returned.

Return value

RET-VALUE

When the operation is successful, 0 is returned. When the thread mode is single thread, no lock is required. Accordingly, 1 is returned without acquiring the lock. There is no problem with the operation.

When the operation is unsuccessful, a negative value is returned. For details, refer to "19.9.3 Error Codes".

19.9.3 Error Codes

This subsection explains return values for errors generated in the thread synchronization control subroutines.

The symbols in the target subroutine column in the table have the following meanings:

- LD: COB_LOCK_DATA
- UD: COB_UNLOCK_DATA
- LO: COB_LOCK_OBJECT
- UO: COB_UNLOCK_OBJECT

Table 19.1 Error codes

Error code	Meaning and action	Target subroutine			
		LD	UD	LO	UO
- 1	The COBOL execution environment is not opened. Use the subroutines after opening the COBOL execution environment.	Notified	Notified	Notified	Notified
- 2	The parameter specification is incorrect. In other words, the specified lock key name may be incorrect (LD, UD); the specified wait time may be incorrect (LD, LO), or the NULL object may be specified (LO, UO) in the object reference.	Notified	Notified	Notified	Notified
- 3	A program of a thread terminated abnormally; accordingly, no lock was released normally. Remove the cause of the error in the program that terminated abnormally, then re-execute the program.	Notified	-	Notified	-
- 4	A wait time for lock acquisition has passed.	Notified	-	Notified	-
- 5	No lock is acquired, or an attempt was made to release the lock acquired by another thread.	-	Notified	-	Notified
-255	A system error occurred. A Windows system error code is set in the ERR-DETAIL parameter.	Notified	Notified	Notified	Notified

19.10 Notes

19.10.1 OO Programming

For a single-thread program, the timing for ending a run unit is the same as that for closing the execution environment. Accordingly, even if the run unit does not end after the NULL object is specified in the object reference data item, the execution environment is closed at the end of the run unit. In addition, the remaining object instances are released from memory. No problems occur in this case.

However, for the multithread program, the timing for ending the run unit differs from that for closing the execution environment. Accordingly, if the run unit ends without specifying the NULL object in the object reference data item, the object instances that are not referenced from any place remain in the memory, resulting in memory shortage.

Therefore, be sure to end the run unit after specifying the NULL object in the object reference data item. For information on the run unit and execution environment of the multithread program, refer to "19.3.1 Execution Environment and Run Unit".

19.10.2 Print Function

This subsection explains the notes on sharing the print file or presentation file by operating the same file connector among threads.

Assume that a single form is generated by executing two or more input-output statements. In this case, if input-output statements are executed from two or more threads for the same file connector, the input-output statement execution order becomes asynchronous. Accordingly, undesirable print results may be obtained because the print data output order is not constant. It is necessary to suppress contention with other threads for input-output statements between the start of a series of processing for the same form and the end of the processing.

To prevent contention with other threads, thread synchronization control is performed before and after a processing series.

For details of the synchronization control for a thread and a file having the same file connector among threads, refer to "19.6.1 Using the Input-Output Module".

19.10.3 Dynamic structure

When a CANCEL statement is executed for a subprogram called with the dynamic structure note:

The program is returned to its initial state by the CANCEL statement. However, in multi-thread mode the DLL of the program specified in the CANCEL statement is not deleted from virtual memory. This means that the program operates normally even if other threads CANCEL the active program.

Even when the DLL is composed of two or more subprograms, the subprograms that were compiled with the compiler option THREAD(MULTI) are returned to their initial state when they are canceled.

Chapter 20 Unicode

This chapter describes how to create a COBOL application that uses Unicode.

20.1 Character Code

The character code is machine-readable code that identifies a set of actual characters. It maps each character to a unique code point (number).

The character code that can be used by NetCOBOL for Windows is as follows:

- Native code (default)

  It is the code system that expresses one character with 1 byte.

  In NetCOBOL, it can be used when the compilation option RCS(ASCII) is specified implicitly or explicitly.

  The encoding of the data item and the code systems of all resources regard it as ASCII.

- Shift-JIS

  When expressing Japanese, it is the code commonly used.

  In NetCOBOL, it can be used when the compilation option RCS(SJIS) is specified.

  The encoding of the data item and the code systems of all resources regard it as Shift-JIS.

  Shift-JIS can also be used as the encoding of the data item in a Unicode application.

  Note

  - When making Shift-JIS the runtime code, specify the compile option RCS(SJIS). At that time, the system locale during compilation should be Japanese. An error will be generated for languages other than Japanese.
  - In the execution of the Shift-JIS program, specify the compile option RCS(SJIS), please set Windows system locale to Japanese (it means the ANSI code page 932). Operation is not guaranteed, except when the system locale is Japanese.

- Unicode

  Unicode is the character code created by the Unicode consortium for the purpose of expressing every character in the world regardless of language, platform, program, etc.

  The character that can be expressed in neither Native code nor Shift-JIS can be used.

20.2 Overview of Unicode Support

For NetCOBOL, data in different languages can be handled by using Unicode.

20.2.1 Specifying the character encoding

20.2.1.1 Encoding form

The table below lists the classes and encoding forms.

Item level	Class	Encoding form
Elementary item	Alphabetic character	ASCII
	Alphanumeric character	ASCII (UTF-8)
	National character	UTF-16 UTF-32(*1)

Item level	Class	Encoding form
Group item	Alphanumeric character	ASCII (UTF-8)

(*1) UTF-32 is supported in NetCOBOL V11 or later.

In Unicode, there are several possible representations.

In NetCOBOL, the alphanumeric items are encoded as UTF-8, and the national data items are encoded as UTF-16 or UTF-32 characters.

In UTF-8, the region length needed to store one character varies from 1 to 4 bytes.

In UTF-16, the region length needed to store one character is 2 or 4 bytes. Only 2 bytes are required if it is in the BMP range. When a surrogate pair is stored, 4 bytes must be used.

In UTF-32, the region length needed to store one character is 4 bytes.

Additionally in UTF-16 and UTF-32, you are able to select big endian in addition to little endian.

Note

- In Windows Server 2012 and Windows 8, you can use the expression format called IVS (Ideographic Variation Sequence), but you cannot use IVS in the current COBOL compiler and runtime system.

20.2.1.2 Encoding specifications

The encoding of the data item is decided by the ENCODING clause.

```
01  DATA1 PIC X(nn) [ENCODING IS alphabet-name1].
01  DATA2 PIC N(nn) [ENCODING IS alphabet-name2].
```

In the ENCODING clause, alphabet-name is specified.

Alphabet name represents the encoding. In NetCOBOL, alphabet-name can be defined for the following encodings.

Character Encoding	Class	Encoding	Remarks
Shift-JIS	Alphanumeric	SJIS	Shift-JIS
	National	SJIS	Shift-JIS
Unicode	Alphanumeric	UTF8	UTF-8
	National	UTF16	UTF-16 Little Endian
		UTF16BE	UTF-16 Big Endian
		UTF16LE	UTF-16 Little Endian
		UTF32	UTF-32 Little Endian
		UTF32BE	UTF-32 Big Endian
		UTF32LE	UTF-32 Little Endian

The encoding format UTF8, UTF16, UTF16LE, UTF16BE, UTF32, UTF32LE and UTF32BE are in accordance with international standards ISO/IEC 10646 regulations.

For example, in a Unicode application, when the alphanumeric data item is UTF-8 and the national item is created by mixing UTF-16LE and UTF-32LE, then encoding is defined in the following manner.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
   ALPHABET
      SJ FOR ALPHANUMERIC IS SJIS
      U8 FOR ALPHANUMERIC IS UTF8  [1]
      U16L FOR NATIONAL IS UTF16LE [2]
      U32L FOR NATIONAL IS UTF32LE [3]
      .
WORKING-STORAGE SECTION.
01 DATA1 PIC X(10) ENCODING IS U8.   *> UTF8
01 DATA2 PIC N(10) ENCODING IS U16L. *> UTF16LE
01 DATA3 PIC N(10) ENCODING IS U32L. *> UTF32LE
```

[1] The alphabet-name "U8" is defined for encoding UTF8.

[2] The alphabet-name "U16L" is defined for encoding UTF16LE.

[3] The alphabet-name "U32L" is defined for encoding UTF32LE.

The ENCODING clause can be specified in the file entry and the group item also.

In this case, it becomes enabled in items where the ENCODING clause is not explicitly specified.

```
01 A ENCODING IS U8 OR U32L.
  02 A1 PIC X(10).                  *> UTF8
  02 A2 PIC N(10) ENCODING IS U16L. *> UTF16LE
  02 A3 PIC N(10).                  *> UTF32LE
```

Various encodings are specified in the following order:

1. ENCODING specification of the basic item
2. ENCODING specification of group item
3. ENCODING specification of the file entry

And, their priorities are in the order of 1, 2, and 3.

Note

You will not be able to use a mixture of Unicode and Shift-JIS within a compile unit as shown below.

```
01 A.
  02 A1 PIC X(10) ENCODING IS SJ.
  02 A2 PIC N(10) ENCODING IS U16L.
```

20.2.1.3 Encoding specifications by the compilation options

As mentioned earlier, the ENCODING clause can be omitted. When the ENCODING clause is omitted, then the encoding specified with the compile option ENCODE is enabled. For details of compilation option ENCODE, refer to A.2.13 ENCODE (encoding form specification) .

When the ENCODING clause is specified and the compilation option is omitted, then the encoding of the data item follows the compilation option RCS. It assumes that there is a compatibility with the old version.

ENCODE(SJIS,SJIS):

```
01 A.
  02 A1 PIC X(10). *> Shift-JIS
  02 A2 PIC N(10). *> Shift-JIS
```

ENCODE(UTF8,UTF16):

```
01 A.
  02 A1 PIC X(10). *> UTF-8
  02 A2 PIC N(10). *> UTF-16LE
```

ENCODE(UTF8,UTF32)

```
01 A.
  02 A1 PIC X(10). *> UTF-8
  02 A2 PIC N(10). *> UTF32LE
```

The priority of the ENCODING clause is higher than the compilation option ENCODE.

Note

If the compilation option ENCODE is omitted, the compilation option RCS is used.
See "A.2.37 RCS(runtime code set specification) for details.

- In versions prior to NetCOBOL V11, the RCS application's runtime code and data item encoding was specified by the compilation option. In V11 and forward use compilation option ENCODE for compatibility.

20.2.1.4 Runtime code

Runtime code is an attribute of the object program created by NetCOBOL and is the factor that determines the code of run-time resources, which are described in "19.1.1.5 Resources". The runtime code is decided in accordance with the specification of the compilation option ENCODE.

The relation of compilation option ENCODE and runtime code is given below.

ENCODE	Encoding		Runtime code
	Alphanumeric	National	
ENCODE(SJIS,SJIS)	Shift-JIS	Shift-JIS	Unicode(*)
ENCODE(UTF8,UTF16[,LE/BE])	UTF-8	UTF-16 LE/BE	Unicode
ENCODE(UTF8,UTF32[,LE/BE])	UTF-8	UTF-32 LE/BE	Unicode

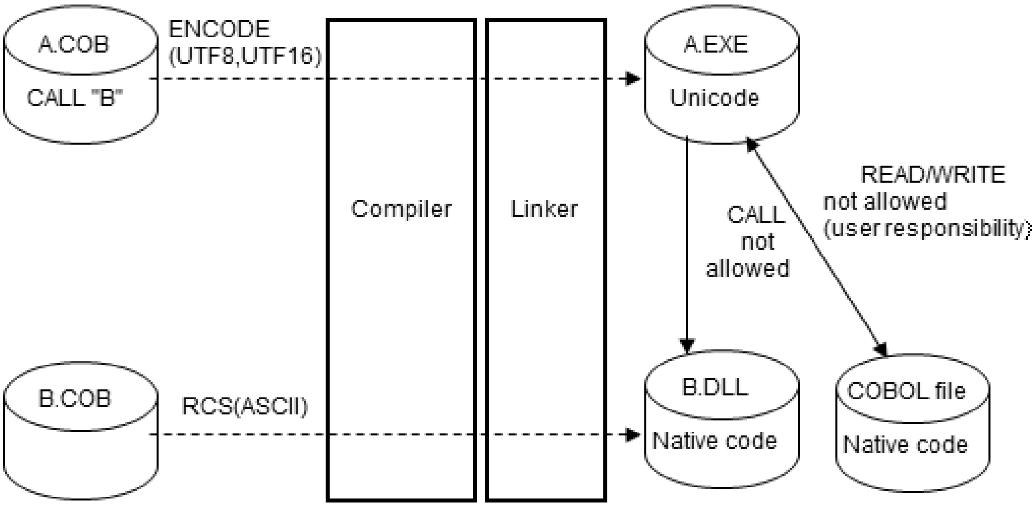
(*)The compilation option RCS(ASCII) is specified explicitly, runtime code is ACP. And, RCS(SJIS) is specified explicitly, runtime code is Shift-JIS.

Note

If the compilation option ENCODE is omitted, the compilation option RCS is used.

See "A.2.37 RCS(runtime code set specification) for details.

Additionally, in NetCOBOL, runtime codes cannot be mixed in one execution unit. For example, as shown in the image below, a run time error is generated when a DLL with a different runtime code is called.



20.2.2 Resources

When Unicode data is processed by NetCOBOL, the translation resource of the source program, etc. can be made with native code or Unicode (UTF-8).

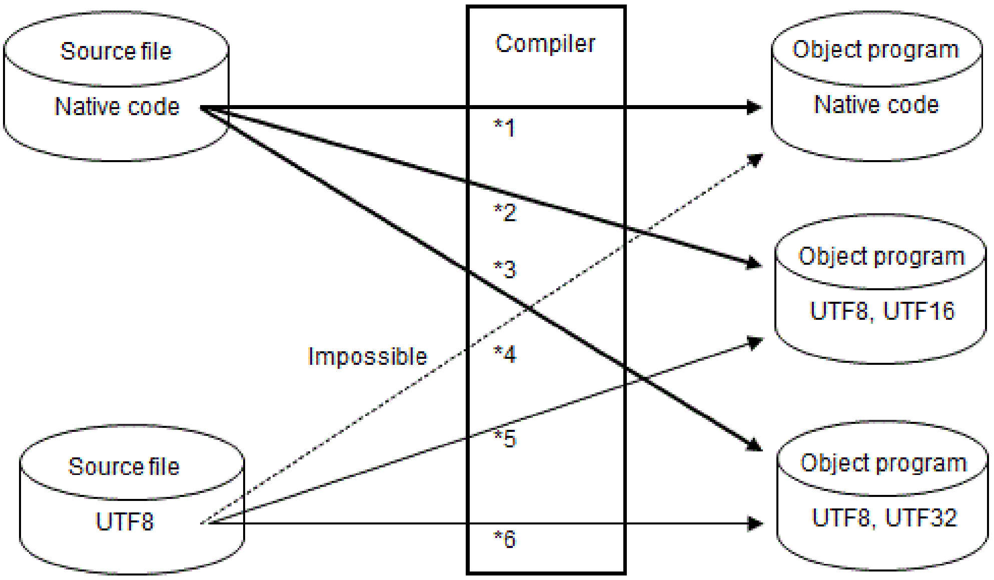
Executing using translation option SCS specifies the code system of the translation resource via the ENCODE option. See the possible combinations in the table and figure below. The default value is Native code.

	ENCODE(SJIS,SJIS)	ENCODE(UTF-8,UTF16)	ENCODE(UTF-8,UTF32)
SCS(ACP)	Can combine	Can combine	Can combine
SCS(SJIS)	Can combine	Can combine	Can combine
SCS(UTF8)	Cannot combine	Can combine	Can combine

Note

If the compilation option ENCODE is omitted, the compilation option RCS is used.

	RCS(ASCII)	RCS(SJIS)	RCS(UTF16)
SCS(ACP)	Can Combine (default)	Can Combine	Can combine
SCS(SJIS)	Can Combine	Can Combine	Can combine
SCS(UTF8)	Cannot combine	Cannot combine	Can combine



*1: The compilation option is not required.

*2: The compilation options SCS(ACP),ENCODE(UTF8,UTF16[,LE/BE]) are required.

*3: The compilation options SCS(ACP),ENCODE(UTF8,UTF32[,LE/BE]) are required.

*4: The compilation options SCS(UTF8),ENCODE(SJIS,SJIS) cannot be specified. A compiler error occurs.

*5: The compilation options SCS(UTF8),ENCODE(UTF8,UTF16[,LE/BE]) are required.

*6: The compilation options SCS(UTF8),ENCODE(UTF8,UTF32[,LE/BE]) are required.

The table below lists the code systems of resources when creating a Unicode application.

	Resource	IN/OUT	Code system
At Compilation	Compilation list	OUT	Native code or Unicode (UTF-8)(*1)
At link	Module definition file	IN	Native code or Unicode (UTF-8)
At execution	Runtime initialization file Entry information file Class information file ODBC information file Print information file Printer information file	IN	Native code or Unicode (UTF-8) (*2)
	Window information file	IN	Native code
	Key definition file	IN	Unicode (UTF-8)
	COBOL file (sequential/line sequential/ relative/indexed)	IN/OUT	Unicode (UTF-8, UTF-16 or UTF-32) (*3)
	ACCEPT/DISPLAY file	IN/OUT	Unicode (UTF-8)

	Resource	IN/OUT	Code system
	TRACE information file	OUT	(*5)
	Message Output file COUNT information file	OUT	Native code or Unicode (UTF-8) (*4)(*5)

*1: The compiler option "A.2.42 SCS(code system of source file)" specifies the code system.

*2: The code is identified by the presence of a BOM. Both codes can be in the same run unit.

*3: The record definition class defines the code system.

*4: If you want to append data to an existing file, it is in the code system of the existing file. If you want to output to a new file, it is in UTF-8. See "5.4.1.56 @MessOutFile(Set a Message Output File) for details.

*5: You can change the code system of input/output files. See "5.4.1.5 @CBR_CODE_SET(Specify the code-set for the output file)

Information

BOM (Byte Order Mark)

BOM is a Unicode character conventionally used as a marker to indicate that text is encoded in UTF-8 and UTF-16. Many Windows applications add a BOM to the Unicode text file. The NetCOBOL compiler and the runtime system assume the use of the BOM character. UNIX applications do not use the BOM character. The source file and the library to be managed in the distribution development environment, etc., can be input even if there is no BOM.

20.2.3 Language element

The language elements related to the character encoding are explained in this section.

In the program example for this section, the following alphabet-name is taken as declared in the environment section.

```
ALPHABET
   U8 FOR ALPHANUMERIC IS UTF8
   U16L FOR NATIONAL IS UTF16LE
   U16B FOR NATIONAL IS UTF16BE
   U32L FOR NATIONAL IS UTF32LE
   U32B FOR NATIONAL IS UTF32BE.
```

Comparison between different encoding

In accordance with "Comparison rules" defined in the COBOL Language Reference, the comparison of data items with different encodings is checked at compile time.

```
WORKING-STORAGE SECTION.
77 DATA1 PIC N(10) VALUE SPACE ENCODING IS U16L. *> UTF-16LE
77 DATA2 PIC N(10) VALUE SPACE ENCODING IS U32L. *> UTF-32LE
*>     :
    IF DATA1 = DATA2 THEN DISPLAY "OK!!".        *> Compiler error
```

In such cases, match the encoding of the comparison target items by using a data item for work. Further, when moving the value in the data item for work, use the MOVE statement (Format 3) described later.

```
77 TEMP PIC N(10) VALUE SPACE ENCODING IS U32L.
*>     :
    MOVE CONV DATA1 TO TEMP.                *> UTF-16LE -> UTF-32LE
    IF TEMP = DATA2 THEN DISPLAY "OK!!".    *> Comparison of UTF-32LE each other
```

Move between different encoding

In accordance with "Move rules" defined in the COBOL Language Reference, the moving of data items having different encoding is checked at compile time.

```
WORKING-STORAGE SECTION.
77 DATA1 PIC N(10) VALUE SPACE ENCODING IS U16L. *> UTF-16LE
77 DATA2 PIC N(10) VALUE SPACE ENCODING IS U32L. *> UTF-32LE
*>    :
    MOVE DATA1 TO DATA2.                         *> Compiler error
```

In such cases, the MOVE STATEMENT (Format 3) is used. The MOVE statement (format 3) changes the sending side items to the encoding form of the receiving side items and performs the move.

```
MOVE CONV DATA1 TO DATA2.
  *> Changing encoding of DATA1 to UTF-32LE, and moving to DATA2
```

A special register CONV-STATUS, CONV-SIZE is prepared, so that it can be decided in the program whether the encoding conversion was successful and whether converted data does not exceed the length of the receiving item.

An error is received in the special register CONV-STATUS when inappropriate characters exist in the source conversion data or if data differs from the source encoding.

In the MOVE statement (Format 3), it is possible to move between the varying character types (alphanumeric character and national language). For details, please refer to "COBOL Language Reference 6.4.28 MOVE STATEMENT".

Class condition

In UTF-16 adopted for Japanese expressions, two digits are necessary for surrogate pairs. The fraction system does not work exactly correct, since the current design accommodates one digit per character. However, characters that correspond to a surrogate pair are only about 300 characters among the JIS2004 addition characters that exceed 4000 characters.

In NetCOBOL, the following two class condition can be used to determine the range of characters stored in the data item.

Class condition	Condition
BMP	True when the character contents of the data item are defined BMP of ISO / IEC 10646-1 in (Basic Multilingual Plane).
UNICODE1	True when the character contents of the data item are defined in UNICODE1.1 (ISO / IEC 10646-1:1993).

The property can be operated by inspecting at the point where JIS2004 is incorporated.

```
 WORKING-STORAGE SECTION.
 01 PIC-N  PIC N(10).
 PROCEDURE DIVISION.
*>     :
     IF PIC-N IS NOT BMP THEN
       DISPLAY "It contains a surrogate pair."
     END-IF.
```

This class condition BMP can be used to confirm whether "Unjustified character division" is generated when the stored number of characters is counted, reference modification, or transcription.

An example of counting the number of characters is as follows.

```
 WORKING-STORAGE SECTION.
 01 PIC-N   PIC N(10).
 01 CHAR-NO PIC S9(4) BINARY VALUE ZERO.
 01 CNT     PIC 9(2).
PROCEDURE DIVISION.
*>     :
    PERFORM VARYING CNT FROM 1 BY 1 UNTIL CNT > 10
      IF PIC-N(CNT:1) IS NOT BMP THEN
          ADD 1 TO CNT
      END-IF
      ADD 1 TO CHAR-NO
    END-PERFORM.
```

20.2.4 Supporting resource conversion

When converting an application in native code to Unicode, note that COBOL files may also be converted, although this is not required.

20.3 Creating a Unicode Application

This section describes the procedure for creating a practical Unicode application. Note the various points when coding the application. See Section "20.4 Notes on Coding", before coding.

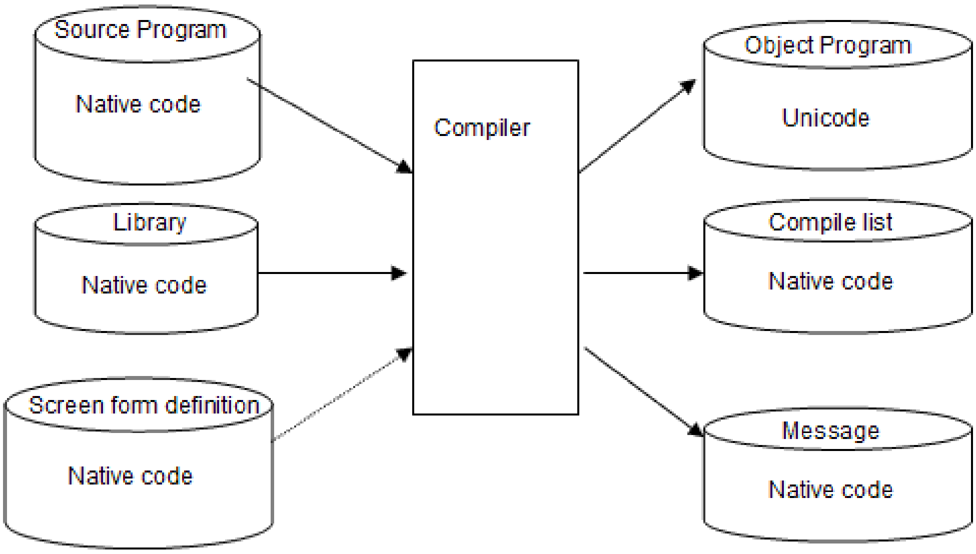
20.3.1 Creating and editing a program

Use native code to create a source program or library. An existing editor can be used to create or edit a program.

20.3.2 Compilation

Specify compiler option ENCODE. Note that some options have locked relationships. See "A.2.13 ENCODE (encoding form specification)" for details.

The figure below shows the code systems of related resources at compilation.



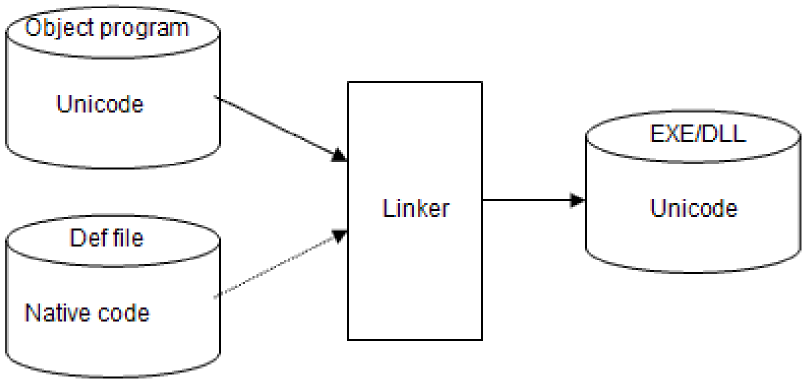
Note

Resources indicated with a dotted line may be omitted.

20.3.3 Link

No special items need be noted at linkage. No special library or link options are required for a Unicode application. Resources can be linked the same way as in native code.

The figure below shows the code systems of related resources at linkage.



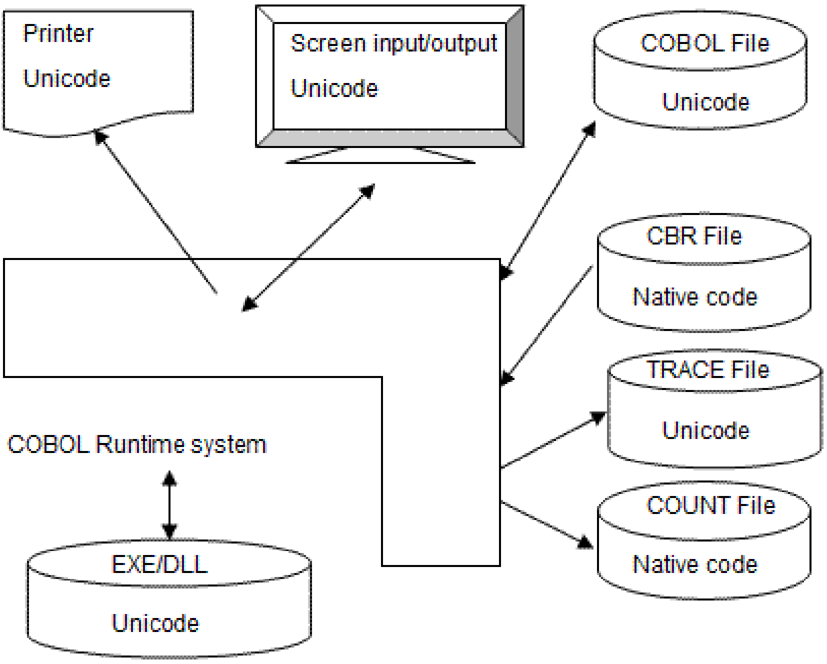
Note

Standard COBOL libraries are omitted from the figure.

20.3.4 Execution

No special items need be noted at execution. Unicode data can be processed the same way as in native code.

The figure below shows the code systems of related resources at execution.



20.3.5 Debugging

A Unicode application can be debugged using the debugging function of NetCOBOL Studio. See the "NetCOBOL Studio User's Guide" for details.

20.4 Notes on Coding

This section contains notes on creating a Unicode application for each language element.

In the coding examples in this section, it is assumed that the following alphabet-names are declared in the ENVIRONMENT DIVISION.

```
ALPHABET
   U8 FOR ALPHANUMERIC IS UTF8
   U16L FOR NATIONAL IS UTF16LE
   U16B FOR NATIONAL IS UTF16BE
   U32L FOR NATIONAL IS UTF32LE
   U32B FOR NATIONAL IS UTF32BE.
```

20.4.1 User-defined word

Only the character within the range of native code can be specified for following user-defined word and nonnumeric literal though source file and library file can be made with Unicode(UTF-8).

- Program-name
- Entry-name
- Class-name
- Method-name
- Property-name
- Program name of CALL statement
- Program name of CANCEL statement
- Class-name and method-name of INVOKE statement

Moreover, the character of a surrogate pair cannot use user-defined word other than the above.

20.4.2 Nonnumeric literals

National hexadecimal nonnumeric literal

Describe the national hexadecimal nonnumeric literal by the hexadecimal number that matches national characters to the encoding of the operand.

```
 WORKING-STORAGE SECTION.
 01 NAME1         PIC N(3) ENCODING IS U16L VALUE NC"ABC".
 01 NAME2         PIC N(3) ENCODING IS U32L VALUE NC"ABC".
 PROCEDURE DIVISION.
*>      :
     IF NAME1 = NX"004100420043" THEN DISPLAY "OK".
     IF NAME2 = NX"000000410000004200000043" THEN DISPLAY "OK".
```

Figurative constant

The figurative constant is determined by the operand.

For example, figurative constant SPACE becomes X"20" when operand is alphanumeric data item, and becomes X"0020" or X"00000020" when operand is national data item. Code value is subject to encode the operand.

20.4.3 National data item

When handling data by Unicode (UTF-16 or UTF-32), national data items must be used. For a national data item, a two-byte or four-byte field is reserved for one digit.

When moving a nonnumeric literal to a national data item, use a national nonnumeric literal. The definition of national character is different according to the encoding form of the operand.

```
WORKING-STORAGE SECTION.
01 ADDR.
   02 CITY    PIC N(20) ENCODING IS U16L.
   02 COUNTRY PIC N(20) ENCODING IS U32L.
PROCEDURE DIVISION.
```

```
*>      :
   MOVE NC"Sanjose" TO CITY.
                       *> CITY=X"00530061006E006A006F00730065"
   MOVE NC"U.S.A." TO COUNTRY.
                      *> COUNTRY=X"000000550000002E000000530000002E000000410000002E"
```

For example, the ASCII space character is not contained in the national characters in Shift-JIS but the ASCII space character is contained in the national characters in Unicode. When the encoding form of the operand is UTF-16 or UTF-32, the literal that contains both of ASCII spaces and national spaces can be declared.

```
     MOVE NC"Sanjose U.S.A." TO ADDR.
```

In the example above, the data item ADDR can be declared in Unicode, but a compile error occurs in Shift-JIS.

20.4.4 Redefining an item

When a national data item is redefined as an alphanumeric data item (using the REDEFINES clause) or an alphanumeric data item is redefined as a national data item, caution is required.

```
 WORKING-STORAGE SECTION.
 01 PERSON.
   02 AGE          PIC 9(3).
   02 NAME         PIC N(8)                 ENCODING IS U16L.
   02 NAME-X       REDEFINES NAME PIC X(16) ENCODING IS U8.
 01 TMP-NAME     PIC X(16).
PROCEDURE DIVISION.
*>      :
     MOVE NAME-X TO TMP-NAME.
     DISPLAY TMP-NAME.    *> Incorrectly displayed characters
```

In Unicode, the encoding form between a national data item (UTF-16 or UTF-32) and alphanumeric data item (UTF-8) is completely different. To reference the same data using another class by redefinition, data conversion that matches the operand may be required. Use the NATIONAL-OF function and DISPLAY-OF function to convert data.

```
 01 TMP-NAME     PIC X(24).
PROCEDURE DIVISION.
*>      :
     MOVE FUNCTION DISPLAY-OF(NAME) TO TMP-NAME.
     DISPLAY TMP-NAME.
     MOVE CONV NAME TO TMP-NAME.
     DISPLAY TMP-NAME.
```

20.4.5 Move

Group item move

When a group item including a national data item is used for move, caution is required in Unicode.

```
 WORKING-STORAGE SECTION.
 01 PERSON.
   02 AGE          PIC 9(3).
   02 NAME         PIC N(20) ENCODING IS U16L.
 01 TMP-AREA     PIC X(80) ENCODING IS U8.
PROCEDURE DIVISION.
*>      :
     MOVE PERSON TO TMP-AREA.
*>      :
     MOVE TMP-AREA TO PERSON.      *> [1]
     DISPLAY "DATA = " TMP-AREA.   *> [2]
```

In the example above, there is no problem when using TMP-AREA as a temporary work area (as shown in [1]). But, when data is directly referenced (as shown in [2]), data is not displayed correctly due to a mixture of encoding forms. In this case, match the temporary area with the original (PERSON) data structure.

Padding with spaces

In COBOL, when the receiving item is longer than the sending item at character move, the receiving item is padded with spaces. In Unicode, ASCII spaces (X"0020" or X"00000020") are used to pad at national move.

Unjustified character division

To store national characters in an alphanumeric item, two digits per character is necessary in Shift-JIS. 2 to 4 digits per characters are necessary for UTF-8. One character is divided by a post, a comparison, and a partial reference, etc. because it is treated as two characters in the language specification, and there is a possibility of it becoming an illegal character even though it is displayed as one character. When one character is divided, it is called "unjustified character division".

For a national item, the consideration of "unjustified character division" is unnecessary in Shift-JIS because the number of digits and the number of characters are the same.

In UTF-16, the character of a surrogate pair is stored in a national data items so two digits are needed. The "unjustified character division" may occur. To address this, make sure that:

- Enough area is reserved
- A surrogate pair is excluded using the class condition
- Character-string handling is done carefully

In UTF-32 the consideration of "unjustified character division" is unnecessary because the number of digits and the number of characters are the same.

20.4.6 Comparison

Group item comparison

When comparing group items, items having different classes can virtually be compared. Caution is required because Unicode has a different encoding form.

```
 WORKING-STORAGE SECTION.
 01 DATA-X.
   02 NAME   PIC X(4) ENCODING IS U8   VALUE "ABCD".   *> X"41424344"
 01 DATA-N.
   02 NAME   PIC N(4) ENCODING IS U16L VALUE NC"ABCD". *> X"0041004200430044"
PROCEDURE DIVISION.
*>      :
     IF DATA-X = DATA-N THEN DISPLAY "OK??".
```

Use the same data structure (declaration) of the operand in a group item comparison.

Nonnumeric comparison

When the encoding forms of the operands are different, a compiler error occurs. Match the encoding form of the operands before the comparison.

In Unicode, caution is required when using a group item including a national data item for a nonnumeric comparison.

```
WORKING-STORAGE SECTION.
 01 GRP.
   02 SMALL    PIC N(1) ENCODING IS U16L VALUE NC"A".   *> X"0041"
 01 LARGE    PIC N(1) ENCODING IS U16L VALUE NC"B".     *> X"0042"
PROCEDURE DIVISION.
*>      :
     IF GRP   < LARGE THEN DISPLAY "OK??".   *>[1]
     IF SMALL < LARGE THEN DISPLAY "OK".     *>[2]
```

In [1], the left side is a group item, so endian conversion is applied to neither the left side nor the right side. A comparison to determine which is larger or smaller is made without converting their little-endian format.

Therefore, the left side is greater, and the result of the expression is not true.

In [2], both the right and left sides are national data items. The compiler converts both sides to big-endian before comparing. The comparison is correct.

For nonnumeric comparison, match the class of the operand with the data structure.

In actual coding, usage shown in the example above is rare. However, the same applies to specifications of an indexed file key, sort-merge file key, and SEARCH ALL key. Therefore, caution is required when using a group item including a national data item in these circumstances.

20.4.7 Creating and editing a program

- In the reference format, the maximum length of 1 line is 251bytes for variable length format and free format, and 80 bytes for fixed length format. This is the physical length, not the display length.
  When the source programs are created in Unicode(UTF-8), the physical length becomes larger than the display length.In this case, split the lines so that you can use the continuation line.
  If the maximum length is exceeded, the compiler ignores the part that has exceeded the maximum length. For this reason, following message is displayed at the time of compilation:

  - cobol: ERROR: system error 'errno=0x016' occurred in 'iconv_error'.

- The character of a surrogate pair cannot be used for user-defined word.

20.4.8 ACCEPT/DISPLAY statement

Simple input-output

Unicode data can be input or output using the ACCEPT or DISPLAY statement. Caution is required when the operand is a group item including a national data item.

```
 WORKING-STORAGE SECTION.
 01 PERSONAL-DATA.
   02 NAME   PIC N(20) ENCODING IS U16L.
   02 TEL    PIC 9(10) ENCODING IS U8.
PROCEDURE DIVISION.
*>      :
     DISPLAY PERSONAL-DATA.   *>[1]
     DISPLAY NAME TEL.        *>[2]
```

The Unicode encoding forms differ depending on the class. When a group item includes a national data item that is displayed using a DISPLAY statement, an illegal character appears (as shown in [1]). In this case, display each elementary item separately (as shown in [2]).

When data is read to a group item where a national item is included using the ACCEPT statement, the character-code of the data becomes the character-code of the alphanumeric item of the character-code specified by the ENCODE option.

When a file is specified for the input-output destination of the ACCEPT or DISPLAY statement, the encoding form of the file is UTF-8.

Note

The encoding form differs from that of a line sequential file. (UTF-16 little-endian is assumed when the record definition class is unique to national data item, as described below.) Therefore, note the difference in the code when using a simple input-output module for line sequential files. See Section "20.4.9 COBOL files".

Others

For the following functions, only characters within native code can be used:

- Simple input-output using the system console
- Command line argument
- Environment variable operation

20.4.9 COBOL files

A record sequential file, relative file, index file, and line sequential file are all input or output without changing Unicode data. For a print file and presentation file (PRT), the code is converted at output according to the class of each item. Note the following:

File identifier (all files)

If a data name specified as the file identifier is a group item including a national data item, a compile-time error occurs in Unicode.

```
 FILE-CONTROL.
     SELECT OUTFILE ASSIGN TO FILE-NAME.
*>      :
 WORKING-STORAGE SECTION.
 01 FILE-NAME.           *>  Compile-time error
   02 F-DRV    PIC X(3) ENCODING IS U8.
   02 F-NAME   PIC N(4) ENCODING IS U16L VALUE NC"DATA".
PROCEDURE DIVISION.
*>      :
     MOVE "C:\" TO F-DRV.
     OPEN OUTPUT OUTFILE.
```

The above example is used to prevent an illegal file name character due to mixed encoding forms. Use only alphanumeric characters for the class to specify a group item in the file identifier.

Line sequential file

A line sequential file is based on format that can be displayed or edited using text editors. One encoding form must be used in a file. The class of items making up a record must be integrated. In NetCOBOL, when the record definition class is unique to alphanumeric characters, UTF-8 is used for input-output. When the class is unique to national characters, UTF-16 or UTF-32 (depending on ENCODE option) is used for input-output. If classes are mixed as shown in the example below, a compile-time error is output. Integrate the class according to usage.

```
 FILE-CONTROL.
     SELECT OUTFILE ASSIGN TO "data.txt"
            ORGANIZATION IS LINE SEQUENTIAL.
*>      :
DATA DIVISION.
 FILE SECTION.
 FD OUTFILE.           *>  Compile-time error
 01 OUT-REC.
   02 REC-ID    PIC X(4) ENCODING IS U8.
   02 REC-DATA  PIC N(20) ENCODING IS U16L.
```

The created line sequential file can be referenced and updated using an editor that can handle Unicode.

Note

The identification code called the signature must be added to the beginning of a Unicode text file. The user need not consider this signature because the COBOL runtime system processes it.

Index file

Caution is required when specifying a group item including a national data item as the index key.

```
 FILE-CONTROL.
     SELECT IX-FILE ASSIGN TO F-NAME
```

```
                    ORGANIZATION IS INDEXED
                    RECORD KEY IS IX-KEY.
*>      :
DATA DIVISION.
 FILE SECTION.
 FD IX-FILE.
 01 IX-REC.
   02 IX-KEY.
     03 KEY1     PIC X(4) ENCODING IS U8.
     03 KEY2     PIC N(8) ENCODING IS U16L.
   02 IX-DATA  PIC X(80).
```

Data is stored using the UTF-16 or UTF-32 little endian in the national data item; the higher and lower-order bytes are reversed. If this data is referenced using the group item, it is handled in the reversed format. Therefore, the data may not be positioned in the designated record by the START statement or other statement.

In this case, W level error is output to indicate a warning at compilation. Make necessary modifications, such as using a sub-record key as required.

Print file

For a print file, the classes in the record need not be integrated. The COBOL runtime system converts the code according to the class of each item. The file operates normally even if there are mixed encoding forms. However, when mixed encoding forms are used for the same class, a compile-time error is output.

```
 FILE-CONTROL.
     SELECT OUT-FILE ASSIGN TO PRINTER.
*>      :
DATA DIVISION.
 FILE SECTION.
 FD OUT-FILE.
 01 OUT-REC  PIC X(120).
 WORKING-STORAGE SECTION.
 01 PRT-DATA  CHARACTER TYPE IS MODE-1.
   02 PRT-NO    PIC 9(4).
   02 PRT-ID    PIC X(4).
   02 PRT-NAME  PIC N(20) ENCODING IS U16L.
   02 PRT-ADDR  PIC N(20) ENCODING IS U32L.
PROCEDURE DIVISION.
*>      :
     OPEN OUTPUT OUT-FILE.
     WRITE OUT-REC FROM PRT-DATA AFTER ADVANCING 1 LINE.
```

If a group item including a national data item is used as a receiving item, code in a group item not matching the class is padded with spaces. Because of this, characters may be disrupted in printing.

```
 FILE-CONTROL.
     SELECT OUT-FILE ASSIGN TO PRINTER.
*>      :
DATA DIVISION.
 FILE SECTION.
 FD OUT-FILE.
 01 OUT-REC  CHARACTER TYPE IS MODE-1 ENCODING IS U16L.
   02 OUT-DATA  PIC N(40).
 WORKING-STORAGE SECTION.
 01 PRT-DATA  CHARACTER TYPE IS MODE-1 ENCODING IS U16L.
   02 PRT-NO    PIC N(4).
   02 PRT-ID    PIC N(4).
   02 PRT-NAME  PIC N(20).
PROCEDURE DIVISION.
*>      :
     OPEN OUTPUT OUT-FILE.
     MOVE  PRT-DATA TO OUT-REC.            *>[1]
     WRITE OUT-REC AFTER ADVANCING 1 LINE.
```

```
*>       :
     MOVE  NC"ABCD" TO OUT-REC.            *> [2]
     WRITE OUT-REC AFTER ADVANCING 1 LINE.
```

If the receiving item is longer than the sending item (as shown in [1]), the receiving item is padded with ASCII spaces according to the group item move rules. Therefore, OUT-DATA stores both UTF-16 and UTF-8 data. When the WRITE statement is executed, OUT-DATA is handled as UTF-16 data. The print result of the part storing UTF-8 data becomes illegal characters. The same result is obtained in [2].

To avoid illegal characters, explicitly specify a national data item for the move target as shown in the example below:

```
*>       :
     WRITE OUT-REC FROM PRT-DATA AFTER ADVANCING 1 LINE.  *> [3]
*>       :
     MOVE  NC"ABCD" TO OUT-DATA.                          *> [4]
     WRITE OUT-REC AFTER ADVANCING 1 LINE.
```

When using the WRITE statement with the FROM clause specified, [3] is printed according to the class of the data item specified in the FROM clause. When a national data item is specified for the move receiving side, [4] is padded with UTF-16 spaces. Therefore, UTF-8 data is not mixed in the national data item.

Form descriptor [print file with FORMAT clause and presentation file (PRT)]

An alphanumeric data item defined in the form descriptor stores only characters coded in one byte. When an alphanumeric data item defined in the form descriptor stores a national character for input-output, the desired result is not obtained in Unicode.

If the data contains national characters, define a mixed-item instead of an alphanumeric data item in the form descriptor. The attribute of data expanded by the COPY statement becomes an alphanumeric data item regardless of whether an alphanumeric data item or a mixed-item is defined. Because this data is handled differently at execution, it is processed normally only when a mixed-item is defined.

20.4.10 Screen function

To use national characters with the screen function, use a national data item.

However, encoding form UTF-32 cannot be used.

Note

A surrogate pair character cannot be used.

20.5 Notes on Execution

20.5.1 Message Output File

When the runtime code set is Unicode, the code set of a file that receives a runtime message (specified using environment variable @MessOutFile) is as described below. See "5.4.1.56 @MessOutFile(Set a Message Output File)"and "5.4.1.5 @CBR_CODE_SET(Specify the code-set for the output file)".

- To add to an existing file, the code of the existing file is used.
- To output to a new file, UTF8 is used.

20.5.2 Font

Use a font that supports Unicode to use the following functions:

COBOL console window

Specify a font that supports Unicode in environment variable @CnslFont. See Section "5.4.1.49 @CnslFont(Set the Console Window Font)" for specification details.

Screen window

Specify a font that supports Unicode in environment variable @ScrnFont. See Section "5.4.1.62 @ScrnFont(Set the Font Used for Screen Handling)" for specification details.

Print file without FORMAT clause

Specify a font that supports Unicode in environment variable @PrinterFontName and font face name specified in the font table. See Section "5.4.1.60 @PrinterFontName(Set the Font Used for Print Files)" and Section "7.1.14 Font table" for specification details.

Print file with FORMAT clause and presentation file

Specify a font that supports Unicode in the printer information file, and font table. See the "FORM Runtime System online manual" and Section "7.1.14 Font table" for specification details.

File that TRACE function and COUNT function output

When the operation mode is Unicode, the code system of the file that TRACE and COUNT function output is as follows:

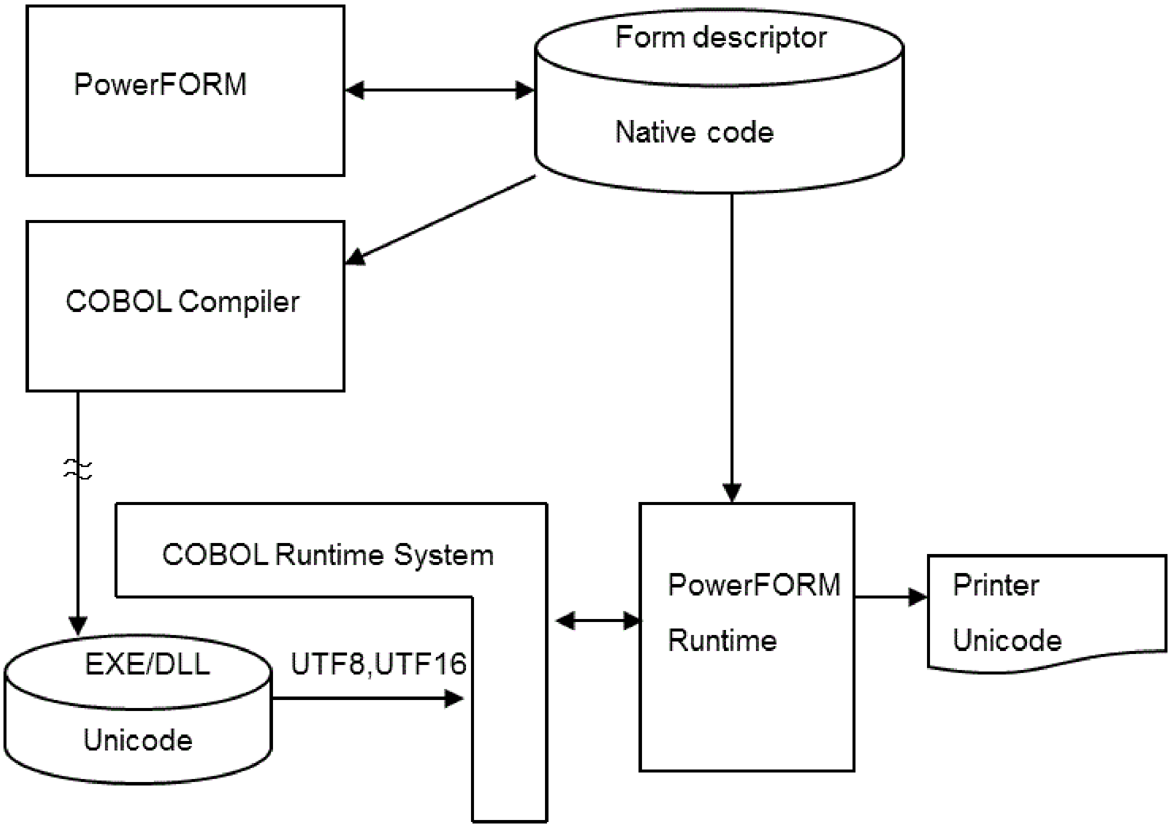
- Code system of the existing file when an addition is written to an existing file.
- UTF-8 when outputting to new file.

20.6 Linkage with Related Products

20.6.1 FORM/FORM Runtime System

A form descriptor created using PowerFORM can be used in a Unicode application. Both existing and newly-created descriptors can be used. Because PowerFORM Runtime System automatically converts the code to Unicode at execution, native code can be used for descriptor code.

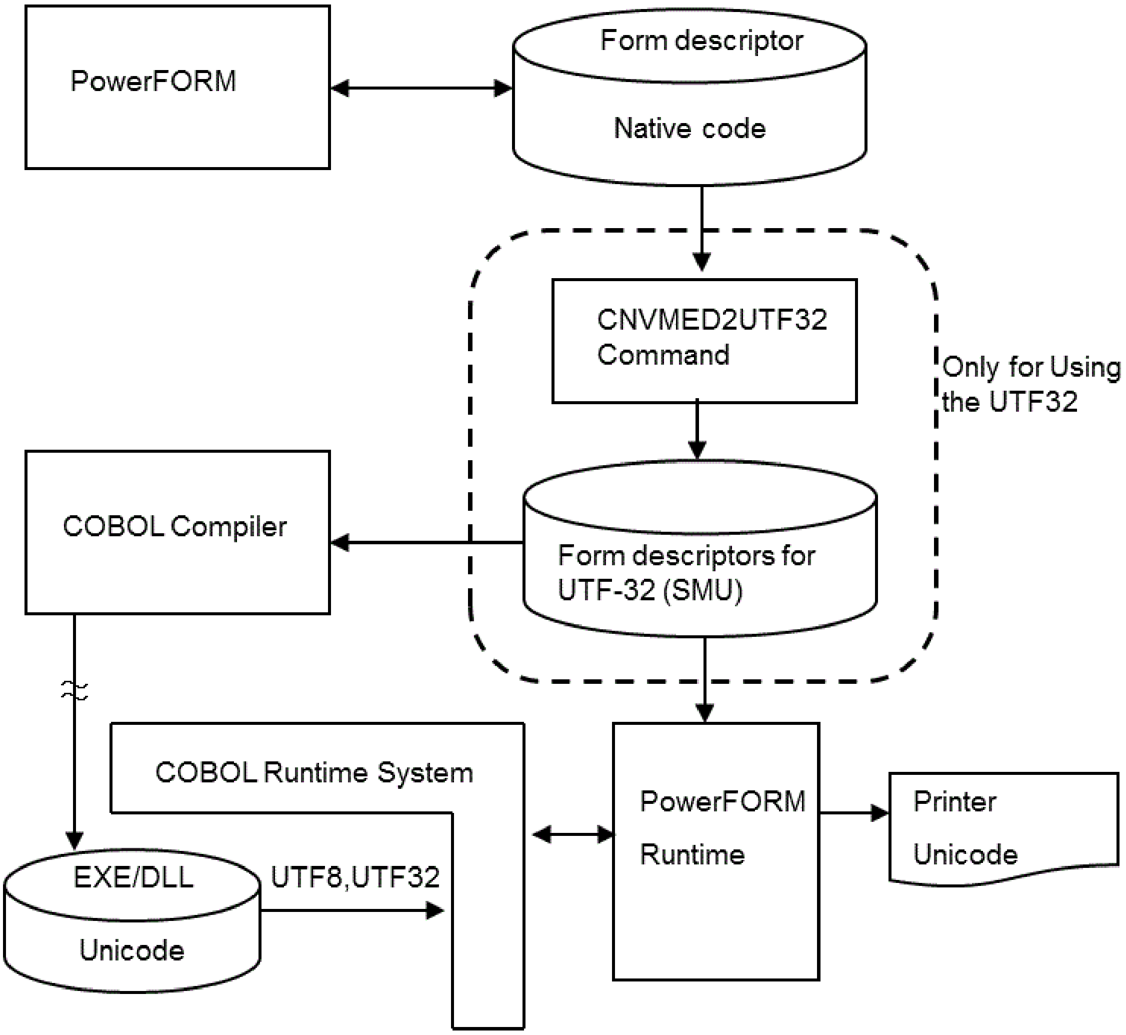
When the code system of the input and output data is UTF-16, the use of a surrogate pair is different depending on the version level of the product used. For details, see the MeFt manual.



The usage and function range of the print file with the FORMAT clause, and presentation file (forms print) are the same as for native code. See "Chapter 7 Printing" for details.

Using the encoding form UTF-32

When using the encoding form UTF-32, it is necessary to convert the form descriptor for UTF-32 using the CNVMED2UTF32 command.



See "I.4 CNVMED2UTF32 Command" for details.

The form descriptor for UTF-32 cannot be renewed directly with PowerFORM.

In this case, convert it again with the CNVMED2UTF32 command after renewing the form descriptor in the conversion origin.

20.6.2 Precompiler

The use of Unicode data when accessing a remote database with a pre-compiler depends on the specification of the pre-compiler. Confirm use with the manual of the pre-compiler of each database.

Moreover, control the encoding form with compilation option ENCODE without specifying the ALPHABET phrase nor the ENCODING phrase of the pre-compiler of each database.

20.6.3 Remote Database Access (ODBC)

Unicode data can be sent or received for a database by using an embedded SQL statement with the remote database access (ODBC) function. When using encoding form UTF-32, the COBOL Runtime System converts it into Unicode data type (UTF-16).

Note the following for the input or output of Unicode data:

- The data source to be accessed must support the input-output of Unicode data. The data source includes the database, ODBC driver, and other network components.

- To input or output Unicode data using a host variable, use a host variable defined as a national data item. In COBOL, the national data item is mapped on the Unicode data type defined in ODBC. See "17.2.11 Correspondence Between ODBC-Handled Data and COBOL-Handled Data".

- Only alphanumeric data items can be handled by alphanumeric host variables. National characters cannot be handled. When national characters are input or output using an alphanumeric host variable, the result of input-output is not guaranteed.

- When compiler option SCS(ACP) is specified, to input Unicode data in a Unicode data type string of the database by specifying an embedded SQL statement value, code the character-string literal or national character-string literal in the native code. In this case, the data source converts native code data to Unicode data.

20.6.4 Other File Systems

Btrieve file

Btrieve files can also be used in a Unicode application the same way as COBOL files (except for the following note). See "20.4.9 COBOL files", for details.

Note

If a national data item is included in the index key (for both elementary and group items) in an indexed organization Btrieve file, data stored in the national data item is compared as little-endian. The file may not be positioned at the desired record.

20.6.5 Interstage Application Server

When creating the CORBA service application and event service, the ENCODE compile options cannot be specified.

Chapter 21 Operation of Comma Separated Value data

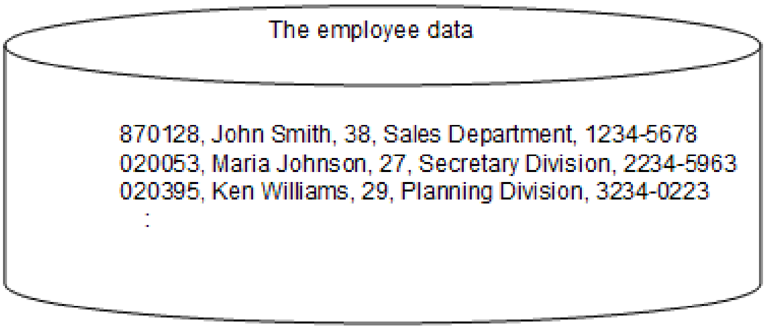
This chapter explains the operation of the Comma Separated Value (CSV) data that uses the STRING and UNSTRING statements.

To manipulate the CSV data easily in NetCOBOL, the extension facility must be prepared in the STRING and UNSTRING statements.

21.1 What is CSV data?

CSV is an implementation of a delimited text file in which a comma is used to separate values. The CSV format has historically been used in spreadsheet and database software, and now can also be used to format the data of various tools and middleware.

For example, CSV delimits a text file as shown below. The employee number, name, age, department, and extension are delimited by commas.



```
870128, John Smith, 38, Sales Department, 1234-5678
020053, Maria Johnson, 27, Secretary Division, 2234-5963
020395, Ken Williams, 29, Planning Division, 3234-0223
  :
```

Normal functions include reading each record as a line sequential file when this data is input by the COBOL program and edited, resolving to the elementary item that is subordinate to the group item of the character-string data delimited by the comma, and general operation. However, this was not easily achieved using previous language specifications.

Consider the case where the employee data above is posted to the following group items.

```
  01 employee.
    02 emp-ID           PIC 9(6).
    02 emp-name         PIC X(30).
    02 emp-age          PIC 9(2).
    02 emp-section      PIC X(30).
    02 emp-extension    PIC X(10).
```

It can be posted using the PERFORM statement while inspecting the CSV data from the beginning, one character at a time, using the UNSTRING statement (format 1).

```
  FILE SECTION.
  FD CSV-FILE.
  01 CSV-REC PIC X(80).
*>  :
  WORKING-STORAGE SECTION.
  01 employee.
    02 emp-ID           PIC 9(6).
    02 emp-name         PIC X(30).
    02 emp-age          PIC 9(2).
    02 emp-section      PIC X(30).
    02 emp-extension    PIC X(10).
PROCEDURE DIVISION.
     OPEN INPUT CSV-FILE.
     READ CSV-FILE.
     UNSTRING CSV-REC
           DELIMITED BY "," INTO emp-ID emp-name emp-age emp-section emp-extension
     END-UNSTRING.
     *>  :
```

The following problems are inherent in the above example.

- The UNSTRING statement should match the class on the receiving side according to the rules for moving data. The REDEFINES clause cannot be used with Unicode because the encoding is different.
- In the CSV data, if the entire data is enclosed with a quotation mark, the comma can be used as data. However, this cannot be processed by the example above.
- When the quotation mark is used as data, one quotation mark is expressed by two consecutive quotation marks. However, this cannot be processed by the example above.

When the CSV data is generated by using the STRING statement (format 1), it has similar problems. The problem is more serious, considering the processing at the trailing blanks, etc.

Operation was facilitated via a syntax that utilizes the CSV data manipulator for the STRING statement and the UNSTRING statement in NetCOBOL.

21.2 Generating CSV data (STRING statement)

This section explains the method of generating CSV data.

21.2.1 Basic operation

The character-string data stored in the group item is used and the STRING statement (format 2) is used for CSV when editing it via a subordinate item unit.

```
  WORKING-STORAGE SECTION.
  77 employee-edit  PIC X(80).
  01 employee.
    02 emp-ID           PIC 9(6).   *> 870128 is stored.
    02 emp-name         PIC X(30).  *> John Smith is stored.
    02 emp-age          PIC 9(2).   *> 38 is stored.
    02 emp-section      PIC X(30).  *> Sales Department is stored.
    02 emp-extension    PIC X(10).  *> 1234-5678 is stored.
PROCEDURE DIVISION.
     PERFORM Get-Employee.
     *>  :
     MOVE SPACE TO employee-edit.
     STRING employee INTO employee-edit BY CSV-FORMAT.
     *>  :
```

When the STRING statement of the example above is executed, the following CSV data is stored in the data item "employee-edit".

```
  870128,John Smith,38,Sales Department,1234-5678
```

When CSV is generated, the stored data is edited as follows:

- The sending item is converted as follows:
  - With Unicode, the encoding is converted according to the encoding of the receiving item when the sending item is alphanumeric or national.
  - When the sending item is a signed numeric, the sign is added to the left end regardless of the specification of the SIGN phrase. When the fraction part is included, the decimal-point character is added.
- When a separator is included in the sending data, the entire data is enclosed in double quotes.
- When double quotes are included in sending data, they are replaced by two consecutive double quotes, and the entire data is enclosed with double quotes.

- When the class of the sending item is alphanumeric character or national character, trailing blanks are removed.
- When the sending item is a numeric item, the leading zeros are removed. However, when the value is 0, the one digit 0 is posted. When a fraction part is included, the trailing zeros are removed.
- The entire data is enclosed with double quotes according to the TYPE specification. Refer to "21.4 Variation of CSV" for details.

For STRING statement details, refer to "COBOL Language Reference".

Note

- Initialize the receiving item before executing a STRING statement.
- In List Creator, even when two double quotes are included in the CSV data, they are not replaced by one double quote.

21.2.2 Error detection

The following conditions indicate errors in CSV data generation.

Table 21.1 CSV data generation errors

(1)	Illegal data is stored in the sending item.
(2)	The receiving side item is small and all data does not enter completely.
(3)	The value of data item with POINTER phrase specified is less than 1.

In a STRING statement, the error in CSV data generation can be detected using the ON OVERFLOW phrase. It is stored where it is normally treatable in the receiving item.

For instance, if the ON OVERFLOW phrase is specified, when the receiving item area size is only 20 bytes, the error can be detected.

```
  WORKING-STORAGE SECTION.
  77 employee-edit  PIC X(20).
  01 employee.
    02 emp-ID           PIC 9(6).   *> 870128 is stored.
    02 emp-name         PIC X(30).  *> John Smith is stored.
    02 emp-age          PIC 9(2).   *> 38 is stored.
    02 emp-section      PIC X(30).  *> Sales Department is stored.
    02 emp-extension    PIC X(10).  *> 1234-5678 is stored.
PROCEDURE DIVISION.
     PERFORM Get-employee.
     *>  :
     MOVE SPACE TO employee-edit.
     STRING employee INTO employee-edit BY CSV-FORMAT
         ON OVERFLOW DISPLAY "Edit failed. Data : " employee-edit
     END-STRING.
     *>  :
```

When a CSV data generation error is encountered, the following occurs when the ON OVERFLOW phrase is not specified.

(1) and (3):

It terminates abnormally after the runtime error is output.

(2):

After the execution error is output, the control is moved to the following statement of the STRING statement.

21.3 Resolution of CSV data (UNSTRING statement)

This section explains the method of CSV data resolution.

21.3.1 Basic operation

When the CSV data is resolved, and a move is done to the item subordinate to the group item, the UNSTRING statement(format 2) is used.

```
  WORKING-STORAGE SECTION.
  77 employee-data  PIC X(80)
            VALUE "870128,John Smith,38,Sales Department,1234-5678".
  01 employee.
    02 emp-ID           PIC 9(6).
    02 emp-name         PIC X(30).
    02 emp-age          PIC 9(2).
    02 emp-section      PIC X(30).
    02 emp-extension    PIC X(10).
PROCEDURE DIVISION.
     *>  :
     UNSTRING employee-data(1:FUNCTION STORED-CHAR-LENGTH(employee-data))
           INTO employee BY CSV-FORMAT.
     *>  :
```

When the UNSTRING statement in the above example is executed, each value separated by commas is stored in each elementary item subordinate to group item "employee".

In the resolution of the Comma Separated Value data, data is edited as follows.

- The sending item is converted as follows:
  - With Unicode, the character code is converted according to the encoding of the receiving item when the sending item is alphanumeric or national.
  - If the receiving item is a signed numeric, the sign is processed according to the specification of SIGN clause of receiving item. The digit match includes the decimal-part.
- When the divided data is enclosed with double quotes, it is posted after the double quotes are excluded.
- When consecutive double quotes are included in the division data enclosed with double quotes, it replaces it with one double quote.
- When the resolved data is posted to the receiving item, it follows the rules of moving data.

For UNSTRING statement details, refer to "COBOL Language Reference".

Note

The program does not operate correctly when the text file of TSV data is read using the line sequential file function and is resolved using the UNSTRING statement.

This is due to the tab being replaced by blanks before the READ statement is executed. Specify high-speed processing (",BSAM") for correctly processing line sequential files.

Refer to "6.7.3 File Processing Results" and "6.7.4 High-Speed File Processing".

21.3.2 Error detection

The following conditions indicate CSV data resolution errors.

Table 21.2 CSV data resolution errors

(1)	Illegal data is stored in the sending item.
(2)	When moving it to the receiving item, an overflow was generated.
(3)	The number of resolved character-string data is greater than the number of receiving.

(4)	The value of data item that specifies POINTER phrase is larger than number of digits of the receiving item.

In the UNSTRING statement, the error in CSV data resolution can be detected using the ON OVERFLOW phrase. It is stored where it is normally treatable in the receiving item.

For instance, if the ON OVERFLOW phrase is specified, when the definition of "emp-extension" is left out of the receiving item, the error can be detected.

```
  WORKING-STORAGE SECTION.
  77 employee-data  PIC X(80)
            VALUE "870128,John Smith,38,Sales Department,1234-5678".
  01 employee.
    02 emp-ID           PIC 9(6).
    02 emp-name         PIC X(30).
    02 emp-age          PIC 9(2).
    02 emp-section      PIC X(30).
PROCEDURE DIVISION.
     *>:
     UNSTRING employee-data(1:FUNCTION STORED-CHAR-LENGTH(employee-data))
           INTO employee BY CSV-FORMAT
           ON OVERFLOW DISPLAY "Data resolution failed."
     END-UNSTRING.
```

Since the character position of the sending data that caused the error can be acquired by specifying the POINTER phrase, more detailed information can be obtained.

When the TALLYING phrase is specified, the number of items that succeeds in the move can be obtained.

```
  77 employee-data  PIC X(80)
            VALUE "870128,John Smith,38,Sales Department,1234-5678".
  01 employee.
    02 emp-ID           PIC 9(6).
    02 emp-name         PIC X(30).
    02 emp-age          PIC 9(2).
    02 emp-section      PIC X(30).
  77 CNT            PIC 9(2).
 PROCEDURE DIVISION.
     *> :
     MOVE 1 TO CNT.
     UNSTRING employee-data(1:FUNCTION STORED-CHAR-LENGTH(employee-data))
           INTO employee BY CSV-FORMAT POINTER CNT
           ON OVERFLOW DISPLAY "Data resolution failed."
                       DISPLAY "Failure data : " employee-data(CNT:)
     END-UNSTRING.
```

When a CSV data resolution error is encountered, the following occurs when the ON OVERFLOW phrase is not specified.

(1) and (4):

It terminates abnormally after the runtime error is output.

(2):

After the execution error is output, the processing of the UNSTRING statement continues.

(3):

After the execution error is output, control is moved to the statement following the UNSTRING statement.

21.4 Variation of CSV

This section explains the variation of CSV.

There is no formally approved specification for CSV, such as ISO standards. Information provided by Microsoft is assumed to be the de facto standard, with some derived forms in use.

In NetCOBOL, the following four variations of the Comma Separated Value generated with the STRING statement (format 2) can be used.

Variation	Meanings
MODE-1	When separator or double quotes exist in sending data, the entire data is enclosed with double quotes. When all blanks are stored in the sending data item whose class is alphanumeric or national, only the separator is posted in the receiving item.
MODE-2	Sending data is enclosed with double quotes. When all blanks are stored in the sending data item whose class is alphanumeric or national, only the separator is posted in the receiving item.
MODE-3	The entire data is enclosed with double quotes, except when the class of the sending data item is a number. When all blanks are stored in the sending data item whose class is alphanumeric or national, only the separator is posted in the receiving item.
MODE-4	The entire data is enclosed with double quotes, except when the class of the sending data item is a number. When all blanks are stored in the sending data item whose class is alphanumeric or national, two consecutive double quotes are posted in the receiving item.

These variations are specified by the TYPE specification or the execution environment variable "5.4.1.9 @CBR_CSV_TYPE(Set the variation of generated CSV type)" of the STRING statement (format 2). MODE-1 is considered the default.

```
 01 employee.
   02 emp-ID            PIC 9(6)  VALUE 870128.
   02 emp-name          PIC X(30) VALUE "John Smith".
   02 emp-section       PIC X(30) VALUE "Sales Department".
   02 emp-position      PIC X(10) VALUE SPACE.
   02 emp-extension     PIC X(20) VALUE "1234-5678,9876".
```

In the example above, when "employee" is specified for the sending item, it becomes the following, depending on MODE:

Variation	Result
MODE-1	870128,John Smith,Sales Department,, "1234-5678,4536"
MODE-2	"870128","John Smith","Sales Department",,"1234-5678,4536"
MODE-3	870128,"John Smith","Sales Department",,"1234-5678,4536"
MODE-4	870128,"John Smith","Sales Department","","1234-5678,4536"

Resolution and move can be done to an item subordinate to the group item using the UNSTRING statement (format 2) for each CSV data.

Chapter 22 Remote Development Support Function

This chapter describes the remote development support function.

22.1 Overview of Remote Development

22.1.1 What is Remote Development?

Using the remote development support functions of NetCOBOL Studio on the client side and NetCOBOL on the server side, you can develop COBOL applications efficiently with a variety of Windows systems.

Client side

NetCOBOL and NetCOBOL Studio must be installed on the client system in order to enable . NetCOBOL Studio to perform development tasks. NetCOBOL Studio can be connected to a server to perform tasks such as compilation, with the results displaying in NetCOBOL Studio.

Server side

NetCOBOL must be installed on the server.

22.1.2 Advantages of Remote Development

Many COBOL applications run on expensive server machines. Developing COBOL application on such machine causes the following problems:

- You must use command line interface because many of these systems do not enable GUI-based environment.
- The system at server side must be shared among the developers.

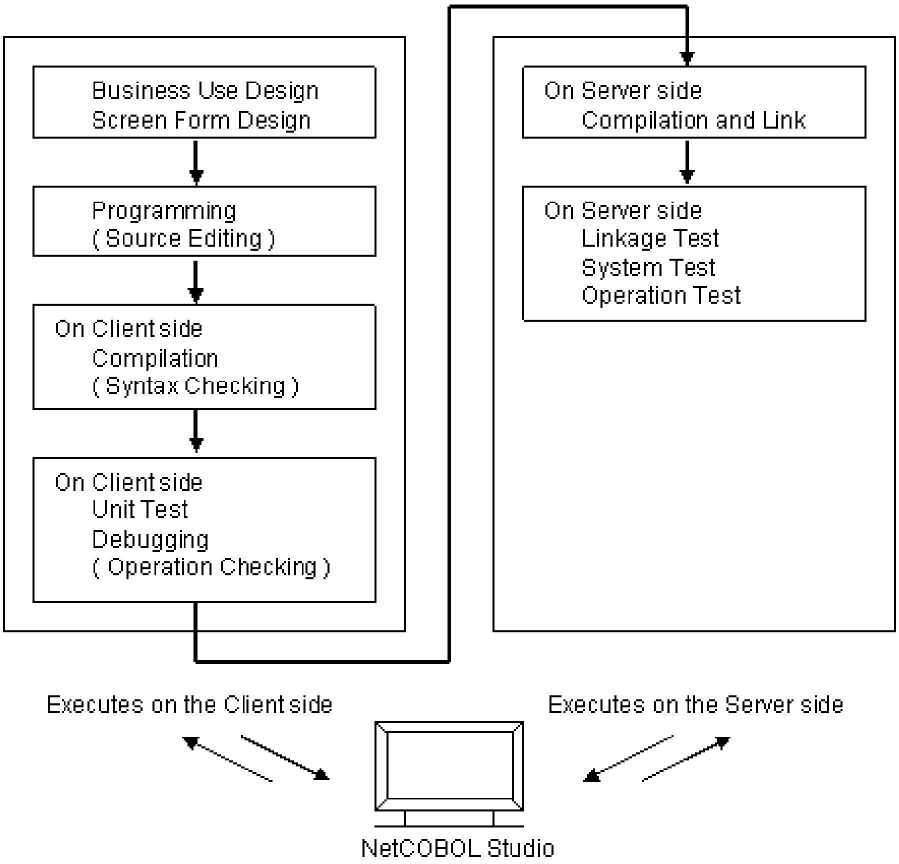
On the other hand, Windows system is easily available as a personal machine and the developer can occupy its GUI-based environment exclusively.

Remote development solves the above problems by performing development tasks on Windows system as far as possible.

- You can develop COBOL application efficiently using GUI-based development environment.
- You can reduce the load on the system at server side by performing development tasks such as source editing etc. on the client side as far as possible.

22.1.3 Flow of Remote Development

The flow of remote development is as follows:



First, create a project with NetCOBOL Studio on the client side, and edit sources.

If possible, perform compilation, unit test/ debugging on the client side.

Then compile, debug and test your application on the server side by using remote build and remote debug function of NetCOBOL Studio.

22.1.4 Notes on Remote Development

Generally, COBOL programs have high portability and, in many cases, you can perform tasks from the creation of your program, to unit testing on the client side.

However, if your program contains server-specific features or platform dependent features, you must test your program on the server side.

Following items are examples of platform dependent features:

Literals or hexadecimal literals

If the client side and the server side use different character encoding, a value that can be compiled on one side may cause a compilation error on the other side.

Key sequence or sort key sequence for character comparison or indexed files

The result of descending order of characters may differ based on the difference between the runtime character encoding on the client side and the server side.

Class conditions

If the runtime character encoding on the client side and the server side are different, the identification results may also differ.

Printing

Available characters and fonts differ according to the target platform

Database functions

The execution results may also differ based on the differences in the database products being used.

Web linkage function

The available functions differ depending on the target platform. In addition, Windows and UNIX systems have different file and path name rules.

Note

This system does not support the Web linkage function.

22.2 Remote Development Support Function

Server side

The NetCOBOL product for the server side system provides "NetCOBOL Remote Development Service" to support remote development. The NetCOBOL Remote Development Service accepts a request from NetCOBOL Studio on the client side and performs development tasks on the server side. For security reasons, this service is not configured to start automatically in its installation. You must start the service before you use remote development functions. See "22.3 NetCOBOL Remote Development Service" for details.

In remote development, a specified account is used to log on the server side system, and development tasks on the server side system are performed with that account.

While performing Remote development, the PATH environment variable in the server side account must include the install folder of NetCOBOL in the server side.

In addition, when remote build is performed from NetCOBOL Studio, the build tasks are executed in the same environment as the NetCOBOL Command Prompt of the server side NetCOBOL.

If you want to perform remote build with your own environment variables, edit the server side environment variables with [System] - [Advanced] in the control panel.

For example, if you want to perform remote build for copy book folder "C: \MyCopybook" in the state specified by COB_COBCOPY environment variables, take the following steps:

1. Log on the server side system with the account to perform remote development.
2. Open [System] - [Advanced settings] in the Control Panel, click the [Environment Variables] button to open the "Environment Variables” dialog.
3. In the "Environment Variables" dialog, add COB_COBCOPY environment variable as a user environment variable and set its value to "C:\MyCopybook".

Generally the steps above are enough to set the environment at the time of remote build. However caution is required while setting the PATH environment variables.

There are system environment variable and user environment variable among environment variables for Windows.

In general, user environment variables are given higher priority to system environment variables. However, for PATH environment variables, the PATH for user environment variable is automatically communicated after the PATH for system environment variable.

Therefore ,in case you change Makefile on the server side to execute your own command, even if you set the folder in which the command exists as the PATH for user environment variable, if a different command of the same name exists on the system PATH environment variable then that command gets executed.

Here, COB_RDENV_X64 environment variable can be used to add your own PATH prior to the system PATH in case it is required. In remote build procedure for NetCOBOL, in case the value of environment variable COB_RDENV_X64 is an existing file path, then that file is considered as batch file and executed by CALL batch command before performing build.

For example, if you create a batch file "C:\MyTools\myenv.bat" with following contents:

```
path C:\MyCommand;%PATH%
```

At this time, if the value of environment variable COB_RDENV_X64 is set in "C:\MyTools\myenv.bat ", then remote build tasks are performed after that batch file is called. As a result, in remote build, path "C:\MyCommand" is given higher priority to the PATH of system environment variable.

Client side

NetCOBOL Studio on the client side is used as the development environment for remote development.

NetCOBOL Studio provides the following remote development features:

- It transfers COBOL resources contained in a COBOL project and generates a Makefile to build the project on the server side.
- It executes the above Makefile on the server side and builds the COBOL application for the server side platform.
- It runs the COBOL application on the server side and debugs it.

Refer to the "NetCOBOL Studio User's Guide" for details.

Server side and Client side Combinations

Refer to the software guide for the supported combinations of the server side and the client side NetCOBOL products.

22.3 NetCOBOL Remote Development Service

The NetCOBOL Remote Development Service (referred as "Remote Development Service" after in this chapter) must be run on the server side system to make use of the remote development functions from NetCOBOL Studio on the client side. The Remote Development Service accepts a request from NetCOBOL Studio, logs on the server side system with the specified account and performs tasks on the server side system with the account.

For security reasons, the Remote Development Service is not configured to start automatically in its installation. You must start the Remote Development Service before you use remote development functions.

22.3.1 Notes on Security

For security reasons, the Remote Development Service is not configured to start automatically in its installation.

In order to maintain security, make sure to disclose the Remote Development Service only for the limited period required. When the release of the Remote Development Service is stopped, restore the changes made in settings of firewall or "Startup type" of Remote Development Service.

Make sure to use remote development functions with the Remote Development Service only inside safe network such as intranet where its security is appropriately managed.

22.3.2 Starting and Stopping the Remote Development Service

This section explains how to start and stop the Remote Development Service.

Starting the Remote Development Service

To start the Remote Development Service, log on the server side system with an administrator account and take the following steps:

1. From the Windows start menu, select [Administrative Tools]-[Services] to display "the Services"
2. From the list of services, select "NetCOBOL Remote Development Services".
3. In the window menu, select [Action]-[Properties], and open the "NetCOBOL Remote Development Services Properties" dialog.
4. In the "NetCOBOL Remote Development Services Properties" dialog, select the [General] tab.
5. Click the "Start" button.

6. If you want the Remote Development Service to start automatically when the system is started, change the "Startup type" to "Automatic".

The Remote Development Service uses port 61999 by default, to disclose its services. If port 61999 is already being used in the system, the port number must be changed. Refer to the description of port setting in "22.3.4 Configuring the Remote Development Service" for details.

If the Windows Firewall or other firewall software is running on the server, you must configure it in a way that it doesn' t block the port which is used by the Remote Development Service. The configuration method for the firewall differs according to the type of firewall software being used. Refer to the document on each firewall software.

Please note the items explained in "22.3.1 Notes on Security" before starting the Remote Development Service.

Stopping the Remote Development Service

To stop the Remote Development Service, log on the server side system with an administrator account and take the following steps:

1. From the Windows start menu, select [Administrative Tools]-[Services] to display the "Services".
2. From the list of services, select "NetCOBOL Remote Development Services".
3. In the window menu, select [Action]-[Properties], and open the "NetCOBOL Remote Development Services Properties" dialog.
4. In the "NetCOBOL Remote Development Services Properties" dialog, select the [General] tab.
5. Click the "Stop" button.
6. If you do not want the Remote Development Service to start automatically when the system is started, change the "Startup type" to "Manual"

When Remote Development Service no longer needs to be disclosed, restore the settings changes that were made in the firewall software or so.

22.3.3 Log files of the Remote Development Service

This section explains the log files output by the remote development service.

Contents of the Log file

The following information is recorded in the log file:

- Connection start date and time
- Client IP address
- Account name
- Whether or not log-on was successful
- Connection end date and time

The commands executed under each user account after logon are not recorded.

The path of the Log file

The path of the log file is as follows:

```
%ProgramData%\Fujitsu\NetCOBOL\RDS\Log\rds.log
```

%ProgramData% is a common application data folder for Windows. The default path for this is as follows:

Windows	default path
Windows Vista or later	C:\ProgramData
Windows Server 2003 or earlier	C:\Documents and Settings\All Users\Application Data

You can change the folder to output the log file by configuration of the Remote Development Service. See the explanation for the logdir setting in "22.3.4 Configuring the Remote Development Service" for details.

You must log on with an administrative account to change or delete log files.

Generations of the Log file

When the size of a log file reaches the maximum size, a backup is made for that log file and a new log file is created.

You can change the maximum size of log files by configuration of the Remote Development Service. Refer to the explanation for the maxlogsize setting in "22.3.4 Configuring the Remote Development Service" for details.

Backup files are located in the same folder as the log file output folder and have the below name.

```
rds.<sequence number>.log
```

Where <sequence number> is a number starting at 001 and has a maximum value of 999. When new backup file is created, a new sequence number is allocated for it. This sequence number is the next sequence number of the backup file with the most recently updated time amongst the backup files in the same folder. If there is no backup file in the same folder, the sequence number 001 is allocated. 001 is also regarded as the next number of 999.

When a new backup file with sequence number n is created, only the backup files which have sequence number between (n - the number of backup generation + 1) and n are retained. All other backup files are deleted. For example, if the new backup file is rds. 007.log and the number of backup generations is 3, all backup files are Deleted except for rds.005.log, rds.006.log, and rds.007.log.

You can change the number of backup generation by configuration of the Remote Development Service. See the explanation for the maxloggen setting in "22.3.4 Configuring the Remote Development Service" for details.

22.3.4 Configuring the Remote Development Service

This section describes the Remote Development Service settings that can be changed and explains how to set them.

The following settings can be changed:

- Port number to be used
- The output folder, maximum size, and number of backup generations for log files

Specify the settings that you want to change as service start parameters. The Remote Development Service must then be restarted for the changes to take effect.

Changing setting of the Remote Development Service

To change setting of the Remote Development Service, log on the server side system with an administrator account and perform the following steps:

1. From the Windows start menu, select [Administrative Tools]-[Services] to display the "Services".
2. From the list of services, select "NetCOBOL Remote Development Services".
3. In the Window menu, select [Action]-[Properties], and open the "NetCOBOL Remote Development Services Properties" dialog.
4. In the "NetCOBOL Remote Development Services Properties" dialog, select the [General] tab.
5. Enter the settings in the "Start parameters".

Note

- To enter more than one setting in the "Start parameters", separate the settings with spaces.
- To enter a setting that contains a space, enclose it in quotation marks (").

```
[Example] /port:61999 "/logdir:C:\log data" /maxlogsize:128 /maxloggen:2
```

- The contents of "Start parameters" are not saved. If you configure the Remote Development Service to start automatically when the system is started, the Remote Development Service starts with default setting when the system is started next time.

Refer to the list below for the settings that can be entered in the "Start parameters".

Setting list of the Remote Development Service

The following table shows the settings that can be changed.

Setting name	Form to specify in start parameters	Default value	Explanation
port	/port:<number>	61999	Specify the TCP/IP port number used by the Remote Development Service. In <number>, specify the port number in decimal form.
logdir	/logdir:<path>	Windows shared application data folder (*)	Specify the folder to which remote development service log files are output. In <path>, specify the folder path. To use a path that contains spaces, enclose the start parameter in quotation marks (").
maxlogsize	/maxlogsize:<size>	128	Specify the maximum size of the Remote Development Service log file. In <size>, specify the maximum size in decimal form. Its unit is kilobytes. If 0 is specified, the Remote Development Service does not output log file
maxloggen	/maxloggen:<number of generations>	2	Specify the number of generations of Remote Development Service log file backups to be retained. In <number of generations>, specify the number of generations in decimal form. If n is specified, n backups (rds.xxx.log) are retained. If a number greater than 999 is specified, 999 is considered to be specified. If 0 is specified, no backup is retained.

* : This depends on the version of Windows. Refer to "The path of the log file" under "22.3.3 Log files of the Remote Development Service".

Appendix A Compiler Options

This appendix explains the COBOL compiler options.

The first section lists compiler options. Review this list to verify the compiler option to be specified, then use the second section to obtain the correct format for the option. The sections in this appendix are:

- List of Compiler Options
- Compiler Option Specification Formats
- Compiler Options That Can Be Specified in the Program Source Code Only
- Compiler Options for the Method Prototype Definition and Separated Method Definition

A.1 List of Compiler Options

This section lists compiler options.

Options that relate to compile time resources

A.2.17 FORMLIB(screen form definition file folder specification)

Specifies the folder for screen and form descriptor files.

A.2.20 LIB(library file folder specification)

Specifies the folder for library files.

A.2.38 REP(repository file I-O destination folder specification)

Specifies I-O folder for REP repository files.

A.2.39 REPIN(repository file input destination folder specification)

Specifies the input folder for repository files.

Options that relate to compile listings

A.2.7 COPY(library text display)

Displays library text.

A.2.21 LINECOUNT(number of rows per page of the compile list)

Specifies the number of lines per page for compiler listings.

A.2.22 LINESIZE(number of characters per row in the compile list)

Specifies the number of characters per line for compiler listings.

A.2.23 LIST(determines whether to output the object program listings)

Determines whether to output the object program listings.

A.2.25 MAP(whether data map listings, program control information listings, and section size listings should be output)

Determines whether to output the data map listing, program control information listing and section size listings.

A.2.26 MESSAGE(whether the optional information list and statistical information list should be output for separately compiled programs)

Determines whether to output the option information listing and compile unit statistical information listing.

A.2.32 NUMBER(source program sequence number area specification)

Specifies the sequence number area of a source program.

A.2.35 PRINT(whether compiler listing should be output and the output destination specification)

Determines whether to output compiler listings and specifies the output folder of each compiler listing.

A.2.46 SOURCE(whether a source program listing should be output)

Determines whether to output a source program listing.

A.2.57 XREF(whether a cross-reference list should be output)

Determines whether to output the cross-reference listing.

Options that relate to compile messages

A.2.6 CONF(whether messages should be output depending on the difference between standards)

Determines whether to output a message indicating incompatibility between the old and new standards.

A.2.15 FLAG(diagnostic message level)

Specifies the level of a diagnostic message.

A.2.16 FLAGSW(whether pointer messages to language elements in COBOL syntax should be displayed)

Determines whether to display a message for the language elements in COBOL syntax.

Options that relate to source program interpretation

A.2.1 ALPHAL(lowercase handling (in the program))

Determines how to treat lowercase letters in user-defined words.

A.2.4 BINARY(binary data item handling)

Determines how to treat binary data items.

A.2.10 CURRENCY(currency symbol handling)

Determines how to treat currency symbols.

A.2.12 DUPCHAR (JIS non-Kanji minus sign specification)

Specifies a JIS non-Kanji minus sign.

A.2.19 LANGLVL(ANSI COBOL standard specification)

Specifies the ANSI COBOL standard.

A.2.18 INITVALUE(initializing items without a VALUE phrase in WORKING-STORAGE)

WORKING-STORAGE item initialization.

A.2.29 NCW(Japanese-language character set specification for the user language)

Specifies a character set of national language user-defined words.

A.2.31 NSPCOMP(Japanese-language space comparison specification)

Determines how to treat the national spaces and ANK spaces.

A.2.36 QUOTE/APOST(figurative constant QUOTE handling)

Determines how to treat the figurative constant QUOTE.

A.2.40 RSV(the type of a reserved word)

Specifies the type of reserved words.

A.2.42 SCS(code system of source file)

Specifies the code type of the source file.

A.2.43 SDS(whether signs in signed decimal data items should be converted)

Selection of sign adjustment for signed decimal data items.

A.2.44 SHREXT(determines how to treat the external attributes in a multithread program)

Determines how to treat the external attributes in a multithread program.

A.2.47 SQLGRP(SQL host variable definition expansion)

SQL host variable definition expansion.

A.2.48 SRF(reference format type)

Specifies the type of reference formats.

A.2.51 STD1(alphanumeric character size specification)

Specifies the collating sequence of alphanumeric characters.

A.2.52 TAB(tab handling)

Specifies a tab interval.

A.2.58 ZWB(comparison between signed external decimal data items and alphanumeric data items)

Compares signed zoned decimal data items with alphanumeric data items .

Options that relate to source program analysis

A.2.41 SAI(whether the source analysis information file should be output)

Determines whether to output a SAI file.

Options that relate to object program generating

A.2.2 ARITHMETIC (Selects the operation mode)

Specified the operation mode

A.2.3 ASCOMP5(interpretation of binary items)

Specifies interpretation of binary data items

A.2.9 CREATE(specifies a creating file)

Specifies a created file.

A.2.11 DLOAD(program structure specification)

Specifies a program structure.

A.2.13 ENCODE (encoding form specification)

Specifies an encoding form.

A.2.24 MAIN(main program/sub-program specification)

Specifies that a COBOL source program is a main program or subprogram.

A.2.27 MODE(ACCEPT statement operation specification)

Specifies operation with the ACCEPT statement.

A.2.28 NAME(whether object files should be output for each separately compiled program)

Specifies a compiler option when source file with two or more compiling units is compiled.

A.2.30 NSP (handling of spaces related to national data item)

Specifies a space related to a national data item.

A.2.33 OBJECT(whether an object program should be output)

Determines whether to output the object programs.

A.2.34 OPTIMIZE(global optimization handling)

Determines whether to create globally optimized object programs.

A.2.37 RCS(runtime code set specification)

Runtime code set specification.

A.2.54 THREAD(specifies multithread program creation)

Specifies a multi-threaded program creation.

Options that relate to runtime control

A.2.14 EQUALS(same key data processing in SORT statements)

Specifies a method of processing data having the same key in the SORT statement.

A.2.56 TRUNC(Truncation Operation)

Selection of column truncation.

Options that relate to runtime resources

A.2.45 SMSIZE(Memory capacity used by PowerBSORT)

Specifying the memory capacity used by PowerBSORT

A.2.49 SSIN(ACCEPT statement data input destination)

Specifies the input destination of data specified in the ACCEPT statement.

A.2.50 SSOUT(DISPLAY statement data output destination)

Specifies the output destination of data specified in the DISPLAY statement.

Options that relate to debugging functions

A.2.5 CHECK(whether the CHECK function should be used)

Determines whether to use the CHECK function.

A.2.8 COUNT(whether the COUNT function should be used)

Determines whether to use the COUNT function.

A.2.53 TEST(whether the debugging function of NetCOBOL Studio and the COBOL Error Report should be used)

Determines whether to use the debugging function of NetCOBOL Studio and the COBOL Error Report.

A.2.55 TRACE(whether the TRACE function should be used)

Determines whether to use the TRACE function.

A.2 Compiler Option Specification Formats

Compiler option specification methods and their priorities

Compiler options can be specified by:

1. Using the NetCOBOL Studio
2. Using the -WC option
3. Using command options other than the -WC option
4. Using a compiler directing statement within a source program (@OPTIONS)

See

Refer to "NetCOBOL Studio User's Guide" for details.

Their priorities are in the order of 4, 3, 2, and 1. There are several methods for specifying option files and -WC options. They are chosen as being effective options according to the following rule.

-WC option

The -WC option can be specified either by:

1. Using the environment variable "COB_OPTIONS"
2. Using the compile command

The specification priority is in the order of 2 and 1. The -WC option can be specified more than once. All specified compile options are valid.

If multiple compile options of the same type are specified, the last specified compile option with the highest specification priority will be valid.

Example

Example 1

```
set COB_OPTIONS=-WC,"MESSAGE,OPTIMIZE"
cobol -WC,"NOOPTIMIZE" a.cob
```

The MESSAGE compile option and NOOPTIMIZE compile option are valid.

Example 2

```
cobol -WC,"CHECK(ALL)" -WC,"MESSAGE" -WC,"CHECK(NUMERIC)" a.cob
```

The MESSAGE compile option and CHECK(NUMERIC) compile option are valid.

Specification by the compiler option

The specification methods may be restricted depending on the compiler option used. Specify the compiler options referencing the following marks:

STUDIO	The compiler option can be specified on the Compiler Options page of NetCOBOL Studio.
-WC	The compiler option can be specified with the -WC command option.
@	The compiler option can be specified in the compiler directing statement.

Other than the above, in the case displayed as "(@)", compiler directing specification is possible but only when plural source code is not contained in the source file.

Specify the folder name and file name

Specify the absolute path name or relative path name for a folder and file name.

A folder or file name that includes one of the following 10 characters must be enclosed in double quotes ("):

```
Space   +   ,   ;   =   [   ]   (   )   '
```

Note

The length of specifiable folder name is up to 2,048 bytes including the separator "; (semi-colon)".

Current folder

The folder where COBOL command is executed is called the current folder.

When using the NetCOBOL Studio, the current folder is the project folder.

Note

The runtime current folder varies with the runtime environment and the operation of the application.

A.2.1 ALPHAL(lowercase handling (in the program))

```
STUDIO,-WC,@
```

```
{  ALPHAL[(   { ALL  }   )]  }
              { WORD }
   NOALPHAL
```

Specify ALPHAL to treat lowercase letters in the source program as uppercase letters. Otherwise, specify NOALPHAL.

ALPHAL treats character constants as follows:

- ALPHAL(ALL)
  In COBOL words, lowercase characters are treated as equivalent to uppercase characters. Lowercase characters in the program name literal, and in constants in CALL statement, CANCEL statement, ENTRY statement, and INVOKE statement, are treated as equivalent to uppercase characters.

- ALPHAL(WORD)
  In COBOL words, lowercase characters are treated as equivalent to uppercase characters. Lowercase characters in the constants including the program name literal, the constants in CALL statement, CANCEL statement, ENTRY statement, and INVOKE statement, are distinguished from uppercase characters.

- NOALPHAL
  Lowercase characters in COBOL words or constants are distinguished from uppercase characters.

See

- "COBOL Word" in Chapter 1, "COBOL Language Reference" for details on COBOL words

- "9.3.4 Compiling Programs"

A.2.2 ARITHMETIC (Selects the operation mode)

```
STUDIO,-WC,@
```

```
ARITHMETIC(   { 18          }   )
              { 31 [, INF]  }
```

This option specifies the 18-digit or 31-digit operation mode.

- ARITHMETIC(18)
  Specifies 18-digit compatible operation mode.
  18-digit mode is used for compatibility between systems and between Version Levels.

- ARITHMETIC(31)
  Specifies 31-digits extension operation mode.
  Up to 31 digits can be used for numeric items, edited numeric items and numeric literals in this mode. 31-digit extension mode is used for functions that have been enhanced to use more than 18 digits.

- ARITHMETIC(31,INF)
  Specifies 31-digit extension operation mode plus the output of I-level diagnostic messages for the following:

  - When the compilation message "JMN3024I-W The intermediate result cannot contain more than 30 digits. The intermediate result is assumed to have 30 digits." is output in 18-digit compatible operation mode.

- When the intermediate result attribute (fixed-point or floating-point) is different from the result in 18-digit compatible operation mode.

Note

- When ARITHMETIC(31) is specified, compiler option BINARY can specify only BINARY(WORD,MLBON).
- Specify ARITHMETIC(18) for compatibility with V10.2.0 or earlier, or with other system.
- Refer to "1.7 Operation mode" in "NetCOBOL Language Reference" for details.

A.2.3 ASCOMP5(interpretation of binary items)

```
STUDIO,-WC,@
```

```
             ┌ NONE   ┐
ASCOMP5(    {  ALL     }  )
             │ BINARY │
             └ COMP   ┘
```

Specify how USAGE BINARY/COMP/COMPUTATIONAL items are interpreted.

- ASCOMP5(NONE)
  Interprets the declarations as they are.
- ASCOMP5(ALL)
  Items declared as USAGE BINARY, USAGE COMP, or USAGE COMPUTATIONAL are interpreted as USAGE COMP-5.
- ASCOMP5(BINARY)
  Items declared as USAGE BINARY are interpreted as USAGE COMP-5.
- ASCOMP5(COMP)
  Items declared as USAGE COMP or USAGE COMPUTATIONAL are interpreted as USAGE COMP-5.

Note

- Note that care is required if you use this option with code that is aware of internal binary numeric formats because the ALL, BINARY, and COMP options will cause the internal formats to change.
- NONE must be specified when using the CBL subroutine.

A.2.4 BINARY(binary data item handling)

```
STUDIO,-WC,@
```

```
          ┌ WORD[,  { MLBON  }  ] ┐
BINARY(  {          { MLBOFF }     }  )
          └ BYTE                  ┘
```

BINARY(WORD) assigns the elementary item of binary data to an area length of 2, 4, or 8 words; BINARY(BYTE) assigns the elementary item of binary data to an area length of 1 to 8 bytes.

How to treat the high order end bit of an unsigned binary data item can also be specified.

- BINARY(WORD,MLBON)
  The high order end bit is treated as a sign.

- BINARY(WORD,MLBOFF)
  The high order end bit is treated as a numeric value.

Note

- BINARY(BYTE) cannot be used in a class definition.
- If BINARY(BYTE) is specified, the high order end bit is treated as a numeric value.
- BINARY(WORD, MLBON) cannot be excluded when specifying ARITHMETIC(31).

The following table shows the relationship between the number of declared digits and the area length.

Table A.1 Length of Assigned Area from Number of PIC Digits

Number of PIC Digits		Assigned Area Length	
Signed	Unsigned	BINARY(BYTE)	BINARY(WORD)
1 - 2	1 - 2	1	2
3 - 4	3 - 4	2	2
5 - 6	5 - 7	3	4
7 - 9	8 - 9	4	4
10 - 11	10 - 12	5	8
12 - 14	13 - 14	6	8
15 - 16	15 - 16	7	8
17 - 18	17 - 18	8	8
19 - 28 (*)	19 - 28	-	12
29 - 31 (*)	29 - 31	-	16

*: For 31-digit extension operation mode.

A.2.5 CHECK(whether the CHECK function should be used)

```
STUDIO,-WC,@
```

```
⎧ CHECK[ ( [n] [,ALL] [,BOUND] [,ICONF] [,NUMERIC] [,PRM] ) ] ⎫
⎩ NOCHECK                                                      ⎭
```

To use the CHECK function, specify CHECK; otherwise, specify NOCHECK.

n indicates the number of times a message is displayed. Specify n with an integer 0 to 999,999. The default value is 1.

- CHECK (ALL)

  Checks BOUND, ICONF NUMERIC, and PRM.

- CHECK(BOUND)
  Checks whether subscripts, indexes, and reference modifications are out of range.

- CHECK(ICONF)
  Check whether parameters in INVOKE statements match those in the methods being invoked.

- CHECK(NUMERIC)
  Checks data exceptions (whether the value matching the attribute format is contained in the numeric data item and whether the divisor is zero).

- CHECK(PRM)
  At compile time, carry out the following checks for data items used in the USING and RETURNING phrases of the CALL statement (except for CALL identifier-name) that calls a internal program, and for data items used in the USING and RETURNING phrases of the internal program.

  - The number of USING phrase parameters match.
  - Existence match of the RETURNING phrase parameter.
  - If the data item is not an object reference, data item length is only verified if the length can be determined at compile time.
  - If the data item is an object reference, the class name specified in the USAGE OBJECT REFERENCE clause, FACTORY, and ONLY match.

  At execution time, the following checks are performed on data items used in the USING and RETURNING phrases of CALL statements that call external programs, and also for data items used in the USING and RETURNING phrases of the external program.

  - The numbers of items in the USING phrases much match as must the lengths of the corresponding data items. However, errors may not be found if the difference in the number of USING phrase parameters is greater than three.
  - The lengths of the RETURNING phrase parameters match. When no RETURNING phrase is specified, PROGRAM-STATUS is implicitly passed and is interpreted as if a 4-byte long data item is specified.

Note

- While the CHECK function is in use, program processing continues until a message is displayed up to n times. However, the program may fail to operate as expected if an error occurs (for example, memory corruption). If 0 is specified for n, program processing continues regardless of the number of times a message is displayed.
- If CHECK is specified, the above check processing is incorporated into the object program. Therefore, execution performance is decreased. When you have finished debugging the program, specify NOCHECK, then recompile the program.
- In arithmetic statements with an ON SIZE ERROR or NOT ON SIZE ERROR phrase, CHECK(NUMERIC) does not check for a zero divisor as the COBOL code already handles that situation.
- A CHECK (NUMERIC) data exception check is made when a display decimal item or a packed decimal item is read, or when an alphabetic or alphanumeric data item or a group item is moved to a display decimal item or a packed decimal item. However, the checks are not made in the following cases:

  - A table element that is specified with ALL as the subscript.
  - Key items in the SEARCH ALL statement (except when the key item is one-dimensional and only one WHEN phrase is specified for the SEARCH statement)
  - Key items in the SORT/MERGE statement
  - Host variables used in the SQL statement
  - The BY REFERENCE parameter for a CALL statement, INVOKE statement, and in-line invocation
  - The following intrinsic function arguments:

    FUNCTION ADDR

    FUNCTION LENG

    FUNCTION LENGTH

  - Moving from an alphanumeric data item or group item to an object property that is either a display decimal item or a packed decimal item.

See

"Using the CHECK Function" in the "NetCOBOL Debugging Guide"

A.2.6 CONF(whether messages should be output depending on the difference between standards)

```
STUDIO,-WC,@
```

```
{ CONF(  { 68  }  ) }
{        { 74  }    }
{        { OBS }    }
{ NOCONF            }
```

Specify CONF to indicate incompatibility between the old and new COBOL standards; otherwise, specify NOCONF. If CONF is specified, an incompatible item is indicated by I-level diagnostic messages.

- CONF(68)
  Indicate items that is interpreted differently between '68 ANSI COBOL and '85 ANSI COBOL.
- CONF(74)
  Indicate items that is interpreted differently between '74 ANSI COBOL and '85 ANSI COBOL.
- CONF(OBS)
  Indicates obsolete language elements and functions.

CONF is effective when a program created according to the existing standard is changed to `85 ANSI COBOL.

Note

The compiler options CONF(68) and CONF(74) are effective only if the compiler option LANGLVL(85) is specified.

See

"A.2.19 LANGLVL(ANSI COBOL standard specification)"

A.2.7 COPY(library text display)

```
STUDIO,-WC,@
```

```
{ COPY   }
{ NOCOPY }
```

To display library text incorporated by the COPY statement in the source program listing, specify COPY; otherwise, specify NOCOPY.

See

"A.2.46 SOURCE(whether a source program listing should be output)"

Note

COPY is only effective when the compiler option SOURCE is specified.

A.2.8 COUNT(whether the COUNT function should be used)

```
STUDIO,-WC,@
```

```
{ COUNT   }
{ NOCOUNT }
```

Specify whether the COUNT function should be used (COUNT) or not (NOCOUNT).

Note

- If COUNT is specified, the process for outputting the COUNT information is incorporated into the object program. Therefore, execution performance is degraded. When debugging is completed, specify NOCOUNT then perform re-compilation.
- COUNT cannot be specified with compiler option TRACE or the -Dr option. If both are specified at the same time, the latter option is valid.

See

"Using the COUNT Function" in the "NetCOBOL Debugging Guide"

A.2.9 CREATE(specifies a creating file)

```
-WC,@
```

```
CREATE [ (    { OBJ }   ) ]
              { REP }
```

Specify the purpose of compilation, that is, either object creation (CREATE (OBJ)) or repository creation (CREATE (REP)).

If CREATE (REP) is specified, the PROCEDURE DIVISION is not analyzed and the object program is not generated.

Note

CREATE (REP) is only for class definition compilation. CREATE (OBJ) is always assumed for compilation other than class definition compilation.

A.2.10 CURRENCY(currency symbol handling)

```
STUDIO,-WC,@
```

```
CURRENCY (      { $               }  )
                { currency-symbol }
```

Specify CURRENCY($) to use "$" as the character used for the currency symbol; specify CURRENCY(currency-symbol) to use another symbol. If CURRENCY(currency-symbol) is specified, refer to the CURRENCY SIGN clause explained in the "COBOL Language Reference" for the currency symbols that can be used.

Note

The character that can be used as currency symbol is \(X'5C') and $(X'24').

A.2.11 DLOAD(program structure specification)

```
STUDIO,-WC,@
```

```
{ DLOAD   }
{ NODLOAD }
```

Specify whether the program structure is a dynamic program structure (DLOAD) or not (NODLOAD).

See

- "4.3 Program Structure"
- "9.1.3 Dynamic Program Structure"
- "16.2 Using Dynamic Program Structure"

A.2.12 DUPCHAR (JIS non-Kanji minus sign specification)

```
STUDIO,-WC,@
```

```
DUPCHAR( { STD } )
         { EXT }
```

Specify whether the JIS non-Kanji minus sign is "MINUS SIGN (STD)" or "FULLWIDTH HYPHEN-MINUS (EXT)" when source programs and library texts are created in UTF-8.

	UTF-8	UTF-16
MINUS SIGN	X"E28892"	X"2212"
FULLWIDTH HYPHEN-MINUS	X"EFBC8D"	X"FF0D"

See

- J.5 About JIS non-Kanji minus sign

A.2.13 ENCODE (encoding form specification)

```
STUDIO,-WC,@
```

```
ENCODE ( { SJIS } [, { SJIS                                } ] )
         {      }     {                                     }
         { UTF8 }     { { UTF16 } [, { LE } ]               }
                      { { UTF32 }    { BE }                 }
```

Specify the encoding of the alphanumeric data item and the national data item.

Endian is specified when the encoding form of national item is Unicode in the first sub operand encode of an alphanumeric character item, in the second sub operand encoding of a national item and in the third sub-operand.

- ENCODE (UTF8, UTF32): The encoding of an alphanumeric character item is specified as UTF8 and the encoding of a national item is specified as UTF32.

When the national item is UTF16 or UTF32, then specify little endian or big endian.

- ENCODE (UTF8, UTF32, BE): The national item is big endian.

When the various sub operands are omitted, encoding is as follows.

- When the encoding of the national item is omitted, it defaults as follows.
  - When the encoding of an alphanumeric character item is SJIS, then the encoding of the national item is SJIS.
  - When the encoding of an alphanumeric character item is UTF8, then the encoding of the national item is UTF16.
- When the encoding of a national item is Unicode and endian is not specified, then it is in accordance with the system's endian.

When compilation option ENCODE is specified, then the runtime code will be Unicode. However, when compilation option RCS is specified, then it is given priority.

Point

When compile option ENCODE (UTF8, UTF16) or ENCODE (UTF8, UTF32) is specified explicitly or implicitly, then spaces related to a national item become alphabetic character spaces. This handling can be done by specifying the compiler option NSP.

See

- "A.2.30 NSP (handling of spaces related to national data item)"

A.2.14 EQUALS(same key data processing in SORT statements)

```
STUDIO,-WC,@
```

```
 ┌ EQUALS   ┐
{           }
 └ NOEQUALS ┘
```

When records that have the same key are input to a SORT statement, specify EQUALS to ensure that the SORT output record order is identical to the input record order. Otherwise, specify NOEQUALS.

If NOEQUALS is specified, the sequence of records having the same key is not defined when the SORT statement outputs records.

Note

If EQUALS is specified, special processing is performed to ensure that the input sequence, for same-key records, is preserved during the sorting operation. Therefore, execution performance decreases.

A.2.15 FLAG(diagnostic message level)

```
STUDIO,-WC,@
```

```
         ┌ I  ┐
FLAG(   {  W   }   )
         └ E  ┘
```

Specify the diagnostic messages to be displayed.

- FLAG(I)
  Displays all diagnostic messages.
- FLAG(W)
  Displays diagnostic messages of only W-level or higher.

- FLAG(E)
  Displays diagnostic messages of only E-level or higher.

Note

The diagnostic message specified in the compiler option CONF is displayed regardless of the FLAG specification.

See

"A.2.6 CONF(whether messages should be output depending on the difference between standards)"

A.2.16 FLAGSW(whether pointer messages to language elements in COBOL syntax should be displayed)

```
STUDIO,-WC,@
```

```
              {    { STDM }              }
              {  [ { STDI } ] [,RPW]     }
  {           {    { STDH }              }       }
  {  FLAGSW(  {                          }   )   }
  {           {  SIA                     }       }
  {           {  GWS                     }       }
  {           {  GSS                     }       }
  {  NOFLAGSW                                    }
```

To display a message indicating a language construct in COBOL syntax, specify FLAGSW; otherwise, specify NOFLAGSW.

The following are language constructs that can be indicated:

- FLAGSW(STDM)
  Language elements that are not in the minimum subset of the standard COBOL85

- FLAGSW(STDI)
  Language elements that are not in the intermediate subset of the standard COBOL85

- FLAGSW(STDH)
  Language elements that are not in the high subset of the standard COBOL85

- FLAGSW(RPW)
  Language elements of the report writer function of the standard COBOL85

- FLAGSW(SIA)
  Language elements that are not in the range of Fujitsu System Integrated Architecture (SIA)

Use FLAGSW(SIA) to create a program that will be executed in another system.

Note

FLAGSW suboperands {STDM | STDI | STDH} and RPW cannot be omitted simultaneously.

A.2.17 FORMLIB(screen form definition file folder specification)

```
STUDIO
```

FORMLIB (*folder-name*[;*folder-name*]...)

If the COPY statement with IN/OF XMDLIB specified is used to fetch a record definition from the screen and form descriptor, specify the folder containing the screen and form descriptor files.
If the screen and form descriptor files exist in more than one folder, specify the folder names separated by semicolons. The folders are searched for in the order specified.

If the option is specified in duplicate, the folder search sequence is as follows:

1. Folders specified in the -m option
2. Folders specified in the compiler option FORMLIB
3. Folders specified in the FORMLIB environment variable
4. Current Folders

See

- "3.5.2.12 -m(Specify the FORM descriptor file folder)"
- "1.2.1 Setting Environment Variables"

A.2.18 INITVALUE(initializing items without a VALUE phrase in WORKING-STORAGE)

```
STUDIO,-WC,@
```

$$\left\{\begin{array}{l}\text{INITVALUE(}\mathit{xx}\text{)}\\ \underline{\text{NOINITVALUE}}\end{array}\right\}$$

Specify whether or not items in WORKING-STORAGE that do not have VALUE phrases are initialized to a specified value

- INITVALUE(xx)
  Specify a double-digit hexadecimal number for xx. The value of xx cannot be omitted. This value will be used to initialize items in WORKING-STORAGE that do not have a VALUE phrase when WORKING-STORAGE is initialized.

- NOINITVALUE
  No initializing action is taken for items that do not have a VALUE phrase.

A.2.19 LANGLVL(ANSI COBOL standard specification)

```
STUDIO,-WC,@
```

$$\text{FLAGLVL(}\left\{\begin{array}{l}\underline{85}\\ 74\\ 68\end{array}\right\}\text{)}$$

Specifies the standard to be used for cases where source program interpretation is different between the old and new COBOL standards.

- LANGLVL(85)
  '85 ANSI COBOL

- LANGLVL(74)
  '74 ANSI COBOL

- LANGLVL(68)
  '68 ANSI COBOL

A.2.20 LIB(library file folder specification)

```
STUDIO
```

LIB (*folder-name*[;*folder-name*]...)

If COPY statements are used, specifies the folder containing the COPY libraries. If libraries exist in more than one folder, specify the folder names separated by a semicolon. The folders are searched for in the order specified.

The following is the priority if the option is specified in duplicate:

1. -I option
2. Compiler option LIB
3. Environment variable COB_COBCOPY
4. Current folder

See

- "3.5.2.10 -I(Specify the library file folder)"
- "1.2.1 Setting Environment Variables"

A.2.21 LINECOUNT(number of rows per page of the compile list)

```
STUDIO,-WC,@
```

LINECOUNT(*n*)

Specifies the number of lines per page in the compiler listing. n can be specified with an integer of up to 3 digits. The default value is 60.

If this option is not specified, LINECOUNT(0) is assumed.

Note

If a value 0 to 12 is specified, display without editing.

A.2.22 LINESIZE(number of characters per row in the compile list)

```
STUDIO,-WC,@
```

LINESIZE(*n*)

Specify the maximum number of characters (value resulting from conversion of alphanumeric characters displayed in the list) for a line in a compiler listing. Value can be specified with 0, 80, or a 3-digit integer 120 or more. When 0 is specified, the compile list is output in one continuous line.

If this option is not specified, LINESIZE(0) is assumed.

Note

- The option information list and diagnosis message list are output with a fixed number (120) of characters regardless of the maximum number of characters specified in LINESIZE.
- The permitted maximum number of characters is 136. If a value greater than 136 is specified in LINESIZE, 136 is used.

A.2.23 LIST(determines whether to output the object program listings)

```
STUDIO,-WC,@
```

```
{ LIST   }
{ NOLIST }
```

Specify whether or not to output (LIST) an object program listing (NOLIST).

When the compiler program listing output is effective by the compiler option PRINT, object program listing is output.

See

- "3.5.2.5 -dp(Specify the compile list file folder)"
- "3.5.2.14 -P(Compile listing file name)"

A.2.24 MAIN(main program/sub-program specification)

```
-WC,@
```

```
{ MAIN[( { WINMAIN } )] }
{        { MAIN    }    }
{                       }
{ NOMAIN                }
```

Specify MAIN to use a COBOL source program as a main program; specify NOMAIN to use a COBOL source program as a subprogram.

- MAIN(WINMAIN)
  Specify this when you use a console window created by COBOL for the input/output target of an ACCEPT statement or DISPLAY statement, and when you use a message box for the output target of an error message at execution time.
- MAIN(MAIN)
  Specify this when you use the system console (command prompt window) for the input/output target of ACCEPT statement, DISPLAY statement and error message at execution time.

If only MAIN is specified, it is handled as MAIN(WINMAIN).

Note

- Please specify optional MAIN for COBOL source program that becomes a main program.
- Specify MAIN(MAIN) only when you need to specify the system console; otherwise, specify MAIN(WINMAIN).
- The MAIN option specified by the @OPTIONS compiler directing statement is only valid in the separately compiled program immediately after the compiler directing statement.

Information

When another language is used for the main program, and you want to use the system console as the input/output target of an ACCEPT statement, a DISPLAY statement, or a runtime error message, you can specify with the environment variable at execution time.

See

"5.3 Setting Runtime Environment Information"

A.2.25 MAP(whether data map listings, program control information listings, and section size listings should be output)

```
STUDIO,-WC,@
```

MAP

NOMAP

Specify MAP to output data map listings, program control information listings, and section size listings, otherwise specify NOMAP.

When the compiler program listing output is effective by the compiler option PRINT, these listings are output.

See

- "3.5.2.5 -dp(Specify the compile list file folder)"
- "3.5.2.14 -P(Compile listing file name)"

A.2.26 MESSAGE(whether the optional information list and statistical information list should be output for separately compiled programs)

```
STUDIO,-WC,@
```

MESSAGE

NOMESSAGE

To output an option information listing and to compile unit statistical information listing, specify MESSAGE; otherwise, specify NOMESSAGE.

A.2.27 MODE(ACCEPT statement operation specification)

```
STUDIO,-WC,@
```

MODE (STD | CCVS)

If an ACCEPT statement where a numeric data item is specified as the receiving item in the format of "ACCEPT unique-name[FROM mnemonic-name]" is executed, specify MODE(STD) to move a right-justified numeric data item to a receiving item.

Specify MODE(CCVS) to move a left-justified character string to a receiving item:

Note

If MODE(CCVS) is specified, only an external decimal data item can be specified as a receiving item in the ACCEPT statement.

A.2.28 NAME(whether object files should be output for each separately compiled program)

```
-WC,@
```

NAME

NONAME

These options are specified when one source file is compiled where two or more compiling units (PROGRAM) are described. The NAME option shows the output to the object files by separating each compiling unit. The NONAME option shows the output to a single object file.

Note

- The object file made specifying the NONAME option (the object at some compilations is stored in one file) cannot be used by the link command.
- If the NAME option is specified, the name of each object file is "each-separately compiled-program-ID.OBJ".

A.2.29 NCW(Japanese-language character set specification for the user language)

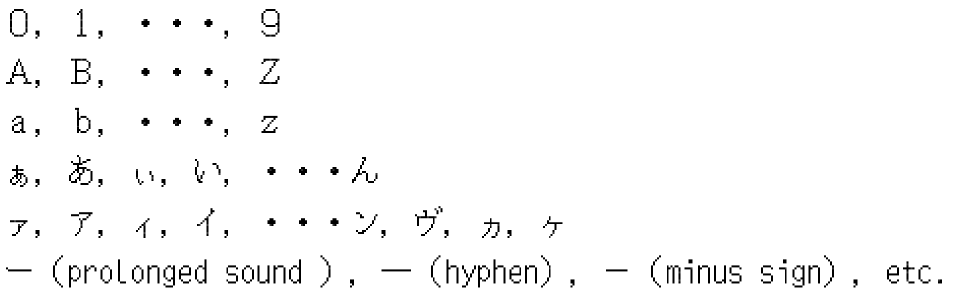
```
STUDIO,-WC,@
```

NCW ({ <u>STD</u> | SYS })

Specify this option for a national character set that can be defined as user-defined words. Specify NCW(STD) to use the national character set as a national character set common to systems; specify NCW(SYS) to use the national character set as a national character set of the computer.

If STD is specified, the following Japanese-language character sets can be used as the user language:

- JIS level 1
- JIS level 2
- JIS non-kanji (the following characters)



０，１，・・・，９
Ａ，Ｂ，・・・，Ｚ
ａ，ｂ，・・・，ｚ
ぁ，あ，ぃ，い，・・・ん
ァ，ア，ィ，イ，・・・ン，ヴ，ヵ，ヶ
ー（prolonged sound），―（hyphen），－（minus sign），etc.

If SYS is specified, the following Japanese-language character sets can be used as the user language:

- Character set with STD specified
- Extended character
- Extended non-kanji
- User-defined characters

- JIS Level non-kanji (the following characters cannot be used)

、　。　，　．　・　：　；　？　！　゛
゜　｀　＾　￣　＿　／　＼　｜　（　）
［　］　｛　｝　「　」　＋　＝　＜　＞
￥　＄　￠　￡　％　＃　＆　＊　＠

A.2.30 NSP (handling of spaces related to national data item)

```
STUDIO,-WC,@
```

{ NSP | <u>NONSP</u> }

Specify whether spaces related to a national data item are handled as national spaces (NSP) or as alphabetic character spaces (NONSP). In this option, encode is enabled for handling the trailing spaces related to a national data item of Unicode and a figurative constant SPACE. It is irrelevant when the encoding of national data items is other than Unicode.

A.2.31 NSPCOMP(Japanese-language space comparison specification)

```
STUDIO,-WC,@
```

NSPCOMP ({ <u>NSP</u> | ASP })

Use this option to specify how to treat a national space in a comparison. Specify NSPCOMP(NSP) to treat the national space as a national space; specify NSPCOMP(ASP) to treat the national space as an ANK space.

When NSPCOMP(ASP) is specified, an encode national space decided by compiler options is treated the same as the ANK blank in two bytes for UTF16, and treated to the same as the ANK blank in four bytes for UTF32.

NSPCOMP(ASP) is effective for the following comparisons:

- National character comparison with a national data item as an operand
- Character comparison with a group item as an operand

NSPCOMP(ASP) is not effective for the following comparisons:

- Comparison between group items that do not include any national data items
- Comparison between group items including an item whose attribute does not specify explicit or implicit display
- Comparison between group items including a national item with a different encode method
- Comparison of a national character that has an encode method different from the encode decided by the compiler options

Note

- In the character and national character comparisons performed by the INSPECT, STRING, and UNSTRING statements, and in index file key operations, national blank characters are not treated as equal to ANK blanks - even if the NSPCOMP(ASP) option is specified.
- If NSPCOMP(ASP) is specified, ANK blanks are treated as Japanese, one of the class condition, when the class condition is JAPANESE.

Information

A national space is equivalent to a 2-byte ANK space in the OSIV code system, whereas it is not in the Windows code system. Thus, specify compiler option NSPCOMP(ASP) for a COBOL program which has been used with an OSIV system to operate under this system.

A.2.32 NUMBER(source program sequence number area specification)

```
STUDIO,-WC,@
```

{ NUMBER
NONUMBER }

Specify this option to determine the value used as the line number in line information. The line information identifies each line of a source program within lists generated at compile time or runtime. Specify NUMBER to use the value of the sequence number area of the source program; specify NONUMBER to use the value generated by the compiler.

When copying text from a library, if a row number that is identical to a subsequent row number is specified, a unique corrected number is added in the same format as that of the statement containing the COPY qualification-value, as follows:

- NUMBER: If the sequence number area includes a nonnumeric character or if the sequence number is not in ascending order, the line number is changed to the previously correct sequence number + 1.
- NONUMBER: Line numbers are assigned in ascending order starting from 1. The increment between successive line numbers is 1. However, when the sequence number is a descending order, the corrected unique number is added in the same format as the COPY qualification-value.

See

"Source Program Listing" in the "NetCOBOL Debugging Guide"

Note

- If NUMBER is specified, a sequence of identical consecutive values is not regarded as an error.
- If NUMBER is specified, FREE cannot be specified for compiler option SRF. If FREE is specified for compiler option SRF, program operation is not guaranteed. See "A.2.48 SRF(reference format type)".
- If the sequence number is in descending order, the source analysis information file output with compiler option SAI specified can only be used by a project browser. See "A.2.41 SAI(whether the source analysis information file should be output)".

A.2.33 OBJECT(whether an object program should be output)

```
STUDIO,-WC,@
```

{ OBJECT [(*folder-name can be specified when using the NetCOBOL Studio*)]
NOOBJECT }

Specify OBJECT to generate an object program; otherwise, specify NOOBJECT. If an object program is output, the file is created in the folder of the source program.

To create the file in another folder, specify the folder name.

See

"3.2 Resources Necessary for Compilation"

A.2.34 OPTIMIZE(global optimization handling)

```
STUDIO,-WC,@
```

{ OPTIMIZE
NOOPTIMIZE }

Specify OPTIMIZE to generate a global-optimized object program; otherwise, specify NOOPTIMIZE.

If this option is specified with option TEST, the debugging information file can be used by the COBOL Error Report, and cannot be used by the debugging function of NetCOBOL Studio.

See

- "Appendix C Global Optimization"
- "A.2.53 TEST(whether the debugging function of NetCOBOL Studio and the COBOL Error Report should be used)"

A.2.35 PRINT(whether compiler listing should be output and the output destination specification)

```
STUDIO
```

PRINT [(*folder-name*)]

Specify this option to generate a compiler listing. If a compiler listing is output, the file is usually created in the folder of the source program. To create the file in another folder, specify the folder name.

A.2.36 QUOTE/APOST(figurative constant QUOTE handling)

```
STUDIO,-WC,@
```

{ QUOTE
APOST }

Specify QUOTE to use quotation marks (") as the values of the QUOTE and QUOTES figurative constants; specify APOST to use apostrophes (") as the values.

Note

Either quotation marks or apostrophes can be used as the delimiters of nonnumeric literals in source programs, regardless of the setting of these options. The beginning and end delimiters must be identical.

A.2.37 RCS(runtime code set specification)

```
STUDIO,-WC,@
```

$$\text{RCS (}\left\{\begin{array}{l}\underline{\text{ASCII}}\\ \text{SJIS}\\ \left\{\begin{array}{l}\text{UTF16}\\ \text{UCS2}\end{array}\right\}\ [,\ \left\{\begin{array}{l}\underline{\text{LE}}\\ \text{BE}\end{array}\right\}\ ]\end{array}\right\}\text{)}$$

Specify whether the runtime code set is "ASCII [RCS (ASCII)]" or "Shift-JIS [RCS (SJIS)]" or "unicode [RCS(UTF16) or RCS (UCS2)]".

RCS(UTF16,LE) and RCS(UCS2,LE) set the runtime code set to little endian.

RCS(UTF16,BE) and RCS(UCS2,BE) set the runtime code set to big endian.

When the RCS(UTF16) and RCS(UCS2) settings produce the same result, the use of RCS(UTF16) is recommended.

Note

- For V11 and later specifying the ENCODE compile option is recommended.
- When compile option RCS is specified, it is assumed that the ENCODE compile option is specified in the following way.
    - When RCS(SJIS) is specified, it is as assumed that ENCODE(SJIS,SJIS) is specified.
    - When RCS(UTF16) or RCS(UCS2) is specified, it is as assumed that ENCODE(UTF8,UTF16) is specified.

See

"Chapter 20 Unicode"

A.2.38 REP(repository file I-O destination folder specification)

STUDIO

REP(*folder-name*)

Normally, the repository file is created in the folder of the source program. To create the file in another folder, specify the folder name.

The specified folder is also used as the input destination folder for the repository file.

Information

If two or more options are specified, folders are searched in the order:

1. Folder specified by the -R option.
2. Folder specified by compiler option REPIN.
3. Folder specified by the COB_REPIN environment variable.
4. Folder specified by the -dr option.
5. Folder specified by compiler option REP.
6. Current folder.

See

- "3.5.2.15 -R(Specify the repository file input folder)"
- "3.5.2.7 -dr(Specify the repository file folder)"
- "A.2.39 REPIN(repository file input destination folder specification)"

- "1.2.1 Setting Environment Variables"

A.2.39 REPIN(repository file input destination folder specification)

```
STUDIO
```

REPIN (*folder-name*[;*folder-name*]...)

Specify the input destination folder for the repository file. If the repository file exists in two or more folders, specify two or more folders separated by semi-colons. If two or more folders are specified, the folders are searched in the order they are specified.

Information

If two or more options are specified, folders are searched in the order:

1. Folder specified by the -R option.
2. Folder specified by compiler option REPIN.
3. Folder specified by environment variable COB_REPIN.
4. Folder specified by the -dr option.
5. Folder specified by compiler option REP.
6. Current folder.

See

- "3.5.2.15 -R(Specify the repository file input folder)"
- "3.5.2.7 -dr(Specify the repository file folder)"
- "A.2.39 REPIN(repository file input destination folder specification)"

A.2.40 RSV(the type of a reserved word)

```
STUDIO,-WC,@
```

RSV ({ ALL | V111 | V112 | V122 | V125 | V30 | V40 | V61 | V70 | V81 | V90 | V1020 | V1040 | VSR2 })

VSR3

Specify the type of a reserved word. The following are the names of reserved-word sets:

- RSV(ALL) : For this product (Fujitsu NetCOBOL latest version)
- RSV(V111) : For GS series COBOL85 V11L11
- RSV(V112) : For GS series COBOL85 V11L20
- RSV(V122) : For GS series COBOL85 V12L20
- RSV(V125) : For COBOL85 V12L50
- RSV(V30) : For COBOL85 V30
- RSV(V40) : For Fujitsu COBOL V4.0
- RSV(V61) : For Fujitsu COBOL V6.1
- RSV(V70) : For Fujitsu NetCOBOL V7.0
- RSV(V81) : For Fujitsu NetCOBOL V8.0
- RSV(V90) : For Fujitsu NetCOBOL V9.0
- RSV(V1020) : For Fujitsu NetCOBOL V10.0, V10.1 and V10.2
- RSV(V1040) : For Fujitsu NetCOBOL V10.4
- RSV(VSR2) : For VS COBOL II REL2.0
- RSV(VSR3) : For VS COBOL II REL3.0

A.2.41 SAI(whether the source analysis information file should be output)

```
STUDIO,-WC,@
```

{ SAI [(*folder-name can be specified when using the NetCOBOL Studio*)]
NOSAI }

Specify SAI to output a source analysis information file; otherwise, specify NOSAI. If the source analysis information file is output, the file is usually created in the folder of the source program.

To create the file in another folder, specify the folder name.

See

- "3.2 Resources Necessary for Compilation"
- "A.2.32 NUMBER(source program sequence number area specification)"

A.2.42 SCS(code system of source file)

```
-WC
```

SCS ({ ACP
SJIS
UTF8 })

To set the code system of the COBOL source file and library to ANSI code page, specify SCS(ACP). To set the code system to Shift-JIS, specify SCS(SJIS). To set the code system to UTF-8, specify SCS(UTF8).

Note

ENCODE(UTF8) must be used when SCS(UTF8) is specified.

See

- "A.2.13 ENCODE (encoding form specification)"

A.2.43 SDS(whether signs in signed decimal data items should be converted)

```
STUDIO,-WC,@
```

```
{ SDS   }
{ NOSDS }
```

Specify this option to determine how to move a signed internal decimal data item to another signed internal decimal data item.

When NOSDS is specified, if it is a value for the sign to mean negative, it changes in minus symbol and if it is a value for the sign to mean positive, it changes to plus symbol. When minus symbol attaches to value 0, it changes to plus symbol.

Example

Program into which operation changes by option

```
    01  A PIC S9V9 VALUE -0.1
    01  B PIC S9.
          :
    MOVE A TO B.
```

When SDS is specified, the decimal point of -0.1 is rounded down and -0 is stored in B.

When NOSDS is specified, the decimal point of -0.1 is rounded down, it becomes -0, convert of the sign is done, and +0 is stored in B.

A.2.44 SHREXT(determines how to treat the external attributes in a multithread program)

```
STUDIO,-WC,@
```

```
{ SHREXT   }
{ NOSHREXT }
```

Use this option when THREAD (MULTI) is specified to generate a multithread object. Specify SHREXT to share data and files having an external attribute (specified by EXTERNAL) between threads; otherwise, specify NOSHREXT.

Note

When THREAD (SINGLE) is specified to generate a single thread object, SHREXT is displayed as the compiler option but compilation is performed as if NOSHREXT were specified.

A.2.45 SMSIZE(Memory capacity used by PowerBSORT)

```
STUDIO,-WC,@
```

SMSIZE(*value*K)

Specify a number for the memory capacity used by PowerBSORT in kilobyte units.

Note

Specify this when you want to restrict the capacity of the memory space used by PowerBSORT that is called from the SORT or MERGE statements. Specify a number in kilobytes. Also set this number in BSORT_PRIME.memory_size of PowerBSORT. For details about which values are valid, refer to the PowerBSORT "Online Manual".

Although this option is equivalent to the smsize option at the time of execution and the value specified in the SORT-CORE-SIZE special register, when these are specified at the same time, the SORT-CORE-SIZE special register has the highest priority, the smsize option at the time of execution has the next highest priority, and the SMSIZE() compiler option has the third highest priority.

Example

```
Special register     MOVE 102400 TO SORT-CORE-SIZE
(102400=100 kilobytes)
Compile option   SMSIZE(500K)
Runtime option  smsize300k
```

In this case, the SORT-CORE-SIZE special register value of 100 kilobytes has the highest priority.

Information

This option is valid if you have installed other PowerBSORT products, and invalid if you have not.

A.2.46 SOURCE(whether a source program listing should be output)

```
STUDIO,-WC,@
```

{ SOURCE
NOSOURCE }

Specify SOURCE to output a source program listing; otherwise, specify NOSOURCE.

When the compiler program listing output is effective by the compiler option PRINT, object program source is output.

See

- "3.5.2.5 -dp(Specify the compile list file folder)"
- "3.5.2.14 -P(Compile listing file name)"

A.2.47 SQLGRP(SQL host variable definition expansion)

```
STUDIO,-WC,@
```

{ SQLGRP
NOSQLGRP }

Specify whether the SQL host variable definition is expanded (SQLGRP) or not (NOSQLGRP).

When NOSQLGRP is specified, a REDEFINE clause cannot be specified to the host variable.

Note

When SQLGRP is specified, you cannot determine whether the data item has a correct format as a host variable just with the data description entry. This means that the syntax check for the host variable definitions during compilation differs as follows:

- The host variables not referenced in an embedded SQL statement are not subject to syntax check. Here, only the host variable names for the group items having global attribute are checked.
- More strict syntax check is made to variable-length character-type host variables. Group items containing items of level number 49 are handled as variable-length character-type host variables.

Specify NOSQLGRP for the conventional syntax check without using the host variables which are defined as group items.

See

"17.2.4 Advanced Data Manipulation"

A.2.48 SRF(reference format type)

```
STUDIO,-WC,@
```

$$\text{SRF(}\left\{\begin{array}{l}\text{FIX}\\\text{FREE}\\\underline{\text{VAR}}\end{array}\right\}\text{ [, }\left\{\begin{array}{l}\text{FIX}\\\text{FREE}\\\text{VAR}\end{array}\right\}\text{])}$$

Specify the reference formats of a COBOL source program and library. Specify FIX for fixed-length format; specify FREE for free format, specify VAR for variable-length format.

If the SRF option is omitted, the reference format specified in the COBOL source program is used.

Note

- If FREE is specified for compiler option SRF, compiler option NUMBER cannot be specified. If NUMBER is specified, program operation is not guaranteed.
- Specify the reference format of COBOL source program, then specify that of the libraries. If the two reference formats are the same, there is no need to specify the library program format.
- FREE cannot be specified when using the NetCOBOL Studio.

A.2.49 SSIN(ACCEPT statement data input destination)

```
STUDIO,-WC,@
```

$$\text{SSIN (}\left\{\begin{array}{l}\textit{runtime-environment-variable-name}\\\underline{\text{SYSIN}}\end{array}\right\}\text{)}$$

Specifies the data input destination of the ACCEPT statement for ACCEPT/DISPLAY function.

- SSIN(runtime-environment-variable-name)
  A file is used as the data input destination . At runtime, specify the path name of the file in the runtime environment variable.
- SSIN(SYSIN)
  The console window is used as the data input destination.

Note

The environment variable name can be specified with up to eight uppercase letters and numeric characters beginning with an uppercase letter (A to Z).

The runtime environment variable name must be unique. The name must be different from a runtime environment variable name (file-identifier) used with another file.

See

"10.1 ACCEPT/DISPLAY Function"

A.2.50 SSOUT(DISPLAY statement data output destination)

```
STUDIO,-WC,@
```

SSOUT ($\left\{ \begin{array}{l} \textit{runtime-environment-variable-name} \\ \underline{\text{SYSOUT}} \end{array} \right\}$)

Specifies the data output destination of the DISPLAY statement for ACCEPT/DISPLAY function.

- SSOUT(runtime-environment-variable-name)
  A file is used as the data output destination. At runtime, specify the path name of the file in the runtime environment variable.

- SSOUT(SYSOUT)
  The console window is used as the data output destination.

Note

The runtime environment variable name can be specified with up to eight uppercase letters and numeric characters beginning with an uppercase letter (A to Z).
The runtime environment variable name must be unique. The name must be different from a runtime environment variable name (file-identifier) used with another file.

See

"10.1 ACCEPT/DISPLAY Function"

A.2.51 STD1(alphanumeric character size specification)

```
STUDIO,-WC,@
```

STD1($\left\{ \begin{array}{l} \text{ASCII} \\ \text{JIS1} \\ \underline{\text{JIS2}} \end{array} \right\}$)

In EBCDIC specification of the ALPHABET clause, specify whether alphanumeric codes (1-byte character standard codes) are treated as ASCII (ASCII), JIS 8-bit codes (JIS1), or JIS 7-bit Roman character codes (JIS2).

Note

EBCDIC is specified in the ALPHABET clause, the character code system used depends on the specification of this option.

- STD1(ASCII): EBCDIC (ASCII)
- STD1(JIS1) : EBCDIC (kana)
- STD1(JIS2) : EBCDIC (lowercase letters)

A.2.52 TAB(tab handling)

```
STUDIO,-WC,@
```

```
TAB (  { 8 }  )
       { 4 }
```

Specifies whether tabs are to be set in units of 4 columns (TAB(4)) or 8 columns (TAB(8)).

Tabs specified as values literally mean tab values.

A.2.53 TEST(whether the debugging function of NetCOBOL Studio and the COBOL Error Report should be used)

```
STUDIO,-WC,@
```

```
{ TEST [ (folder-name can be specified when using the NetCOBOL Studio) ] }
{ NOTEST                                                                }
```

To use the debugging function of NetCOBOL Studio and the COBOL Error Report, specify TEST; otherwise, specify NOTEST. If TEST is specified, the debugging information file used with the debugging function and the COBOL Error Report is created in the folder of the source program. To create the file in another folder, specify the folder name.

Upon linking, the linkage option should also be set to use the debugging function and the COBOL Error Report.

Note

If this option is specified with OPTIMIZE, the debugging information file can be used by the COBOL Error Report, and cannot be used by debugging function of NetCOBOL Studio. Please don't specify OPTIMIZE when you use debugging function of NetCOBOL Studio. Refer to the "A.2.34 OPTIMIZE(global optimization handling)".

See

- "3.2 Resources Necessary for Compilation"
- "4.5 LINK Command Format"
- "NetCOBOL Debugging Guide"
- "NetCOBOL Studio User's Guide"

A.2.54 THREAD(specifies multithread program creation)

```
STUDIO,-WC,@
```

```
THREAD (  { MULTI  }  )
          { SINGLE }
```

Specify THREAD (MULTI) to generate a multithread object; otherwise, specify THREAD (SINGLE).

Note

If THREAD (MULTI) is to generate a multithread object, the distributed development support function cannot be used.

See

"Chapter 19 Multithread Programs"

A.2.55 TRACE(whether the TRACE function should be used)

```
STUDIO,-WC,@
```

```
  ┌ TRACE [(n)] ┐
 {              }
  └ NOTRACE     ┘
```

To use the TRACE function, specify TRACE; otherwise, specify NOTRACE.

n indicates the number of the trace information items to be output. Specify n with an integer 1 to 999,999. The default value is 200.

Note

- If TRACE is specified, processing for displaying trace information is incorporated into the object program. Consequently, execution performance decreases. When debugging is completed, specify NOTRACE, then recompile the program.
- TRACE cannot be specified along with compiler option COUNT. If they are specified together, only the latter is used. Refer to "A. 2.8 COUNT(whether the COUNT function should be used)".

See

"Using the TRACE Function" in the "NetCOBOL Debugging Guide"

A.2.56 TRUNC(Truncation Operation)

```
STUDIO,-WC,@
```

```
  ┌ TRUNC    ┐
 {           }
  └ NOTRUNC  ┘
```

Specify a method of truncating high-order digits when a number is moved with a binary data item as a receiving item.

- TRUNC: The high-order digits of the result value are truncated according to the specification of the PICTURE clause of the receiving item. The resulting value after truncation is stored in the receiving item.

  If this option is specified with the compiler option OPTIMIZE, the high-order digits of a variable moved from an external decimal data item or internal decimal data item through optimization are also truncated. Digits are truncated as explained above only if the number of digits in the integer part of the sending item is greater than that of the receiving item.
- NOTRUNC: The execution speed of the object program is of top priority. If the execution is faster without truncation than with truncation, digits are not truncated.

Example

With the following PICTURE clauses:

Moving S999V9 (3 digits in the integer part) to S99V99 (2 digits in the integer part)

-> Truncation

Moving S9V999 (1 digit in the integer part) to S99V99 (2 digits in the integer part)

-> No truncation

See

"A.2.34 OPTIMIZE(global optimization handling)"

Note

- With NOTRUNC, if the number of digits in the integer part of the sending item is greater than the number of digits in the integer part of the receiving item, results cannot be guaranteed.
- If NOTRUNC is specified, the program must be designed so that a digit count exceeding the digit count specified in the PICTURE clause is not stored in the receiving item even when no digit is truncated.
- If NOTRUNC is specified, whether to truncate digits depends on the compiler. Thus, a program using NOTRUNC may be incompatible with another system.

A.2.57 XREF(whether a cross-reference list should be output)

```
STUDIO,-WC,@
```

$$\left\{ \begin{array}{l} \text{XREF} \\ \underline{\text{NOXREF}} \end{array} \right\}$$

Specify whether a cross-reference list is output (XREF) or not (NOXREF).

The cross-reference list shows user words and special registers in ascending order of character size.

When the compiler program listing output is effective by the compiler option PRINT, cross-reference listing is output.

See

- "3.5.2.5 -dp(Specify the compile list file folder)"
- "3.5.2.14 -P(Compile listing file name)"

Note

If the highest severity code is S or higher as a result of compilation with compiler option XREF specified, a cross-reference list is not output.

A.2.58 ZWB(comparison between signed external decimal data items and alphanumeric data items)

```
STUDIO,-WC,@
```

```
{ ZWB   }
{ NOZWB }
```

Specify this option to determine how to treat the sign part when a signed external decimal data item is compared with an alphanumeric field.

Specify ZWB for comparison ignoring the sign part of the external decimal data item; specify NOZWB for comparison including the sign part.

Alphanumeric characters include alphanumeric data items, alphabetic data items, alphanumeric edited data items, numeric edited data items, nonnumeric literals, and figurative constants other than ZERO.

Example

```
77  ED   PIC S9(3) VALUE  +123.
77  AN   PIC  X(3) VALUE  "123".
```

The conditional expression ED = AN is defined as shown below in this example:

- With ZWB specified: True
- With NOZWB specified: False

A.3 Compiler Options That Can Be Specified in the Program Definition (Source Code) Only

The following compiler options can be specified only when the program definition is compiled:

- BINARY (BYTE)
- CONF (OBS)
- FLAGSW other than NOFLAGSW
- LANGLEVEL (74) or LANGLVL (68)
- MAIN, MAIN(WINMAIN), or MAIN(MAIN)

A.4 Compiler Options for Method Prototype Definitions and Separated Method Definitions

The compiler option that is specified when a separated method definition is compiled must match the compiler option that is specified when the method prototype definition is compiled.

However, the following compiler options do not have to be matched:

- CHECK
- CONF
- COPY
- COUNT
- CURRENCY

- DLOAD
- DUPCHAR
- FLAG
- FORMLIB
- LIB
- LINECOUNT
- LINESIZE
- LIST
- MAP
- MESSAGE
- NUMBER
- OBJECT
- PRINT
- QUOTE/APOST
- REP
- REPIN
- SAI
- SOURCE
- SRF
- TEST
- TRACE
- XREF

Appendix B I-O Status List

Table B.1 lists the values and meanings that can be returned to the data item specified in the FILE STATUS clause of the file control entry. These I-O status values are set whenever I-O statements are executed.

Information

Related manuals

- FORM RTS online manual
- PowerFORM RTS online manual
- RDB/7000 Handbook
- SymfoWARE7000 RDB_7000 Linkage Help
- SymfoNET Handbook: Database Linkage
- ACM Control Handbook
- IDCM Software Development Kit Online Help

Table B.1 I-O Status List

Classification	I-O Status Value	Detailed Information	Meaning
Successful	00	--	Input-output statements were executed successfully.
	02	--	The status is one of the following: - The value of the reference key of the record read by executing the READ statement is the same as that of the next record. - The record having the same record key value as the record written by executing the WRITE or REWRITE statement already exists in a file. This is not an error, however, because a record key value may be duplicated.
	04	--	The length of an input record is greater than the maximum record.
		FORM RTS	One of the following occurred in the presentation file. - An input error occurred in required full-field input items or required input items. - A data error occurred. (A national language data error, an ANK data error, a numeric data item configuration data error, a numeric data item sign error, a numeric data item decimal point error, a redundancy check error) - Data entered in the data item is invalid.
	05	--	One of the following status occurred in the file where the OPTIONAL clause is specified. - The OPEN statement in INPUT, I-O, or EXTEND mode was executed for a file, but the file has yet to be created. - The OPEN statement in INPUT mode was executed for a nonexistent file. In this case, the file is not created and at end condition occurs (input-output status value = 10) during execution of the first READ statement.

Classification	I-O Status Value	Detailed Information	Meaning
			- The OPEN statement in I-O or EXTEND mode was executed for a nonexistent file. In this case, the file is created.
	07	--	Input-output statements were successfully executed, however, the file referenced by one of the following methods exists in a non-reel or unit medium. - OPEN statement or CLOSE statement with NO REWIND specified - CLOSE statement with REEL/UNIT specified
AT END condition	10	--	AT END condition occurred in a sequentially accessed READ statement. - File end reached. - The first READ statement was executed for a non-existent optional input file. That is, a file with the OPTIONAL clause specified was opened in INPUT mode but the file was not created.
	14	--	AT END condition occurred in a sequentially accessed READ statement. - The valid digits of a read relative record number are greater than the size of the relative key item of the file.
Invalid key condition	21	--	Record key order is invalid. One of the following status occurred. - During sequential access, the prime record key value was changed between the READ statement and subsequent REWRITE statement. - During random access or dynamic access of a file with DUPLICATES specified as the prime key, the prime record key value was changed between the READ statement and subsequent REWRITE or DELETE statement. - During sequential access, prime record key values are not in ascending order at WRITE statement execution.
	22	--	During WRITE or REWRITE statement execution, the value of the prime record key, or alternate record key to be written already exists in a file, except when DUPLICATES is specified in the prime record key or alternate record key.
Invalid Key Condition	23	--	The record was not found. - During START statement or random access READ, REWRITE, or DELETE statement execution, the record having the specified key value does not exist in the file. - Relative record number 0 was specified for a relative file.
	24	--	One of the following statuses occurred. - Area shortage occurred during WRITE statement or CLOSE statement execution. - During WRITE statement execution, the specified key is out of the key range. - After overflow writing, an attempt was made to execute the WRITE statement again.

Classification	I-O Status Value	Detailed Information	Meaning
Permanent error condition	30	--	A physical error occurred.
	34	--	Area shortage occurred during WRITE statement or CLOSE statement execution.
	35	--	The OPEN statement in INPUT, I-O, or EXTEND mode was executed for a file without the OPTIONAL clause specification. The file has yet to be created, however.
	37	--	The specified function is not supported.
	38	--	The OPEN statement was executed for a file for which CLOSE LOCK was executed before.
	39	--	During OPEN statement execution, a file whose attribute conflicts with the one specified in the program was assigned.
Logical error condition	41	--	The OPEN statement was executed for an opened file.
	42	--	The CLOSE statement was executed for an unopened file.
Logical Error Condition	43	--	During sequential access or DELETE or REWRITE statement execution for a file with DUPLICATES specified as the prime key, the preceding input-output statement was not a successful READ statement.
	44	--	One of the following statuses occurred. - The record length during WRITE or REWRITE statement execution is greater than the maximum record length specified in the program. Or, an invalid numeric was specified as the record length. - During REWRITE statement execution, the record length is not equal to that of the record to be rewritten.
	46	--	During sequential call READ statement execution, the file position indicator is undefined due to one of the following reasons. - The preceding START statement is unsuccessful. - The preceding READ statement is unsuccessful (including at end condition).
	47	--	The READ or START statement was executed for a file not opened in INPUT or I-O mode.
	48	--	The WRITE statement was executed for a file not opened in OUTPUT, EXTEND (sequential, relative, or indexed), or I-O (relative or indexed) mode.
	49	--	The REWRITE or DELETE statement was executed for a file not opened in I-O mode.
Other errors	90	--	These errors are other than those previously classified. The following conditions are assumed. - File information is incomplete or invalid. - During OPEN or CLOSE statement execution, an error occurred in an OPEN or CLOSE function. - An attempt was made to execute an input-output statement for a file where the CLOSE statement was unsuccessful due to input-output status value 90.

Classification	I-O Status Value	Detailed Information	Meaning
			- Resources such as main storage are unavailable. - An attempt was made to execute the OPEN statement for a file not closed correctly. - An attempt was made to write records after an error occurred due to overflow writing. - An attempt was made to write records after a no-space condition occurred. - An invalid character is contained in a text file record. - A character cannot be converted to the code set. - Many applications executed an OPEN statement for the same file. Therefore, the lock table has run out of space and cannot provide new locks. - Necessary related product loading failed. - An error other than the above occurred. This is the only information about input-output operations where the error occurred. - A system error occurred.
		FORM RTS/Power FORM RTS	FORM RTS/Power FORM RTS detected an error.
	91	--	- No file is assigned. - At OPEN statement execution, file identification does not correspond to a physical file name.
	93	--	An exclusive error occurred. (File lock)
	99	--	An exclusive error occurred. (Record lock)
		FORM RTS	A system error occurred.
	9E	FORM RTS	The execution waiting by the READ statement was compulsorily released.

Note

Refer to the FORM RTS and PowerForm RTS error code for FORM RTS and PowerForm RTS in the detailed information column.

Information

Related manuals

- FORM RTS online manual
- PowerFORM RTS online manual

Appendix C Global Optimization

This appendix explains the global optimization performed by the COBOL compiler.

C.1 Optimization

Global optimization divides the procedure division into sets of statements (called basic blocks), each consisting of one entry, one exit, and statements executable in physical sequence between the entry and the exit. Control flow and data use status are then analyzed, mainly in connection with loops (program parts to execute repeatedly).

Specifically, the following operations occur during global optimization:

- Removing common expressions
- Shifting an invariant
- Optimizing induction variables
- Optimizing PERFORM statements
- Integrating adjacent moves
- Eliminating unnecessary substitutions

C.2 Removing a Common Expression

For arithmetic or conversion processing, if possible, the previous results are reserved and used without additional execution of arithmetic or conversion processing.

Example

Example 1

```
 77 I  PIC S99 BINARY.
 77 J  PIC S99 BINARY.
 77 K  PIC S99 BINARY.
 01 REC-1.
   02 A0  OCCURS 25 TIMES.
     03 A  PIC XX OCCURS 10 TIMES.
   02 B0  OCCURS 35 TIMES.
     03 B  PIC XX OCCURS 10 TIMES.
*>        :
     MOVE SPACE TO A   (I, J).           *>(1)
*>        :
     MOVE SPACE TO B   (I, K).           *>(2)
```

If I does not change in value between (1) and (2) in Example 1, the address calculation formula (*1) for "A (I, J)" and address calculation formula (*2) for "B (I, K)" will have "I*20" in common.

Therefore, COBOL optimizes the latter to use the result of the former.

(*1) A-22+I*20+J*2

(*2) B-22+I*20+K*2

Example 2

```
 77 Z1  PIC S9(9)  DISPLAY.
 77 Z2  PIC S9(9)  DISPLAY.
 77 B1  PIC S9(4)  BINARY.
 77 B2  PIC S9(4)  BINARY.
*>
     COMPUTE Z1 = B1 * B2.          *>(1)
```

```
*>        :
     COMPUTE Z2 = B1 * B2.          *>(2)
```

In Example 2, B1 and B2 remain unchanged between (1) and (2). B1 * B2 is common and (2) is optimized so it uses results of (1).

C.3 Shifting an Invariant

For arithmetic or conversion processing done within a loop, the processing can be performed outside the loop if the same results would be obtained regardless of whether the processing is done within or outside the loop.

Example

```
 77 I   PIC S9(4)  BINARY.
 77 ZD  PIC S9(7)  DISPLAY.
 01 REC-1.
   02 B1  PIC S9(7)  BINARY OCCURS 20 TIMES.
*>     :
 PROCEDURE DIVISION.
     MOVE 1  TO  I.
 LOOP-1.                                     *>--+
     IF B1(I) = ZD GO TO EXIT-1.             *>  |
*>       :                                   *>  | Loop
     ADD 1 TO I.                             *>  |
     IF I IS <= 20 GO TO LOOP-1.             *>--+
 EXIT-1.
```

If ZD never changes throughout the loop in Example 1, the compiler shifts ZD-to-Binary conversion processing by the IF statement to the outside of the loop.

C.4 Optimizing an Induction Variable

If the compiler finds a loop containing a partial expression that uses an item (induction item) defined recursively by a constant (as in ADD 1 TO I., for example) or by an item with an unchanging value, it introduces a new induction variable and changes the multiplication of the subscript calculation formula to an addition.

Example

```
77 I  PIC S9(4)  BINARY.
 01 REC-1.
   02 A  PIC X(10)  OCCURS 20 TIMES.
*>     :
 PROCEDURE DIVISION.
*>     :
LOOP-1.                                   *>--+
     IF A(I) = ...               *>(1)    *>  |
*>        :                               *>  | Loop
     END-IF.                              *>  |
     ADD 1 TO I.                 *>(2)    *>  |
     IF I IS <= 20 GO TO LOOP-1.  *>(3)   *>--+
```

Because I is defined recursively by a constant within the loop in Example 1, the compiler introduces a new induction variable to replace the multiplication "I * 10" in the address calculation formula for A(I) "A - 10 + I * 10" with "t", and generates "ADD 10 TO t" after (2).

Furthermore, if I is not used anywhere else in the loop and, concurrently, the value of I calculated inside the loop is not used after control exits from the loop, the compiler replaces (3) with "IF t IS <= 20 GO TO LOOP-1" and deletes (2).

C.5 Optimizing a PERFORM Statement

The compiler expands the PERFORM statement into some instructions for saving, setting and restoring the return address for the return mechanism. The exit of the PERFORM statement transfers control to the other statement, generally. Otherwise, some of the machine instructions for the return mechanism turn out to be redundant.

The compiler then removes these redundant machine instructions.

C.6 Integrating Adjacent Moves

If different alphanumeric move statements transfer data from contiguous items to identically contiguous items, the compiler integrates these move statements into one.

Example

```
   02 A1  PIC X(32).
   02 A2  PIC X(16).
*>        :
   02 B1  PIC X(32).
   02 B2  PIC X(16).
*>        :
     MOVE A1 TO B1.                        *>(1)
*>       :
     MOVE A2 TO B2.                        *>(2)
```

If A2 and B2 do not change between (1) and (2) and, concurrently, B2 is not referenced in Example 1, the compiler deletes MOVE statement (2). Then MOVE statement (1) sets A1 and A2, and B1 and B2, as one area, respectively, to perform a move.

C.7 Eliminating Unnecessary Substitutions

Substitutions are eliminated for data items that are not going to be either implicitly or explicitly referenced.

C.8 Notes on Global Optimization

If the compiler option OPTIMIZE is specified, the compiler generates a globally optimized object program. For details, refer to "A.2.34 OPTIMIZE(global optimization handling)".

The following explains notes on global optimization:

Use of the linkage function

Some parameters for a called program share all or part of a storage area (for example, CALL "SUB" USING A, A. or CALL "SUB" USING A, B. where A and B share part of the area).

If the content of the linkage area is overwritten by the called program, optimization of the called program may not lead to the expected result.

No execution of global optimization

The compiler does not perform global optimization on the following programs:

- Programs that do not define items and index names having attributes targeted by global optimization
- Programs using the segmentation function (segmentation module)

Less effective global optimization

Global optimization is less effective on the following programs:

- Programs mainly performing I/O operations, and programs that usually do not use the CPU, i.e., containing only data definitions, not executable code.
- Program using no numeric data items but only alphanumeric data items
- Programs referencing nondeclaratives from declaratives
- Programs referencing declaratives from nondeclaratives
- Programs specifying the compiler option TRUNC. For details, refer to "A.2.56 TRUNC(Truncation Operation)".

Notes on Debugging

- Since global optimization causes the deletion, shifting and modification of one or more statements, a program interruption (for example, data exception) can occur a different number of times or at a different location.
- If the program is interrupted while data items are being written, those data items may not actually be set.
- With compiler option NOTRUNC, the program may not operate properly if the internal or external decimal data item is to be recursively defined. For details, refer to "A.2.56 TRUNC(Truncation Operation)".
- If the compiler option OPTIMIZE is specified, the debugging function of NetCOBOL Studio cannot be used.

Appendix D Intrinsic Function List

D.1 Function Types and Coding

Each function has a type. The location in a program at which a function can be written varies depending on the type. See Table D.1, "Intrinsic Functions," for the correspondence between functions and types.

The coding of individual function types is explained below. A statement written in accordance with the function call format is normally called a function identifier, but is referred to as a function.

Integer function

An integer function can only be written in an arithmetic expression. No integer function can be written outside an arithmetic expression, such as in the send side of the MOVE statement.

The following example uses an integer function in the COMPUTE statement.

Example

Day calculation

```
DATA            DIVISION.
 WORKING-STORAGE SECTION.
  01 INT         PIC S9(9) COMP-5.
  01 IN-YMD      PIC 9(8).
  01 OUT-YMD     PIC 9(8).
  01 OUT-YMD-ED  PIC XXXX/XX/XX.
 PROCEDURE       DIVISION.
     MOVE 20021225 TO IN-YMD.
*   Day calculation from date
     COMPUTE INT = FUNCTION INTEGER-OF-DATE(IN-YMD).
     DISPLAY "December 25, 2002 is "
       INT "th day from the reference date.".
*   Date calculation from the number of days
     COMPUTE OUT-YMD = FUNCTION DATE-OF-INTEGER (INT).
     MOVE OUT-YMD TO OUT-YMD-ED.
     DISPLAY "The " INT "th day from the reference date is "
       OUT-YMD-ED ".".
```

Execution result

```
December 25, 2002 is the +000146821th day from the reference date.
The +000146821th day from the reference date is 2002/12/25.
```

Numeric function

A numeric function can only be written in an arithmetic expression, as is the case with an integer function. No numeric function can be written outside an arithmetic expression, such as in the send side of the MOVE statement.

Alphanumeric function

An alphanumeric function can be written anywhere an alphanumeric data item can be written.

Example

UPPER-CASE function

```
DATA            DIVISION.
 WORKING-STORAGE SECTION.
 01 LOWCASE      PIC X(13) VALUE "fujitsu cobol".
 PROCEDURE DIVISION.
     DISPLAY "Before conversion: " LOWCASE.
     DISPLAY "After conversion: " FUNCTION UPPER-CASE(LOWCASE).
```

Execution result

```
Before conversion: fujitsu cobol
After conversion: FUJITSU COBOL
```

In this example, the DISPLAY statement is used to display characters converted into upper case. However, the converted characters can be written in the send side of the MOVE statement to move them to a working area.

National function

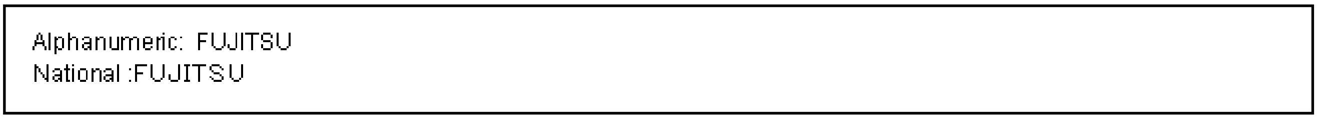
A national function can be written anywhere a national data item can be written. The following example transcribes converted characters and then displays them.

Example

NATIONAL function

```
DATA            DIVISION.
 WORKING-STORAGE SECTION.
 01 NCHAR  PIC N(7).
 01 CHAR   PIC X(7) VALUE "FUJITSU".
 PROCEDURE DIVISION.
*>  National characters that are converted from alphanumeric
*>  characters are displayed.
     MOVE FUNCTION NATIONAL(CHAR) TO NCHAR.
     DISPLAY "Alphanumeric: " CHAR.
     DISPLAY "National: " NCHAR.
```

Execution result



```
Alphanumeric: FUJITSU
National :ＦＵＪＩＴＳＵ
```

Pointer data function

For information on the coding of the pointer data function, see "12.3 Using the ADDR and LENG Functions".

D.2 Function Type Determined by Argument Type

Some function types are determined by the type of the argument. An example of this function type is shown using a function for determining the maximum value.

Example

MAX function

```
 01 C1     PIC X(10).
 01 C2     PIC X(5).
 01 C3     PIC X(5).
 01 V1     PIC S9(3).
 01 V2     PIC S9(3)V9(2).
 01 V3     PIC S9(3).
 01 MAXCHAR  PIC X(10).
 01 MAXVALUE PIC S9(3)V9(2).
 PROCEDURE DIVISION.
     MOVE FUNCTION MAX(C1 C2 C3) TO MAXCHAR.             *>[1]
*>         :
     COMPUTE MAXVALUE =  FUNCTION MAX(V1 V2 V3).         *>[2]
```

The MAX function is the function that requests the maximum value and the function type is determined by the argument type.

The MAX function in [1] is an alphanumeric function because the argument type is alphanumeric. An alphanumeric function cannot be written in an arithmetic expression. The MAX function in [2] is a numeric function because the argument type is numeric. A numeric function can only be written in an arithmetic expression.

D.3 Obtaining the Year Using the CURRENT-DATE Function

The ACCEPT/DISPLAY function used for date input can only obtain the last two digits of the year. The CURRENT-DATE function can obtain four digits of the year.

Example

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 TODAY.
  02 YEAR   PIC X(4).
  02        PIC X(17).
 PROCEDURE DIVISION.
     MOVE FUNCTION CURRENT-DATE TO TODAY.
*>       :
```

A four-digit year is indicated by the YEAR variable after move operation.

You can also vary the date returned by the CURRENT-DATE function by using the @CBR_JOBDATE environment variable, as described in the "5.4.1 Environment Variables".

Example

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 TODAY.
  02 YEAR   PIC X(4).
  02 MONTH  PIC X(2).
  02 DAY    PIC X(2).
  02        PIC X(13).
 PROCEDURE DIVISION.
     MOVE FUNCTION CURRENT-DATE TO TODAY.
*>       :
```

1999.09.01 is set to the environment variable @CBR_JOBDATE at execution time.

1999 is set to the YEAR, 09 is set to the MONTH and 01 is set to the DAY after move operation.

Refer to "10.1.8 Setting and Accepting a Given Date" for details.

Note

In the example, the group item move is made because the receiving side of the MOVE statement is a group item. If the receiving side is a numeric data item, a numeric data move is made. Different rules apply to the numeric data move than to the group item move. In this example, therefore, a four-digit year cannot be obtained even if a four-digit area is allocated as shown below.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77 LAG     PIC 9(4).
PROCEDURE DIVISION.
    MOVE FUNCTION CURRENT-DATE TO LAG.
```

The variable LAG after a move is made does not indicate the year but indicates the difference from Greenwich Mean Time (digits 18 to 21 of the value of the CURRENT-DATE function).

D.4 Calculating Days from an Arbitrary Reference Date

The number of days from an arbitrary reference date can be obtained by calculating the difference from the value obtained by day calculation. The following example calculates the number of days from the specified reference date to the current date, and calculates interest within the period.

Example

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 TODAY     PIC X(8).
 01 TODAY-R   REDEFINES TODAY  PIC 9(8).
 01 FROM-DAY  PIC 9(8).
 01 INT       PIC 9(5).
 01 PAY       PIC 9(6).
 01 EDT-PAY   PIC ZZZ,ZZ9.
 01 EDT-INT   PIC ZZZZ9.
 01 EDT1      PIC XXXX/XX/XX.
 01 EDT2      PIC XXXX/XX/XX.
 PROCEDURE DIVISION.
*> Set the reference date.
     ACCEPT FROM-DAY
     MOVE FROM-DAY TO EDT1
*> Obtain the current date.
     MOVE FUNCTION CURRENT-DATE TO TODAY EDT2
*> Calculate the number of days between the two dates.
     COMPUTE INT =  (FUNCTION INTEGER-OF-DATE(FROM-DAY))
                  - (FUNCTION INTEGER-OF-DATE(TODAY-R))
*> Calculate the interest (example:  fixed to 133.3 yen per
*> 20 days).
     COMPUTE PAY = FUNCTION INTEGER-PART((INT / 20) * 133.3)

     MOVE PAY TO EDT-PAY.
     MOVE INT TO EDT-INT.
*> Display the result.
     DISPLAY EDT-INT
       " days have passed from the deposit date(" EDT1 ").".
     DISPLAY "The total interest as of " EDT2 " is "
       EDT-PAY " yen.".
```

Execution result (Suppose "19920521" is input for the reference date.)

```
2,272 days have passed from the deposit date (May 21, 1992).
The total interest as of August 10, 1998 is 15,062 yen.
```

D.5 Intrinsic Function List

Table D.1 lists the intrinsic functions that can be used with NetCOBOL.

Table D.1 Intrinsic Functions

Classification	Function	Explanation	Function type
Length	LENGTH	Obtains the length of a data item or literal.	Integer
	LENG	Obtains number of bytes.	Integer
	STORED-CHAR-LENGTH	Obtains the effective character length.	Integer
Size	MAX	Obtains the maximum value.	Integer, numeric, or alphanumeric
	MIN	Obtains the minimum value.	Integer, numeric, or alphanumeric
	ORD-MAX	Obtains the ordinal position of the maximum value.	Integer
	ORD-MIN	Obtains the ordinal position of the minimum value.	Integer
Conversion	REVERSE	Reverses the order of character strings.	Alphanumeric
	LOWER-CASE	Converts uppercase characters to lowercase characters.	Alphanumeric
	UPPER-CASE	Converts lowercase characters to uppercase characters.	Alphanumeric
	NUMVAL	Converts numeric characters to numeric values.	Numeric
	NUMVAL-C	Converts numeric characters including a comma and currency sign to numeric values.	Numeric
	NATIONAL	Converts characters into national characters.	National
	CAST-ALPHANUMERIC	Converts characters into Alphanumeric characters.	Alphanumeric
	UCS2-OF	Converts the encoding system into UCS2.	National
	UTF8-OF	Converts the encoding system into UTF8.	Alphanumeric
	DISPLAY-OF	Converts national characters into alphanumeric characters.	Alphanumeric
	NATIONAL-OF	Converts alphanumeric characters into national characters.	National
Character operation	CHAR	Obtains a character at a specified position in the collating sequence of a program.	Alphanumeric
	ORD	Obtains the ordinal position of a specified character in the collating sequence of a program.	Integer
Numeric value operation	INTEGER	Obtains the maximum integer within a specified range.	Integer
	INTEGER-PART	Obtains integer parts.	Integer
	RANDOM	Obtains random numbers.	Numeric
Calculation of interest rate	ANNUITY	Obtains the approximate value of the equal payment rate to 1 (the principal) according to the rate of interest and the period.	Numeric
	PRESENT-VALUE	Obtains the current price according to the reduction rate.	Numeric
Date operation	CURRENT-DATE	Obtains the current date and time and the difference between Greenwich Mean Time.	Alphanumeric

Classification	Function	Explanation	Function type
	DATE-OF-INTEGER	Obtains the date corresponding to the day of the year.	Integer
	DAY-OF-INTEGER	Obtains the year and day corresponding to the day of the year.	Integer
	INTEGER-OF-DATE	Obtains the day of the year corresponding to the date.	Integer
	INTEGER-OF-DAY	Obtains the day of the year corresponding to the year and day.	Integer
	WHEN-COMPILED	Obtains the date and time the program was compiled.	Alphanumeric
Arithmetic calculation	SQRT	Obtains the approximate value of a square root.	Numeric
	FACTORIAL	Obtains factorials.	Integer
	LOG	Obtains natural logarithms.	Numeric
	LOG10	Obtains common logarithms.	Numeric
	MEAN	Obtains average values.	Numeric
	MEDIAN	Obtains medians.	Numeric
	MIDRANGE	Obtains the average values of the maximum and minimum.	Numeric
	RANGE	Obtains the difference between the maximum and minimum.	Integer or numeric
	STANDARD-DEVIATION	Obtains standard deviations.	Numeric
	MOD	Obtains a specified value in the specified modulus.	Integer
	REM	Obtains remainders.	Numeric
	SUM	Obtains sums.	Integer or numeric
	VARIANCE	Obtains variances.	Numeric
Trigonometric function	SIN	Obtains the approximate value of a sine.	Numeric
	COS	Obtains the approximate value of a cosine.	Numeric
	TAN	Obtains the approximate value of a tangent.	Numeric
	ASIN	Obtains the approximate value of an inverse sine.	Numeric
	ACOS	Obtains the approximate value of an inverse cosine.	Numeric
	ATAN	Obtains the approximate value of an inverse tangent.	Numeric
Pointer	ADDR	Obtains the first address.	Pointer data

Appendix E Writing Special Literals

This appendix explains how to write literals for specifying names (for example, program and file names) defined in the system.

E.1 Program Name Literal

Any characters can be used for a literal specifying a program name in the PROGRAM-ID paragraph, CALL statement, or CANCEL statement. The length of the literal must be 60 bytes or less. There are no configuration rules. You must determine if the literal complies with the linker rules.

E.2 Text Name Literal

For the text name literal to be written in the COPY statement, specify the name of the file containing library text in the following format:

```
"[drive-name: ] [path-name]  file-name [.extension]"
```

drive-name

Specify the drive-name by a single character from A to Z. When the drive-name is omitted, the current drive will be used.

path-name

Specify a folder path in the following format:

```
[\][folder-name [\ folder-name] ...]\
```

- When the path-name is omitted, the file will be placed in the current folder. When the relativity path-name is specified, the current folder is added to the front.

File-name

Specify the file-name.

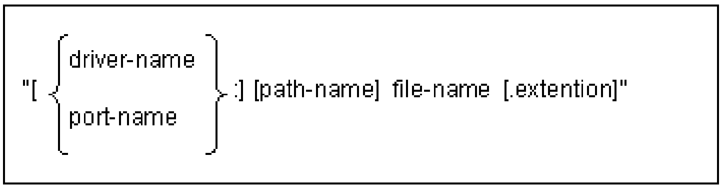
extension

Specify the file extension if possible. Note: "CBL" and "COB" may not be used. For example:

- "C:\COPY\A.CBL"
- "A.CPY"
- "C:\COPY\A"

E.3 File-Identifier Literal

For the file-identifier literal to be written in the ASSIGN clause of the file control entry, specify the file to be processed in the following manner:



```
"[ { driver-name } :] [path-name]  file-name  [.extention]"
   { port-name   }
```

drive-name

Specify a drive name with an alphabetic character from A to Z. A name not specified with an alphabetic character from A to Z is regarded as a port name. When the drive name is omitted, the current drive will be used.

port-name

A port name can be specified for sequential files only. If a port name is specified, specification of a path name and file-reference-name is invalid.

To specify a printer with a port name, use port name LPT1, LPT2, or LPT3.

path-name

Specify the folder storing the file in the following format:

```
[\][folder-name[\folder-name] ...]\
```

If a path name is omitted the current folder is used.

file-name

Specify a file name.

extension

Specify any character string for identifying the type of the file.

For example:

- "A:\COBOL\A.DAT"
- "LPT1:"
- "B.TXT"

E.4 Literal for Specifying an External Name

An external name can be assigned by specifying a literal in the AS phrase to each of the following names defined in the identification division. If the AS phrase is omitted, the internal name is the same as the external name.

- Program name

```
PROGRAM-ID.  CODE-GET  AS  "XY1234".
```

- Class name

```
CLASS-ID.  special-processing  AS  "SP-CLASS-001".
```

- Method name

```
METHOD-ID.  GET-VALUE   AS "VALUE".
```

The literal specified in the AS phrase must not begin with an underscore. There are no other restrictions on the type and composition of characters making up the literal. The user must confirm that linker rules are observed.

Appendix F OSIV-series Function Comparison

Table G.1 shows the functional comparison between the OSIV-series (M-series) and COBOL97.

GS-series

The successor to the M-series; it is the same as the M-series.

Global server

GS-series (M-series) as it is seen from the system.

GS program

A program that runs with the GS-Series (M-series).

GS-specific function

A function that can only be used with the GS-series (M-series) system.

F.1 Function comparison

The following symbols are used in the "Comparison" column of Table G.1:

A : Can be used in the same manner as the OSIV-series.

B : Can be used under given conditions in the same manner as the OSIV-series.

C : Cannot be used with the OSIV-series. (The function is specific to this system or incompatible with the OSIV-series.)

D : Compilation can be done, but the function is disabled during execution.

E : Can be used, but the function works differently than when it works with the OSIV-series.

F : Cannot be used with this system.

Table F.1 Functional comparison between the OSIV-series and this system

Function Classification			Comparison	Remarks
Classification		Function outline		
Character set		All types of characters that can be used in the program	A	
Code		Unicode	C	
		System dependence	A	
COBOL words	User-defined word	All types of user-defined words	A	Use of the underscore (_) character has a specific function in this system.
	Figurative constant	All figurative constants that can be used in the program	A	
	Special register	SHIFT-IN,SHIFT-OUT	D	
		PROGRAM-STATUS, RETURN-CODE	B	The attributes are different. OSIV-series: S9(4)BINARY This system: S9(18)COMP-5
		Other than the above	A	
	Function name	SYSPUNCH, STACKER-01 to 12, CSP, S01, S02 SYSPCH, BUSHU, SOKAKU, ON-YOMI, KUN-YOMI	D	
		SWITCH-8, SYSERR	C	

Function Classification			Comparison	Remarks
Classification		Function outline		
		CHANNEL02 to 12, C02 to C12	B	The FCB control statement must be specified.
		Other than the above	A	
COBOL words	Literal	National literal, National alphanumeric literal, National association literal, National language literal	F	
		Hexadecimal literal, National hexadecimal literal, National code literal	E	Note differences among codes.
		Other than the above	A	
	Others	Specification of quotation mark as a constant	C	OSIV-Series: Compile option APOST/QUOTE is followed This system: Automatically determined. Free format
Method of writing program	Reference format	Sequence number	A	
		Free format	C	
		Fixed format, variable format	A	
Data definition	Data description	All clauses that can be described in the data description entry	A	
		Named literal (78 item)	C	
		Type definition	C	
		Data item that refers to type	C	
	Data type	BINARY-CHAR UNSIGNED BINARY-SHORT BINARY-LONG BINARY-DOUBLE	C	
		Other than the above	A	
Expression	Arithmetic expression	Binary arithmetic operator, unary arithmetic operator	A	
	Conditional expression	All relational operators that can be used	A	
	Linkage expression	Use of linkage expression	C	
	Class condition	All class conditions that can be used	A	

Function Classification			Comparison	Remarks
Classification		Function outline		
	Other conditions	Condition-name condition, sign condition, switch-status condition	A	
Nucleus	Environment definition	SUBSCHEMA-NAME paragraph	F	
		ALPHABET clause (EBCDIC)	C	
		Other than the above	A	
Sequential file	Environment definition	APPLY WRITE-ONLY clause, MULTIPLE FILE TAPE clause, RERUN clause, PASSWORD clause, RESERVE AREA clause	D	
		Data name specified in the ASSIGN clause, DISK specified in the ASSIGN clause, PRINTER specified in the ASSIGN clause, LOCK MODE clause	C	
		Other than the above	A	
	File definition	CODE-SET clause	D	
		BLOCK CONTAINS clause	B	No functional meaning for this system. The program that was operating with the OSIV-series operates as before.
		Other than the above	A	
	Input-output statement	WITH LOCK specified in an input-output statement, UNLOCK statement	C	
		Other than the above	A	
	I control record	Form overlay	B	KOL2 only
Line sequential file		All	C	
Relative file	Environment definition	PASSWORD clause, RERUN clause	D	
		Data name specified in the ASSIGN clause, DISK specified in the ASSIGN clause, LOCK MODE clause	C	
		Other than the above	A	
	File definition	CODE-SET clause	D	
		BLOCK CONTAINS clause	B	No functional meaning for this system. The program that was operating with the OSIV-series operates as before.
		Other than the above	A	

Function Classification			Comparison	Remarks
Classification		Function outline		
	Input-output statement	WITH LOCK specified in an input-output statement, UNLOCK statement	C	
		Other than the above	A	
Indexed file	Environment definition	PASSWORD clause, RERUN clause	D	
		Data name specified in the ASSIGN clause, DISK specified in the ASSIGN clause, LOCK MODE clause	C	
		Permission to form a single key with multiple data items	B	ESP III system only. Can be used with this system under given conditions. (*)
		Other than the above	A	
	File definition	CODE-SET clause	D	
		BLOCK CONTAINS clause	B	No functional meaning for this system. The program that was operating with the OSIV-series operates as before .
		Other than the above	A	
	Input-output statement	WITH LOCK specified in an input-output statement, UNLOCK statement	C	
		POSITIONING POINTER specified in the START statement	F	ESP III system only
		Other than the above	A	
Sort-merge	Environment definition	File-identifier constant of ASSIGN clause	C	
	Mnemonic name	BUSHU,SOKAKU, ON-YOMI,KUN-YOMI	D	
	Special register	SORT-CORE-SIZE	E	
		SORT-MESSAGE	D	
		Other than the above	A	
	Others	All	A	
Inter-program communication	PROCEDURE DIVISION	WITH specification	F	
		BY VALUE specification	C	
		WITH specification	C	
		RETURNING specification	C	
		Other than the above	A	
	Others	All	A	

Function Classification			Comparison	Remarks
Classification		Function outline		
Source text manipulation	COPY statement	OF/IN SYSDBDCT specification	F	ESP III system only
		OF/IN XMDLIB and XFDLIB specification	C	
		Library text name literal	C	
		JOINING specification only	C	
Report writer feature	File definition	BLOCK CONTAINS clause, CODE clause	F	
		Other than the above	F	
	Others	All	F	
Presentation file	Environment definition	CMD, TRM, WST, DSP, APL and ACM specified in the SYMBOLIC DESTINATION clause	F	
		APPLY MULTICONVERSATION-MODE clause	C	
		PROCESSING TIME clause	D	
		DESTINATION CONTROL clause	D	
		MESSAGE SEQUENCE clause	D	
		Other than the above	A	
	File definition	EXTERNAL clause	B	Cannot be specified for a file opened by the OPEN statement with the INPUT or I-O phrase in the OSIV series.
	Input-output statement	All	A	
	Special register	All	A	
		Forms overlay	B	KOL2 only
		Screen form descriptor	B	Note the available function scope.
Debugging functions		COUNT	A	
		CHECK	E	
		CHECK(EXTEND)	D	
		CHECK(Other than the above)	C	
		TRACE	C	
		Other than the above	D	
Segmentation		All	D	Compilation only if used with the OSIV-series
Communication		All	D	

Function Classification			Comparison	Remarks
Classification		Function outline		
Extension	System control	All	D	
	Network database	All	F	
	AIM/RDB	All	F	
	SD function	All	A	
Floating point		All	E	Because OSIV system and the representation are different, the result might be different.
Intrinsic function		CURRENT-DATE function	A	
		Other than the above	C	
Screen handling		All	C	
Command line argument handling and environment variable handling		All	C	
Object-oriented features		All	C	
Compiler option customizing function		All	F	
User tailoring function		All	F	

* : For details, refer to "How to use Indexed file".

F.2 Checking Program Operation

Programs created in common function scope can be checked under this system. Some functions may cause program execution methods and execution results to differ between systems.

The parameter of OSIV system is passed in the during starting of the program and the parameter is passed

When operating a program using the inter-program communication function, if the program is activated through a system, the method of specifying the parameters to be passed to the program is different.

To pass parameters in the global server system format under this system, use the initialization file (COBOL85.CBR) or Runtime Environment Setup window to specify the parameters. Refer to "5.3 Setting Runtime Environment Information" for the method of specifying the global server system format parameters in the initialization file (COBOL85.CBR) and the Runtime Environment Setup window.

Example

- [COBOL program description]

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. A.
*>  :
 LINKAGE SECTION.
 01 wparam.
   03 wparam-length PIC 9(4) BINARY.
   03 wparam-character-string.
     05 wchara-1 PIC X
        OCCURS 1 TO 100 TIMES
        DEPENDING ON wparam-length.
 PROCEDURE DIVISION USING wparam.
*>   :
```

- [Execution method and result with OSIV series]
    - [Input command]

      CALL 'X9999.A.LOAD(A)' 'ABCDE'

    - [Parameter contents]

5	A	B	C	D	E

- [Initialization file for getting same results with OSIV series]
    - [The contents of the initialization file]

```
[A]
 :
 :
@MGPRM="ABCDE"
 :
```

Note

Only one parameter can be passed on.

If a program uses a function specific to the OSIV-series, special processing is required for operating the program under this system.

- To substitute another resource and operate a program using the communication function, execute input-output statements for the sequential file. Check the contents of the sequential file to determine whether the program operates as expected.

The program that makes signed zoned decimal item an object of comparison is compiled specifying compiler option NOZWB and operated

When the program that compares signed zoned decimal item with alphabetic and alphanumeric field (alphabetic and alphanumeric data item, alphabetic item, alphanumeric edited data item, numeric edited data item, nonnumeric literal and figurative constant (other than ZERO)) is compiled specifying compiler option NOZWB, the operation result might be different in OSIV system and this system.

This is because the representation form of data is different.

The program is compiled specifying compiler option ZWB and confirm the operation result.

Example

- [COBOL program description]

```
IDENTIFICATION DIVISION.
   :
 WORKING-STORAGE SECTION.
   :
  01  A PIC S9(6) VALUE 200912.
  01  B PIC  9(8) VALUE 20091201.
   :
 PROCEDURE DIVISION
        IF (A <= B(1:6))
   :
```

- [Data representation and comparison result in OSIV ststem]

  "A => F2F0F0F9F1C2" and "B => F2F0F0F9F1F2" are compared.

Comparison result : A < B

- [Data representation and comparison result in this system]

  "A => 323030393142" and "B => 323030393132" are compared.

  Comparison result : A > B

Note

- If a hexadecimal non-numeric literal and national hexadecimal nonnumeric literal are used in the program, take the code into consideration.
- If the presentation file function is used, the expansion format of a presentation file record fetched from a screen and form descriptors (this system) or format descriptor (OSIV-series) is different when the COBOL source program is compiled.
- If a function name (CHANNEL02 to CHANNEL12, or C02 to C12) is used, the FCB control statement is required. The FCB control statement format used with the global server ADJUST is also used under this system.
- The available function scope of forms overlay and screen form descriptor varies depending on the operating system used. Refer to "Notes on OS" (system.txt) of FORM for details.
- For conversion of the screen form descriptor, refer to "Host and Workstation Linkage" in the OS IV PSAM User's Guide.

Appendix G Special Windows Subroutines

This appendix describes NetCOBOL subroutines. The topics are as follows:

- Subroutines called by COBOL programs.
- Subroutines used to link to another language.

G.1 Subroutines Called by COBOL programs

This section explains the subroutines provided by NetCOBOL for COBOL programs.

G.1.1 Subroutine for Obtaining the Instance Handle

The instance handle of the Windows COBOL application can be obtained by using subroutine JMPBWINS.

Data definition

```
01  data-name.
  02 data-name-1    PIC S9(18)   COMP-5.
  02 FILLER         PIC X(24).
```

Specification of the CALL statement

```
CALL "JMPBWINS" USING data-name.
```

Interface

For data-name, specify the area for storing the instance handle received by the subroutine. The instance handle is stored in data-name-1.

Note

- When the subroutine is called with the dynamic program structure, the following entry information is required. Refer to "5.4.2 Entry Information for Subprograms" for the method of specifying entry information

```
[ENTRY]
 JMPBWINS=F4AGPRCT.DLL
```

- When calling the subroutine with dynamic-link structure, link F4AGCIMP.LIB in linking the calling program.

G.1.2 Subroutine for Outputting an Event Log

The user can output information such as the character strings defined in the COBOL source program to the event log by using subroutine COB_REPORT_EVENT.

The specification method is explained below.

Data definition

```
  WORKING-STORAGE SECTION.
*> Data items specified in the USING clause
    01 data-name-1.
     02 event-number              PIC 9(9)  COMP-5.
     02 event-type                PIC 9(4)  COMP-5.
     02 FILLER                    PIC 9(4)  COMP-5.
     02 event-data.
      03 data-length              PIC 9(9)  COMP-5.
      03 data-address             POINTER.
     02 detailed-error-information  PIC 9(9)  COMP-5.
     02 FILLER                      PIC 9(9)  COMP-5.
     02 source-name                 PIC X(256).
     02 description                 PIC X(1024).

*> Data item specified in the RETURNING clause
    01 data-name-2                  PIC S9(9)  COMP-5.

  CONSTANT SECTION.
*> Type values
    01 type-Information    PIC  9(4)  COMP-5  VALUE 0.
    01 type-Warning        PIC  9(4)  COMP-5  VALUE 1.
    01 type-Error          PIC  9(4)  COMP-5  VALUE 2.
```

Specification of the CALL statement

```
INITIALIZE data-name-1.
CALL   "COB_REPORT_EVENT" USING     data-name-1
                          RETURNING data-name-2.
```

Interface

The interface is explained as follows.

- For event-number, specify a value from 0 to 999 for the event ID.
- For event-type, specify a value from 0 to 2. Specify 0 for Information, 1 for Warning, and 2 for Error.
- For data-length, specify the length of the data output area.
- For data-address, specify the start address of the data output area.
- If subroutine processing failed, a detail error code may be returned to detailed-error-information.
- For source-name, specify space or the registry key name defined in the output destination computer. If space is specified, "NetCOBOL Application x64" is output.
- For description, specify text (character string) using up to 1,024 bytes.
- A return code is set in data-name-2.

Return Codes

A return code from the subroutine is set in data-name-2.

- Return code 0 : Information has been output to the event log successfully.
- Return code 1 : An invalid event number is specified.
- Return code 2 : An invalid type is specified.
- Return code 3: The operating system is not Windows(x64).
- Return code 99: Event log output failed for a reason other than the above. A system error code is returned to detailed-error-information of data-name-1. Remove the cause of error while referring to "System Error Codes" in "NetCOBOL Messages" or Visual C++ Online Help.

Note

When using the event log output subroutine, note the following:

- To prevent malfunction by initialization failure of data-name-1, it is recommended to initialize data-name-1 with the INITIALIZE statement immediately before setting it to a value.
- The event log can be output to another computer (Windows(x64)) in the network. The output destination computer is the same as the runtime message event log output destination. As with the runtime messages, NetCOBOL or NetCOBOL runtime system

must be installed in the output destination computer. See "18.2.1 Function for Outputting Runtime Messages to the Event Log" for details.

- To output a character string other than the default to source, set information in advance in the registry in the output destination computer. To set or delete registry information for this function, use the "event log output subroutine registry key add/delete" tool.

  A user having registry key access authority (value referencing or setting, or subkey creation or deletion) such as a user in the Administrators group should set or delete registry key information.

  Character string of "NetCOBOL" and "NetCOBOL SNAP" are reserved by this product and cannot be specified for source.

- The COBOL interface allows a character string to be specified for source using up to 256 bytes but the size is dependent on the quantitative limit of the system.

- The following entry information is needed if you specify a DLOAD option during compilation of the COBOL program that invokes a subroutine. For how to specify such entry information, see "5.4.2 Entry Information for Subprograms".

```
[Program-name.ENTRY]
 COB_REPORT_EVENT=F4AGEFNC.DLL
```

G.1.3 Subroutine for Obtaining a Process ID

This subroutine can obtain the process ID of the process by using subroutine COB_GET_PROCESSID.

The specification method is explained as follows.

Data definition

```
01  data-name  PIC  9(9) COMP-5.
```

Specification of the CALL statement

```
CALL "COB_GET_PROCESSID" USING  BY  REFERENCE data-name.
```

Interface

For data-name, specify the area in which the process ID posted by the subroutine is to be stored.

Note

When using the process ID acquisition subroutine, note the following:

When the subroutine is called with the dynamic program structure, the following entry information is required. See "5.4.2 Entry Information for Subprograms" for information on how to specify entry information.

```
[Program-name.ENTRY]
  COB_GET_PROCESSID=F4AGEFNC.DLL
```

- When calling the subroutine with dynamic-link structure, link F4AGCIMP.LIB in linking the calling program.

When subroutine COB_GET_THREADID is called, the value of the special register PROGRAM-STATUS is updated to the irregular value. Please describe data item (PIC S9(9) COMP-5) of the dummy in RETURNING clause of CALL statement to prevent it.

G.1.4 Subroutine for Obtaining a Thread ID

This subroutine can obtain the thread ID of the thread by using subroutine COB_GET_THREADID.

The specification method is explained as follows.

Data definition

```
01  data-name  PIC  9(9) COMP-5.
```

Specification of the CALL statement

```
CALL  "COB_GET_THREADID" USING  BY  REFERENCE data-name.
```

Interface

For data-name, specify the area in which the thread ID posted by the subroutine is to be stored.

Note

When using the thread ID acquisition subroutine, note the following:

When the subroutine is called with the dynamic program structure, the following entry information is required. See "5.4.2 Entry Information for Subprograms" for information on how to specify entry information.

```
[Program-name.ENTRY]
  COB_GET_THREADID=F4AGEFNC.DLL
```

- When calling the subroutine with dynamic-link structure, link F4AGCIMP.LIB in linking the calling program.

When subroutine COB_GET_THREADID is called, the value of the special register PROGRAM-STATUS is updated to the irregular value. Please describe data item (PIC S9(9) COMP-5) of the dummy in RETURNING clause of CALL statement to prevent it.

G.1.5 Deadlock Exit Subroutine

When using the NetCOBOL database functions (ODBC), or precompiler, if a deadlock is recognized in a program accessing the database, the deadlock exit subroutine is used to return control to the deadlock procedure defined with a USE FOR DEAD-LOCK statement. For more details, see "17.2.14 Deadlock Exits".

The specification method is explained below.

Data Definition

There are no parameters.

Return Codes

There is no return code from the subroutine.

Specification of the CALL Statement

```
CALL  "COB_DEADLOCK_EXIT".
```

CALL Conditions

When databases are accessed, a return code is set in SQLSTATE. If the return code is a value that indicates a deadlock, the deadlock exit subroutine can be invoked to transfer control to a procedure designed to recover from a deadlock situation.

Note

- If the program that invoked the subroutine, or one of its parent programs doesn't contain a deadlock exit, the deadlock exit schedule fails, the runtime error message JMP0024I-U is output, and the program ends abnormally.

- If any programs between the program that invoked the subroutine and the program containing the deadlock exit are written in a different language, such programs are not recovered. Also, when restarting processes following a deadlock exit, programs written in a different language must be re-entered. This means that programs written in a different language must be re-enterable and not require resource recovery.

- If the subroutine is invoked from a program written in a language other than COBOL, operations cannot be guaranteed.
- It is up to the COBOL code to determine whether or not a program is deadlocked as described under "CALL Conditions" above.
- The subroutine's behavior is not guaranteed for any purpose other than responding to a deadlock.
- The COB_DEADLOCK_EXIT subroutine can be used in multithreaded environments.

G.1.6 The Subroutines for Memory Allocation

COBOL provides two subroutines for memory allocation:

- COB_ALLOC_MEMORY: Memory is dynamically allocated.
- COB_FREE_MEMORY: Dynamically allocated memory is freed.

When COB_FREE_MEMORY is not called, memory is freed as follows:

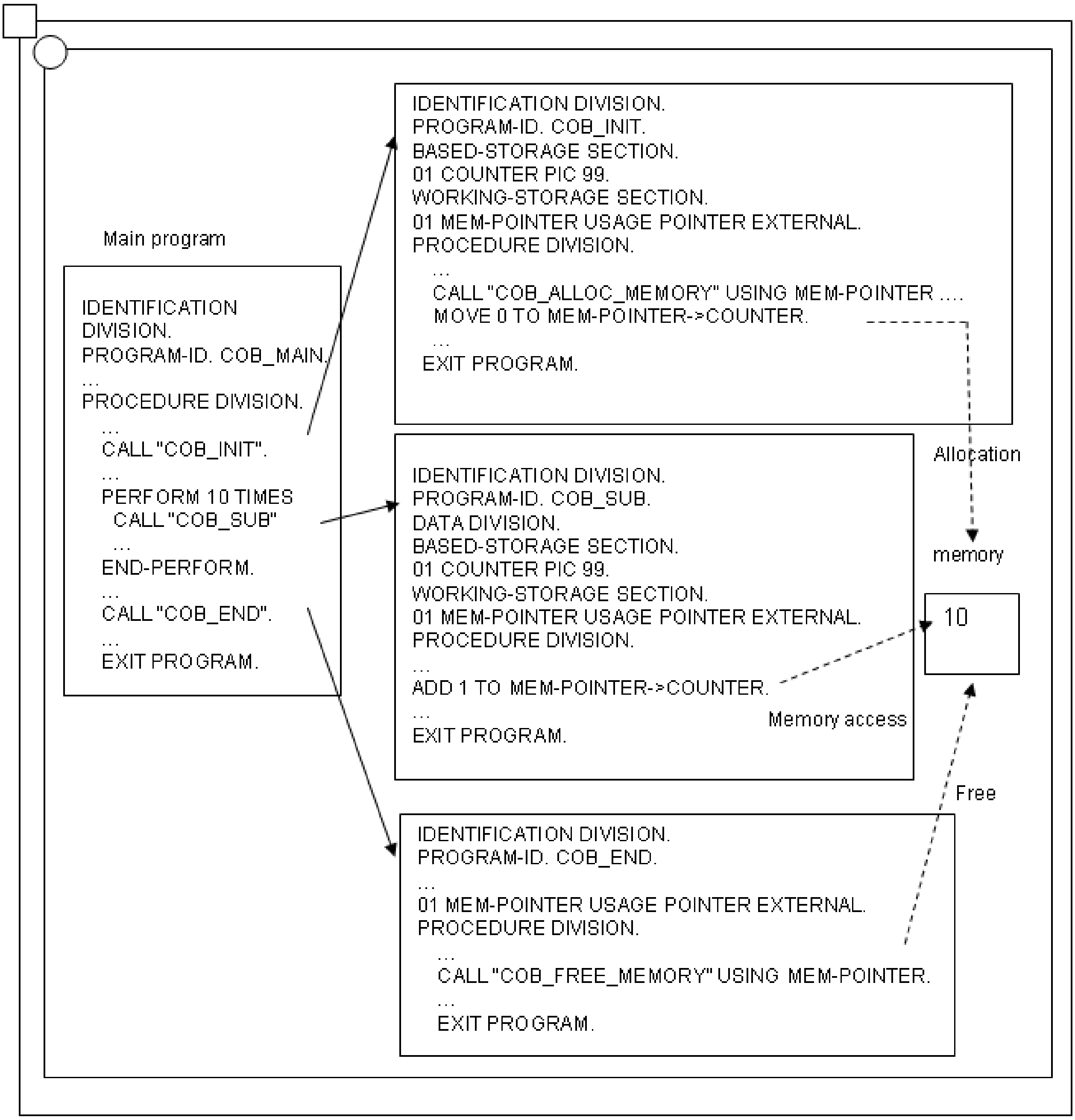
Thread Mode	Type of memory specified with COB_ALLOC_MEMORY	Memory Freed
Singlethread mode	-	When execution environment is closed
Multithread mode	Process	When execution environment is closed
	Thread	When run unit is ended

The type of memory specified with COB_ALLOC_MEMORY is meaningless in singlethread mode. In multithread mode, the type of the memory is selected according to the purpose. When memory is shared between threads, "process" is selected. When memory used is for a specific thread, "thread" is selected. Refer to "19.2.2 What is a Multithread Program?" for more information about singlethread mode and the multithread mode.

Singlethread mode

When the main program is COBOL

When the main program is COBOL, subroutine COB_ALLOC_MEMORY is called by program COB_INIT and the memory is allocated. There is no difference in operation between the "process" or "thread" types of memory. The allocated memory can be accessed from the program COB_SUB that exists on the same run unit.



```
IDENTIFICATION
DIVISION.
PROGRAM-ID. COB_MAIN.
...
PROCEDURE DIVISION.
  ...
  CALL "COB_INIT".
  ...
  PERFORM 10 TIMES
    CALL "COB_SUB"
    ...
  END-PERFORM.
  ...
  CALL "COB_END".
  ...
  EXIT PROGRAM.
```

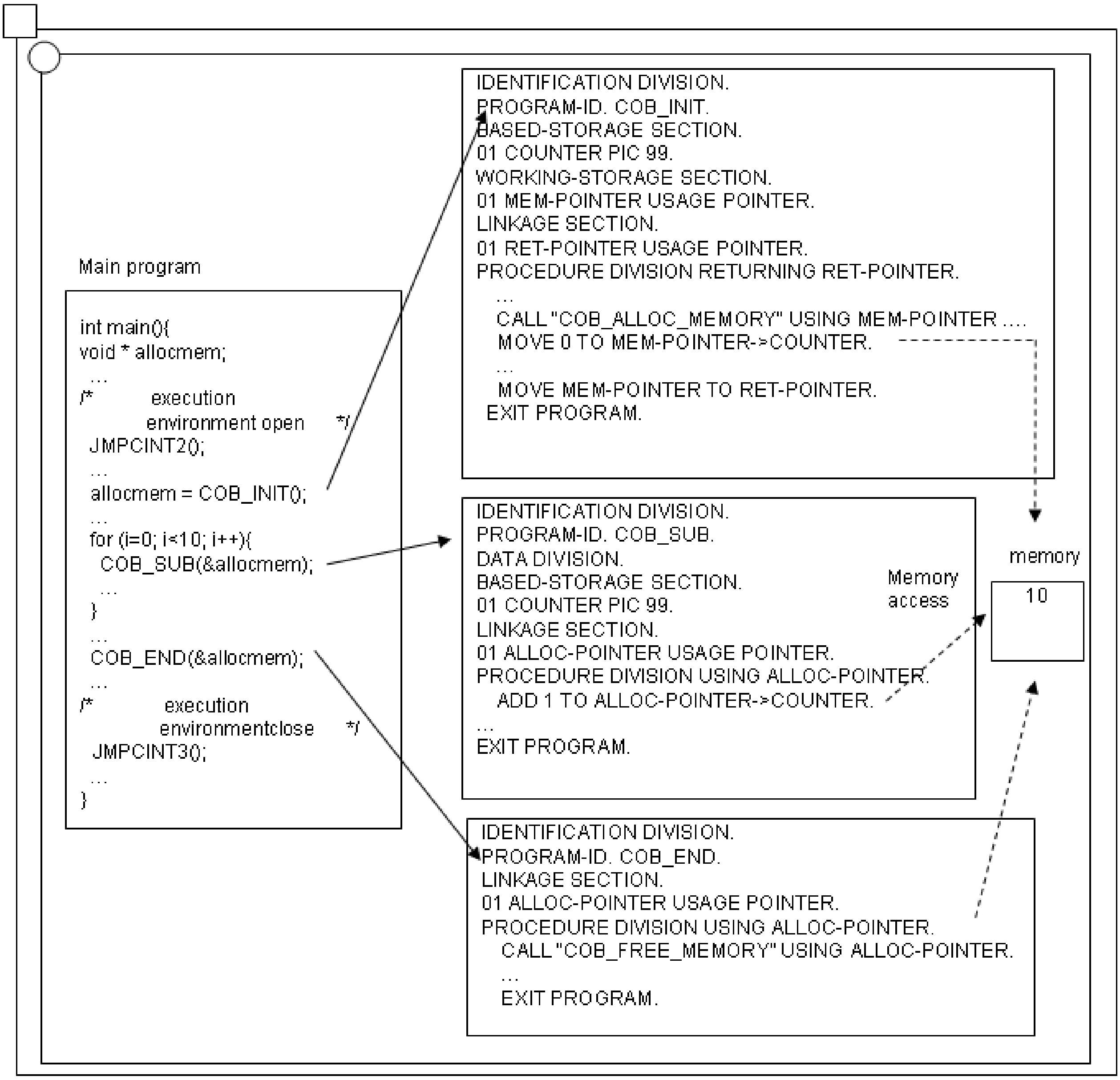
```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_INIT.
BASED-STORAGE SECTION.
01 COUNTER PIC 99.
WORKING-STORAGE SECTION.
01 MEM-POINTER USAGE POINTER EXTERNAL.
PROCEDURE DIVISION.
  ...
  CALL "COB_ALLOC_MEMORY" USING MEM-POINTER ....
  MOVE 0 TO MEM-POINTER->COUNTER.
  ...
 EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_SUB.
DATA DIVISION.
BASED-STORAGE SECTION.
01 COUNTER PIC 99.
WORKING-STORAGE SECTION.
01 MEM-POINTER USAGE POINTER EXTERNAL.
PROCEDURE DIVISION.
...
ADD 1 TO MEM-POINTER->COUNTER.
...
EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_END.
...
01 MEM-POINTER USAGE POINTER EXTERNAL.
PROCEDURE DIVISION.
  ...
  CALL "COB_FREE_MEMORY" USING MEM-POINTER.
  ...
  EXIT PROGRAM.
```

When the main program is not COBOL

When the main program is a different language and operating two or more programs on the same run unit, operation similar to the COBOL program can be achieved by calling JMPCINT2 and JMPCINT3. When program COB_INIT is ended, the allocated memory is freed when neither JMPCINT2 nor JMPCINT3 are called, and the memory cannot be accessed from the program COB_SUB. Refer to "G.2 Subroutines Used to Link to Another Language" for JMPCINT2 and JMPCINT3 usage.



```
int main(){
void * allocmem;
 ...
/*        execution
         environment open      */
 JMPCINT2();
 ...
 allocmem = COB_INIT();
 ...
 for (i=0; i<10; i++){
   COB_SUB(&allocmem);
   ...
 }
 ...
 COB_END(&allocmem);
 ...
/*          execution
           environmentclose     */
 JMPCINT3();
 ...
}
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_INIT.
BASED-STORAGE SECTION.
01 COUNTER PIC 99.
WORKING-STORAGE SECTION.
01 MEM-POINTER USAGE POINTER.
LINKAGE SECTION.
01 RET-POINTER USAGE POINTER.
PROCEDURE DIVISION RETURNING RET-POINTER.
   ...
   CALL "COB_ALLOC_MEMORY" USING MEM-POINTER ....
   MOVE 0 TO MEM-POINTER->COUNTER.
   ...
   MOVE MEM-POINTER TO RET-POINTER.
 EXIT PROGRAM.
```

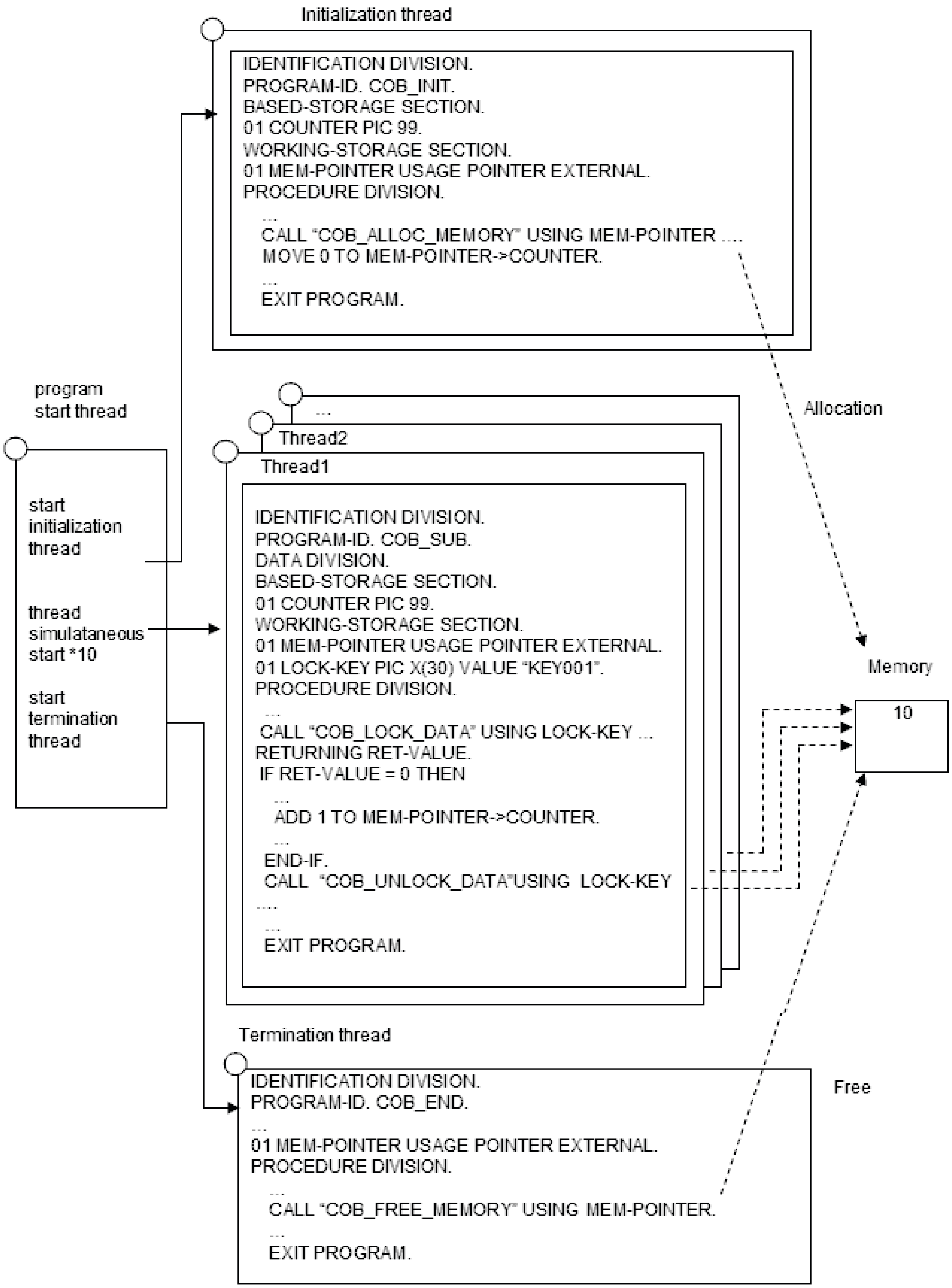
```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_SUB.
DATA DIVISION.
BASED-STORAGE SECTION.
01 COUNTER PIC 99.
LINKAGE SECTION.
01 ALLOC-POINTER USAGE POINTER.
PROCEDURE DIVISION USING ALLOC-POINTER.
   ADD 1 TO ALLOC-POINTER->COUNTER.
...
EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_END.
LINKAGE SECTION.
01 ALLOC-POINTER USAGE POINTER.
PROCEDURE DIVISION USING ALLOC-POINTER.
   CALL "COB_FREE_MEMORY" USING ALLOC-POINTER.
   ...
   EXIT PROGRAM.
```

Multithread mode

Process specification

COB_ALLOC_MEMORY is called by "Process" in program COB_INIT of the initialization thread called first, and the memory is allocated. The memory allocated here can be accessed from program COB_SUB with a different thread because it is not freed until the end of the process. However, when memory is shared between threads as shown in figure, synchronous control by the data lock subroutine is needed. Refer to "19.4 Resource Sharing among Threads" for details about synchronous control between threads.



```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_INIT.
BASED-STORAGE SECTION.
01 COUNTER PIC 99.
WORKING-STORAGE SECTION.
01 MEM-POINTER USAGE POINTER EXTERNAL.
PROCEDURE DIVISION.
  ...
  CALL “COB_ALLOC_MEMORY” USING MEM-POINTER ....
  MOVE 0 TO MEM-POINTER->COUNTER.
  ...
  EXIT PROGRAM.
```

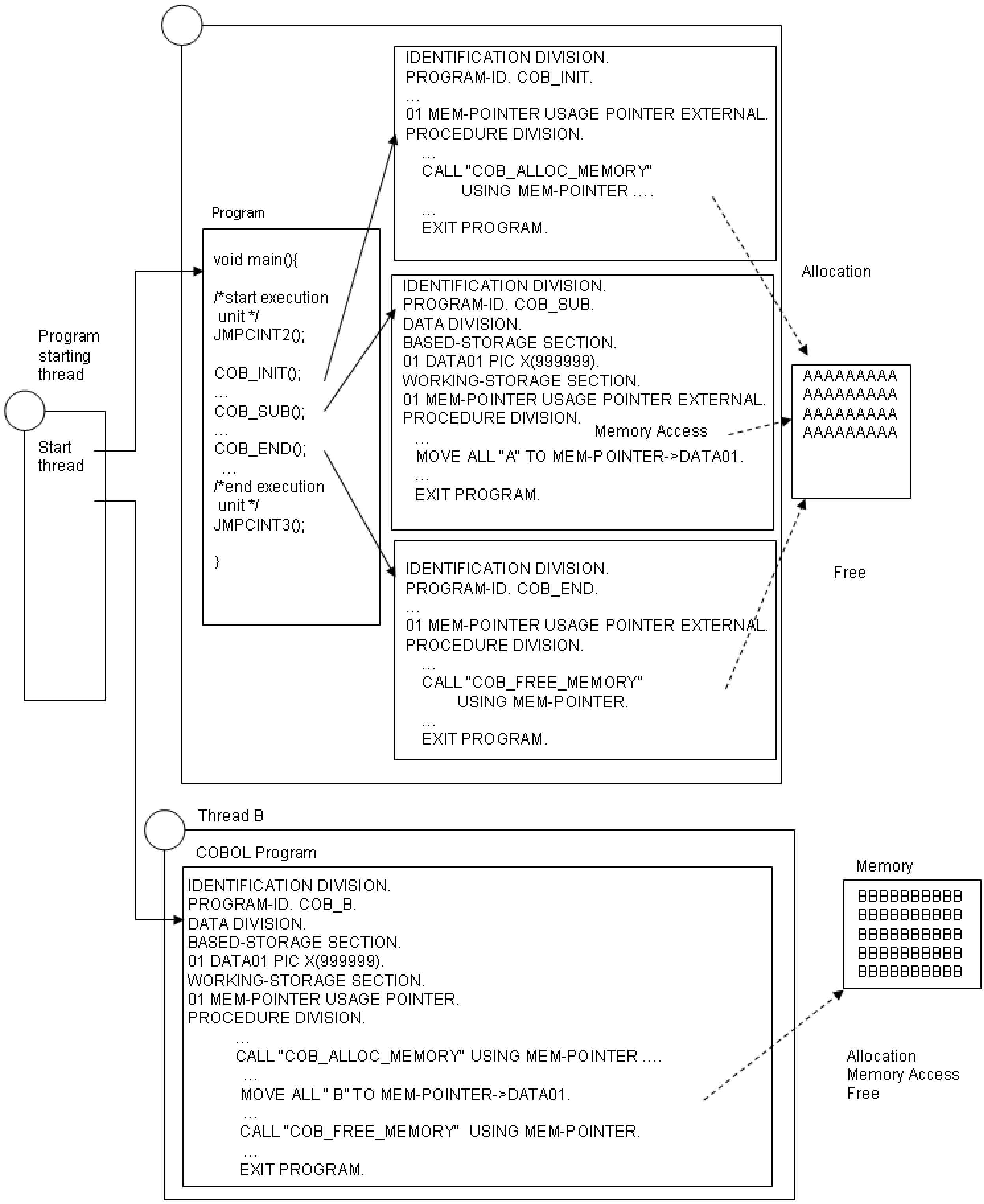
```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_SUB.
DATA DIVISION.
BASED-STORAGE SECTION.
01 COUNTER PIC 99.
WORKING-STORAGE SECTION.
01 MEM-POINTER USAGE POINTER EXTERNAL.
01 LOCK-KEY PIC X(30) VALUE “KEY001”.
PROCEDURE DIVISION.
 ...
 CALL “COB_LOCK_DATA” USING LOCK-KEY ...
RETURNING RET-VALUE.
 IF RET-VALUE = 0 THEN
   ...
   ADD 1 TO MEM-POINTER->COUNTER.
   ...
  END-IF.
  CALL  “COB_UNLOCK_DATA”USING  LOCK-KEY
....
  ...
  EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_END.
...
01 MEM-POINTER USAGE POINTER EXTERNAL.
PROCEDURE DIVISION.
   ...
   CALL “COB_FREE_MEMORY” USING MEM-POINTER.
   ...
   EXIT PROGRAM.
```

Thread specification

In thread A program COB_INIT and thread B program COB_B, COB_ALLOC_MEMORY is called respectively by "Thread" and the memory is allocated.

When the main program of the thread is COBOL, the free processing is done for the non-COBOL language at the call of JMPCINT3, when the main program is ended. Use "Thread" when memory used is for a specific thread,.



```
void main(){

/*start execution
 unit */
JMPCINT2();

COB_INIT();
…
COB_SUB();
…
COB_END();
 …
/*end execution
 unit */
JMPCINT3();

}
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_INIT.
…
01 MEM-POINTER USAGE POINTER EXTERNAL.
PROCEDURE DIVISION.
   …
   CALL "COB_ALLOC_MEMORY"
         USING MEM-POINTER ....
   …
   EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_SUB.
DATA DIVISION.
BASED-STORAGE SECTION.
01 DATA01 PIC X(999999).
WORKING-STORAGE SECTION.
01 MEM-POINTER USAGE POINTER EXTERNAL.
PROCEDURE DIVISION.
 …
 MOVE ALL "A" TO MEM-POINTER->DATA01.
 …
 EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_END.
…
01 MEM-POINTER USAGE POINTER EXTERNAL.
PROCEDURE DIVISION.
   …
   CALL "COB_FREE_MEMORY"
         USING MEM-POINTER.
   …
   EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COB_B.
DATA DIVISION.
BASED-STORAGE SECTION.
01 DATA01 PIC X(999999).
WORKING-STORAGE SECTION.
01 MEM-POINTER USAGE POINTER.
PROCEDURE DIVISION.
         …
         CALL "COB_ALLOC_MEMORY" USING MEM-POINTER ....
          …
          MOVE ALL " B" TO MEM-POINTER->DATA01.
          …
          CALL "COB_FREE_MEMORY"  USING MEM-POINTER.
          …
          EXIT PROGRAM.
```

Note

To call a subroutine using DLOAD option, you must use the following entry information file. For details about entry information file, refer to "5.4.2 Entry Information for Subprograms".

```
[Program-name.ENTRY]
  Subroutine-name=F4AGEFNC.DLL
```

COB_ALLOC_MEMORY

The COB_ALLOC_MEMORY subroutine can be used to dynamically allocate memory.

Call Format

```
CALL  "COB_ALLOC_MEMORY" USING BY REFERENCE data-name-1
                               BY VALUE data-name-2
                               BY VALUE data-name-3
                         RETURNING data-name-4.
```

Explanation of Parameter

Data-name-1

```
01 data-name-1   USAGE POINTER.
```

Specify the pointer to allocated memory. The allocated memory is not initialized.

Data-name-2

```
01 data-name-2   PIC  9(9) COMP-5.
```

Specify the number of bytes to be allocated.

Data-name-3

```
01 data-name-3   PIC  9(9) COMP-5.
```

Specify the type of memory required. The following values can be specified:

- 0 : Allocate the memory in process scope. In this case, the allocated memory exists until the runtime environment closes.
- 1 : Allocate the memory in thread scope. In this case, the allocated memory exists until the run unit ends.

Explanation of Return value

Data-name-4

```
01 data-name-4   PIC S9(9) COMP-5.
```

If the operation succeeds, the return value is 0. If the operation fails, the return value is as follows:

- -1 : Parameter error
- -2 : Insufficient memory
- -3 : The run unit is ended

COB_FREE_MEMORY

The COB_FREE_MEMORY subroutine can be used to free dynamically allocated memory.

Call Format

```
CALL  "COB_FREE_MEMORY" USING BY REFERENCE data-name-1
                        RETURNING data-name-2.
```

Explanation of Parameter

Data-name-1

```
01 data-name-1   USAGE POINTER.
```

Specify the pointer returned when the memory was allocated using COB_ALLOC_MEMORY.

Explanation of Return value

Data-name-2

```
01 data-name-2   PIC S9(9) COMP-5.
```

If the operation succeeds, the return value is 0. If the operation fails, the return value is -1 since the allocated memory is already freed, destroyed, or not allocated using COB_ALLOC_MEMORY.

Note

When you allocate the memory in thread scope, allocate and free it in the same run unit. If the run unit is different, the processing to free dynamically allocated memory fails.

G.1.7 Subroutines to forcibly end the CALL Process

COBOL offers the COB_EXIT_PROCESS subroutines to forcibly end the CALL process and all of its threads.

A subroutine that forcibly terminates the process is provided in COBOL.

The method of ending the process includes the following two methods.

The process is normally ended

The process is terminated normally, and the value specified for the parent process is returned. It is convenient to use this to return a value from the subprogram to the parent process.

The process is abnormally ended

An application error is generated, and the process is terminated abnormally. Diagnosis function can confirm the generation part of the error, the program linkage, and the state of the application from the current state and the output diagnosis report. When diagnosis function is assumed to be invalid, the following functions can be used similar to other application errors:

- Visual C++ Just-In-Time Debugger
- Windows Error reporting

Note

We recommend that you use the call only when a critical problem has been detected in the application. Calling the routine in a multi-threaded application could cause severe problems when threads are terminated in the middle of critical file operations.

In sub routine COB_EXIT_PROCESS call, unconditional close of the file might not be done.

Please call this routine in the file under the opening after it closes as much as possible.

COB_EXIT_PROCESS

Call Format

```
CALL  "COB_EXIT_PROCESS" USING BY VALUE data-name-1
                               BY VALUE data-name-2
                         RETURNING data-name-3.
```

Parameter

Data-name-1

```
01 data-name-1   PIC 9(9) COMP-5.
```

Specify how to end the CALL process. The following values can be specified:

- 0 : Terminate the calling process normally and return the value specified in data-name-2 to the parent of the calling process.
- 1 : An application error is generated, and the process is terminated abnormally.

Data-name-2

```
01 data-name-2   PIC 9(9) COMP-5.
```

Specify the value returned to the parent of the calling process when data-name -1 is 0. The valid value range is 0-255.

Return value

Data-name-3

```
01 data-name-3   PIC S9(9) COMP-5.
```

If the operation succeeds, there is no return value. If the operation fails due to a parameter error, the return value is -1.

G.1.8 Endian conversion Subroutines

In NetCOBOL, the endian of national data items can be selected.

Functions to convert little-endian to big-endian, and big-endian to little-endian are provided.

Call Format

#NATBETOLE

A data item converted from big-endian to little-endian.

```
CALL  "#NATBETOLE" USING [BY REFERENCE] identifier.
```

#NATLETOBE

A data item converted from little-endian to big-endian.

```
CALL  "#NATLETOBE" USING [BY REFERENCE] identifier.
```

Explanation of Parameter

- When the identifier is an elementary item, the converted items can be national data items and national editing data items only.
- When the identifier is a group item and the following items are included in the subordinate items, the items will not be converted:
  - An item that is neither a national data item or a national editing data item
  - An item for which a REDEFINES clause is specified and the item subordinate to the item for which a REDEFINES clause is specified
  - An item for which a RENAMES clause is specified
- In the following cases, the execution results are not guaranteed:
  - The format description is wrong.
  - When calls #NATBETOLE, the data format of the item or the data format of national item or national editing data item in the item is not big-endian.
  - When calls #NATLETOBE, the data format of the item or the data format of national item or national editing data item in the item is not little-endian.
  - The identifier is a group item and the item uses OCCURS clause with DEPENDING phrase in a subordinate items.
  - The identifier is a data item that is defined in the constant section.
  - The identifier is a data item that is defined by reference modification.

G.2 Subroutines Used to Link to Another Language

This section explains the subroutines provided by COBOL for linking to other languages.

G.2.1 Run Unit Start Subroutine

The run unit start subroutine is used when multiple COBOL programs are called from another language program and operated in a single run unit.

The specification method is explained as follows.

Data definition (calling with C language)

```
Type declaration part
  extern void JMPCINT2(void);

Procedure part
  JMPCINT2();
```

Note

To call the subroutine from a source to be compiled for Visual C++, specify `extern "C" void JMPCINT2 (void);' in the type declaration.

Interface

No parameter is required for calling.

Return Code

The subroutine posts no return code.

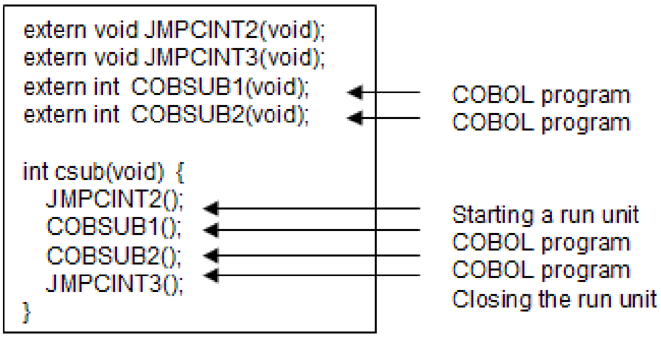
Note

When using the run unit start subroutine, note the following:

After this subroutine is called, the user must call the JMPCINT3 subroutine to close the run unit.

For more information on the end of a run unit, see "9.1.2 COBOL Inter-Language Environment" and "19.3.1 Execution Environment and Run Unit".

Example



```
extern void JMPCINT2(void);
extern void JMPCINT3(void);
extern int  COBSUB1(void);
extern int  COBSUB2(void);

int csub(void) {
   JMPCINT2();
   COBSUB1();
   COBSUB2();
   JMPCINT3();
}
```

G.2.2 Run Unit End Subroutine

After a run unit is started by the run time start subroutine, the run unit end subroutine is used from another-language program to close the run unit.

The specification method is explained as follows.

Data definition (calling with C language)

```
Type declaration
  extern void JMPCINT3(void);

Procedure
  JMPCINT3();
```

Note

To call the subroutine from a source to be compiled for Visual C++, specify `extern "C" void JMPCINT3 (void);' in the type declaration.

Interface

No parameter is required for calling.

Return Code

The subroutine posts no return code.

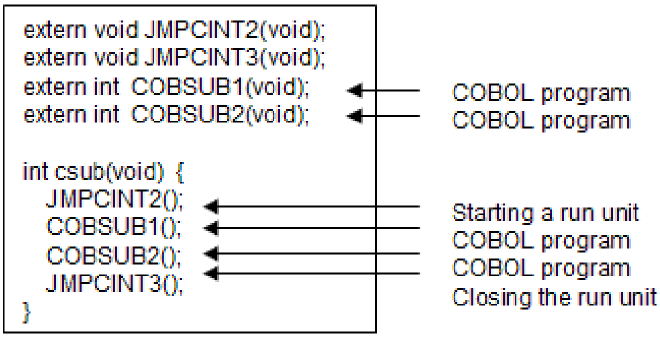
Note

When using the run unit end subroutine, note the following:

Before calling this subroutine, the user must call the JMPCINT2 subroutine to start a run unit.

For more information on the end of a run unit, see "9.1.2 COBOL Inter-Language Environment" and "19.3.1 Execution Environment and Run Unit".

Example



```
extern void JMPCINT2(void);
extern void JMPCINT3(void);
extern int  COBSUB1(void);
extern int  COBSUB2(void);

int csub(void) {
   JMPCINT2();
   COBSUB1();
   COBSUB2();
   JMPCINT3();
}
```

G.2.3 Subroutine for Closing the Runtime Environment

A runtime environment can be closed by calling JMPCINT4 from another-language program after all run units in the process are finished.

The specification method is explained as follows.

Data definition (calling with C language)

```
Type declaration part
  extern void JMPCINT4(void);

Procedure part
  JMPCINT4();
```

Note

To call the subroutine from a source to be compiled for Visual C++, specify `extern "C" void JMPCINT4 (void);' in the type declaration.

Interface

No parameter is required for calling.

Return Code

The subroutine posts no return code.

Note

When using the runtime environment closing subroutine, note the following:

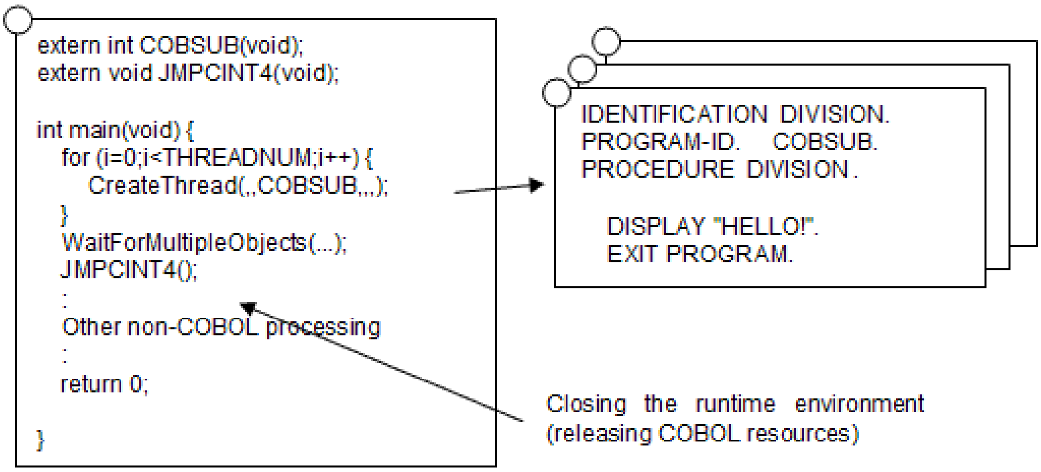
Before this subroutine is called, COBOL program execution in all threads of the process must be finished. If the subroutine is called during COBOL program execution, the COBOL program in execution terminates abnormally.

Before this subroutine is called, the user must call the JMPCINT3 subroutine to close the run unit if the JMPCINT2 subroutine has been called previously to start the run unit.

For more information on the end of a run unit, see "9.1.2 COBOL Inter-Language Environment" and "19.3.1 Execution Environment and Run Unit".

In a single thread program, the runtime environment is closed automatically when all run units end. Therefore, if the JMPCINT4 subroutine is called after run units end in a single thread program, the subroutine returns without responding.

Example



```
extern int COBSUB(void);
extern void JMPCINT4(void);

int main(void) {
   for (i=0;i<THREADNUM;i++) {
      CreateThread(,,COBSUB,,,);
   }
   WaitForMultipleObjects(...);
   JMPCINT4();
   :
   Other non-COBOL processing
   :
   return 0;
}
```

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.    COBSUB.
PROCEDURE  DIVISION.

   DISPLAY "HELLO!".
   EXIT PROGRAM.
```

Appendix H Incompatible Syntax Between Standard and OO COBOL

A number of standard COBOL features cannot be used in class definitions or separately coded methods (using the prototype method definition feature). This appendix lists these features.

H.1 Features not Allowed in Class Definitions

Table I.1 lists the features that cannot be used in any part of a class definition - factory, object or method.

Table H.1 Features not allowed in class definitions

Feature	Description
ANSI`85 standard obsolete elements	All functions that were declared obsolete in the ANSI`85 standard cannot be used in class definitions. These functions are: - Any transcription or relation between an ALL literal in which ALL is immediately followed by a literal of two or more characters and a numeric data item or numeric- edited data item - AUTHOR paragraph, INSTALLATION paragraph, DATE-WRITTEN paragraph, DATE-COMPILED paragraph, and SECURITY paragraph - RERUN clause - MULTIPLE FILE TYPE clause - LABEL RECORD clause - VALUE OF clause - DATA RECORDS clause - ALTER statement - ENTER statement - GO TO statement omitted procedure-name-1 - OPEN statement with REVERSED phrase - STOP statement with literal value
Specification of names using literals	Literals cannot be used to specify class names in CLASS-ID paragraphs and method names in METHOD-ID paragraphs.
SOURCE-COMPUTER paragraph and OBJECT- COMPUTER paragraph	SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs cannot be specified in the ENVIRONMENT DIVISION of class definitions.
APPLY clause	APPLY MULTICONVERSATION-MODE and APPLY SAVED-AREA clauses cannot be specified in INPUT-OUTPUT sections in the ENVIRONMENT DIVISION of FACTORY, OBJECT, and METHOD definitions.
Screen Handling	Screen handling functions cannot be used in FACTORY, OBJECT, and METHOD definitions.
CHARACTER TYPE clause	The CHARACTER TYPE clause cannot be specified in the DATA DIVISION of FACTORY, and OBJECT definitions. Also, the CHARACTER TYPE phrase cannot be specified in the LINKAGE SECTION of PROTOTYPE method definitions.
EXTERNAL clause	The EXTERNAL clause cannot be specified in the DATA DIVISION of FACTORY, and OBJECT definitions.

Feature	Description
	However, the EXTERNAL clause can be specified in the DATA DIVISION of method definitions.
GLOBAL clause	The GLOBAL clause cannot be specified in the DATA DIVISION of FACTORY, OBJECT, and METHOD definitions.
LINAGE clause	The LINAGE clause cannot be specified in FILE description entries in FACTORY and OBJECT definitions. However, the LINAGE clause can be specified in FILE description entries in METHOD definitions as long as it is not used with the EXTERNAL clause.
PRINTING POSITION clause	The PRINTING POSITION clause cannot be specified in the LINKAGE SECTION of METHOD prototype definitions.
Special Register PROGRAM-STATUS	Special Register PROGRAM-STATUS cannot be used in METHOD definitions.
ENTRY statement	ENTRY statements cannot be written in the PROCEDURE DIVISION of METHOD definitions.
ODBC	The DATA BASE functions of ODBC drivers cannot be used in FACTORY definitions.

H.2 Additional Features not Allowed in Separate Method Definitions

Table I.2 lists the features that cannot be used in separately compiled method definitions. These are in addition to the features listed in Table I.1.

Table H.2 Features not allowed in separate method definitions

Function	Explanation
SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs	SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs cannot be specified in the ENVIRONMENT DIVISION the method definition compiled separately.
SPECIAL-NAMES paragraph	The SPECIAL-NAMES paragraph cannot be specified in the ENVIRONMENT DIVISION.
WRITE statements with ADVANCING	WRITE statements with an ADVANCING clause can only be used if one of the following conditions is satisfied: - PRINTER or PRINTER-n (n is an integer from 1 to 9) is specified in the ASSIGN clause. - The statement is specified for files defined in the same compilation unit. - The file is a print file with the FORMAT clause.

Appendix I Other Commands

This section explains the commands other than the complication command and link command provided with COBOL.

I.1 LINK command /DUMP option

Command Synax

```
LINK /DUMP[options] file-name
```

Operand

Each option and the file-name should be separated by one or more blanks.

A full path name or a relative path name can be specified for the file-name.

Options

Table J.2 lists the /DUMP option of LINK command options.

Table I.1 Options of /DUMP option of LINK command

Options	Explanation
/EXPORTS	Displays all the definition information exported from an executable file (EXE) or dynamic link library (DLL). The external symbol names (program names and function names) included in an executable file (EXE) or dynamic link library (DLL) can be displayed using this option.
/IMPORTS	Displays all the definition information imported into an executable file (EXE) or dynamic link library (DLL). The external symbol names (program names and function names) called from an executable file (EXE) or dynamic link library (DLL) can be displayed using this option.
/ALL	Displays all the information (apart from a reverse assembly).
/OUT:output file name	Specifies a file to which the DUMPBIN results are output.

Note

To display a full list of the /DUMP option of LINK command options, simply enter "LINK /DUMP" without any parameters:

```
LINK /DUMP
```

File name

Specifies the name of the file to be examined.

The types of file that can be specified are:

- Executable file (*.EXE)
- Dynamic link library (*.DLL)
- Object file (*.OBJ)
- Import library (*.LIB)
- Standard library (*.LIB)

Example

Example 1

List programs called by a DLL:

```
LINK /DUMP /EXPORTS /OUT:output-file-name  DLL-name
```

Example 2

List called programs in an import library:

```
LINK /DUMP /ALL /OUT:output-file-name Import-library-name
```

Example 3

Find out if there are called programs in an executable file:

```
LINK /DUMP /IMPORTS /OUT:output-file-name Executable-file-name
```

I.2 INSDBINF Command

The following problems have been seen for interoperation between the COBOL compiler and the precompiler:

- Because the compiler error detection line numbers output by the compiler are line numbers in the intermediate file (source file after precompilation), the user must correct the original source program while referring to the intermediate file. (The original source program means a COBOL source program before precompilation in which an embedded SQL statement is written.)
- The builder error jump function does not support an error jump in the original source program.
- The debugger cannot debug the original source program.

To solve the above problems, the INSDBINF command generates an intermediate file in which line number and file name control information is embedded.

Line number control information (#LINE information) is used to post the line numbers in the source program to be precompiled to the compiler or precompiler to be run later.

File name control information (#FILE information) is used to post the file name of the original source program or the name of the file included by the precompiler.

The COBOL compiler can reference line number and file name control information using the intermediate file generated by the INSDBINF command as an input file. Based on control information, the COBOL compiler obtains the file names of the original source program and the included file, and the line numbers in the original source program are related to those in the precompiled source file. Thus, the INSDBINF command can solve prior problems involving interoperation between the COBOL compiler and precompiler.

Command syntax

```
INSDBINF [option-list] input-file-name-1 [input-file-name-2]
```

Operand

Options and file names must be separated by one or more blanks. The parameter enclosed in square brackets can be omitted. Either an absolute path or relative path can be specified for the file name as follows:

option-list

Specify INSDBINF command options. Table J.3, "INSDBINF command options," lists the INSDBINF command options that can be specified.

Table I.2 INSDBINF command options

Option	Explanation
-Iinclude-file-folder-name	When an include file to be processed by the precompiler exists, specify the folder containing the include file following the -I option. If include files exist in multiple

Option	Explanation
	folders, specify as many -I options as there are folders. If two or more -I options are specified, the folders are searched in the order they are specified. If the -I option is specified with no include file folder name or the -I option is omitted, the current folder is searched.
-Sextension[,extension]	Specify the extension of the include file to be processed by the precompiler. If two or more extensions are specified, the command searches for the extensions in the order they are specified. If the file name does not have an extension, specify the character string "None" for the extension. If the -S option is omitted, the command searches for extensions in order of (1) no extension and (2) COB.
-Ooutput-file-name	Specify the name of the file to which the program generated by the command is to be output. If the -O option is omitted, the command assumes that input-file-name-1 with its extension replaced by COB is specified.
-C	Specify when BINARY, COMP, and COMPUTATIONAL are specified as the host variable and the precompiler is not expanded to COMP-5 or COMPUTATIONAL-5. (For Pro*COBOL, specify when "no" is specified for the comp5 option and the output post-precompile expansion source is input.) If this option is omitted, the source is handled as source expanded to COMP-5 or COMPUTATIONAL-5. The default value of the Pro*COBOL comp5 option is "yes".
-D	Specify when only the declaration inside the embedded SQL BEGIN/END DECLARE section is expanded as a host variable. (For Pro*COBOL, specify when "yes" is specified for the declare_section option and the output post-precompile expansion source is input.) If the option is omitted, the declaration outside the embedded SQL BEGIN/END DECLARE section is also handled as a host variable. The default value of the Pro*COBOL declare_section option is "no".
-U	Specify when the code system of original source program is Unicode(UTF-8). If the -U option is omitted, it is treated as an ANSI code page.

input-file-name-1

Specify the file name of the original source program in which an embedded SQL statement is written.

input-file-name-2

Specify the name of the file to which the precompiler-expanded program is output by the precompiler. If this option is omitted, the command assumes that input-file-name-1 with its extension replaced by CBL is specified.

Note on accessing to the ORACLE Database

- When the original source program uses BINARY, COMP, and COMPUTATIONAL as the host variable data types, if "yes" is specified as the value of the Pro*COBOL comp5 option (expand the type to COMP-5), you do not need to use the -C INSDBINF option. If "no" is specified as the value of the comp5 option and the expanded source is input, specify the -C option. Note that this does not work in a byte-swap system. If attempted, the following message is output:

```
Programs expanded with BINARY, COMP, or COMPUTATIONAL do not run in a byte-swap system. Processing
continues.
```

- In Pro*COBOL, when declare_section=yes is specified, the comp5 option is only valid for the BINARY, COMP, or COMPUTATIONAL data types set in the embedded SQL BEGIN/END DECLARE section. To input source expanded by defining BINARY, COMP, or COMPUTATIONAL data outside the BEGIN/END DECLARE section of the original source and specifying "yes" as the value of the declare_section option, specify the -D option. With this option only items within the BEGIN/END DECLARE section are handled as host variables.
- When a mistake is found in the input file and the option specification, the following message is output:

```
Line number control information and file name control information cannot be correctly output.
Check the input file or the command option.
```

    - The file input to this command should be a pre-compiler extract program that inputs an original source program to the pre-compiler. Proper original source programs are always precompiled. Do not correct the pre-compiler extract program.

    - The specification of -C or -D of comp5 or declare_section and this command of Pro*COBOL do not correspond. Specify -C for comp5=no of Pro*COBOL and specify -D for declare_section=yes. Release each option when -C is specified with comp5=yes or -D is specified with declare_section=no.

    - Refer to "SOFTWARE RELEASE GUIDE" when not using any of the above.

- Do not use tab characters in the original source program. Otherwise, this tool generates a wrong output file, thereby causing an error during compilation, or malfunctioning during execution of the debugger. In such a case, change the tab characters in the original source program to blank characters, then precompile the program.

Note on accessing with SymfoWARE/RDB Database

COBOL precompiler for SymfoWARE/RDB database has the option to embed line number control information and file name control information.

If SymfoWARE/RDB database is used, please use its option.

For more information, please refer to "SymfoWARE Server User's Guide".

Verification of Precompiler Operation

Refer to the "NetCOBOL software release guide" for information on precompilers whose operation has been verified and for notes on using these precompilers.

I.3 cobmkmf Command

The cobmkmf command collects the source file names in the current directory, and generates the Makefile automatically. The generated Makefile can be executed by the nmake command, and supports compiling and linking COBOL source programs.

Command Syntax

```
cobmkmf [ -w | -l ] [ MAIN = main-program-name ] [ macro-name = macro-definition-value ]
```

Operand

Insert one or more spaces between each option and its associated file name. Items in [brackets] are optional.

File names can be specified using either an absolute path or a relative path.

-w | -l

Specifies the usage of the generated makefile.

-w : An executable program that uses the COBOL console

-l : For A dynamic link library

Omit this specification to generate an executable program that uses the system console (command prompt window).

MAIN = main-program-name

Specifies the COBOL source file that is to be used as the main program in the Makefile. The COBOL source file does not contain a path name in its specification.

macro-name = macro-definition-value

"macro-name = macro-definition-value" overrides the "macro-name =" line in the makefile.

Enclose character strings that contain spaces with double quotation marks. If a double quotation mark is used in a macro-definition-value, the double quotation mark must be preceded by a backslash: \"

The following macro-names can be used with cobmkmf:

macro-name	macro-definition-value
MAKFILE	Makefile name

macro-name	macro-definition-value
PROGRAM	Executable program or dynamic link library name
COBFLAGS	COBOL compiler option
COBLDFLAGS	COBOL option related to compilation
LDFLAGS	Linkage option
LDLIBS	Library to link (System library etc.)

Generated File

The cobmkmf command generates the following files:

File Name	Description
Makefile	A list of commands that specifies compiling and linking, and the order of execution. If Makefile already exists in the current folder, a backup file called "Makefile.old" is created.
SRCLIST	Specifies source file dependencies. If SRCLIST already exists in the current folder, a backup file called "SRCLIST.old" is created.

The cobmkmf command generates the Makefile assuming that the COBOL source file is named according to the following rules:

- Source file in Program definition

```
program-name.cob
```

- Source file in Class definition

```
class-name.cob
```

- Source file in Separated method definition

```
method-name.cob
```

The source file type is determined by its extension. A list of source file names is output to the SRCLIST file. Ensure that the extension is one of the following before outputting to the SRCLIST file.

Extension	Source File Type
.COB, .COBOL	COBOL source program
.CBL	Library Text
.SMD, .PMD, .PXD	Form Descriptor
.REP	Class Information
.LIB	Library and Import Library

Target

The following targets can be used in the generated Makefile:

Target	Function
all	Compile and link to create the executable program or dynamic link library.
clean	Delete all objects and debugging information, etc.
rebuild	Execute nmake clean and then execute nmake all.
srclist	Search the current directory and update the SRCLIST.

Target	Function
depend	Update dependencies according to the SRCLIST.
opt	Execute nmake all in COBOL optimization mode.
debug	Execute nmake all so that the COBOL debug function can be used.
thread	Execute nmake all in COBOL multi-thread mode.
trace	Execute nmake all so that the COBOL trace function can be used.
check	Execute nmake all so that the COBOL check function can be used.
count	Execute nmake all so that the COBOL count function can be used.
list	Execute nmake all using the COBOL list output function.

When a source file that affects dependencies is added or modified, do one of the following to redefine the dependencies:

- Execute the cobmkmf again.
- Recreate the SRCLIST using nmake srclist or update the SRCLIST using an editor, and then re-execute nmake depend.

Example

- To create a Makefile that generates MAIN01.EXE that uses the COBOL console with COBOL compiler option -WC,"ALPHAL(WORD)" (The COBOL source file name of the main program is assumed to be MAIN01.COB):

```
cobmkmf -w PROGRAM=MAIN01.EXE MAIN=MAIN01.COB "COBFLAGS=-WC,\"ALPHAL(WORD)\""
```

- To create a Makefile that generates MAIN02.EXE that uses the system console with COBOL compiler option -WC,"ALPHAL(WORD)" (The COBOL source file name of the main program is assumed to be MAIN02.COB):

```
cobmkmf PROGRAM=MAIN02.EXE MAIN=MAIN02.COB "COBFLAGS=-WC,\"ALPHAL(WORD)\""
```

- To create a Makefile that generates SUBLIB01.DLL with COBOL compiler option -WC,"ALPHAL(WORD)":

```
cobmkmf PROGRAM=SUBLIB01.DLL "COBFLAGS=-WC,\"ALPHAL(WORD)\""
```

- To create a debug library using the generated Makefile:

```
nmake debug
```

- To rebuild the executable programs and libraries using the generated Makefile in multi-thread mode:

```
nmake clean thread
```

Cautions

- cobmkmf generates the Makefile based on the assumption that all necessary resources - and only those necessary resources - to make an executable program or dynamic link library are in the current directory . Before executing the cobmkmf command:
  - Delete all unnecessary files from the current directory.
  - Copy all necessary files into the current directory.
- cobmkmf does not perform strict parsing. For example, an incorrect Makefile may be created if there is an erroneous dependency that generates a syntax error. Moreover, the contents of the current directory may need to be corrected before cobmkmf command execution time, since the SRCLIST is generated according to the contents of the current directory. Make corrections using a text editor, and use the generated Makefile only after it builds without errors.
- The following syntaxes in the COBOL source file are valid when determining dependencies:
  - PROGRAM-ID paragraph / Program-name paragraph / Method-name paragraph

- End program header / End class header / End method header
- The class specifier in the REPOSITORY paragraph
- COPY statement
- CALL statement

In the following cases, dependency may not be correctly identified:

- Two or more separately compiled programs exist in the same source file.
- The program name or class name defined in the source differs from the actual file name.
- A valid syntax is replaced by a source statement operation function.
- A valid syntax continues beyond a single line.
- A COPY statement occurs in the middle of a valid syntax.
- Only one of the following is included in the library text: the definition of the stored program or the CALL statement that calls it.

- cobmkmf executes nmake depend internally, and generates the dependency definition. When a dependency error is detected, an alert message is displayed.
- If the Makefile created by cobmkmf contains a recursive dependency such as a cross reference or an indirect reference, some source files may be compiled and linked at make command execution time, even if the source file has not been updated.
- cobmkmf can only be used to create a Makefile that generates one executable program or dynamic link library. cobmkmf cannot be used to create a Makefile that contains a cross-reference or an indirect reference between libraries.
- Execute the nmake command in the environment in which the compile and link environment variables are set.

I.4 CNVMED2UTF32 Command

The CNVMED2UTF32 command is a command that converts form descriptors created with PowerFORM into form descriptors using UTF-32.

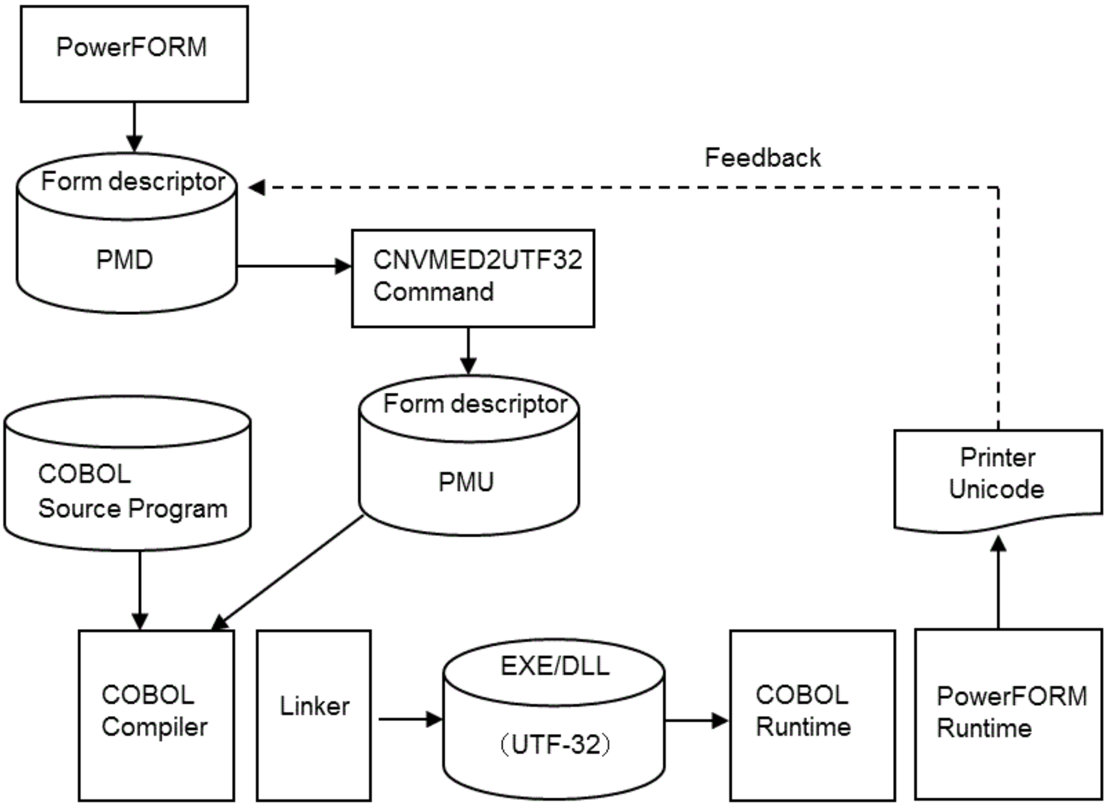
Form descriptors created with PowerFORM treat a UCS2 field as one character = two bytes. The programs do not treat one character = four bytes when using the encoding of a UCS2 field is assumed to be UTF-32 by the COBOL program. In the CNVMED2UTF32 command, form descriptors that convert the data length of a UCS2 field defined in form descriptors to UTF-32 are generated.

In the translation of a COBOL program that uses print file/presentation file (PRT) where the encoding of a UCS2 field is assumed to be UTF-32 with the FORMAT phrase, input form descriptors converted by the CNVMED2UTF32 command.

The following explains the procedure for development of the program that uses form descriptors when the CNVMED2UTF32 command is used and the form of the command.

Program development procedure

A standard program development procedure with form descriptors using the CNVMED2UTF32 command is shown below.



Command syntax

Command	Operand
CNVMED2UTF32	conversion-former-file-name [/D output-directory-name] [/C] [/P] [/Y]

Note

The CNVMED2UTF32 command is stored in the PowerFORM installation directory.

Operand

Options and file names must be separated by one or more blanks. The parameter enclosed in square brackets can be omitted. Either an absolute path or relative path can be specified for the file name as follows:

Table I.3 CNVMED2UTF32 command options

Options	Explanation
conversion-former-file-name	Identifies the form descriptor in the conversion origin. Multiple files can be specified by using an asterisk in the file name. However, files in the subdirectory are not converted. When using a blank for the file name, enclose it with a double quotation.
/D output-directory-name	Identifies the output directory for form descriptors. Output goes to the same directory as the former conversion file when a directory is not specified.
/C	Processing is continued if an error occurs during conversion.

Options	Explanation
/P	Error messages are not displayed.
/Y	Overwrite the existing file when the same file name exists.

Conversion former file name

Form descriptors made with PowerFORM become objects. The CNVMED2UTF32 command receives an error in the following situations.

- A file is not a form descriptor made with PowerFORM
- Form descriptors for e/c level exceed the 17th edition
- A broken form descriptors
- Form descriptors for UTF-32

Output file

The conversion further output file is form descriptors for UTF-32. The extension of the output file is as follows.

- file-name.PMD -> file-name.PMU

Note

- The extension of the output form descriptor does not change.
- Output for the form descriptor is used by translating and executing the COBOL program of which UTF-32 is the encoding of UCS2 Field.
- When the output form descriptor is assumed, the input of the COBOL program that makes the encoding of UCS2 Field excluding UTF-32, a translation error or a run-time error will generate.
- Output form descriptors cannot be changed with PowerFORM. Instead, change the original form descriptors and reconvert using the CNVMED2UTF32 command.

Error list

The message is displayed in the standard output.

Return value	Message	Corrective action
0	"Input file" was normally converted.	None
-1	Invalid parameter. Usage:CNVMED2UTF32.exe "file name" [/D "directory name at output destination"] [/C] [/P] [/Y]	Confirm the parameter.
-2	Out of memory.	Execute again after terminating unnecessary applications.
-101	The error occurred at the time of reading "Input file".	Confirm the state of the device that stores the input file.
-102	"Input file" does not exist.	Confirm the specification of the file name.
-103	The access to "Input file" was refused.	Confirm the access authority of the input file.
-104	"Input file" is not form descriptor.	Specify the form descriptors.
-105	"Input file" cannot be converted because the descriptor is not supported.	Specify the supported form descriptors.

Return value	Message	Corrective action
-110	The length of record area of the item exceeds 65535 bytes, "Input file" cannot be converted.	Adjust the length of the record area of the defined field.
-201	"Output directory" does not exist.	Confirm the output directory.
-202	"Output file name" exists at the output destination.	Confirm the file in the output directory. When overwriting a file, specify the /Y option.
-204	The access to "Output file" was refused.	Confirm the access authority of the output file and the output directory.
-205	The error occurred at the time of writing "Output file".	Confirm the disk free space of the output directory and the state of the device.
-206	The path of "Output file" is too long.	Use a short path for the output directory.

Appendix J Character Code-types

A character code is a mechanism that a computer uses to express characters. You do not have to be conscious of this fact if you are developing applications using COBOL, except in special circumstances. COBOL applies consistent rules for all operations, from character descriptions for describing source, to the assignment and comparison of characters used as data.

However, there are exceptions if you are developing applications. These exceptions are the following:

- Applications that operate in other systems
- Porting applications that operate in other systems to Windows systems

In this kind of situation, problems might occur because of differences between the character code used in Windows and the other system. You must understand enough about character codes to be able to resolve these problems. For this reason, this section explains the following topics:

- Character code summary
- Code-types that support COBOL products in each operating system
- Character code conversion
- Problems converting code to Shift JIS

Note

The phrase "character code" can mean the particular value of characters assigned individually, and it can also mean specifying the system for the assignment rules. In this section, however, it mainly has the latter meaning. The word "code" has the former meaning in this section.

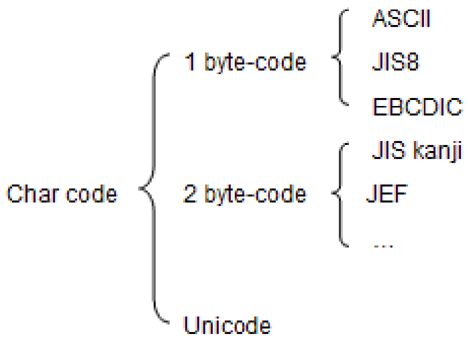
J.1 Character code summary

Character codes contain a number of systems, and the classification method contains a number of types. For this reason, you cannot just compare one code-type with another code-type. However, this section does explain the following code-type classifications in simplified terms:

- Difference in the number of bytes for expressing characters
- Mixed system character types

J.1.1 Code-type classification according to difference in the number of bytes for expressing characters

From the perspective of the number of bytes for expressing characters, the character code-types are classified as follows:



1 byte-type for expressing alphanumeric character signs is prepared, and other characters are contained in extensions after that.

ASCII

This is a code-type that was established by the US country standardization organization ANSI. It includes upper- and lower-case letters, numbers, control characters, and a few signs.

JIS8

This is a code that was established by the Japanese standardization organization JIS. It has inherited most of the ASCII code-types, but has made the following modifications:

- It uses "\" instead of the backslash
- It has added kana characters

EBCDIC

This is a code-type that was originally devised by IBM. It includes upper- and lower-case letters, numbers, control characters, and a few signs. A part of the character assignment now allows any assignment, and different variants exist according to country, region, and purpose. In Japan, the following code-types are mainly used:

- EBCDIC (kana): This contains added kana characters, and signs that are unique to Japanese
- EBCDIC (ASCII): This contains added lower-case letters
- EBCDIC (lower-case letters): This contains added lower-case letters, and signs that are unique to Japanese

2 byte-types have been prepared for expressing Japanese characters. As well as kanji, this includes katakana, hiragana, letters, numbers, and other signs. The general term for this type of code is "Kanji code".

JIS Kanji

This is Kanji code that has been defined using JIS X 0208. It was established in 1978, and revised twice afterwards. In 1983, the revision included the modification of about 400 characters. To establish which version is under discussion, revisions before this one are often called "78JIS", and revisions after are called "83JIS".

JEF

This code is based on 78JIS. It is a Kanji code that is unique to Fujitsu, and has been created for use with EBCDIC. It has the following characteristics:

- It contains many Kanji that are not contained in JIS Kanji.
- It contains all the Kanji contained in 78JIS, and in the same character order.
- It contains all the characters that have been added to 83JIS.

In Japan, these character code-types are not often used separately. They are generally used with mixed code-types, so that 1 byte-type codes and 2 byte-type codes can be mixed. This is explained in the next section.

The initial aim of Unicode was to bring together characters used throughout the world and unify them as a single code-type. For this reason, the behavior of Unicode is completely different to that of all other code-types explained up until now. For details about Unicode, refer to "Unicode".

J.1.2 Classification according to mixed system character types

The way to use mixed characters, in other words those that are expressed using 1 byte, such as alphanumeric characters, and those that are expressed using 2 bytes, has different variants according to the way that the character type is determined, and the character type that can be used.

This section explains the three main character types. These are shown below.

Shift JIS (SJIS)

This is a code-type that is widely used in PCs. It can be used to mix alphanumeric characters and kana characters (JIS8), and Japanese characters (JIS Kanji), and contains the following characteristics:

- You cannot use Shift code for character type switching.
- Alphanumeric characters and kana that are expressed using 1 byte contain the same values as JIS8 code-type.

- Although JIS Kanji is split into 4 areas, it supports calculations in line with regulations, and matches the order of the character code values with JIS Kanji.

Shift JIS also has an area for adding characters that are not contained in JIS Kanji. There are a number of different variants, depending on the differences in the characters added to the area. For example, 78JIS characters that are unique to Fujitsu, OASYS signs (R90), and Microsoft characters (MS-SJIS) have been added. In Windows systems, MS-SJIS is the normal standard.

EUC-JIS

This is a code-type that is widely used in UNIX-types. It can be used to mix kana (JIS kana) and Japanese character (JIS Kanji) with alphanumeric characters (ASCII) according to the ISO 2022 extension method, and contains the following characteristics:

- Although characters are expressed by 1 to 3 bytes, the character type can be determined in the first byte.
- Alphanumeric characters expressed using 1 byte contains the same values as ASCII code-types.
- All JIS Kanji are contained as 1 area.
- JIS kana is displayed with 1 byte Shift code.

There is also an area (G3) that can be used for adding characters according to the ISO 2022 extension system. This area also contains EUC (U90). This takes into account the area compatibility with SJIS (R90) and JEF before adding extension Kanji.

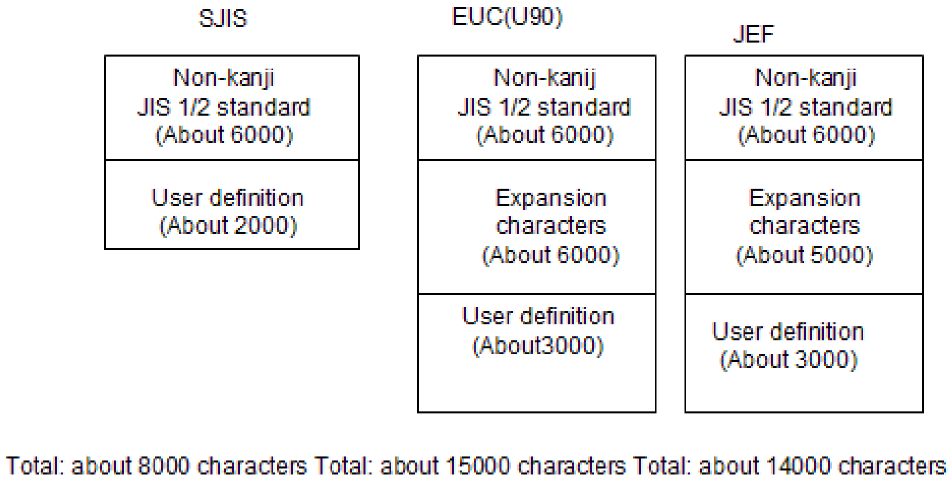
EBCIDIC-JEF

This is a code-type that is used in the Fujitsu OSIV series. It can be used to mix alphanumeric characters and kana characters (EBCDIC) with Japanese characters (JEF), and contains the following characteristics:

- It uses Shift code for switching character types.
- Each character code matches EBCDIC and JEF completely.

In each character code-type, some characters corresponding to the code contain undefined areas. A user can assign any characters to these areas. This is what is known as an "external character", or a "user defined character".

The table below gives an overview of the character range that can be used in each of the code-types.



J.2 Code-types supported in Fujitsu COBOL products

The character code-types that are supported in each Fujitsu COBOL product operating system are shown in table below.

Table J.1 Supported COBOL product code-types

System		Character code -type	Product name
Host-type			COBOL85
UNIX-type	Solaris	EUC	Fujitsu COBOL NetCOBOL
		SJIS	

System		Character code -type	Product name
		Unicode	NetCOBOL
	Linux	EUC	NetCOBOL
	HP-UX	EUC	Fujitsu COBOL
		SJIS	
Windows-type	Windows 2000 Windows XP Windows Server 2003	SJIS	Fujitsu COBOL NetCOBOL
		Unicode	
		EBCDIC+JEF	JEF option

J.3 Character code conversion

This is the way to exchange data between systems that have different code-types to code conversion.

The main ways to exchange code are as follows:

- Conversion where the characters match

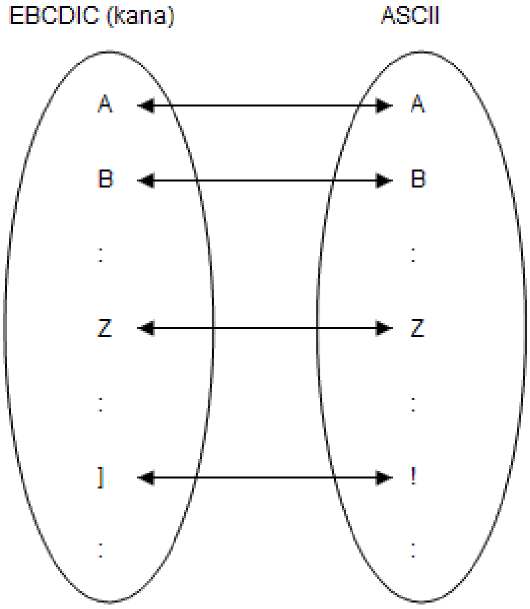
  The character type contained in the conversion target code-type is equal to or greater than the character type contained in the conversion source code-type. You can restore the original characters by converting in the reverse direction.

- Conversion where the characters do not match

  The character type contained in the conversion target code-type is less than the character type contained in the conversion source code-type, or a simple inclusive relationship cannot be sustained.

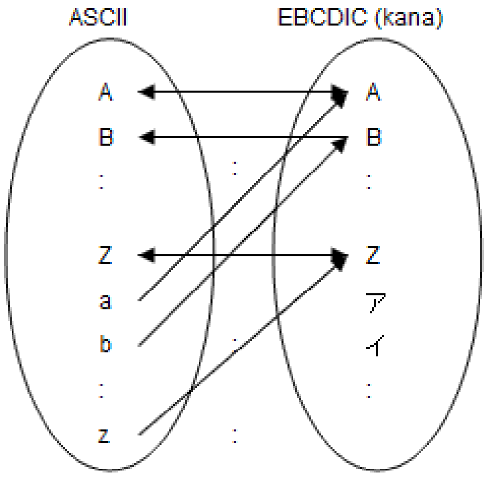
If you continue conversion in the latter case, you will have to convert to characters that are different from those in the conversion source. In this case, there are a further two methods that you may use.

Although you will have to use characters that are different from those in the conversion source, sometimes 1 to 1 support is retained for conversion. For example, to convert from EBCDIC to ASCII, although "!" is converted to "]", you can revert to the original state by converting in the reverse direction.



This kind of conversion is referred to as "conversion by substitute characters".

As well as support for using characters that are different to those in the conversion source, there is also support for changing converted characters back to source characters. For example, if you convert EBCDIC (kana) to ASCII, both "a" and "A" are converted to "A". If you have converted "a" to "A", however, you cannot change it back to "a" again even by reverse conversion.



This kind of conversion is referred to as "conversion by degeneration".

J.4 Problems converting code to Shift JIS

If you develop applications to be operated on other systems, or port applications operated on other systems to a Windows system, you must convert a part of the source program or data file code to Shift JIS.

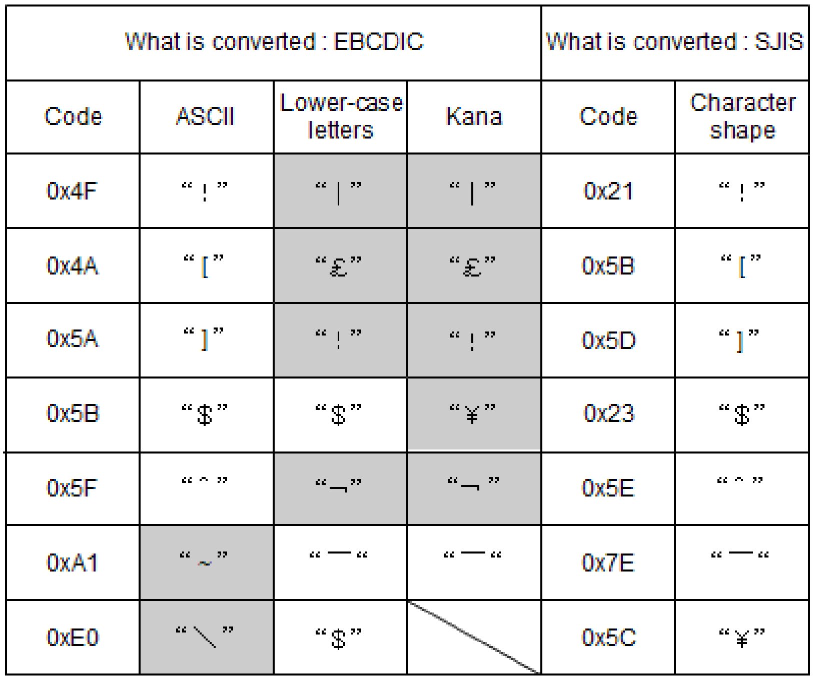
At this time, the following problems might arise because of incompatibility between Shift JIS and the original source program, and Shift JIS and the code-type used in the data file.

- An error occurs at the time of compilation
- An erroneous result or character distortion occurs at the time of execution

  The main causes of this trouble are shown below.

- Converting EBCIDC to Shift JIS using substitute characters

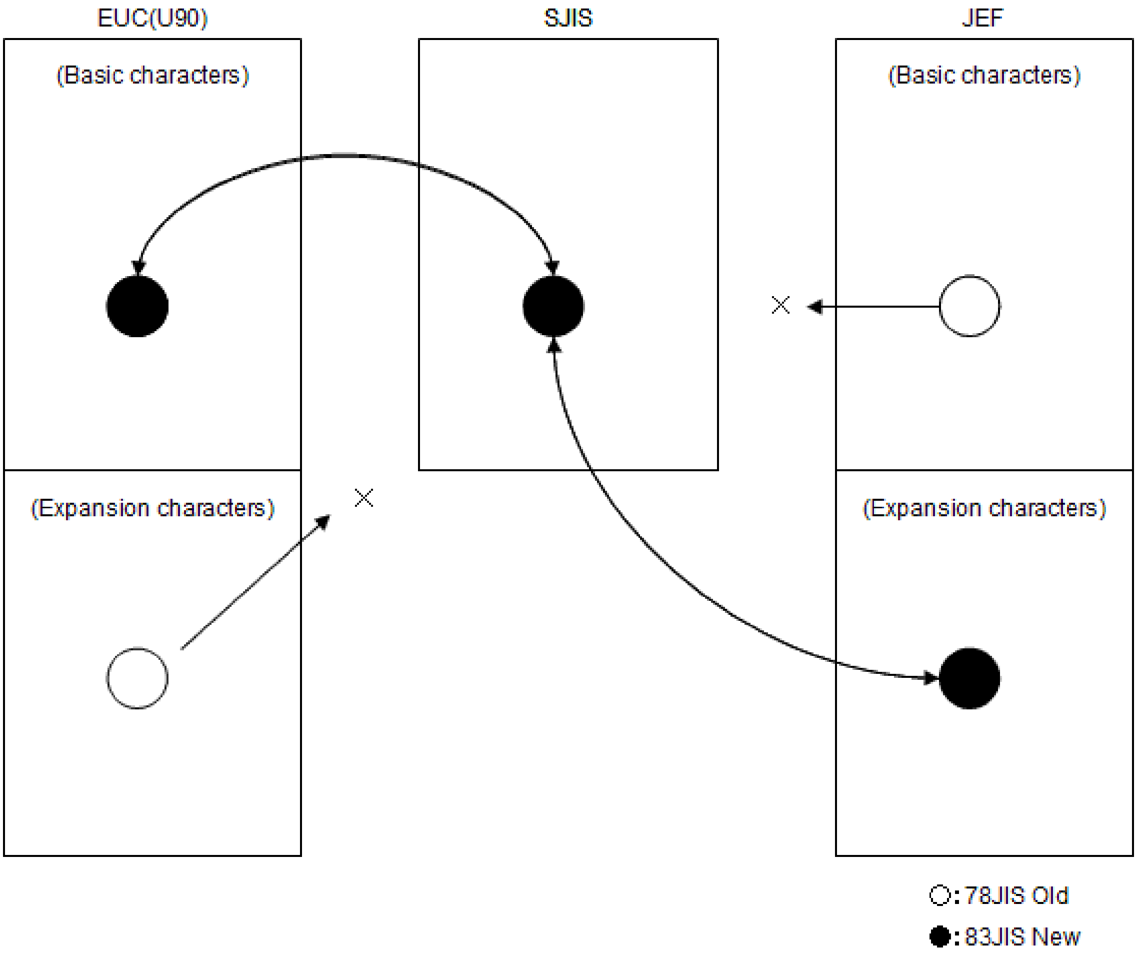
  The shaded part in the table below indicates conversion using substitute characters.



What is converted : EBCDIC				What is converted : SJIS	
Code	ASCII	Lower-case letters	Kana	Code	Character shape
0x4F	“ ! ”	“ | ”	“ | ”	0x21	“ ! ”
0x4A	“ [”	“£”	“£”	0x5B	“ [”
0x5A	“] ”	“ ! ”	“ ! ”	0x5D	“] ”
0x5B	“$”	“$”	“¥”	0x23	“$”
0x5F	“ ^ ”	“¬”	“¬ ”	0x5E	“ ^ ”
0xA1	“ ~ ”	“ — “	“ — “	0x7E	“ — “
0xE0	“ ＼ ”	“$”		0x5C	“¥”

- Converting characters that are incompatible with 78SJIS and 83JIS

  Although JEF and EUC (U90) are assigned a different code for each character style for characters that are incompatible with 78SJIS and 83JIS, only one code is assigned in SJIS. For this reason, 78JIS character information that is contained in JEF and EUC (U90) is generally lost.

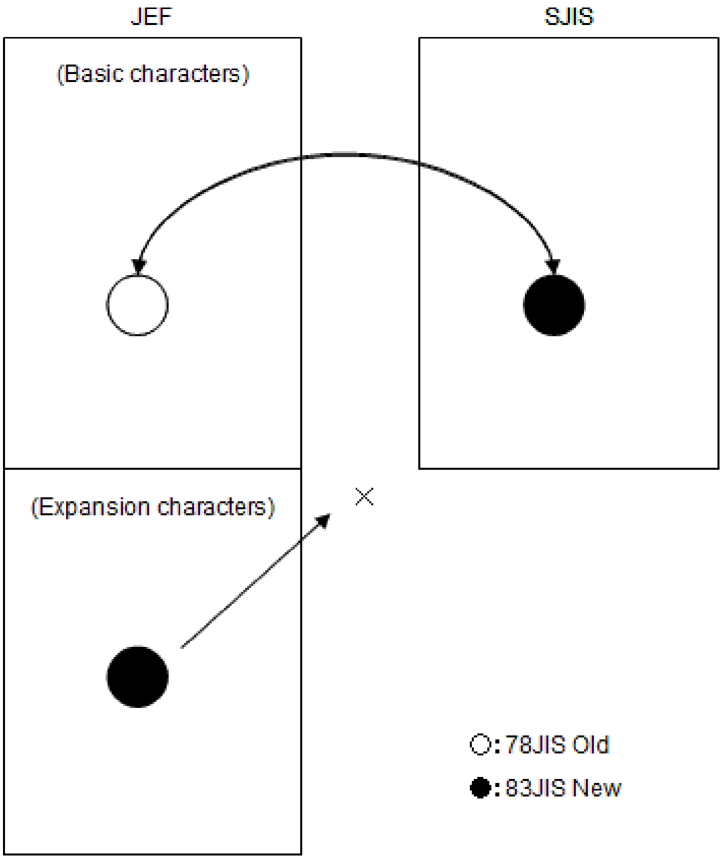


  If the source program or data file before conversion is EUC (U90), 78JIS old character style characters will be contained in extension character sets. For this reason, there will not be many problems regarding conversion.

  However, if the source program or data file before conversion is JEF, 78JIS old character style characters will be contained in basic character sets. For this reason, there might be a more serious effect. In this kind of situation, you must use one of the following conversion methods.

- Conversion that ignores character style

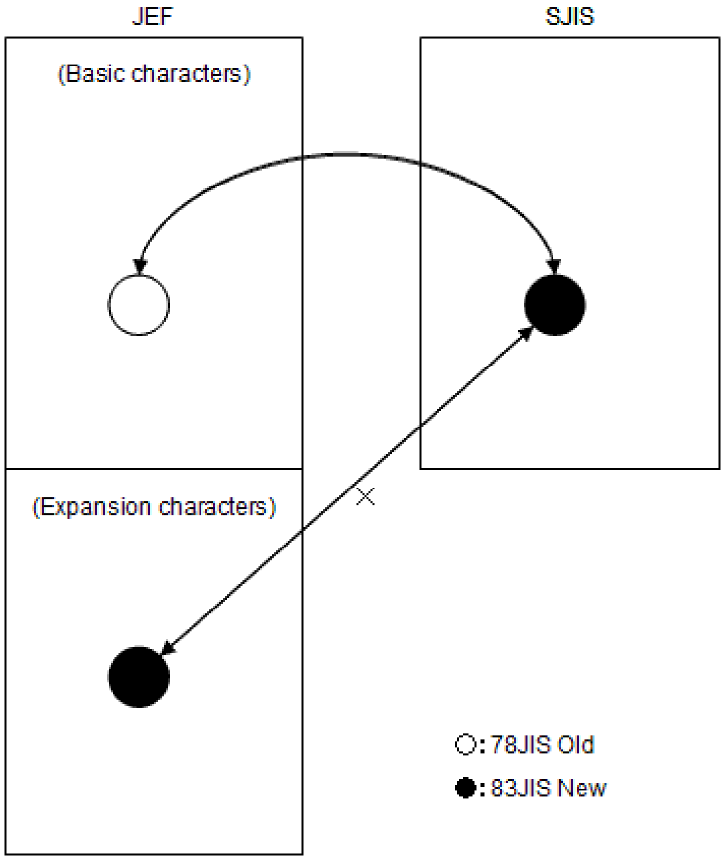
  This ignores the differences between 78JIS old characters and 83JIS new characters, and converts as follows:



If you use this method, the character style displayed on your PC will be different to the style before conversion. However, this method is recommended for OSIV application distribution development.

- Conversion that uses degeneration

  This uses 78JIS old character and 83JIS new character degeneration to convert.



If you use this method, you will not be able to distinguish between 78JIS old characters and 83JIS new characters. For this reason, this method cannot be recommended. However, this conversion method might be efficient if you are not particularly conscious about differences between character styles for porting applications from OSIV.

J.5 About JIS non-Kanji minus sign

When a national language user-defined word is specified to DISJOINING or to a JOINING phrase of a COPY statement, a JIS non-Kanji minus sign is used as a separator.

In Unicode, two minus sign characters are defined.

	UTF-8	UTF-16
MINUS SIGN	X"E28892"	X"2212"
FULLWIDTH HYPHEN-MINUS	X"EFBC8D"	X"FF0D"

When source programs and library texts are created in UTF-8, the NetCOBOL compiler uses "FULLWIDTH HYPHEN-MINUS" as a separator. If "MINUS SIGN" is used in national language user-defined words, the program doesn't work as intended.

This separator can be specified with the DUPCHAR compiler option.

- DUPCHAR(STD) : MINUS SIGN
- DUPCHAR(EXT) : FULLWIDTH HYPHEN-MINUS (default)

Appendix K Security

In a network environment, there is always a danger that the system and resources might be tampered with or destroyed, or that information might be leaked, because of illegal access. For this reason, you should use Web server user authorization functions and encrypted communication functions that are appropriate for the architecture of your system. Additionally, you should implement self-defense measures, such as restricting what users are able to do with applications.

K.1 Safeguarding Resources

To safeguard resources (such as database files, and input and output files), and definition and information files required for the operation of programs from illegal access and tampering, restrict access to OS functions and programs. Keep particularly important resources in a reliable and secure business segment (intranet environment).

To safeguard the information files that are required for programs and the operation of programs that have been installed on the Web server from illegal access and tampering, restrict access to OS functions.

K.2 Guidelines About Creating Applications

Consider the following areas before creating applications that take security issues into account.

- Initial check and notification of the processing result

  In the case of processing interactive dialogs or responses, check that everything is OK and notify the processing result before you access or process important data. If necessary, execute a design that can detect erroneous processing. It also helps the analysis of the processing if you record the log.

- Anonymity

  Take into account the risk of using data user that can reveal the real name and identify of a user, particular with regard to the leakage of information.

- Interface inspection

  For external interfaces, take into account buffer overflow (buffer overrun) and cross-site scripting to prevent the creation of a security hole. To prevent buffer overflow, it is helpful to inspect the length, type, and attributes of the external interface input data. To prevent cross-site scripting, you can make it so that unintended tags are not contained in pages that are created dynamically. For example, you can escape meta characters when they are output.

- Repeated executions

  Take into account restrictions on the number of requests that can be made from the same terminal within a certain period of time.

- Inspection log record

  Create a Web server or OS inspection log function, or an application log output process, to record security-related events and take into account the method to analyze and pursue security breaches when they arise.

- Rulemaking for security

  To create robust applications that do not suffer from fragile security-related processing, it is helpful to specify important resources that will safeguard against the threat of security breaches, and to make particular rules for access to resources and interface design.

K.3 Cross Site Scripting

Cross site scripting refers to the practice of inserting malicious content in the form of scripts such as JavaScript into Web pages. Users accessing the Web page can then unknowingly execute these scripts with diverse detrimental effects. Malicious script can also be used to intercept and falsify Cookie data, so that the Cookie can be used to gain authentication and thus hijack another user's session. There is also the danger that, using HTML tags rather than scripts, HTML that hides its true intention can be displayed.

NetCOBOL's CGI, ISAPI and SAF subroutines provide the ability to check, or sanitize, data in conversion character strings ensuring they don't contain characters that are vulnerable to cross site scripting attacks. See the subroutine user's guides for details of this support.

K.4 Remote Development Function

Although this product provides a remote development function for performing builds from other computers in a network, the remote development function has not been designed or made for use over the Internet. Use it only in a reliable and secure business segment (intranet environment).

Appendix L COBOL coding techniques

This section describes techniques for coding COBOL programs.

L.1 Efficient Programming

Programming changes that are made to enhance performance should also consider the performance aspects of the algorithm being used. Examining an algorithm for efficiency may produce better results than making any particular detailed coding changes. Taking an iterative approach to reviewing code for bottlenecks and making improvements is advised.

L.1.1 General Notes

Items in WORKING-STORAGE SECTION

- Examine the alignment of data in the WORKING-STORAGE section. Placing often-used data items close together can improve the behavior of the processor's memory cache and overall performance.
- Avoid unnecessary moves. For example, when initializing a group item, prefer instructions that do not write to the same elementary item more than once or that write to elementary items that do not require initialization. Code that first initializes an entire group item followed by instructions that initialize individual elementary items within that group item will cause additional machine level instructions to be executed.
- For constants whose value is not intended to be changed as part of program execution, set the initial value of the item using the VALUE clause.

Loop optimization

In loops, consider patterns such as the following that can reduce the number of instructions executed within a loop (as opposed to outside of the loop):

- Prefer the use of subscripts that are defined as table index names instead of using data names.
- Move items used in loop comparisons and loop index assignments to data items that exactly match the type of the values being compared against or moved into prior to the beginning of the loop.

Short circuit expression evaluation

Compound Boolean conditions connected by AND or OR are evaluated sequentially from left to right assuming there are no parentheses to alter the order of evaluation. Execution time can be shortened by writing such compound conditions as follows.

- Conditions that most often evaluate to true should be written prior to other conditions when using an OR operator.
- Conditions that most often evaluate to false should be written prior to other conditions when using an AND operator.

L.1.2 Selection of data item attribute

Alphanumeric versus numeric data items

Favor the use of alphanumeric items over numeric items in data definitions, because numeric items must always be translated to a numeric value internally prior to computations or comparisons.

For example, the bit pattern of a zoned decimal item such as a PIC S9 DISPLAY item, considers the bit patterns X'39' and X'49' to both represent the numeric value +9. The object code needed to interpret the numeric value is slower than what would be required in any byte-by-byte evaluation used with an alphanumeric item.

Numeric USAGE DISPLAY data items (zoned decimal data items)

Use this data item attribute for numeric values that are intended for display. The number of bytes required to represent numeric values using zoned decimal items and the amount of object code required to interpret values in this format is slower than any of the other numeric data description formats.

Numeric USAGE PACKED-DECIMAL data items (internal decimal data items)

Using this format is less costly in space and time than using zoned decimal data items, but slower than using binary items.

Numeric USAGE BINARY/COMP/COMP-5 data items (binary data items)

This format is best suited for operations and subscripts not intended to be displayed. Operations and comparisons are faster than with zoned decimal data items and internal decimal data items. Using COMP-5 is faster than using BINARY/COMP on little endian processors, such as on Windows, because it is the native representation of integers on such processors.

Sign of numeric data item

Avoid storing values that potentially have a sign in unsigned values, since this generates instructions to compute the absolute value of an expression prior to storing the value in the receiving field. Additionally, it is better not to specify either SIGN LEADING or SIGN SEPARATE when declaring the receiving item, as this also results in an extra instruction to handle the sign.

L.1.3 Numeric moves, numeric comparisons, and arithmetic operations

Data item attributes

- For operands in a move, for comparison operators, and for arithmetic operations, try to use operands that have the same USAGE clause, as this avoids data translation.
- For operands in a move, for comparison operators, and for addition and subtraction operations, try to have the operands match in the number of fractional digits declared. For multiplication and division operations, try to have the receiving item match in the number of fractional digits declared with the number of digits that the intermediate results of the multiplication or division will have.
    - Where the precision does not meet these guidelines, machine instructions must be inserted to convert and align the precision of the operands prior to each operation and comparison.
    - For an operation such as C=B/A, the precision of the operands is aligned when dc=db-da, where dc, db and da denote the number of fractional digits associated with data items C, B, and A, respectively. Similarly, for an operation such as C=B*A the precision of the operands is aligned when dc=db+da.
- When grouping operations in an arithmetic expression, try to specify an ordering of operations in which the precision and scale of the operands is either the same or increasing.

Number of digits

Do not use more digits than necessary when declaring data items, since operations and comparisons on data items with larger numbers of digits consumes more time.

Exponent

Using integer constants as the exponent in an exponent operation results in the best performance. Performance is much worse if non-integer values are specified for the exponent.

ROUNDED specification

Try to use the ROUNDED phrase sparingly, since machine instructions must be generated to adjust the result after computing an extra digit of precision.

ON SIZE ERROR specification

Try to use the ON SIZE ERROR phrase sparingly.

To check overflow when ON SIZE ERROR is specified, the following machine instructions are generated:

- When the result is computed in a binary item, the absolute value of the result is compared against the maximum value.
- When the result is computed in an internal decimal item, number of character positions of the result must be compared with the number of character positions of the receiving item.

TRUNC option

- Try to design programs so that the use of the TRUNC option is not required.
- When the TRUNC option is specified, binary item results must be divided by 10** n (n is the receiving item size) to determine the amount by which to round down results prior to storing the result. Therefore, specifying the TRUNC option can greatly impact performance in programs that heavily use binary data items.
- When the NOTRUNC option is specified, it is necessary to design programs so that programs do not attempt to store numeric values that exceed the number of character positions declared for receiving side items. Try to use the NOTRUNC option after ensuring that the program correctly excludes input values that do not match the scale and precision of the program's data item declarations.

L.1.4 Alphanumeric move and alphanumeric comparison

Boundary alignment

In general, processing can be more efficient when alphanumeric items are aligned to four or eight byte boundaries. Since the SYNCHRONIZED clause is only an annotation when applied to alphanumeric items, it is necessary to manually insert slack bytes in order to align alphanumeric items. Since overall memory usage is also increased by creating aligned data, try using this optimization for heavily used data items.

Item length

- Alphanumeric character comparisons perform best when the lengths of the operands are equal. Alphanumeric character moves perform best when the length of the sending item is equal to or greater than the length of the receiving item. When the sending item is a constant, performance can be improved by matching the length of the constant to the length of the receiving item.
- The above cases do not apply to a large item of hundreds of bytes or more.

Moving group items

When all items of a group item are moved, using one MOVE statement t move the group provides the best performance. The performance decreases when each item is moved using a separate MOVE statement.

L.1.5 Input/Output

SAME RECORD AREA clause

Use the SAME RECORD AREA clause only in the following cases:

- When the content of the record area is to be shared with two or more files.
- When it is necessary to use the record after the WRITE statement is executed.

Performance decreases when the READ/WRITE statement in the sequential file for which the SAME RECORD AREA clause is specified moves the record between the record area and the buffer area.

ACCEPT and DISPLAY statements

Avoid using the ACCEPT and DISPLAY statements where large amounts of data are input or output. Where larger amounts of data are involved, READ and WRITE statements should be used.

OPEN and CLOSE statements

Avoid executing OPEN and CLOSE statements whenever possible, since they are expensive operations.

L.1.6 Inter-program communication

Standard of subprogram division

When one system is composed of a lot of programs, better performance can be achieved by not subdividing programs more than necessary.

- The fewest number of machine instructions executed for a subprogram call is ten instructions in the case of a statically linked call. If the subprogram itself consists of very few instructions, the cost of making the subroutine call will have a more noticeable impact on performance. Where the number of instructions executed in the subprogram is in the hundreds of lines or more, the overhead of the call itself is negligible.
- More resources are used when system is divided into small subprograms, because each subprogram has initialization instructions, a termination routine, a work area, etc.

Dynamic program structure and dynamic link structure

Avoid using dynamic program calls (calls made using a "CALL identifier" statement or dynamic because of the use of the DLOAD compiler option), except when it is necessary to remove subprograms from memory to save on virtual memory in a very big system.

Using the dynamic link structure (linking with import libraries) is the preferred way to link your applications.

- Where dynamic program calls are made, in order to make the call, the runtime must first search to see if the subprogram has already been loaded and, if not, search for the program. Since this kind of processing needs to occur on every call, the overhead of dynamic program calls is greater than static program structures.
- In the dynamic linking structure, after loading the subprogram, the call is done directly. The overhead of this program structure is only slightly greater than the overhead of using the static link structure.

CANCEL statement

Avoid using the CANCEL statement as much as possible when using the dynamic program structure, since using it comes at a performance cost.

Number of parameters

Each parameter described in the USING phrase of the CALL statement, ENTRY statement, or PROCEDURE DIVISION header has its address set. Therefore, it is more efficient to have a smaller number of parameters in the USING phrase by combining parameter values into group items.

L.1.7 Debugging

- Remove the CHECK, COUNT, and TRACE compiler options after you complete debugging and recompile your source code to avoid the performance impacts of these options.
- Execution time can slow down by as much as a factor of two when the CHECK compiler option is specified. When the CHECK(NUMERIC) option is used, the performance impact of its use can be mitigated by replacing decimal items with binary items as much as possible.

L.2 Notes on numeric items

To avoid the occurrence of invalid data, review the following notes.

L.2.1 Decimal items

Decimal item input

- When an input record contains decimal items, data read from the file may contain invalid data. The compiler does not the check bit pattern when a READ statement is executed. Use a class condition (IF ... IS NUMERIC) to confirm that the bit pattern is correct after a READ is performed.
- The compiler does not check the bit pattern when storing data. Use CORRESPONDING in conjunction with a MOVE to assure valid bit patterns. If CORRESPONDING is not used, invalid bit patterns may result.

Invalid bit patterns

- The zone bits (excluding the sign part) of an external decimal item should be 0x3.
- When an item with an invalid bit pattern is moved, the result is undefined.

Checking characters during MOVE

When alphanumeric items containing spaces are moved to items defined as digits that do not allow spaces, the result is unpredictable. In (a) in the example below, the alphanumeric SND-DATA) is programmed to move to the RSV-DATA that is defined as a digit item PIC 9(4). If a space is encountered in the value of SND-DATA, the result is not predictable. Since the MOVE in (a) is not predictable, the comparison in (b) is also not predictable.

```
 01 SND-DATA PIC X(4).
 01 RSV-DATA PIC 9(4).
*>...
 PROCEDURE DIVISION.
     MOVE  SPACE  TO  SND-DATA
     *>...
     MOVE  SND-DATA  TO  RSV-DATA  *>... (a)
     IF  RSV-DATA  =  SPACE  THEN  *>... (b)
```

(SND-DATA) 20 20 20 20 --move--> (RSV-DATA) xx xx xx xx

Class conditions can be used to confirm that correct values are being stored. In the corrected program below, the class condition IS NUMERIC is used (c) to confirm that a correct value is stored.

```
 01 SND-DATA PIC X(4).
 01 RSV-DATA PIC 9(4).
*>...
PROCEDURE DIVISION.
    MOVE  SPACE  TO  SND-DATA
    MOVE   0     TO  RSV-DATA
    *>  ...
    IF SND-DATA IS NUMERIC THEN     *>... (c)
      MOVE  SND-DATA  TO  RSV-DATA
    ELSE
      DISPLAY "abnormal data" SND-DATA
    END-IF
```

L.2.2 Large value binary items

The value of a binary item can be larger than the value that the PICTURE clause describes. Furthermore, if two large binary items are added together, the PICTURE clause can erroneously describe a large negative value. Use the NOTRUNC compiler option to resolve this issue.

Only the digits that are contained in the PICTURE clause are displayed with the DISPLAY statement, even though the binary item may contain a value that is larger than that in the PICTURE clause. Use ON SIZE ERROR or NOT ON SIZE ERROR to check for overflow as part of the operation.

A binary item with a value larger than the PICTURE clause can describe may cause abnormal program termination.

L.2.3 Floating-point items

Conversion into Fixed-point

Conversion errors can be reduced by storing the result of arithmetic operations of floating point items in fixed-point.

L.2.4 Numeric items

Order of multiplication and division

In the following program, the [a] and [b] computations are similar, the only difference being the order of the multiplication and division operations. The [a] computation results in Z=333.0, and the [b] computation results in Z=333.33.

```
  77  X  PIC  S99  VALUE 10.
  77  Y  PIC  S9   VALUE 3.
  77  Z  PIC  S999V99.
```

```
PROCEDURE DIVISION.
    COMPUTE  Z = X / Y * 100.     *> [a]
    COMPUTE  Z = X * 100 / Y.     *> [b]
```

The different results are caused by the different number of decimal places for X and Z (described to the second decimal place), versus Y. In [a], X/Y = 3.33, which is then multiplied by 100 to produce 333.0. In [b], the intermediate result of X*100 is 1000, which is then divided by Y, resulting in 333.33.

It is therefore recommended that multiplication and division operations be ordered as shown in [b] to produce accurate results.

Move converted into absolute value

- When storing a signed value in a data item declared to be unsigned, the absolute value of the sending item is stored.
- When storing a signed value in a data item declared to be alphanumeric, the absolute value of the sending item is stored.

L.3 Notes

External Boolean item bit pattern

- The content of an external Boolean item must be hexadecimal 0x30 or 0x31. No other values are permitted.
- If Boolean items contain improper values in the top 6 bits, the result is unspecified.
- It is recommended that Boolean items be tested for proper values.

Reference to record area

- Do not refer to the record area of the file before an OPEN statement is executed or after executing a CLOSE statement.

Index